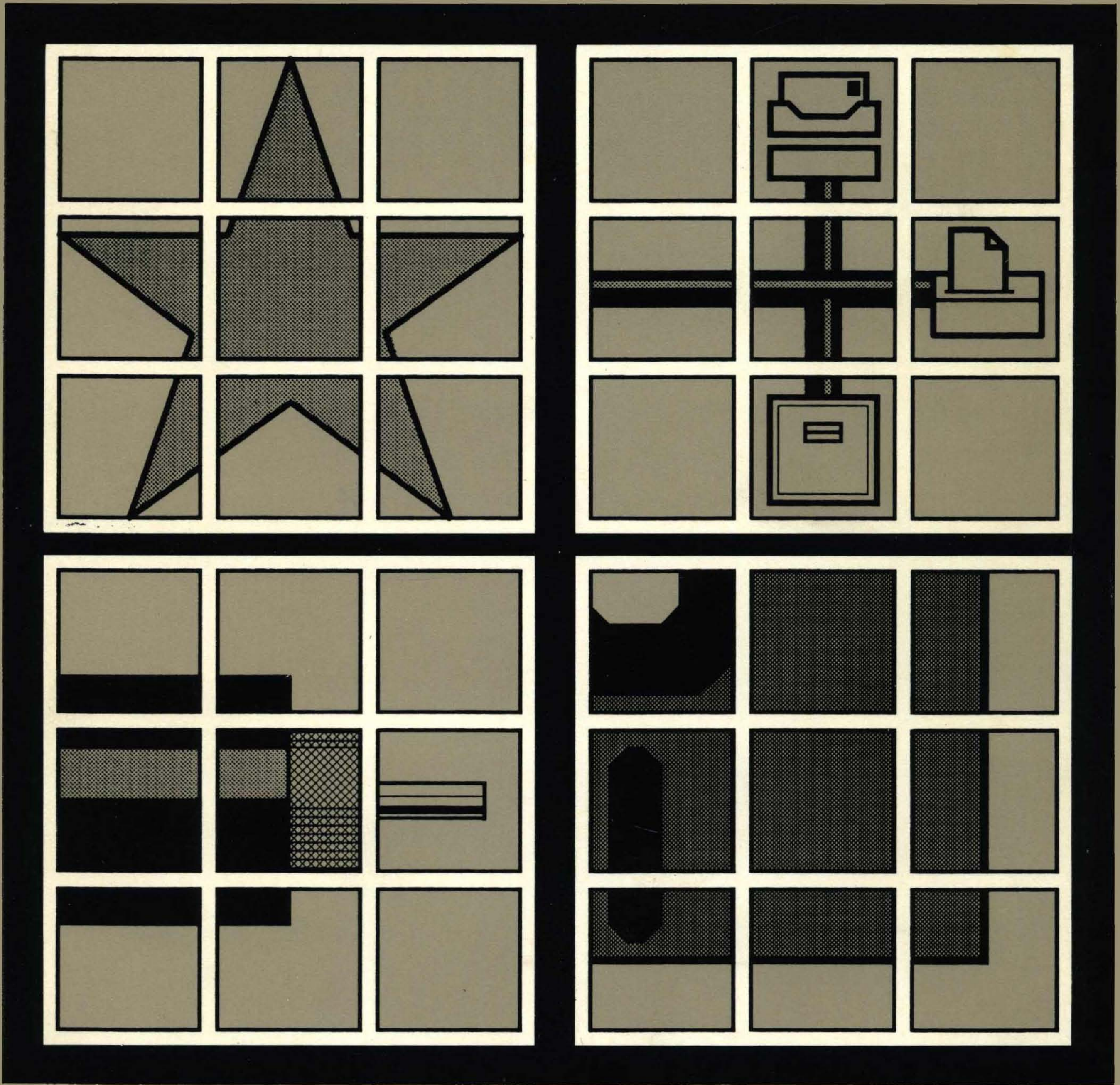
XEROX

Office Systems Technology

A Look into the World of the Xerox 8000 Series Products: Workstations, Services, Ethernet, and Software Development



Revised Edition

Office Systems Technology

A Look into the World of the Xerox 8000 Series Products: Workstations, Services, Ethernet, and Software Development

Managing Editors:
Ted Linden and Eric Harslem

OSD-R8203A
January 1984

XEROX

Office Systems Division
3333 Coyote Hill Road
Palo Alto, California 94304

Second Printing, 1984

All of the articles in this volume have been or will be published in other publications (as indicated in the Table of Contents). All articles are reprinted with permission.

Copyright © 1982, 1984 by Xerox Corporation.

Dedication

This book is dedicated to all the hardworking and creative people of Xerox' Systems Development Department who have seen the promise in the office of the future and made it into a reality.

Preface

This version of *Office Systems Technology* is basically identical to OSD-R8203 published in 1982, except for a few minor corrections, and some revision of the two articles starting on pages 65 and 91.

Notice

The papers reproduced in this book were written by different authors and were originally published at various times. Neither the original publication nor their republication implies any specific endorsement by the Xerox Corporation. Some statements in these papers are not valid concerning current products. Furthermore, no statement in these papers should be construed to imply a commitment or warranty by the Xerox Corporation concerning past, present, or future products.

Introduction

The members of the Xerox Systems Development Department, while creating the Xerox 8000 Series products, have explored many new frontiers in office systems technology. Many of their technical breakthroughs have been recorded in the open literature. This book gathers the majority of these publications together to make them more readily accessible.

This book is organized as follows: papers about new features that are visible to users of these products come first; papers about underlying technology come later. The first section has the papers about the user interface and functionality of the *8010 Workstation*; the second section has papers about the *Network Services* that support this and other workstations. The three succeeding sections cover: *Ethernet and Communications Protocols*, *Programming Language and Operating System*, and *Processor Architecture*. The final section has papers about the *Software Engineering* methodology that was used during the development of all these products.

In the first section dealing with the 8010 workstation, the first two papers describe the dramatically new user interface concepts that are employed–the first focusing on workstation features and the second on the user interface design goals. The next two papers describe, respectively, the design of the integrated graphics facility and the records processing functionality. The final paper in this section contains a comparative evaluation of text editors.

An office *system* is not just a collection of workstations. Network Services provide the functionality that make the difference between a collection of workstations and an office system. There are three papers about Network Services. The first describes the Clearinghouse, which enables a workstation to locate named resources in a widely distributed office system. User authentication is the cornerstone of most security and audit controls and presents some challenging problems in a distributed system–as discussed in the next paper. The final paper in this section describes the mail service developed by researchers at Xerox PARC. It has served as a prototype for the Mail Service and for other distributed services in the 8000 Series products. There are no published papers about the 8000 Series Print Service, File Service, or External Communication Service.

The glue that holds together all of the previous functions is the Ethernet and the Xerox Network Systems Communication Protocols. The first paper is an overview of communications and the office. The next paper describes the evolution of the Ethernet local area network. Office communications are not always local, and the remaining papers in this section deal with issues about building individual local networks into an effective, geographically-dispersed internetwork. The use of multiple local networks is covered in the third paper in this section, the fourth deals with addressing in an internetwork using 48-bit addresses, and the fifth describes the higher-level communication protocols.

Behind the scenes for all of these products is a programming language and operating system capable of supporting the incremental growth of a large office system. The fourth section deals with these topics. First there are two papers about Mesa, a practical programming language that incorporates many recent ideas from research on programming languages. The following paper on multiple inheritance subclassing describes the approach that was used to support object-oriented programming in the design and implementation of the 8000 Series products. The final paper discusses Pilot, the operating system used in all Xerox 8000 Series products.

The processor architecture for the Xerox 8000 Series products is the subject of the two papers in the fifth section. The first provides an overview of the Mesa processor architecture and the second reports the findings from an analysis of the Mesa instruction set.

Building an integrated office system is a large software engineering project. Pilot, the operating system in the 8000 Series products, provides one case study in software engineering which is discussed from different viewpoints in the first and fourth papers in this section. The Mesa language was designed to encourage the use of better software engineering methods, and that topic is examined in the second paper in this section. The third paper describes the software engineering techniques that were used during the development of the application code for the 8000 Series products.

This book itself exemplifies the use of the technology that it describes. The front cover design and front matter of this book were created using 8000 Series products. All of the recent papers were created using the Xerox 8000 Series products. While some of them were typeset for their original publication, the following papers are reproduced exactly as they were created and printed using 8000 Series products:

Star Graphics: An Object-Oriented Implementation
The Design of Star's Records Processing
The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment
Authentication in Office System Internetworks
Traits - An Approach to Multiple-Inheritance Subclassing
A Retrospective on the Development of Star

Acknowledgments

We are indebted to Paula Ann Balch and Stan Suk for many hours of work producing this collective volume. The front cover design was created by Norman Cox of the Xerox Office Products Division. Bill Verplank provided valuable assistance in the cover design.

No book on Xerox office systems technology would be complete without an acknowledgment of the pioneering research in this area done by our colleagues at the Xerox Palo Alto Research Center (PARC). Without them this book would not have been possible. The volume of PARC publications on office systems technology has generally prohibited us from including their publications here–unless one or more of the authors was a member of the Systems Development Department.

Office Systems Technology

Table of Contents

The 8010 Workstation

Network Services

Ethernet and Communications Protocols

Table of Contents (continued)

Programming Language and Operating System

Processor Architecture

Software Engineering

The star user interface: an overview

by DAVID CANFIELD SMITH, CHARLES IRBY, and RALPH KIMBALL
Xerox Corporation
Palo Alto, California
and
ERIC HARSLEM
Xerox Corporation
El Segundo, California

ABSTRACT

In April 1981 Xerox announced the 8010 Star Information System, a new personal computer designed for office professionals who create, analyze, and distribute information. The Star user interface differs from that of other office computer systems by its emphasis on graphics, its adherence to a metaphor of a physical office, and its rigorous application of a small set of design principles. The graphic imagery reduces the amount of typing and remembering required to operate the system. The office metaphor makes the system seem familiar and friendly; it reduces the alien feel that many computer systems have. The design principles unify the nearly two dozen functional areas of Star, increasing the coherence of the system and allowing user experience in one area to apply in others.

INTRODUCTION

In this paper we present the features in the Star system without justifying them in detail. In a companion paper,[1] we discuss the rationale for the design decisions made in Star. We assume that the reader has a general familiarity with computer text editors, but no familiarity with Star.

The Star hardware consists of a processor, a two-page-wide bit-mapped display, a keyboard, and a cursor control device. The Star software addresses about two dozen functional areas of the office, encompassing document creation; data processing; and electronic filing, mailing, and printing. Document creation includes text editing and formatting, graphics editing, mathematical formula editing, and page layout. Data processing deals with homogeneous databases that can be sorted, filtered, and formatted under user control. Filing is an example of a network service using the Ethernet local area network.[2,3] Files may be stored on a work station's disk (Figure 1), on a file server on the work station's network, or on a file server on a different network. Mailing permits users of work stations to communicate with one another. Printing uses laser-driven xerographic printers capable of printing both text and graphics. The term *Star* refers to the total system, hardware plus software.

As Jonathan Seybold has written, "This is a very different product: Different because it truly bridges word processing and typesetting functions; different because it has a broader range of capabilities than anything which has preceded it; and different because it introduces to the commercial market radically new concepts in human engineering."[4]

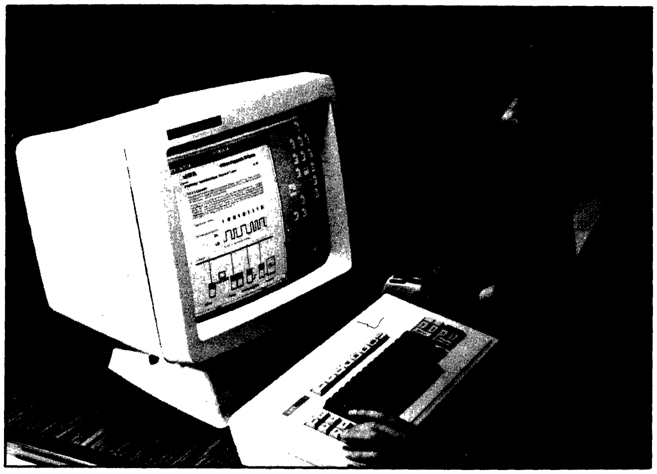


Figure 1—A Star workstation showing the processor, display, keyboard and mouse

The Star hardware was modeled after the experimental Alto computer developed at the Xerox Palo Alto Research Center.[5] Like Alto, Star consists of a Xerox-developed high-bandwidth MSI processor, local disk storage, a bit-mapped display screen having a 72-dot-per-inch resolution, a pointing device called the mouse, and a connection to the Ethernet. Stars are higher-performance machines than Altos, being about three times as fast, having 512K bytes of main memory (vs. 256K bytes on most Altos), 10 or 29M bytes of disk memory (vs. 2.5M bytes), a 10½-by-13½-inch display screen (vs. a 10½-by-8¼-inch one), 1024 × 808 addressable screen dots (vs. 606 × 808), and a 10M bits-per-second Ethernet (vs. 3M bits). Typically, Stars, like Altos, are linked via Ethernets to each other and to shared file, mail, and print servers. Communication servers connect Ethernets to one another either directly or over phone lines, enabling internetwork communication to take place. This means, for example, that from the user's perspective it is no harder to retrieve a file from a file server across the country than from a local one.

Unlike the Alto, however, the Star user interface was designed before the hardware or software was built. Alto software, of which there was eventually a large amount, was developed by independent research teams and individuals. There was little or no coordination among projects as each pursued its own goals. This was acceptable and even desirable in a research environment producing experimental software. But it presented the Star designers with the challenge of synthesizing the various interfaces into a single, coherent, uniform one.

ESSENTIAL HARDWARE

Before describing Star's user interface, we should point out that there are several aspects of the Star (and Alto) architecture that are essential to it. Without these elements, it would have been impossible to design a user interface anything like the present one.

Display

Both Star and Alto devote a portion of main memory to the bit-mapped display screen: 100K bytes in Star, 50K bytes (usually) in Alto. Every screen dot can be individually turned on or off by setting or resetting the corresponding bit in memory. This gives both systems substantial ability to portray graphic images.

Memory Bandwidth

Both Star and Alto have a high memory bandwidth—about 50 MHz, in Star. The entire Star screen is repainted from memory 39 times per second. This 50-MHz video rate would swamp most computer memories, and in fact refreshing the screen takes about 60% of the Alto's memory bandwidth. However, Star's memory is double-ported; therefore, refreshing the display does not appreciably slow down CPU memory access. Star also has separate logic devoted solely to refreshing the display.

Microcoded Personal Computer

Both Star and Alto are personal computers, one user per machine. Therefore the needed memory access and CPU cycles are consistently available. Special microcode has been written to assist in changing the contents of memory quickly, permitting a variety of screen processing that would otherwise not be practical.[6]

Mouse

Both Star and the Alto use a pointing device called the mouse (Figure 2). First developed at SRI,[7] Xerox's version has a ball on the bottom that turns as the mouse slides over a flat surface such as a table. Electronics sense the ball rotation and guide a cursor on the screen in corresponding motions. The mouse is a "Fitts's law" device: that is, after some practice you can point with a mouse as quickly and easily as you can with the tip of your finger. The limitations on pointing speed are those inherent in the human nervous system.[8,9] The mouse has buttons on top that can be sensed under program control. The buttons let you point to and interact with objects on the screen in a variety of ways.

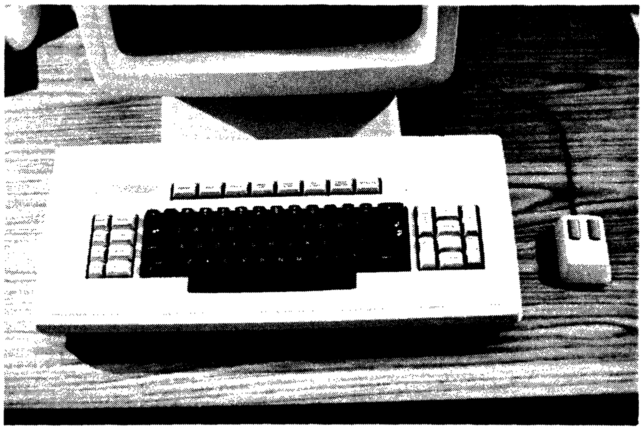


Figure 2—The Star keyboard and mouse

The keyboard has 24 easy-to-understand function keys. The mouse has two buttons on top.

Local Disk

Every Star and Alto has its own rigid disk for local storage of programs and data. Editing does not require using the network. This enhances the personal nature of the machines, resulting in consistent behavior regardless of how many other machines there are on the network or what anyone else is doing. Large programs can be written, using the disk for swapping.

Network

The Ethernet lets both Stars and Altos have a distributed architecture. Each machine is connected to an Ethernet. Other machines on the Ethernet are dedicated as *servers,* machines that are attached to a resource and that provide access to that resource. Typical servers are these:

1. *File server*—Sends and receives files over the network, storing them on its disks. A file server improves on a work station's rigid disk in several ways: (a) Its capacity is greater—up to 1.2 billion bytes. (b) It provides backup facilities. (c) It allows files to be shared among users. Files on a work station's disk are inaccessible to anyone else on the network.
2. *Mail server*—Accepts files over the network and distributes them to other machines on behalf of users, employing the Clearinghouse's database of names and addresses (see below).
3. *Print server*—Accepts print-format files over the network and prints them on the printer connected to it.
4. *Communication server*—Provides several services: The *Clearinghouse service* resolves symbolic names into network addresses.[10] The *Internetwork Routing service* manages the routing of information between networks over phone lines. The *Gateway service* allows word processors and dumb terminals to access network resources.

A network-based server architecture is economical, since many machines can share the resources. And it frees work stations for other tasks, since most server actions happen in the background. For example, while a print server is printing your document, you can edit another document or read your mail.

PHYSICAL OFFICE METAPHOR

We will briefly describe one of the most important principles that influenced the form of the Star user interface. The reader is referred to Smith et al.[1] for a detailed discussion of all the principles behind the Star design. The principle is to apply users' existing knowledge to the new situation of the computer. We decided to create electronic counterparts to the objects in an office: paper, folders, file cabinets, mail boxes, calculators, and so on—an electronic metaphor for the physical office. We hoped that this would make the electronic world seem more familiar and require less training. (Our initial experiences with users have confirmed this.) We further decided to make the electronic analogues be *concrete objects.*

Star documents are represented, not as file names on a disk, but as pictures on the display screen. They may be selected by pointing to them with the mouse and clicking one of the mouse buttons. Once selected, documents may be moved, copied, or deleted by pushing the MOVE, COPY, or DELETE key on the keyboard. Moving a document is the electronic equivalent of picking up a piece of paper and walking somewhere with it. To file a document, you move it to a picture of a file drawer, just as you take a piece of paper to a physical filing cabinet. To print a document, you move it to a picture of a printer, just as you take a piece of paper to a copying machine.

Though we want an analogy with the physical world for familiarity, we don't want to limit ourselves to its capabilities. One of the *raisons d'être* for Star is that physical objects do not provide people with enough power to manage the increasing complexity of their information. For example, we can take advantage of the computer's ability to search rapidly by providing a search function for its electronic file drawers, thus helping to solve the problem of lost files.

THE DESKTOP

Every user's initial view of Star is the Desktop, which resembles the top of an office desk, together with surrounding furniture and equipment. It represents a working environment, where current projects and accessible resources reside. On the screen (Figure 3) are displayed pictures of familiar office objects, such as documents, folders, file drawers, in-baskets, and out-baskets. These objects are displayed as small pictures, or *icons*.

You can "open" an icon by selecting it and pushing the OPEN key on the keyboard. When opened, an icon expands into a larger form called a *window,* which displays the icon's contents. This enables you to read documents, inspect the contents of folders and file drawers, see what mail has arrived, and perform other activities. Windows are the principal mechanism for displaying and manipulating information.

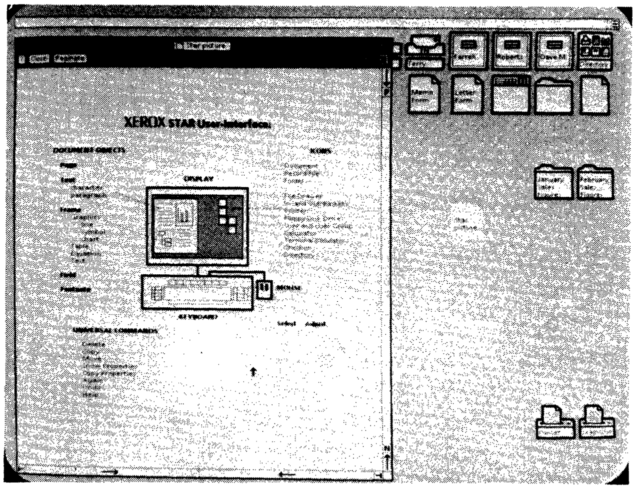


Figure 3—A "Desktop" as it appears on the Star screen

This one has several commonly used icons along the top, including documents to serve as "form pad" sources for letters, memos and blank paper. There is also an open window displaying a document.

The Desktop surface is displayed as a distinctive grey pattern. This is restful and makes the icons and windows on it stand out crisply, minimizing eye strain. The surface is organized as an array of 1-inch squares, 14 wide by 11 high. An icon may be placed in any square, giving a maximum of 154 icons. Star centers an icon in its square, making it easy to line up icons neatly. The Desktop always occupies the entire display screen; even when windows appear on the screen, the Desktop continues to exist "beneath" them.

The Desktop is the principal Star technique for realizing the physical office metaphor. The icons on it are visible, concrete embodiments of the corresponding physical objects. Star users are encouraged to think of the objects on the Desktop in physical terms. You can move the icons around to arrange your Desktop as you wish. (Messy Desktops are certainly possible, just as in real life.) You can leave documents on your Desktop indefinitely, just as on a real desk, or you can file them away.

ICONS

An *icon* is a pictorial representation of a Star object that can exist on the Desktop. On the Desktop, the size of an icon is approximately 1 inch square. Inside a window such as a folder window, the size of an icon is approximately ¼-inch square. Iconic images have played a role in human communication from cave paintings in prehistoric times to Egyptian hieroglyphics to religious symbols to modern corporate logos. Computer science has been slow to exploit the potential of visual imagery for presenting information, particularly abstract information. "Among [the] reasons are the lack of development of appropriate hardware and software for producing visual imagery easily and inexpensively; computer technology has been dominated by persons who seem to be happy with a simple, very limited alphabet of characters used to produce linear strings of symbols."[11] One of the authors has applied icons to an environment for writing programs; he found that they greatly facilitated human-computer communication.[12] Negroponte's Spatial Data Management system has effectively used iconic images in a research setting.[13] And there have been other efforts.[14,15,16] But Star is the first computer system designed for a mass market to employ icons methodically in its user interface. We do not claim that Star exploits visual communication to the ultimate extent; we do claim that Star's use of imagery is a significant improvement over traditional human-machine interfaces.

At the highest level the Star world is divided into two classes of icons, (1) data and (2) function icons:

Data Icons

Data icons (Figure 4) represent objects on which actions are performed. All data icons can be moved, copied, deleted, filed, mailed, printed, opened, closed, and have a variety of other operations performed on them. The three types of data icons are document, folder, and record file.

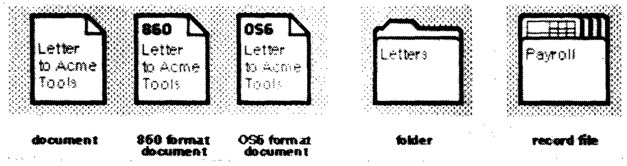


Figure 4—The "data" icons: document, folder and record file

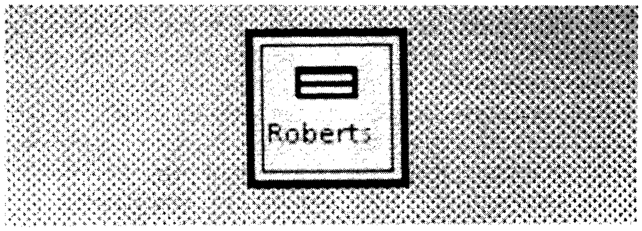


Figure 5—A file drawer icon

Document

A document is the fundamental object in Star. It corresponds to the standard notion of what a document should be. It most often contains text, but it may also include illustrations, mathematical formulas, tables, fields, footnotes, and formatting information. Like all data icons, documents can be shown on the screen, rendered on paper, sent to other people, stored on a file server or floppy disk, etc. When opened, documents are always rendered on the display screen exactly as they print on paper (informally called "what you see is what you get"), including displaying the correct type fonts, multiple columns, headings and footings, illustration placement, etc. Documents can reside in the system in a variety of formats (e.g., Xerox 860, IBM OS6), but they can be edited only in Star format. Conversion operations are provided to translate between the various formats.

Folder

A folder is used to group data icons together. It can contain documents, record files, and other folders. Folders can be nested inside folders to any level. Like file drawers (see below), folders can be sorted and searched.

Record file

A record file is a collection of information organized as a set of records. Frequently this information will be the variable data from forms. These records may be sorted, subset via pattern matching, and formatted into reports. Record files provide a rich set of information storage and retrieval functions.

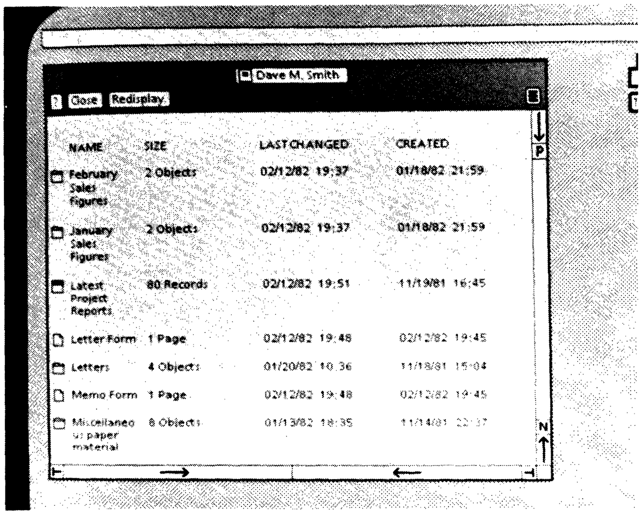
Function Icons

Function icons represent objects that perform actions. Most function icons will operate on any data icon. There are many kinds of function icons, with more being added as the system evolves:

File drawer

A file drawer (Figure 5) is a place to store data icons. It is modeled after the drawers in office filing cabinets. The organization of a file drawer is up to you; it can vary from a simple list of documents to a multilevel hierarchy of folders containing other folders. File drawers are distinguished from other storage places (folders, floppy disks, and the Desktop) in that (1) icons placed in a file drawer are physically stored on a file server, and (2) the contents of file drawers can be shared by multiple users. File drawers have associated access rights to control the ability of people to look at and modify their contents (Figure 6).

Although the design of file drawers was motivated by their physical counterparts, they are a good example of why it is neither necessary nor desirable to stop with just duplicating real-world behavior. People have a lot of trouble finding things in filing cabinets. Their categorization schemes are frequently ad hoc and idiosyncratic. If the person who did the categorizing leaves the company, information may be permanently lost. Star improves on physical filing cabinets by taking advantage of the computer's ability to *search rapidly*. You can search the contents of a file drawer for an object having a certain name, or author, or creation date, or size, or a variety of other attributes. The search criteria can use fuzzy patterns containing match-anything symbols, ranges, and other predicates. You can also sort the contents on the basis of those criteria. The point is that whatever information retrieval facilities are available in a system should be applied to



NAME	SIZE	LAST CHANGED	CREATED
February Sales Figures	2 Objects	02/12/82 19:37	01/18/82 21:59
January Sales Figures	2 Objects	02/12/82 19:37	01/18/82 21:59
Latest Project Reports	80 Records	02/12/82 19:51	11/19/81 16:45
Letter Form	1 Page	02/12/82 19:48	02/12/82 19:45
Letters	4 Objects	01/20/82 10:36	11/18/81 15:04
Memo Form	1 Page	02/12/82 19:48	02/12/82 19:45
Miscellaneous paper material	8 Objects	01/13/82 18:35	11/14/81 22:37

Figure 6—An open file drawer window

Note that there is a miniature icon for each object inside the file drawer.

the information in files. Any system that does not do so is not exploiting the full potential of the computer.

In basket and Out basket

These provide the principal mechanism for sending data icons to other people (Figure 7). A data icon placed in the Out basket will be sent over the Ethernet to a mail server (usually the same machine as a file server), thence to the mail servers of the recipients (which may be the same as the sender's), and thence to the In baskets of the recipients. When you have mail waiting for you, an envelope appears in your In basket icon. When you open your In basket, you can display and read the mail in the window.

Any document, record file, or folder can be mailed. Documents need not be limited to plain text, but can contain illustrations, mathematical formulas, and other nontext material. Folders can contain any number of items. Record files can be arbitrarily large and complex.

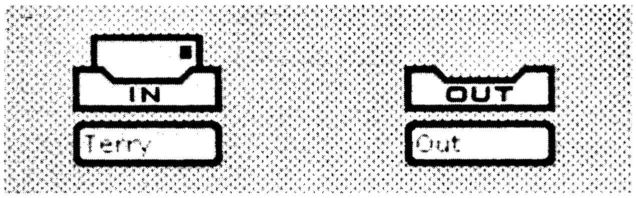


Figure 7—In and Out basket icons

Printer

Printer icons (Figure 8) provide access to printing services. The actual printer may be directly connected to your work station, or it may be attached to a print server connected to an Ethernet. You can have more than one printer icon on your Desktop, providing access to a variety of printing resources. Most printers are expected to be laser-driven raster-scan xerographic machines; these can render on paper anything that can be created on the screen. Low-cost typewriter-based printers are also available; these can render only text.

As with filing and mailing, the existence of the Ethernet greatly enhances the power of printing. The printer represented by an icon on your Desktop can be in the same room as your work station, in a different room, in a different building, in a different city, even in a different country. You perform exactly the same actions to print on any of them: Select a data icon, push the MOVE key, and indicate the printer icon as the destination.

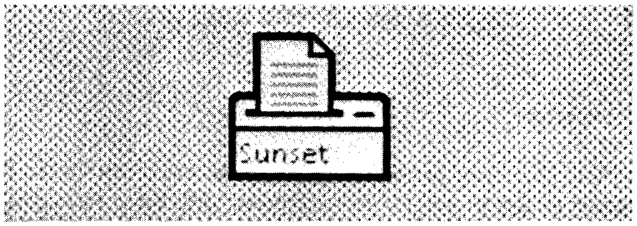


Figure 8—A printer icon

Floppy disk drive

The floppy disk drive icon (Figure 9) allows you to move data icons to and from a floppy disk inserted in the machine. This provides a way to store documents, record files and folders off line. When you open the floppy disk drive icon, Star reads the floppy disk and displays its contents in the window. Its window looks and acts just like a folder window: icons may be moved or copied in or out, or deleted. The only difference is the physical location of the data.

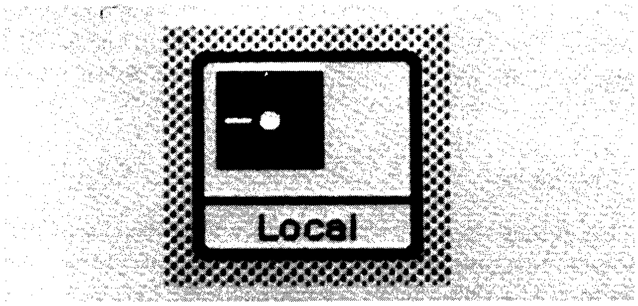


Figure 9—A floppy disk drive icon

User

The user icon (Figure 10) displays the information that the system knows about each user: name, location, password (invisible, of course), aliases if any, home file and mail servers, access level (ordinary user, system administrator, help/training writer), and so on. We expect the information stored for each user to increase as Star adds new functionality. User icons may be placed in address fields for electronic mail.

User icons are Star's solution to the naming problem. There is a crisis in computer naming of people, particularly in electronic mail addressing. The convention in most systems is to

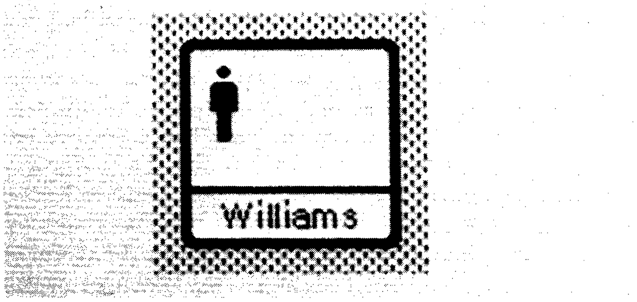


Figure 10—A user icon

use last names for user identification. Anyone named Smith, as is one of the authors, knows that this doesn't work. When he first became a user on such a system, *Smith* had long ago been taken. In fact, "D. Smith" and even "D. C. Smith" had been taken. He finally settled on "DaveSmith", all one word, with which he has been stuck to this day. Needless to say, that is *not* how he identifies himself to people. In the future, people will not tolerate this kind of antihumanism from computers. Star already does better: it follows society's conventions. User icons provide unambiguous unique references to individual people, using their normal names. The information about users, and indeed about all network resources, is physically stored in the Clearinghouse, a distributed database of names. In addition to a person's name in the ordinary sense, this information includes the name of the organization (e.g., Xerox, General Motors) and the name of the user's division within the organization. A person's linear name need be unique only within his division. It can be fully spelled out if necessary, including spaces and punctuation. Aliases can be defined. User icons are *references* to this information. You need not even know, let alone type, the unique linear representation for a user; you need only have the icon.

User group

User group icons (Figure 11) contain individual users and/or other user groups. They allow you to organize people according to various criteria. User groups serve both to control access to information such as file drawers (access control lists) and to make it easy to send mail to a large number of people (distribution lists). The latter is becoming increasingly important as more and more people start to take advantage of computer-assisted communication. At Xerox we have found that as soon as there were more than a thousand Alto users, there were almost always enough people interested in any topic whatsoever to form a distribution list for it. These user groups have broken the bonds of geographical proximity that have historically limited group membership and communication. They have begun to turn Xerox into a nationwide "village," just as the Arpanet has brought computer science researchers around the world closer together. This may be the most profound impact that computers have on society.

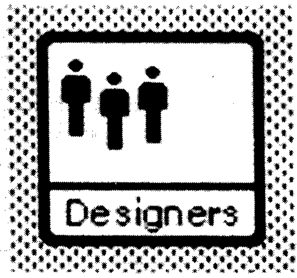


Figure 11—A user group icon

Calculator

A variety of styles of calculators (Figure 12) let you perform arithmetic calculations. Numbers can be moved between Star documents and calculators, thereby reducing the amount of typing and the possibility of errors. Rows or columns of tables can be summed. The calculators are user-tailorable and extensible. Most are modeled after pocket calculators—business, scientific, four-function—but one is a tabular calculator similar to the popular Visicalc program.

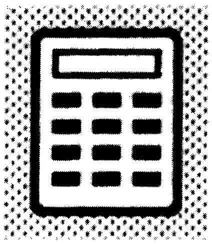


Figure 12—A calculator icon

Terminal emulators

The terminal emulators permit you to communicate with existing mainframe computers using existing protocols. Initially, teletype and 3270 terminals are emulated, with additional ones later (Figure 13). You open one of the terminal icons and type into its window; the contents of the window behave exactly as if you were typing at the corresponding terminal. Text in the window can be copied to and from Star documents, which makes Star's rich environment available to them.

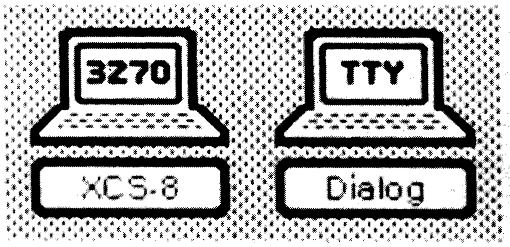


Figure 13—3270 and TTY emulation icons

Directory

The Directory provides access to network resources. It serves as the source for icons representing those resources; the Directory contains one icon for each resource available (Figure 14). When you are first registered in a Star network,

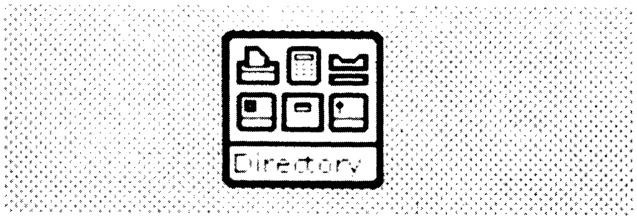
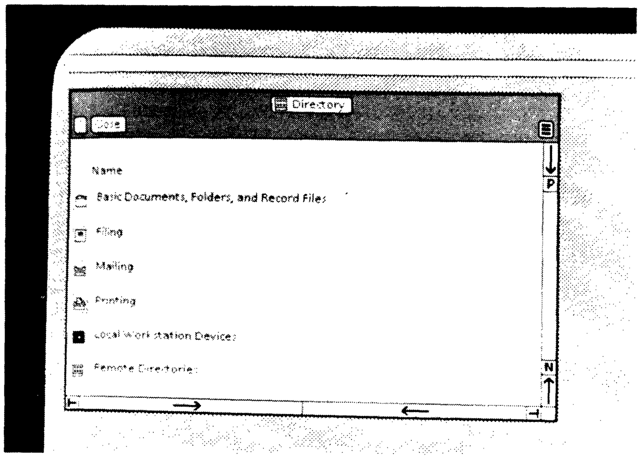


Figure 14—A Directory icon

your Desktop contains nothing but a Directory icon. From this initial state, you access resources such as file drawers, printers, and mail baskets by opening the Directory and copying out their icons. You can also get blank data icons out of the Directory. You can retrieve other data icons from file drawers. Star places no limits on the complexity of your Desktop except the limitation imposed by physical screen area (Figure 15). The Directory also contains Remote Directories representing resources available on other networks. These can be opened, recursively, and their resource icons copied out, just as with the local Directory. You deal with local and remote resources in exactly the same way.



Name

Basic Documents, Folders, and Record Files

Filing

Mailing

Printing

Local Work station Devices

Remote Directories

Figure 15—The Directory window, showing the categories of resources available

The important thing to observe is that although the functions performed by the various icons differ, the way you interact with them is the same. You select them with the mouse. You push the MOVE, COPY, or DELETE key. You push the OPEN key to see their contents, the PROPERTIES key to see their properties, and the SAME key to copy their properties. This is the result of rigorously applying the principle of uniformity to the design of icons. We have applied it to other areas of Star as well, as will be seen.

WINDOWS

Windows are rectangular areas that display the contents of icons on the screen. Much of the inspiration for Star's design came from Alan Kay's Flex machine[17] and his later Smalltalk programming environment on the Alto.[18] The Officetalk treatment of windows was also influential; in fact, Officetalk, an experimental office-forms-processing system on the Alto, provided ideas in a variety of areas.[19] Windows greatly increase the amount of information that can be manipulated on a display screen. Up to six windows at a time can be open in Star. Each window has a header containing the name of the icon and a menu of commands. The commands consist of a standard set present in all windows ("?", CLOSE, SET WINDOW) and others that depend on the type of icon. For example, the window for a record file contains commands tailored to information retrieval. CLOSE removes the window from the display screen, returning the icon to its tiny size. The "?" command displays the online documentation describing the type of window and its applications.

Each window has two scroll bars for scrolling the contents vertically and horizontally. The scroll bars have jump-to-end areas for quickly going to the top, bottom, left, or right end of the contents. The vertical scroll bar also has areas labeled N and P for quickly getting the next or previous screenful of the contents; in the case of a document window, they go to the next or previous page. Finally, the vertical scroll bar has a jumping area for going to a particular part of the contents, such as to a particular page in a document.

Unlike the windows in some Alto programs, Star windows do not overlap. This is a deliberate decision, based on our observation that many Alto users were spending an inordinate amount of time manipulating windows themselves rather than their contents. This manipulation of the medium is overhead, and we want to reduce it. Star automatically partitions the display space among the currently open windows. You can control on which side of the screen a window appears and its height.

PROPERTY SHEETS

At a finer grain, the Star world is organized in terms of *objects* that have *properties* and upon which *actions* are performed. A few examples of objects in Star are text characters, text paragraphs, graphic lines, graphic illustrations, mathematical summation signs, mathematical formulas, and icons. Every object has properties. Properties of text characters include type style, size, face, and posture (e.g., bold, italic). Properties of paragraphs include indentation, leading, and alignment. Properties of graphic lines include thickness and structure (e.g., solid, dashed, dotted). Properties of document icons include name, size, creator, and creation date. So the properties of an object depend on the type of the object. These ideas are similar to the notions of classes, objects, and messages in Simula[20] and Smalltalk. Among the editors that use these ideas are the experimental text editor Bravo[21] and the experimental graphics editor Draw,[22] both developed at the Xerox Palo Alto Research Center. These all supplied valuable knowledge and insight to Star. In fact, the text editor aspects of Star were derived from Bravo.

In order to make properties visible, we invented the notion of a property sheet (Figure 16). A property sheet is a two-dimensional formlike environment which shows the proper-

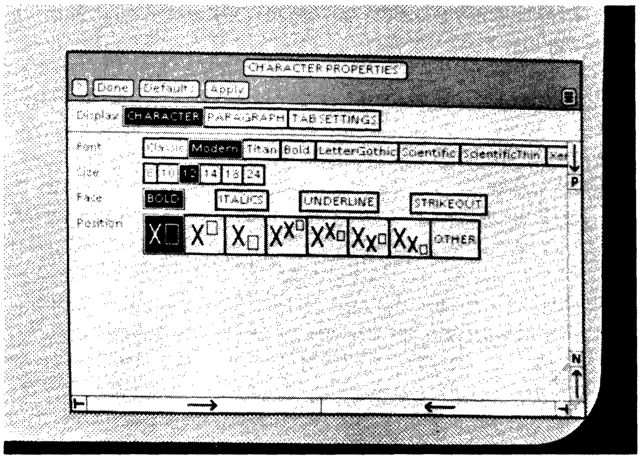


Figure 16—The property sheet for text characters

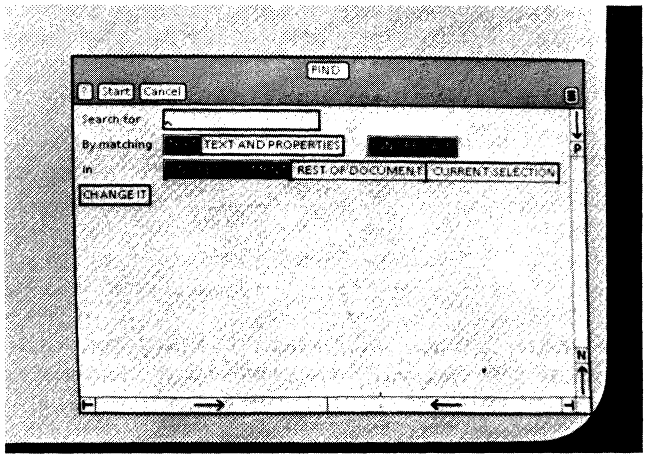


Figure 17—The option sheet for the Find command

ties of an object. To display one, you select the object of interest using the mouse and push the PROPERTIES key on the keyboard. Property sheets may contain three types of parameters:

1. *State*—State parameters display an independent property, which may be either on or off. You turn it on or off by pointing to it with the mouse and clicking a mouse button. When on, the parameter is shown video reversed. In general, any combination of state parameters in a property sheet can be on. If several state parameters are logically related, they are shown on the same line with space between them. (See "Face" in Figure 16.)
2. *Choice*—Choice parameters display a set of mutually exclusive values for a property. Exactly one value must be on at all times. As with state parameters, you turn on a choice by pointing to it with the mouse and clicking a mouse button. If you turn on a different value, the system turns off the previous one. Again the one that is on is shown video reversed. (See "Font" in Figure 16.) The motivation for state and choice parameters is the observation that it is generally easier to take a multiple-choice test than a fill-in-the-blanks one. When options are made visible, they become easier to understand, remember, and use.
3. *Text*—Text parameters display a box into which you can type a value. This provides a (largely) unconstrained choice space; you may type any value you please, within the limits of the system. The disadvantage of this is that the set of possible values is not visible; therefore Star uses text parameters only when that set is large. (See "Search for" in Figure 17.)

Property sheets have several important attributes:

1. A small number of parameters gives you a large number of combinations of properties. They permit a rich choice space without a lot of complexity. For example, the character property sheet alone provides for 8 fonts, from 1 to 6 sizes for each (an average of about 2), 4 faces (any combination of which can be on), and 8 positions relative to the baseline (including OTHER, which lets you type in a value). So in just four parameters, there are over $8 \times 2 \times 2^4 \times 8 = 2048$ combinations of character properties.
2. They show all of the properties of an object. None is hidden. You are constantly reminded what is available every time you display a property sheet.
3. They provide progressive disclosure. There are a large number of properties in the system as a whole, but you want to deal with only a small subset at any one time. Only the properties of the selected object are shown.
4. They provide a "bullet-proof" environment for altering the characteristics of an object. Since only the properties of the selected object are shown, you can't accidentally alter other objects. Since only valid choices are displayed, you can't specify illegal properties. This reduces errors.

Property sheets are an example of the Star design principle that *seeing and pointing* is preferred over *remembering and typing*. You don't have to remember what properties are available for an object; the property sheet will show them to you. This reduces the burden on your memory, which is particularly important in a functionally rich system. And most properties can be changed by a simple pointing action with the mouse.

The three types of parameters are also used in *option sheets*. (Figure 18). Option sheets are just like property sheets, except that they provide a visual interface for *arguments to commands* instead of *properties of objects*. For example, in the Find option sheet there is a text parameter for the string to search for, a choice parameter for the range over which to search, and a state parameter (CHANGE IT) controlling whether to replace that string with another one. When CHANGE IT is turned on, an additional set of parameters appears to contain the replacement text. This technique of having some parameters appear depending on the settings of others is another part of our strategy of progressive disclosure: hiding information (and therefore complexity) until it is needed, but making it visible when it is needed. The various sheets appear simpler than if all the options were always shown.

COMMANDS

Commands in Star take the form of noun-verb pairs. You specify the object of interest (the noun) and then invoke a command to manipulate it (the verb). Specifying an object is called *making a selection.* Star provides powerful selection mechanisms, which reduce the number and complexity of commands in the system. Typically, you exercise more dexterity and judgment in making a selection than in invoking a command. The ways to make a selection are as follows:

1. With the mouse—Place the cursor over the object on the screen you want to select and click the first (SELECT) mouse button. Additional objects can be selected by using the second (ADJUST) mouse button; it adjusts the selection to include more or fewer objects. Most selections are made in this way.
2. With the NEXT key on the keyboard—Push the NEXT key, and the system will select the contents of the next field in a document. Fields are one of the types of special higher-level objects that can be placed in documents. If the selection is currently in a table, NEXT will step through the rows and columns of the table, making it easy to fill in and modify them. If the selection is currently in a mathematical formula, NEXT will step through the various elements in the formula, making it easy to edit them. NEXT is like an intelligent step key; it moves the selection between semantically meaningful locations in a document.
3. With a command—Invoke the FIND command, and the system will select the next occurrence of the specified text, if there is one. Other commands that make a selection include OPEN (the first object in the opened window is selected) and CLOSE (the icon that was closed becomes selected). These optimize the use of the system.

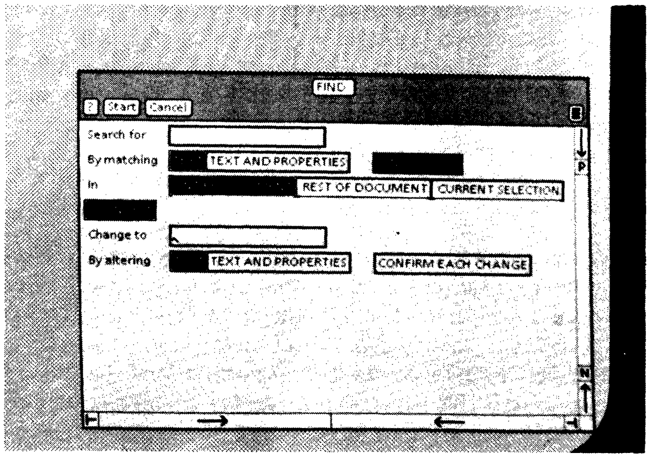


Figure 18—The Find option sheet showing Substitute options *(The extra options appear only when CHANGE IT is turned on)*

The object (noun) is almost always specified before the action (verb) to be performed. This makes the command interface *modeless;* you can change your mind as to which object to affect simply by changing the selection before invoking the command.[23] No "accept" function is needed to terminate or confirm commands, since invoking the command is the last step. Inserting text does not require a command; you simply make a selection and begin typing. The text is placed after the end of the selection. A few commands require more than one operand and hence are modal. For example, the MOVE and COPY commands require a destination as well as a source.

GENERIC COMMANDS

Star has a few commands that can be used throughout the system: MOVE, COPY, DELETE, SHOW PROPERTIES, COPY PROPERTIES, AGAIN, UNDO, and HELP. Each performs the same way regardless of the type of object selected. Thus we call them generic commands. For example, you follow the same set of actions to move text in a document as to move a document in a folder or a line in an illustration: select the object, move the MOVE key, and indicate the destination. Each generic command has a key devoted to it on the keyboard. (HELP and UNDO don't use a selection.)

These commands are more basic than the ones in other computer systems. They strip away extraneous application-specific semantics to get at the underlying principles. Star's generic commands are derived from *fundamental computer science concepts* because they also underlie operations in programming languages. For example, program manipulation of data structures involves moving or copying values from one data structure to another. Since Star's generic commands embody fundamental underlying concepts, they are widely applicable. Each command fills a host of needs. Few commands are required. This simplicity is desirable in itself, but it has another subtle advantage: it makes it easy for users to form a model of the system. What people can understand, they can use. Just as progress in science derives from simple, clear theories, so progress in the usability of computers depends on simple, clear user interfaces.

Move

MOVE is the most powerful command in the system. It is used during text editing to rearrange letters in a word, words in a sentence, sentences in a paragraph, and paragraphs in a document. It is used during graphics editing to move picture elements such as lines and rectangles around in an illustration. It is used during formula editing to move mathematical structures such as summations and integrals around in an equation. It replaces the conventional "store file" and "retrieve file" commands; you simply move an icon into or out of a file drawer or folder. It eliminates the "send mail" and "receive mail" commands; you move an icon to an Out basket or from an In basket. It replaces the "print" command; you move an icon to a printer. And so on. MOVE strips away much of the historical clutter of computer commands. It is more fundamental than the myriad of commands it replaces. It is simultaneously more powerful and simpler.

MOVE also reinforces Star's physical metaphor: a moved object can be in only one place at one time. Most computer file transfer programs only make copies; they leave the originals behind. Although this is an admirable attempt to keep information from accidentally getting lost, an unfortunate side effect is that sometimes you lose track of where the most recent information is, since there are multiple copies floating around. MOVE lets you model the way you manipulate information in the real world, should you wish to. We expect that during the *creation* of information, people will primarily use MOVE; during the *dissemination* of information, people will make extensive use of COPY.

Copy

COPY is just like MOVE, except that it leaves the original object behind untouched. Star elevates the concept of copying to the level of *a paradigm for creating.* In all the various domains of Star, you *create by copying.* Creating something out of nothing is a difficult task. Everyone has observed that it is easier to modify an existing document or program than to write it originally. Picasso once said, "The most awful thing for a painter is the white canvas. . . . To copy others is necessary."[24] Star makes a serious attempt to alleviate the problem of the "white canvas," to make copying a practical aid to creation. Consider:

- You create new documents by copying existing ones. Typically you set up blank documents with appropriate formatting properties (e.g., fonts, margins) and then use those documents as *form pad* sources for new documents. You select one, push COPY, and presto, you have a new document. The form pad documents need not be blank; they can contain text and graphics, along with fields for variable text such as for business forms.
- You place new network resource icons (e.g., printers, file drawers) on your Desktop by copying them out of the Directory. The icons are registered in the Directory by a system administrator working at a server. You simply copy them out; no other initialization is required.
- You create graphics by copying existing graphic images and modifying them. Star supplies an initial set of such images, called *transfer symbols.* Transfer symbols are based on the idea of dry-transfer rub-off symbols used by many secretaries and graphic artists. Unlike the physical transfer symbols, however, the computer versions can be modified: they can be moved, their sizes and proportions can be changed, and their appearance properties can be altered. Thus a single Star transfer symbol can produce a wide range of images. We will eventually supply a set of documents (transfer sheets) containing nothing but special images tailored to one application or another: people, buildings, vehicles, machinery. Having these as sources for graphics copying helps to alleviate the "white canvas" feeling.
- In a sense, you can even type characters by copying them from keyboard windows. Since there are many more characters (up to 2^{16}) in the Star character set than there are keys on the keyboard, Star provides a series of keyboard interpretation windows (Figure 19), which allow you to see and change the meanings of the keyboard keys. You are presented with the options; you look them over and choose the ones you want.

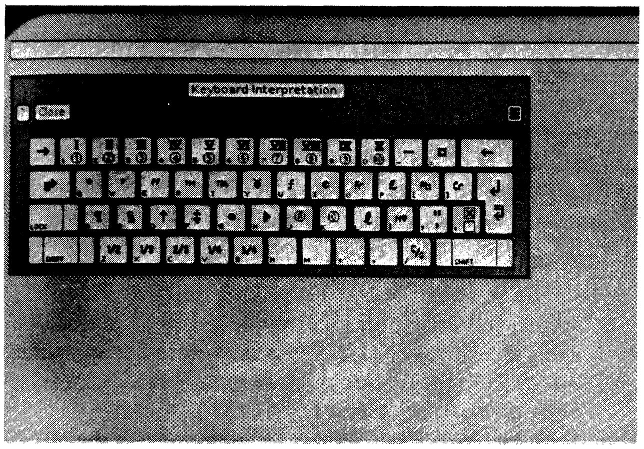


Figure 19—The Keyboard Interpretation window

This displays other characters that may be entered from the keyboard. The character set shown here contains a variety of common office symbols.

Delete

This deletes the selected object. If you delete something by mistake, UNDO will restore it.

Show Properties

SHOW PROPERTIES displays the properties of the selected object in a property sheet. You select the object(s) of interest, push the PROPERTIES (PROP'S) key, and the appropriate property sheet appears on the screen in such a position as to not overlie the selection, if possible. You may change as many properties as you wish, including none. When finished, you invoke the Done command in the property sheet menu. The property changes are applied to the selected objects, and the property sheet disappears. Notice that SHOW PROPERTIES is therefore used both to examine the current properties of an object and to change those properties.

Copy Properties

You need not use property sheets to alter properties if there is another object on the screen that already has the desired properties. You can select the object(s) to be changed, push the SAME key, then designate the object to use as the source. COPY PROPERTIES makes the selection look the "same" as the source. This is particularly useful in graphics editing. Frequently you will have a collection of lines and symbols whose appearance you want to be coordinated (all the same line width, shade of grey, etc.). You can select all the objects to be changed, push SAME, and select a line or symbol having the desired appearance. In fact, we find it helpful to set up a document with a variety of graphic objects in a variety of appearances to be used as sources for copying properties.

Again

AGAIN repeats the last command(s) on a new selection. All the commands done since the last time a selection was made are repeated. This is useful when a short sequence of commands needs to be done on several different selections; for example, make several scattered words bold and italic and in a larger font.

Undo

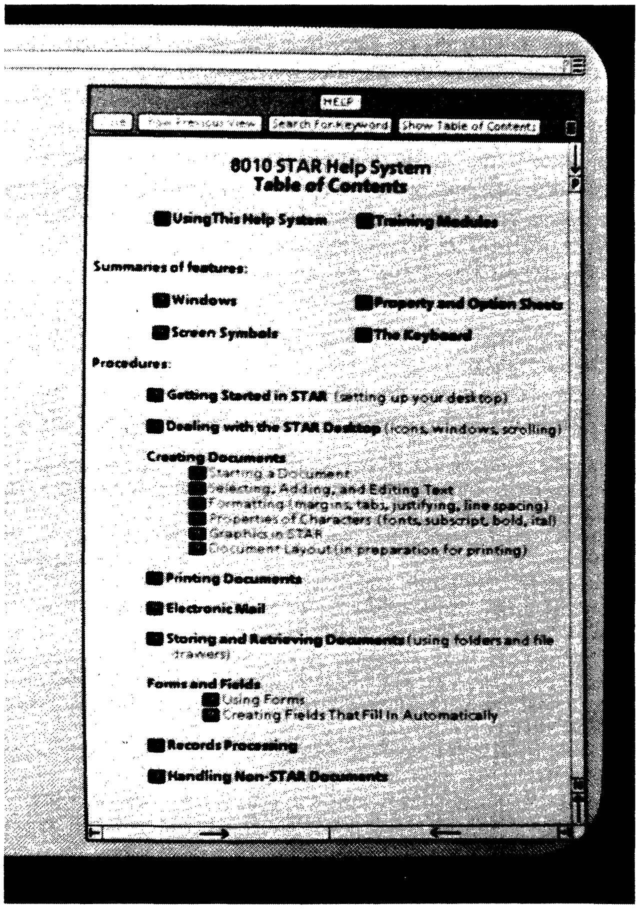
UNDO reverses the effects of the last command. It provides protection against mistakes, making the system more forgiving and user-friendly. Only a few commands cannot be repeated or undone.

Help

Our effort to make Star a personal, self-contained system goes beyond the hardware and software to the tools that Star provides to teach people how to use the system. Nearly all of its teaching and reference material is on line, stored on a file server. The Help facilities automatically retrieve the relevant material as you request it.

The HELP key on the keyboard is the primary entrance into this online information. You can push it at any time, and a window will appear on the screen displaying the Help table of contents (Figure 20). Three mechanisms make finding information easier: *context-dependent invocation, help references,* and a *keyword search command.* Together they make the online documentation more powerful and useful than printed documentation.

- *Context-dependent invocation*—The command menu in every window and property/option sheet contains a "?" command. Invoking it takes you to a part of the Help documentation describing the window, its commands, and its functions. The "?" command also appears in the message area at the top of the screen; invoking that one takes you to a description of the message (if any) currently in the message area. That provides more detailed explanations of system messages.
- *Help references*—These are like menu commands whose effect is to take you to a different part of the Help material. You invoke one by pointing to it with the mouse, just as you invoke a menu command. The writers of the material use the references to organize it into a network of interconnections, in a way similar to that suggested by Vannevar Bush[25] and pioneered by Doug Engelbart in his NLS system.[26,27] The interconnections permit cross-referencing without duplication.
- The *SEARCH FOR KEYWORD command*—This command in the Help window menu lets you search the available documentation for information on a specific topic. The keywords are predefined by the writers of the Help material.



8010 STAR Help System
Table of Contents

Using This Help System
Training Modules

Summaries of features:

Windows
Property and Option Sheets
Screen Symbols
The Keyboard

Procedures:

Getting Started in STAR (setting up your desktop)

Dealing with the STAR Desktop (icons, windows, scrolling)

Creating Documents
Starting a Document
Selecting, Adding, and Editing Text
Formatting (margins, tabs, justifying, line spacing)
Properties of Characters (fonts, subscript, bold, ital)
Graphics in STAR
Document Layout (in preparation for printing)

Printing Documents

Electronic Mail

Storing and Retrieving Documents (using folders and file drawers)

Forms and Fields
Using Forms
Creating Fields That Fill In Automatically

Records Processing

Handling Non-STAR Documents

Figure 20—The Help window, showing the table of contents

Selecting a square with a question mark in it takes you to the associated part of the Help documentation.

SUMMARY

We have learned from Star the importance of formulating the user's conceptual model *first, before* software is written, rather than tacking on a user interface *afterward.* Doing good user interface design is not easy. Xerox devoted about thirty work-years to the design of the Star user interface. It was designed *before* the functionality of the system was fully decided. It was designed *before* the computer hardware was even built. We worked for two years *before* we wrote a single line of actual product software. Jonathan Seybold put it this way: "Most system design efforts start with hardware specifications, follow this with a set of functional specifications for the software, then try to figure out a logical user interface and command structure. The Star project started the other way around: the paramount concern was to define a conceptual model of how the user would relate to the system. Hardware and software followed from this."[4]

Alto served as a valuable prototype for Star. Over a thousand Altos were eventually built, and Alto users have had several thousand work-years of experience with them over a period of eight years, making Alto perhaps the largest prototyping effort in history. There were dozens of experimental programs written for the Alto by members of the Xerox Palo Alto Research Center. Without the creative ideas of the authors of those systems, Star in its present form would have been impossible. On the other hand, it was a real challenge to bring some order to the different user interfaces on the Alto. In addition, we ourselves programmed various aspects of the Star design on Alto, but every bit (sic) of it was throwaway code. Alto, with its bit-mapped display screen, was powerful enough to implement and test our ideas on visual interaction.

REFERENCES

1. Smith, D. C., E. F. Harslem, C. H. Irby, R. B. Kimball, and W. L. Verplank. "Designing the Star User Interface." *Byte,* April 1982.
2. Metcalfe, R. M., and D. R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM,* 19 (1976), pp. 395–404.
3. Intel, Digital Equipment, and Xerox Corporations. "The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (version 1.0)." Palo Alto: Xerox Office Products Division, 1980.
4. Seybold, J. W. "Xerox's 'Star.'" *The Seybold Report.* Media, Pennsylvania: Seybold Publications, 10 (1981), 16.
5. Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. "Alto: A Personal Computer." In D. Siewiorek, C. G. Bell, and A. Newell (eds.), *Computer Structures: Principles and Examples.* New York: McGraw-Hill, 1982.
6. Ingalls, D. H. "The Smalltalk Graphics Kernel." *Byte,* 6 (1981), pp. 168–194.
7. English, W. K., D. C. Engelbart, and M. L. Berman. "Display-Selection Techniques for Text Manipulation." *IEEE Transactions on Human Factors in Electronics,* HFE-8 (1967), pp. 21–31.
8. Fitts, P. M. "The Information Capacity of the Human Motor System in Controlling Amplitude of Movement." *Journal of Experimental Psychology,* 47 (1954), pp. 381–391.
9. Card, S., W. K. English, and B. Burr. "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT." *Ergonomics,* 21 (1978), pp. 601–613.
10. Oppen, D. C., and Y. K. Dalal. "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment." Palo Alto: Xerox Office Products Division, OPD-T8103, 1981.
11. Huggins, W. H., and D. Entwisle. *Iconic Communication.* Baltimore and London: The Johns Hopkins University Press, 1974.
12. Smith, D. C. *Pygmalion, A Computer Program to Model and Stimulate Creative Thought.* Basel and Stuttgart: Birkhäuser Verlag, 1977.
13. Bolt, R. *Spatial Data-Management.* Cambridge, Massachusetts: Massachusetts Institute of Technology Architecture Machine Group, 1979.
14. Sutherland, I. "Sketchpad, A Man-Machine Graphical Communication System." *AFIPS, Proceedings of the Fall Joint Computer Conference* (Vol. 23), 1963, pp. 329–346.
15. Sutherland, W. "On-Line Graphical Specifications of Computer Procedures." Cambridge, Massachusetts: Massachusetts Institute of Technology, 1966.
16. Christensen, C. "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language." In M. Klerer and J. Reinfelds (eds.), *Interactive Systems for Experimental and Applied Mathematics.* New York: Academic Press, 1968.
17. Kay, A. C. *The Reactive Engine.* Salt Lake City: University of Utah, 1969.
18. Kay, A. C., and the Learning Research Group. "Personal Dynamic Media." Xerox Palo Alto Research Center Technical Report SSL-76-1, 1976. (A condensed version is in *IEEE Computer,* March 1977, pp. 31–41.)
19. Newman, W. M. "Officetalk-Zero: A User's Manual." Xerox Palo Alto Research Center Internal Report, 1977.
20. Dahl, O. J., and K. Nygaard. "SIMULA–An Algol-Based Simulation Language." *Communications of the ACM,* 9 (1966), pp. 671–678.
21. Lampson, B. "Bravo Manual." In *Alto User's Handbook,* Xerox Palo Alto Research Center, 1976 and 1978. (Much of the design and all of the implementation of Bravo was done by Charles Simonyi and the skilled programmers in his "software factory.")
22. Baudelaire, P., and M. Stone. "Techniques for Interactive Raster Graphics." *Proceedings of the 1980 Siggraph Conference,* 14 (1980), 3.
23. Tesler, L. "The Smalltalk Environment." *Byte,* 6 (1981), pp. 90–147.
24. Wertenbaker, L. *The World of Picasso.* New York: Time-Life Books, 1967.
25. Bush, V. "As We May Think." *Atlantic Monthly,* July 1945.
26. Engelbart, D. C. "Augmenting Human Intellect: A Conceptual Framework." Technical Report AFOSR-3223, SRI International, Menlo Park, Calif., 1962.
27. Engelbart, D. C., and W. K. English. "A Research Center for Augmenting Human Intellect." *AFIPS Proceedings of the Fall Joint Computer Conference* (Vol. 33), 1968, pp. 395–410.

Designing the Star User Interface

The Star user interface adheres rigorously to a small set of principles designed to make the system seem friendly by simplifying the human-machine interface.

Dr. David Canfield Smith, Charles Irby,
Ralph Kimball, and Bill Verplank
Xerox Corporation
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Eric Harslem
Xerox Corporation
El Segundo, CA 90245

In April 1981, Xerox announced the 8010 Star Information System, a new personal computer designed for offices. Consisting of a processor, a large display, a keyboard, and a cursor-control device (see photo 1), it is intended for business professionals who handle information.

Star is a multifunction system combining document creation, data processing, and electronic filing, mailing, and printing. Document creation includes text editing and formatting, graphics editing, mathematical formula editing, and page layout. Data processing deals with homogeneous, relational databases that can be sorted, filtered, and formatted under user control. Filing is an example of a network service utilizing the Ethernet local-area network (see references 9 and 13). Files may be stored on a work station's disk, on a file server on the work station's network, or on a file server on a different network. Mailing permits users of work stations to communicate with one another. Printing utilizes laser-driven raster printers capable of printing both text and graphics.

As Jonathan Seybold has written, "This is a very different product: Different because it truly bridges word processing and typesetting functions; different because it has a broader range of capabilities than anything which has preceded it; and different because it introduces to the commercial market radically new concepts in human engineering." (See reference 15.)

The Star user interface adheres rigorously to a small set of design principles. These principles make the system seem familiar and friendly, simplify the human-machine interaction, unify the nearly two dozen functional areas of Star, and allow user experience in one area to apply in others. In reference 17, we presented an overview of the features in Star. Here, we describe the principles behind those features and illustrate the principles with examples. This discussion is addressed to the designers of other computer programs and systems—large and small.

Star Architecture

Before describing Star's user interface, several essential aspects of the Star architecture should be pointed out. Without these elements, it would have been impossible to design an interface anything like the present one.

The Star hardware was modeled after the experimental Xerox Alto computer (see reference 19). Like Alto, Star consists of a Xerox-developed, high-bandwidth, MSI (medium-scale integration) processor; local disk storage; a bit-mapped display screen having a 72-dots-per-inch resolution; a pointing device called the "mouse"; and a connection to the Ethernet network. Stars are higher-performance machines than Altos, being about three times as fast, having 512K bytes of main memory (versus 256K bytes on most Altos), 10

About the Authors

These five Xerox employees have worked on the Star user interface project for the past five years. Their academic backgrounds are in computer science and psychology.

Photo 1: *A Star work station showing the processor, display, keyboard, and mouse.*

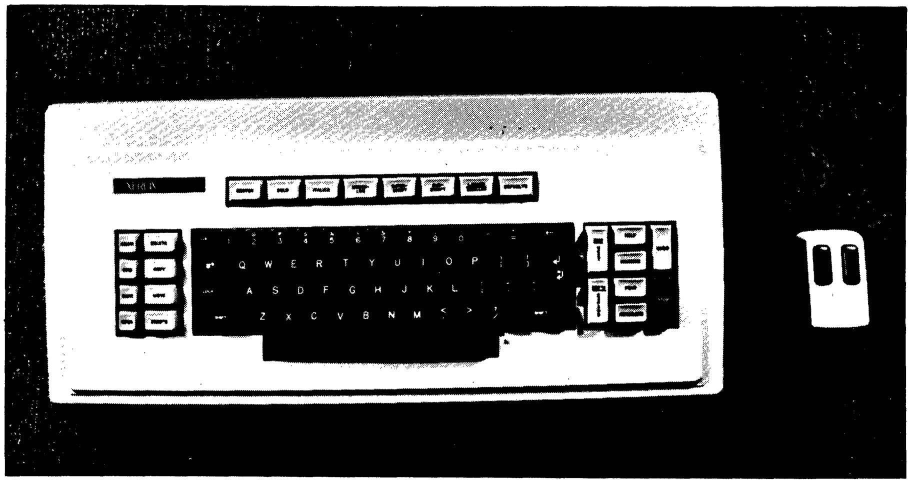


Photo 2: *The Star keyboard and mouse. Note the two buttons on top of the mouse.*

or 29 megabytes of disk memory (versus 2.5 megabytes), a 10½- by 13½-inch display screen (versus 10½ by 8 inches), and a 10-megabits-per-second Ethernet (versus 3 megabits). Typically, Stars, like Altos, are linked via Ethernets to each other and to shared file, mail, and print servers. Communication servers connect Ethernets to one another either directly or over telephone lines, enabling internetwork communication. (For a detailed description of the Xerox Alto computer, see the September 1981 BYTE article "The Xerox Alto Computer" by Thomas A. Wadlow on page 58.)

The most important ingredient of the user interface is the bit-mapped display screen. Both Star and Alto devote a portion of main memory to the screen: 100K bytes in Star, 50K bytes (usually) in Alto. Every screen dot can be individually turned on or off by setting or resetting the corresponding bit in memory. It should be obvious that this gives both computers an excellent ability to portray visual images. We believe that all impressive office systems of the future will have bit-mapped displays. Memory cost will soon be insignificant enough that they will be feasible even in home computers. Visual communication is effective, and it can't be exploited without graphics flexibility.

There must be a way to change dots on the screen quickly. Star has a high memory bandwidth, about 90 megahertz (MHz). The entire Star screen is repainted from memory 39 times per second, about a 50-MHz data rate between memory and the screen. This would swamp most computer memories. However, since Star's memory is double-ported, refreshing the display does not appreciably slow down processor memory access. Star also has separate logic devoted solely to refreshing the display. Finally, special microcode has been written to assist in changing the contents of memory quickly, permitting a variety of screen processing that would not otherwise be practical (see reference 8).

People need a way to quickly point to items on the screen. Cursor step keys are too slow; nor are they suitable for graphics. Both Star and Alto use a pointing device called the mouse (see photo 2). First developed at Stanford Research Institute (see reference 6), Xerox's version has a ball on the bottom that turns as the mouse slides over a flat surface such as a table. Electronics sense the ball rotation and guide a cursor on the screen in corresponding motions. The mouse possesses several important attributes:

- It is a "Fitts's law" device. That is, after some practice you can point with a mouse as quickly and easily as you can with the tip of your finger. The limitations on pointing speed are those inherent in the human nervous system (see references 3 and 7).
- It stays where it was left when you are not touching it. It doesn't have to be picked up like a light pen or stylus.
- It has buttons on top that can be sensed under program control. The buttons let you point to and interact with objects on the screen in a variety of ways.

Every Star and Alto has its own hard disk for local storage of programs and data. This enhances their personal nature, providing consistent access to information regardless of how many other machines are on the network or what anyone else is doing. Larger programs can be written, using the disk for swapping.

The Ethernet lets both Stars and Altos have a distributed architecture. Each machine is connected to an Ethernet. Other machines on the Ethernet are dedicated as "servers"—machines that are attached to a resource and provide access to that resource.

Star Design Methodology

We have learned from Star the importance of formulating the fundamental concepts (the user's conceptual model) *before* software is written, rather than tacking on a user interface *afterward*. Xerox devoted about thirty work-years to the design of the Star user interface. It was designed *before* the functionality of the system was fully decided. It was even designed *before* the computer hardware was built. We worked for two years *before* we wrote a single line of actual product software. Jonathan Seybold put it this way, 'Most system design efforts start with hardware specifications, follow this with a set of functional specifications for the software, then try to figure out a logical user interface and command structure. The Star project started the other way around: the paramount concern was to define a conceptual model of how the user would relate to the system. Hardware and software followed from this." (See reference 15.)

In fact, before we even began designing the model, we developed a methodology by which we would do the design. Our methodology report (see reference 10) stated:

> One of the most troublesome and least understood aspects of interactive systems is the *user interface*. In the design of user interfaces, we are concerned with several issues: the provision of languages by which users can express their commands to the computer; the design of display representations that show the state of the system to the user; and other more abstract issues that affect the user's understanding of the system's behavior. Many of these issues are highly subjective and are therefore often addressed in an *ad hoc* fashion. We believe, however, that more rigorous approaches to user interface design can be developed. . . .
>
> These design methodologies are all unsatisfactory for the same basic reason: they all omit an essential step that must precede the design of any successful user interface, namely *task analysis*. By this we mean the analysis of the task performed by the user, or users, prior to introducing the proposed computer system. Task analysis involves establishing who the users are, what their goals are in performing the task, what information they use in performing it, what information they generate, and what methods they employ. The descriptions of input and output information should include an analysis of the various *objects*, or individual types of information entity, employed by the user. . . .
>
> The purpose of task analysis is to simplify the remaining stages in user interface design. The *current task description*, with its breakdown of the information objects and methods presently employed, offers a starting point for the definition of a corresponding set of objects and methods to be provided by the computer system. The idea behind this phase of design is to build up a new *task environment* for the user, in which he can work to accomplish the same goals as before, surrounded now by a different set of objects, and employing new methods.

Prototyping is another crucial element of the design process. System designers should be prepared to implement the new or difficult concepts and then to *throw away* that code when doing the actual implementation. As Frederick Brooks says, the question "is not *whether* to build a pilot system and throw it away. You *will* do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers. . . . Hence *plan to throw one away; you will, anyhow.*" (See reference 2.) The Alto served as a valuable prototype for Star. Over a thousand Altos were eventually built. Alto users have had several thousand work-years of experience with them over a period of eight years, making Alto perhaps the largest prototyping effort ever. Dozens of experimental programs were written for the Alto by members of the Xerox Palo Alto Research Center. Without the creative ideas of the authors of those systems, Star in its present form would have been impossible. In addition, we ourselves programmed various aspects of the Star design on Alto, but all of it was "throwaway" code. Alto, with its bit-mapped display screen, was powerful enough to implement and test our ideas on visual interaction.

Some types of concepts are inherently difficult for people to grasp. Without being too formal about it, our experience before and during the Star design led us to the following classification:

Easy	*Hard*
concrete	abstract
visible	invisible
copying	creating
choosing	filling in
recognizing	generating
editing	programming
interactive	batch

The characteristics on the left were incorporated into the Star user's conceptual model. The characteristics on the right we attempted to avoid.

Principles Used

The following main goals were pursued in designing the Star user interface:

- familiar user's conceptual model
- seeing and pointing versus remembering and typing
- what you see is what you get
- universal commands
- consistency
- simplicity
- modeless interaction
- user tailorability

We will discuss each of these in turn.

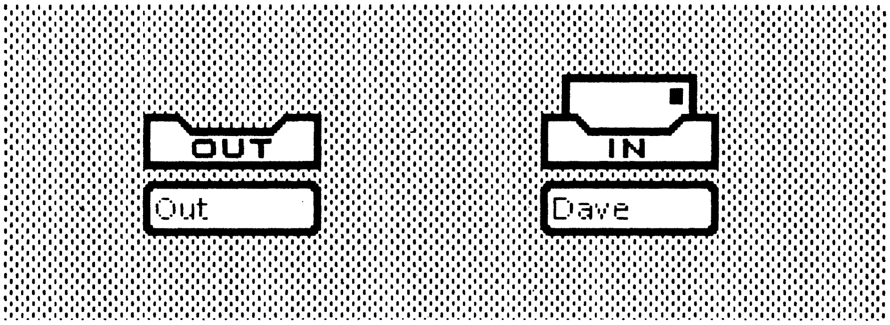


Figure 1: *In-basket and out-basket icons. The in-basket contains an envelope indicating that mail has been received. (This figure was taken directly from the Star screen. Therefore, the text appears at screen resolution.)*

Familiar User's Conceptual Model

A *user's conceptual model* is the set of concepts a person gradually acquires to explain the behavior of a system, whether it be a computer system, a physical system, or a hypothetical system. It is the model developed in the mind of the user that enables that person to understand and interact with the system. The first task for a system designer is to decide what model is preferable for users of the system. This extremely important step is often neglected or done poorly. The Star designers devoted several work-years at the outset of the project discussing and evolving what we considered an appropriate model for an office information system: the metaphor of a physical office.

The designer of a computer system can choose to pursue familiar analogies and metaphors or to introduce entirely new functions requiring new approaches. Each option has advantages and disadvantages. We decided to create electronic counterparts to the physical objects in an office: paper, folders, file cabinets, mail boxes, and so on—an electronic metaphor for the office. We hoped this would make the electronic "world" seem more familiar, less alien, and require less training. (Our initial experiences with users have confirmed this.) We further decided to make the electronic analogues be *concrete objects*. Documents would be more than file names on a disk; they would also be represented by pictures on the display screen. They would be selected by pointing to them with the mouse and clicking one of the buttons. Once selected, they would be moved, copied, or deleted by pushing the appropriate key. Moving a document became the electronic equivalent of picking up a piece of paper and walking somewhere with it. To file a document, you would move it to a picture of a file drawer, just as you take a physical piece of paper to a physical file cabinet.

The reason that the user's conceptual model should be decided *first* when designing a system is that the approach adopted *changes the functionality of the system*. An example is electronic mail. Most electronic-mail systems draw a distinction between *messages* and *files* to be sent to other people. Typically, one program sends messages and a different program handles file transfers, each with its own interface. But we observed that offices make no such distinction. Everything arrives through the mail, from one-page memos to books and reports, from intraoffice mail to international mail. Therefore, this became part of Star's physical-office metaphor. Star users mail documents of any size, from one page to many pages. Messages are short documents, just as in the real world. User actions are the same whether the recipients are in the next office or in another country.

A physical metaphor can simplify and clarify a system. In addition to eliminating the artificial distinctions of traditional computers, it can eliminate commands by taking advantage of more general concepts. For example, since moving a document on the screen is the equivalent of picking up a piece of paper and walking somewhere with it, there is no "send mail" command. You simply move it to a picture of an outbasket. Nor is there a "receive mail" command. New mail appears in the in-basket as it is received. When new mail is waiting, an envelope appears in the picture of the in-basket (see figure 1). This is a simple, familiar, nontechnical approach to computer mail. And it's easy once the physical-office metaphor is adopted!

While we want an analogy with the physical world for familiarity, we don't want to limit ourselves to its capabilities. One of the raisons d'être for Star is that physical objects do not provide people with enough power to manage the increasing complexity of the "information age." For example, we can take advantage of the computer's ability to search rapidly by providing a search function for its electronic file drawers, thus helping to solve the long-standing problem of lost files.

The "Desktop"

Every user's initial view of Star is the "Desktop," which resembles the top of an office desk, together with surrounding furniture and equipment. It represents your working environment—where your current projects and accessible resources reside. On the screen are displayed pictures of familiar office objects, such as documents, folders, file drawers, in-baskets, and out-baskets. These objects are displayed as small pictures or "icons," as shown in figure 2.

You can "open" an icon to deal with what it represents. This enables you to read documents, inspect the contents of folders and file drawers, see what mail you have received, etc. When opened, an icon expands into a larger form called a "window," which displays the icon's contents. Windows are the principal mechanism for displaying and manipulating information.

The Desktop "surface" is displayed as a distinctive gray pattern. This restful design makes the icons and windows on it stand out crisply, minimizing eyestrain. The surface is organized as an array of one-inch squares, 14 wide by 11 high. An icon can be placed in any square, giving a maximum of 154 icons. Star centers an icon in its square, making it easy to line up icons neatly. The Desktop always occupies the entire display screen; even when windows appear on the screen, the Desktop continues to exist "beneath" them.

The Desktop is the principal Star technique for realizing the physical-office metaphor. The icons on it are visible, concrete embodiments of the corresponding physical objects. Star users are encouraged to think of the objects on the Desktop in physical terms. Therefore, you can move the icons around to arrange your Desktop as you wish. (Messy Desktops are certainly possible, just as in real life.) Two icons cannot occupy the same space (a basic law of physics). Although moving a document to a Desktop resource such as a printer involves transferring the document icon to the same square as the printer icon, the printer immediately "absorbs" the document, queuing it for printing. You can leave

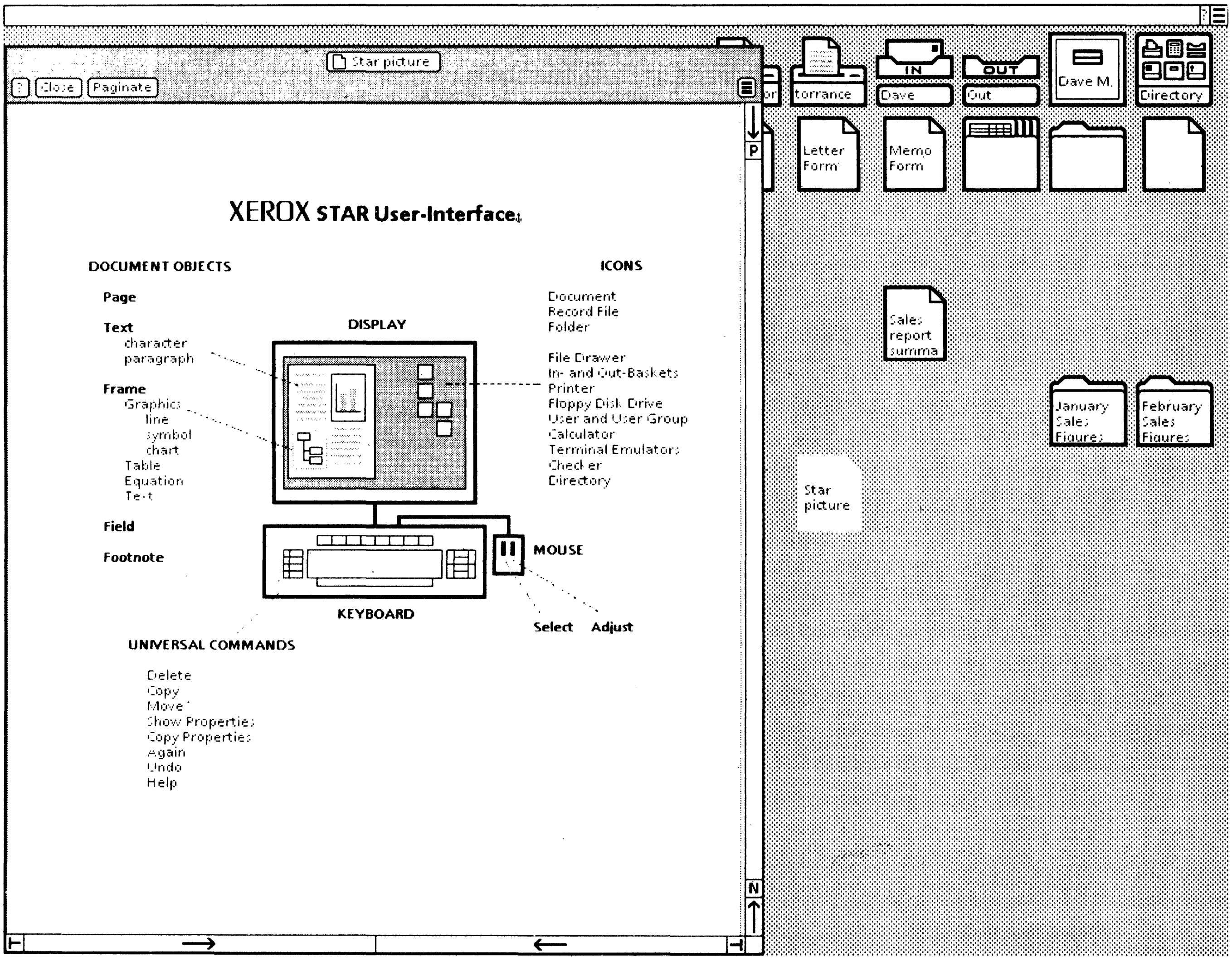


Figure 2: *A Desktop as it appears on the Star screen. Several commonly used icons appear across the top of the screen, including documents to serve as "form-pad" sources for letters, memos, and blank paper. An open window displaying a document containing an illustration is also shown.*

documents on your Desktop indefinitely, just as on a real desk, or you can file them away in folders or file drawers. Our intention and hope is that users will *intuit* things to do with icons, and that those things will indeed be part of the system. This will happen if:

(a) Star models the real world accurately enough. Its *similarity* with the office environment preserves your familiar way of working and your existing concepts and knowledge.
(b) Sufficient *uniformity* is in the system. Star's principles and "generic" commands (discussed below) are applied throughout the system, allowing lessons learned in one area to apply to others.

The model of a physical office provides a simple base from which learning can proceed in an incremental fashion. You are not exposed to entirely new concepts all at once. Much of your existing knowledge is embedded in the base.

In a functionally rich system, it is probably not possible to represent everything in terms of a single model. There may need to be more than one model. For example, Star's records-processing facility cannot use the physical-office model because physical offices have no "records processing" worthy of the name. Therefore, we invented a different model, a record file as a collection of *fields*. A record can be displayed as a row in a *table* or as filled-in fields in a *form*. Querying is accomplished by filling in a blank example of a record with predicates describing the desired values, which is philosophically similar to Zloof's "Query-by-Example" (see reference 21).

Of course, the number of different user models in a system must be kept to a minimum. And they should not overlap; a new model should be introduced only when an existing one does not cover the situation.

Seeing and Pointing

A well-designed system makes everything relevant to a task visible on the screen. It doesn't hide things under CODE+key combinations or force you to remember conventions. That burdens your memory. During conscious thought, the brain utilizes several levels of memory, the most important being the "short-term memory." Many studies have analyzed the short-term memory and its role in thinking. Two conclusions stand out: (1) conscious thought deals with concepts in the short-term memory (see reference 1) and (2) the capacity of the short-term memory is limited (see reference 14). When everything being dealt with in a computer system is visible, the display screen relieves the load on the short-term memory by acting as a sort of "visual cache." Thinking becomes easier and more productive. A well-designed computer system can actually improve the *quality* of your thinking (see reference 16). In addition, visual communication is often more efficient than linear communication; a picture is worth a thousand words.

A subtle thing happens when everything is visible: *the display becomes reality*. The user model becomes identical with what is on the screen. Objects can be understood purely in terms of their visible characteristics. Actions can be understood in terms of their effects on the screen. This lets users *conduct experiments* to test, verify, and expand their understanding—the essence of experimental science.

In Star, we have tried to make the objects and actions in the system *visible*. Everything to be dealt with and all commands and effects have a visible representation on the display screen or on the keyboard. You never have to remember that, for example, CODE+Q does something in one context and something different in another context. In fact, our desire to eliminate this possibility led us to abolish the CODE key. (We have yet to see a computer system with a CODE key that doesn't violate the principle of visibility.) You never invoke a command or push a key and have nothing visible happen. At the very least, a message is posted explaining that the command doesn't work in this context, or it is not implemented, or there is an error. It is disastrous to the user's model when you invoke an action and the system does nothing in response. We have seen people push a key several times in one system or another trying to get a response. They are not sure whether the system has "heard" them or not. Sometimes the system is simply throwing away their keystrokes. Sometimes it is just slow and is *queuing* the keystrokes; you can imagine the unpredictable behavior that is possible.

We have already mentioned icons and windows as mechanisms for making the concepts in Star visible. Other such mechanisms are Star's *property and option sheets*. Most objects in Star have properties. A property sheet is a two-dimensional, form-like environment that displays those properties. Figure 3 shows the character property sheet. It appears on the screen whenever you make a text selection and push the PROPERTIES key. It contains such properties as type font and size; bold, italic, underline, and strikeout face; and superscript/subscript positioning. Instead of having to remember the properties of characters, the current settings of those properties, and, worst of all, how to change those properties, property sheets simply *show everything on the screen. All* the options are presented. To change one, you point to it with the mouse and push a button. Properties in effect are displayed in reverse video.

This mechanism is used for *all* properties of *all* objects in the system. Star contains a couple of hundred properties. To keep you from being overwhelmed with information, property sheets display only the properties relevant to the type of object currently selected (e.g., character, paragraph, page, graphic line, formula element, frame, document, or folder). This is an example of "progressive disclosure": hiding complexity until it is needed. It is also one of the clearest examples of how an emphasis on visibility can reduce the amount of remembering and typing required.

Property sheets may be thought of as an *alternate representation* for ob-

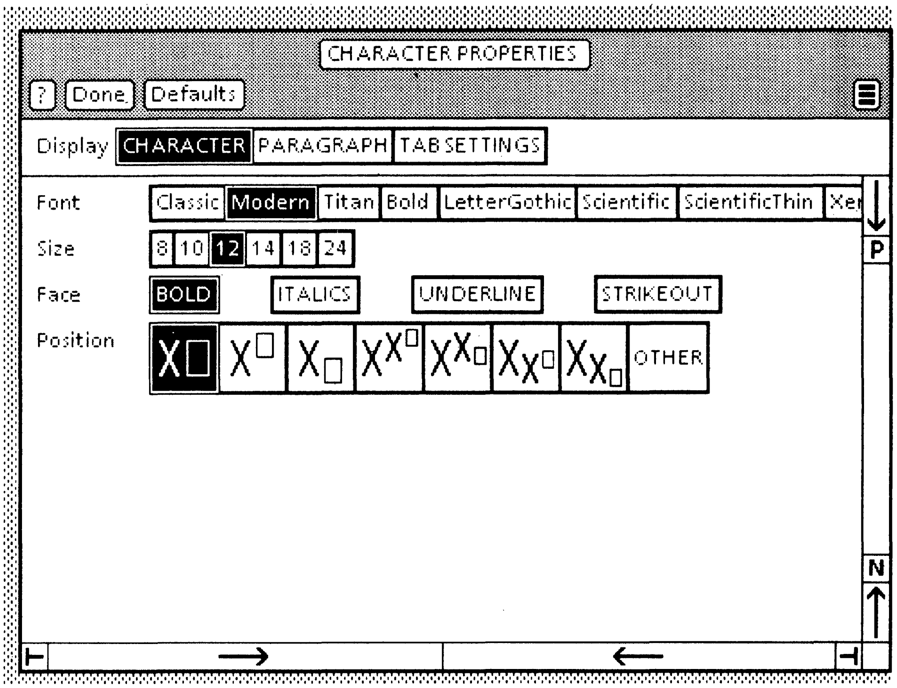

CHARACTER PROPERTIES
? Done Defaults
Display CHARACTER PARAGRAPH TAB SETTINGS
Font Classic Modern Titan Bold LetterGothic Scientific ScientificThin Xe
Size 8 10 12 14 18 24
Face BOLD ITALICS UNDERLINE STRIKEOUT
Position OTHER
P
N

Figure 3: *The property sheet for text characters.*

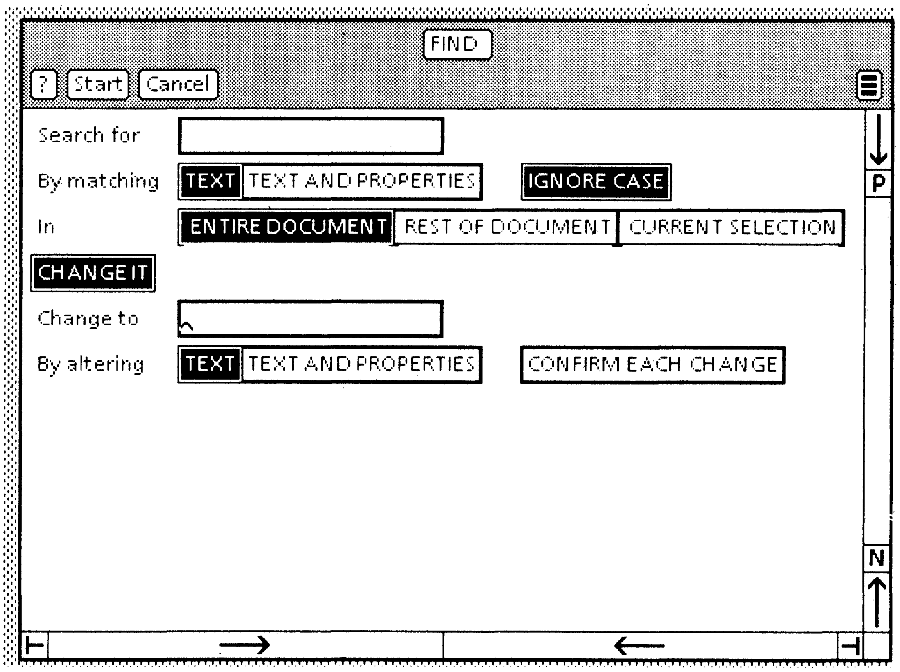

FIND
? Start Cancel
Search for
By matching TEXT TEXT AND PROPERTIES IGNORE CASE
In ENTIRE DOCUMENT REST OF DOCUMENT CURRENT SELECTION
CHANGE IT
Change to
By altering TEXT TEXT AND PROPERTIES CONFIRM EACH CHANGE
P
N

Figure 4: *The option sheet for the Find command showing both the Search and Substitute options. The last two lines of options appear only when CHANGE IT is turned on.*

jects. The screen shows you the visible characteristics of objects, such as the type font of text characters or the names of icons. Property sheets show you the underlying structure of objects as they make this structure visible and accessible.

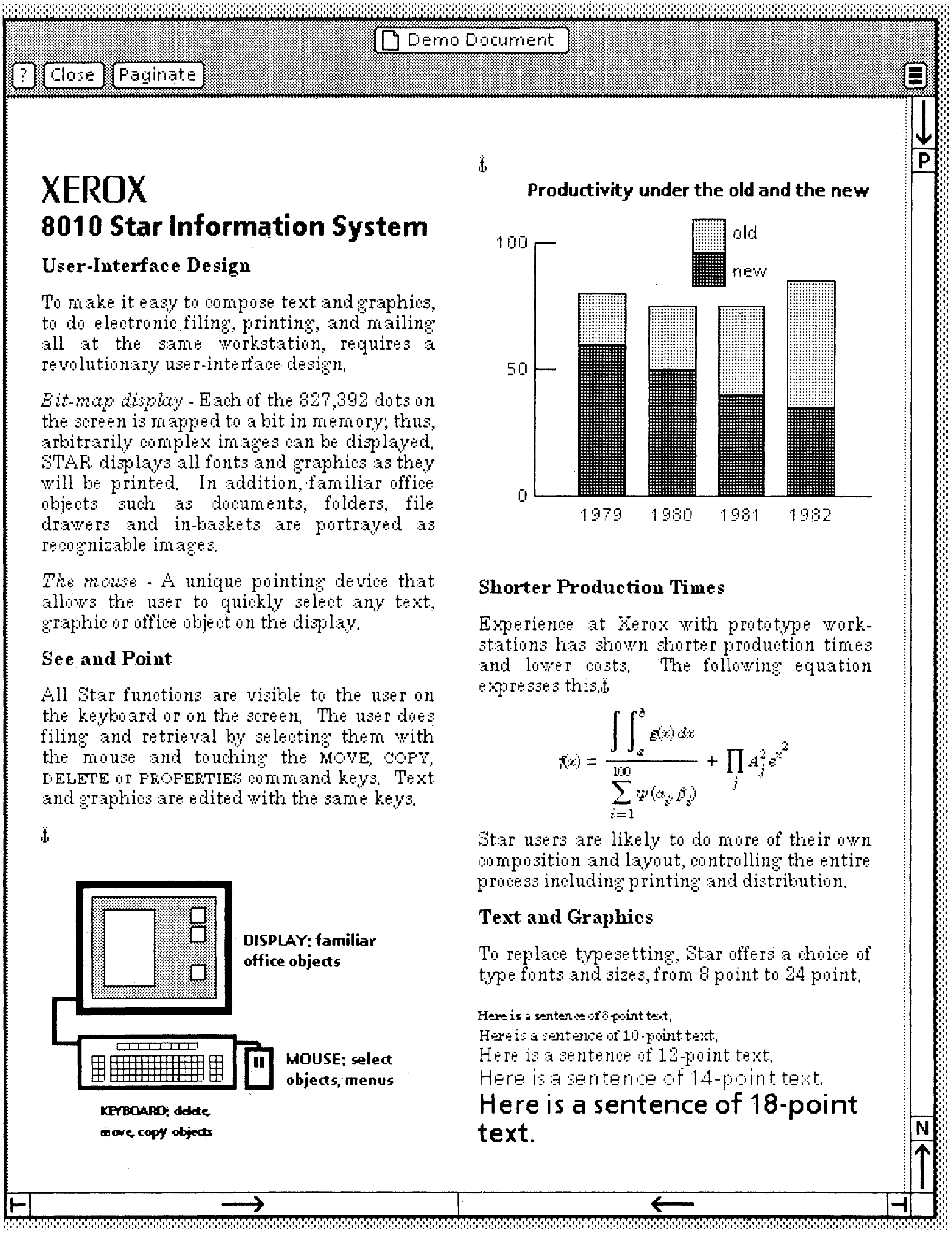
Invisibility also plagues the commands in some systems. Commands often have several arguments and options that you must remember with no assistance from the system. Star addresses this problem with *option sheets* (see figure 4), a two-dimensional, form-like environment that displays the arguments to commands. It serves the same function for command arguments that property sheets do for object properties.

What You See Is What You Get

"What you see is what you get" (or WYSIWYG) refers to the situation in which the display screen portrays an accurate rendition of the printed page. In systems having such capabilities as multiple fonts and variable line spacing, WYSIWYG requires a bit-mapped display because only that has sufficient graphic power to render those characteristics accurately.

WYSIWYG is a simplifying technique for document-creation systems. All composition is done *on the screen*. It eliminates the iterations that plague users of document compilers. You can examine the appearance of a page *on the screen* and make changes until it looks right. The printed page will look the same (see figure 5). Anyone who has used a document compiler or post-processor knows how valuable WYSIWYG is. The first powerful WYSIWYG editor was Bravo, an experimental editor developed for Alto at the Xerox Palo Alto Research Center (see reference 12). The text-editor aspects of Star were derived from Bravo.

Trade-offs are involved in WYSIWYG editors, chiefly having to do with the lower resolution of display screens. It is never possible to get an *exact* representation of a printed page on the screen since most screens have only 50 to 100 dots per inch (72 in Star), while most printers have higher resolution. Completely accurate character positioning is not possible. Nor is it usually possible to represent shape differences for fonts smaller than eight points in size since there are too few dots per character to be recognizable. Even 10-point ("normal" size) fonts may be uncomfortably small on the screen, necessitating a magnified mode for viewing text.



XEROX

8010 Star Information System

User-Interface Design

To make it easy to compose text and graphics, to do electronic filing, printing, and mailing all at the same workstation, requires a revolutionary user-interface design.

Bit-map display - Each of the 827,392 dots on the screen is mapped to a bit in memory; thus, arbitrarily complex images can be displayed. STAR displays all fonts and graphics as they will be printed. In addition, familiar office objects such as documents, folders, file drawers and in-baskets are portrayed as recognizable images.

The mouse - A unique pointing device that allows the user to quickly select any text, graphic or office object on the display.

See and Point

All Star functions are visible to the user on the keyboard or on the screen. The user does filing and retrieval by selecting them with the mouse and touching the MOVE, COPY, DELETE or PROPERTIES command keys. Text and graphics are edited with the same keys.

DISPLAY: familiar office objects
MOUSE: select objects, menus
KEYBOARD: delete, move, copy objects

Shorter Production Times

Experience at Xerox with prototype workstations has shown shorter production times and lower costs. The following equation expresses this.

$$f(x) = \frac{\int\int_a^b g(x)\,dx}{\sum_{i=1}^{100} \psi(\alpha_i, \beta_i)} + \prod_j A_j^2 e^{x^2}$$

Star users are likely to do more of their own composition and layout, controlling the entire process including printing and distribution.

Text and Graphics

To replace typesetting, Star offers a choice of type fonts and sizes, from 8 point to 24 point.

Here is a sentence of 8-point text.

Here is a sentence of 10-point text.

Here is a sentence of 12-point text.

Here is a sentence of 14-point text.

Here is a sentence of 18-point text.

Figure 5: *A Star document showing multicolumn text, graphics, and formulas. This is the way the document appears on the screen. It is also the way it will print (at higher resolution, of course).*

WYSIWYG requires very careful design of the screen fonts in order to keep text on the screen readable and attractive. Nevertheless, the increase in productivity made possible by WYSIWYG editors more than outweighs these difficulties.

Universal Commands

Star has a few commands that can be used throughout the system: MOVE, COPY, DELETE, SHOW PROPERTIES, COPY PROPERTIES, AGAIN, UNDO, and HELP. Each performs the same way regardless of the type of object selected. Thus, we call them "universal" or "generic" commands. For example, you follow the same set of actions to move text in a document and to move a line in an illustration or a document in a folder: select the object, push the MOVE key, and indicate a destination. (HELP and UNDO don't use a selection.) Each generic command has a key devoted to it on the keyboard.

These commands are far more basic than the commands in other computer systems. They strip away the extraneous application-specific semantics to get at the underlying principles. Star's generic commands derive from fundamental computer-science concepts because they also underlie operations in programming languages. For example, much program manipulation of data structures involves moving or copying values from one data structure to another. Since Star's generic commands embody fundamental underlying concepts, they are widely applicable. Each command fills a variety of needs, meaning fewer commands are required. This simplicity is desirable in itself, but it has another subtle advantage: it makes it easy for users to form a model of the system. People can use what they understand. Just as progress in science derives from simple, clear theories, progress in the usability of computers is coming to depend on simple, clear user interfaces.

MOVE is the most powerful command in the system. It is used during text editing to rearrange letters in a word, words in a sentence, sentences in a paragraph, and paragraphs in a document. It is used during graphics editing to move picture elements, such as lines and rectangles, around in an illustration. It is used during formula editing to move mathematical structures, such as summations and integrals, around in an equation. It replaces the conventional "store file" and "retrieve file" commands; you simply move an icon into or out of a file drawer or folder. It eliminates the "send mail" and "receive mail" commands; you move an icon to an out-basket or from an in-basket. It replaces the "print" command; you move an icon to a printer. And so on. MOVE strips away much of the historical clutter of computer commands. It is more fundamental than the myriad of commands it replaces. It is simultaneously more powerful and simpler.

Much simplification comes from Star's object-oriented interface. The action of setting properties also replaces a myriad of commands. For example, changing paragraph margins is a command in many systems. In Star, you do it by selecting a paragraph object and setting its MARGINS property. (For more information on object-oriented languages, see the August 1981 BYTE.)

Consistency

Consistency asserts that mechanisms should be used in the same way wherever they occur. For example, if the left mouse button is used to select a character, the same button should be used to select a graphic line or an icon. Everyone agrees that consistency is an admirable goal. However, it is perhaps the single hardest characteristic of all to achieve in a computer system. In fact, in systems of even moderate complexity, consistency may not be well defined.

A question that has defied consensus in Star is what should happen to a document after it has been printed. Recall that a user prints a document by selecting its icon, invoking MOVE, and designating a printer icon. The printer absorbs the document, queuing it for printing. What happens to that document icon after printing is completed? The two plausible alternatives are:

1. The system deletes the icon.
2. The system does not delete the icon, which leads to several further alternatives:
   - 2a. The system puts the icon back where it came from (i.e., where it was before MOVE was invoked).
   - 2b. The system puts the icon at an arbitrary spot on the Desktop.
   - 2c. The system leaves the icon in the printer. You must move it out of the printer explicitly.

The consistency argument for the first alternative goes as follows: when you move an icon to an out-basket, the system mails it and then deletes it from your Desktop. When you move an icon to a file drawer, the system files it and then deletes it from your Desktop. Therefore, when you move an icon to a printer, the system should print it and then delete it from your Desktop. Function icons should behave consistently with one another.

The consistency argument for the second alternative is: the user's conceptual model at the Desktop level is the physical-office metaphor. Icons are supposed to behave similarly to their physical counterparts. It makes sense that icons are deleted after they are mailed because after you put a piece of paper in a physical out-basket and the mailperson picks it up, it *is* gone. However, the physical analogue for printers is the office copier, and there is no notion of deleting a piece of paper when you make a copy of it. Function icons should behave consistently with their physical counterparts.

There is no one right answer here. Both arguments emphasize a dimension of consistency. In this case, the dimensions happen to overlap. We eventually chose alternative 2a for the following reasons:

1. *Model dominance*—The physical metaphor is the stronger model at the Desktop level. Analogy with physical counterparts *does* form the basis for people's understanding of what icons are and how they behave. Argument 1 advocates an *implicit* model that must be learned; argument 2 advocates an *explicit* model that people already have when they are introduced to the system. Since people do use their existing knowledge when confronted with new situations, the design of the system should be based on that knowledge. This is especially important if people are to be able to *intuit* new uses for the features they have learned.

2. *Pragmatics*—It is dangerous to delete things when users don't expect it. The first time a person labors over a document, gets it just right, prints it, and finds that it has disappeared, that person is going to become *very nervous*, not to mention angry. We also decided to put it back where it came from (2a instead of 2b or 2c) for the pragmatic reason that this involves slightly less work on the user's part.

3. *Seriousness*—When you file or nail an icon, it is not deleted entirely from the system. It still exists in the file drawer or in the recipients' in-baskets. If you want it back, you can move it back out of the file drawer or send a message to one of the recipients asking to have a copy sent back. Deleting after printing, however, is final; if you move a document to a printer and the printer deletes it, that document is gone for good.

One way to get consistency into a system is to adhere to *paradigms* for operations. By applying a successful way of working in one area to other areas, a system acquires a unity that is both apparent and real. Paradigms that Star uses are:

- *Editing*—Much of what you do in Star can be thought of as editing. In addition to the conventional text, graphics, and formula editing, you manage your files by *editing filing windows*. You arrange your working environment by *editing your Desktop*. You alter properties by *editing property sheets*. Even programming can be thought of as *editing data structures* (see reference 16).
- *Information retrieval*—A lot of power can be gained by applying information-retrieval techniques to information wherever it exists in a system. Star broadens the definition of "database." In addition to the traditional notion as represented by its record files, Star views file drawers as databases of documents, in-baskets as databases of mail, etc. This teaches users to think of information retrieval as a general tool applicable throughout the system.
- *Copying*—Star elevates the concept of "copying" to a high level: that of a paradigm for creating. In all the various domains of Star, you *create by copying*. Creating something out of nothing is a difficult task. Everyone has observed that it is easier to modify an existing document or program than to write it originally. Picasso once said, "The most awful thing for a painter is the white canvas . . . To copy others is necessary." (See reference 20.) Star makes a serious attempt to alleviate the problem of the "white canvas" by making copying a practical aid to creation. For example, you create new icons by copying existing ones.

Graphics are created by copying existing graphic images and modifying them. In a sense, you can even type characters in Star's 2^{16}-character set by "copying" them from keyboard windows (see figure 6).

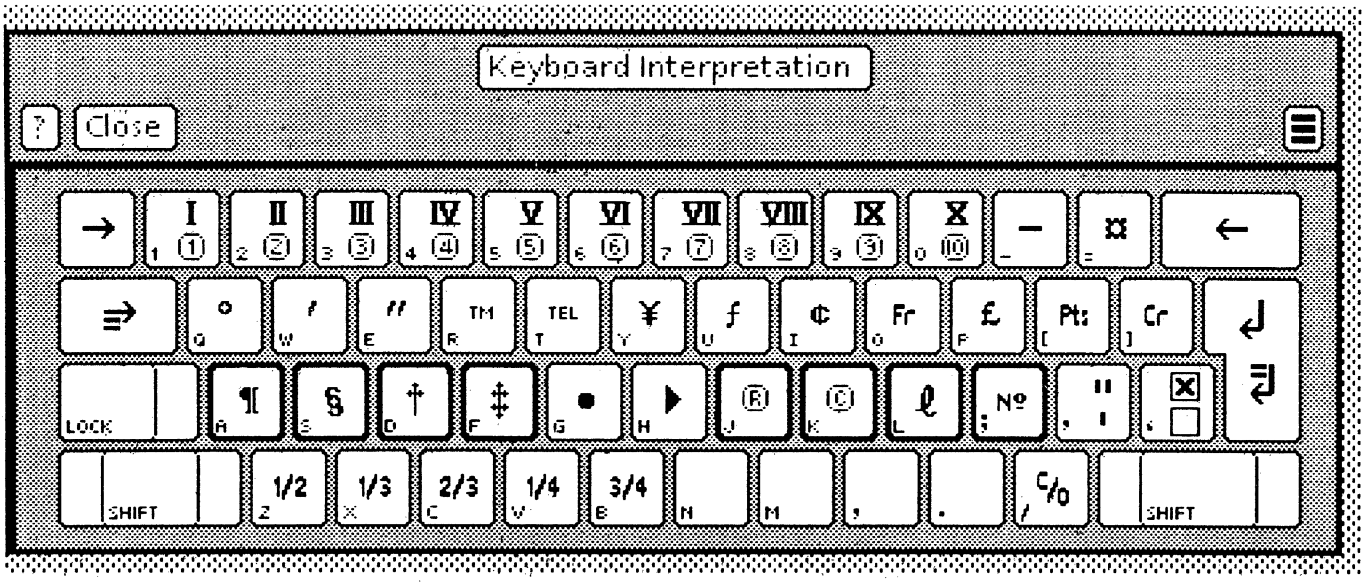

Keyboard Interpretation
?
Close
→ I 1 ① II 2 ② III 3 ③ IV 4 ④ V 5 ⑤ VI 6 ⑥ VII 7 ⑦ VIII 8 ⑧ IX 9 ⑨ X 0 ⑩ – _ ¤ = ←
⇛ ° Q ′ W ″ E TM R TEL T ¥ Y ƒ U ¢ I Fr O £ P Pt: [Cr] ↲
LOCK ¶ A § S † D ‡ F ● G ▶ H ® J © K ℓ L № ; ″ ′ , ☒ □ . ↲
SHIFT 1/2 Z 1/3 X 2/3 C 1/4 V 3/4 B N M , . c/o / SHIFT

Figure 6: *The keyboard-interpretation window serves as the source of characters that may be entered from the keyboard. The character set shown here contains a variety of office symbols.*

These paradigms *change the very way you think*. They lead to new habits and models of behavior that are more powerful and productive. They can lead to a *human-machine synergism*.

Star obtains additional consistency by using the class and subclass notions of Simula (see reference 4) and Smalltalk (see reference 11). The clearest example of this is classifying icons at a higher level into *data icons* and *function icons*. Data icons represent objects on which actions are performed. Currently, the three types (i.e., subclasses) of data icons are documents, folders, and record files. Function icons represent objects that perform actions. Function icons are of many types, with more being added as the system evolves: file drawers, in- and out-baskets, printers, floppy-disk drives, calculators, terminal emulators, etc.

In general, anything that can be done to one data icon can be done to all, regardless of its type, size, or location. All data icons can be moved, copied, deleted, filed, mailed, printed, opened, closed, and a variety of other operations applied. Most function icons will accept any data icon; for example, you can move any data icon to an out-basket. This use of the class concept in the user-interface design reduces the artificial distinctions that occur in some systems.

Simplicity

Simplicity is another principle with which no one can disagree. Obviously, a simple system is better than a complicated one if they have the same capabilities. Unfortunately, the world is never as simple as that. Typically, a trade-off exists between easy novice use and efficient expert use. The two goals are not always compatible. In Star, we have tried to follow Alan Kay's maxim: "simple things should be simple; complex things should be possible." To do this, it was sometimes necessary to make common things simple at the expense of uncommon things being harder. Simplicity, like consistency, is not a clear-cut principle.

One way to make a system appear simple is to make it uniform and consistent, as we discussed earlier. Adhering to those principles leads to a *simple user's model*. Simple models are easier to understand and work with than intricate ones.

Another way to achieve simplicity is to minimize the redundancy in a system. Having two or more ways to do something increases the complexity without increasing the capabilities. The ideal system would have a minimum of powerful commands that obtained all the desired functionality and that did not overlap. That was the motivation for Star's "generic" commands. But again the world is not so simple. General mechanisms are often inconvenient for high-frequency actions. For example, the SHOW PROPERTIES command is Star's general mechanism for changing properties, but it is too much of an interruption during typing. Therefore, we added keys to optimize the changing of certain character properties: BOLD, ITALICS, UNDERLINE, SUPERSCRIPT, SUBSCRIPT, LARGER/SMALLER (font), CENTER (paragraph). These significantly speed up typing, but they don't add any new functionality. In this case, we felt the trade-off was worth it because typing is a frequent activity. "Minimum redundancy" is a good but not absolute guideline.

In general, it is better to introduce new *general* mechanisms by which "experts" can obtain accelerators rather than add a lot of special one-purpose-only features. Star's mechanisms are discussed below under "User Tailorability."

Another way to have the system as a whole appear simple is to make each of its parts simple. In particular, the system should avoid overloading the semantics of the parts. Each part should be kept conceptually clean. Sometimes, this may involve a major redesign of the user interface. An example from Star is the mouse, which has been used on the Alto for eight years. Before that, it was used on the NLS system at Stanford Research Institute (see reference 5). All of those mice have three buttons on top. Star has only two. Why did we depart from "tradition"? We observed that the dozens of Alto programs all had different semantics for the mouse buttons. Some used them one way, some another. There was no consistency between systems. Sometimes, there was not even consistency *within* a system. For example, Bravo uses the mouse buttons for selecting text, scrolling windows, and creating and deleting windows, depending on where the cursor is when you push a mouse button. Each of the three buttons has its own meaning in each of the different regions. It is difficult to remember which button does what where.

Thus, we decided to simplify the mouse for Star. Since it is apparently quite a temptation to overload the semantics of the buttons, we eliminated temptation by eliminating buttons. Well then, why didn't we use a one-button mouse? Here the plot thickens. We did consider and prototype a one-button mouse interface. One button is sufficient (with a little cleverness) to provide all the functionality needed in a mouse. But when we tested the interface on naive users, as we did with a variety of features, we found that they had a lot of trouble making selections with it. In fact, we prototyped and tested six different semantics for the mouse buttons: one one-button, four two-button, and a three-button design. We were chagrined to find that while some were better than others, *none of them* was completely easy to use, even though, a priori, it seemed like all of them would work! We then took the most successful features of two of the two-button designs and prototyped and tested them as a seventh design. To our relief, it not only tested better than any of the other six, everyone found it simple and trouble-free to use.

This story has a couple of morals:

- The intuition of designers is error-prone, no matter how good or bad they are.

- The critical parts of a system should be tested on representative users, preferably of the "lowest common denominator" type.
- What is simplest along any one dimension (e.g., number of buttons) is not necessarily conceptually simplest for users; in particular, minimizing the number of keystrokes may not make a system easier to use.

Modeless Interaction

Larry Tesler defines a *mode* as follows:

> A mode of an interactive computer system is a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation on operator input. (See reference 18.)

Many computer systems use modes because there are too few keys on the keyboard to represent all the available commands. Therefore, the interpretation of the keys depends on the mode or state the system is in. Modes can and do cause trouble by making habitual actions cause unexpected results. If you do not notice what mode the system is in, you may find yourself invoking a sequence of commands quite different from what you had intended.

Our favorite story about modes, probably apocryphal, involves Bravo. In Bravo, the main typing keys are normally interpreted as commands. The "i" key invokes the Insert command, which puts the system in "insert mode." In insert mode, Bravo interprets keystrokes as letters. The story goes that a person intended to type the word "edit" into his document, but he forgot to enter insert mode first. Bravo interpreted "edit" as the following commands:

E(verything)	select everything in the document
D(elete)	delete it
I(nsert)	enter insert mode
t	type a "t"

The entire contents of the document were replaced by the letter "t." This makes the point, perhaps too strongly, that modes should be introduced into a user interface with caution, if at all.

Commands in Star take the form of noun-verb. You specify the object of interest (the noun) and then invoke a command to manipulate it (the verb). Specifying an object is called "making a selection." Star provides powerful selection mechanisms that reduce the number and complexity of commands in the system. Typically, you will exercise more dexterity and judgment in making a selection than in invoking a command. The object (noun) is almost always specified before the action (verb) to be performed. This helps make the command interface modeless; you can change your mind as to which object to affect simply by making a new selection before invoking the command. No "accept" function is needed to terminate or confirm commands since invoking the command is the last step. Inserting text does not even require a command; you simply make a selection and begin typing. The text is placed after the end of the selection.

The noun-verb command form does not by itself imply that a command interface is modeless. Bravo also uses the noun-verb form; yet, it is a highly modal editor (although the latest version of Bravo has drastically reduced its modalness). The difference is that Bravo tries to make one mechanism (the main typing keys) serve more than one function (entering letters and invoking commands). This inevitably leads to confusion. Star avoids the problem by having special keys on the keyboard devoted solely to invoking functions. The main typing keys only enter characters. (This is another example of the simplicity principle: avoid overloading mechanisms with meanings.)

Modes are not necessarily bad. Some modes can be helpful by simplifying the specification of extended commands. For example, Star uses a "field fill-in order specification mode." In this mode, you can specify the order in which the NEXT key will step through the fields in the document. Invoking the SET FILL-IN ORDER command puts the system in the mode. Each field you now select is added to the fill-in order. You terminate the mode by pushing the STOP key. Star also utilizes temporary modes as part of the MOVE, COPY, and COPY PROPERTIES commands. For example, to move an object, you select it, push the MOVE key that puts the system in "move mode," and then select the destination. These modes work for two reasons. First, *they are visible*. Star posts a message in the Message Area at the top of the screen indicating that a mode is in effect. The message remains there for the duration of the mode. Star also changes the shape of the cursor as an additional indication. You can always tell the state of the system by inspection (see figure 7). Second, *the allowable actions are constrained during modes*. The only action that is allowed—except for actions directly related to the mode—is scrolling to another part of the document. This constraint makes it even more apparent that the system is in an unusual state.

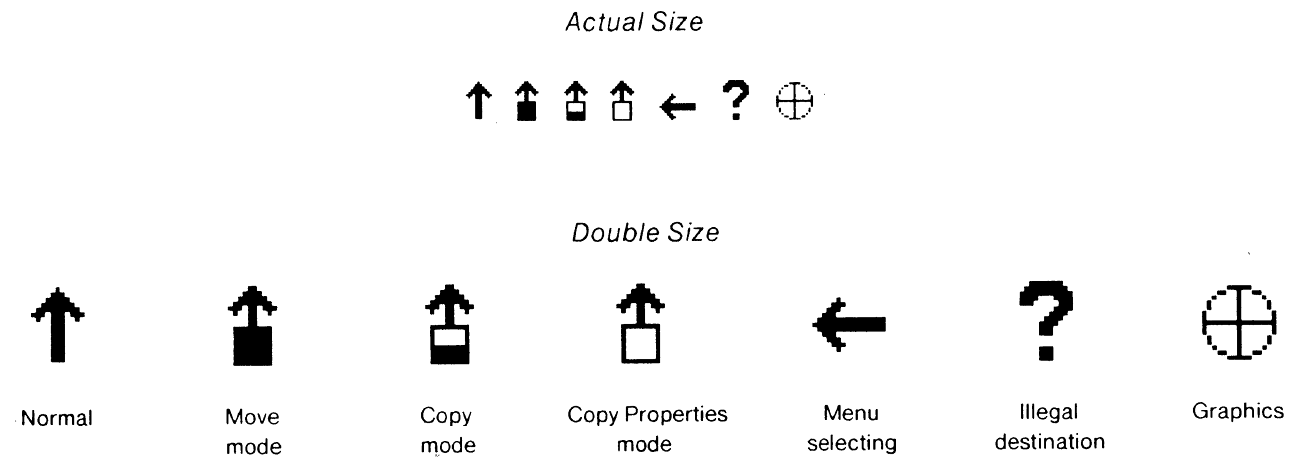


Figure 7: *Some of the cursor shapes used by the Star to indicate the state of the system. The cursor is a 16- by 16-bit map that can be changed under program control.*

User Tailorability

No matter how general or powerful a system is, it will never satisfy all its potential users. People always want ways to speed up often-performed operations. Yet, everyone is different. The only solution is to design the system with provisions for user extensibility built in. The following mechanisms are provided by Star:

- You can tailor the appearance of your system in a variety of ways. The simplest is to choose the icons you want on your Desktop, thus tailoring your working environment. At a more sophisticated level, a work station can be purchased with or without certain functions. For example, not everyone may want the equation facility. Xerox calls this "product factoring."
- You can set up blank documents with text, paragraph, and page layout defaults. For example, you might set up one document with the normal text font being 10-point Classic and another with it being 12-point Modern italic. The documents need not be blank; they may contain fixed text and graphics, and fields for variable fill-in. A typical form might be a business-letter form with address, addressee, salutation, and body fields, each field with its own default text style. Or it might be an accounting form with lines and tables. Or it might be a mail form with To, From, and Subject fields, and a heading tailored to each individual. Whatever the form or document, you can put it on your Desktop and make new instances of it by selecting it and invoking COPY. Thus, each form can act like a "pad of paper" from which new sheets can be "torn off."

Interesting documents to set up are "transfer sheets," documents containing a variety of graphics symbols tailored to different applications. For example, you might have a transfer sheet containing buildings in different sizes and shapes, or one devoted to furniture, animals, geometric shapes, flowchart symbols, circuit components, logos, or a hundred other possibilities. Each sheet would make it easier to create a certain type of illustration. Graphics experts could even construct the symbols on the sheets, so that users could create high-quality illustrations without needing as much skill.

- You can tailor your filing system by changing the sort order in file drawers and folders. You can also control the filing hierarchy by putting folders inside folders inside folders, to any desired level.
- You can tailor your record files by defining any number of "views" on them. Each view consists of a filter, a sort order, and a formatting document. A filter is a set of predicates that produces a subset of the record file. A formatting document is any document that contains fields whose names correspond to those in the record file. Records are always displayed through some formatting document; they have no inherent external representation. Thus, you can set up your own individual subset(s) and appearance(s) for a record file, even if the record file is shared by several users.
- You can define "meta operations" by writing programs in the *CUS*tomer Programming language CUSP. For example, you can further tailor your forms by assigning computation rules expressed in CUSP to fields. Eventually, you will be able to define your own commands by placing CUSP "buttons" into documents.
- You can define abbreviations for commonly used terms by means of the abbreviation definition/expansion facility. For example, you might define "sdd" as an abbreviation for "Xerox Systems Development Department." The expansion can be an entire paragraph, or even multiple paragraphs. This is handy if you

create documents out of predefined "boilerplate" paragraphs, as the legal profession does. The expansion can even be an illustration or mathematical formula.

- Every user has a unique name used for identification to the system, usually the user's full name. However, you can define one or more *aliases* by which you are willing to be known, such as your last name only, a shortened form of your name, or a nickname. This lets you personalize your identification to the rest of the network.

Summary

In the 1980s, the most important factors affecting how prevalent computer usage becomes will be reduced cost, increased functionality, improved availability and servicing, and, perhaps most important of all, progress in user-interface design. The first three alone are necessary, but not sufficient for widespread use. Reduced cost will allow people to *buy* computers, but improved user interfaces will allow people to *use* computers. In this article, we have presented some principles and techniques that we hope will lead to better user interfaces.

User-interface design is still an art, not a science. Many times during the Star design we were amazed at the depth and subtlety of user-interface issues, even such supposedly straightforward issues as consistency and simplicity. Often there is no one "right" answer. Much of the time there is no scientific evidence to support one alternative over another, just intuition. Almost always there are trade-offs. Perhaps by the end of the decade, user-interface design will be a more rigorous process. We hope that we have contributed to that progress. ■

References

1. Arnheim, Rudolf. *Visual Thinking.* Berkeley: University of California Press, 1971.
2. Brooks, Frederick. *The Mythical Man-Month.* Reading, MA: Addison-Wesley, 1975.
3. Card, Stuart, William English, and Betty Burr. "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT." *Ergonomics,* vol. 21, no. 8, 1978, pp. 601-613.
4. Dahl, Ole-Johan and Kristen Nygaard. "SIMULA—An Algol-Based Simulation Language." *Communications of the ACM,* vol. 9, no. 9, 1966, pp. 671-678.
5. Engelbart, Douglas and William English. "A Research Center for Augmenting Human Intellect." *Proceedings of the AFIPS 1968 Fall Joint Computer Conference,* vol. 33, 1968, pp. 395-410.
6. English, William, Douglas Engelhart, and M. L. Berman. "Display-Selection Techniques for Text Manipulation." *IEEE Transactions on Human Factors in Electronics,* vol. HFE-8, no. 1, 1967, pp. 21-31.
7. Fitts, P. M. "The Information Capacity of the Human Motor System in Controlling Amplitude of Movement." *Journal of Experimental Psychology,* vol. 47, 1954, pp. 381-391.
8. Ingalls, Daniel. "The Smalltalk Graphics Kernel." BYTE, August 1981, pp. 168-194.
9. Intel, Digital Equipment, and Xerox Corporations. *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications.* Version 1.0, 1980.
10. Irby, Charles, Linda Bergsteinsson, Thomas Moran, William Newman, and Larry Tesler. *A Methodology for User Interface Design.* Systems Development Division, Xerox Corporation, January 1977.
11. Kay, Alan and the Learning Research Group. *Personal Dynamic Media.* Xerox Palo Alto Research Center Technical Report SSL-76-1, 1976. (A condensed version is in *IEEE Computer,* March 1977, pp. 31-41.)
12. Lampson, Butler. "Bravo Manual." *Alto User's Handbook,* Xerox Palo Alto Research Center, 1976 and 1978. (Much of the design of all the implementation of Bravo was done by Charles Simonyi and the skilled programmers in his "software factory.")
13. Metcalfe, Robert and David Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM,* vol. 19, no. 7, 1976, pp. 395-404.
14. Miller, George. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information." In *The Psychology of Communication,* by G. Miller, New York: Basic Books, 1967. (An earlier version appeared in *Psychology Review,* vol. 63, no. 2, 1956, pp. 81-97.
15. Seybold, Jonathan. "Xerox's 'Star'." In *The Seybold Report,* Media, PA: Seybold Publications, vol. 10, no. 16, 1981.
16. Smith, David Canfield. *Pygmalion, A Computer Program to Model and Stimulate Creative Thought.* Basel, Switzerland: Birkhauser Verlag, 1977.
17. Smith, David Canfield, Charles Irby, Ralph Kimball, and Eric Harslem. "The Star User Interface: An Overview." Submitted to the AFIPS 1982 National Computer Conference.
18. Tesler, Larry. Private communication; but also see his excellent discussion of modes in "The Smalltalk Environment." BYTE, August 1981, pp. 90-147.
19. Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. "Alto: A Personal Computer." In *Computer Structures: Principles and Examples,* edited by D. Siewiorek, C. G. Bell, and A. Newell, New York: McGraw-Hill, 1982.
20. Wertenbaker, Lael. *The World of Picasso.* New York: Time-Life Books, 1967.
21. Zloof, M. M. "Query-by-Example." *Proceedings of the AFIPS 1975 National Computer Conference,* vol. 44, 1975, pp. 431-438.

Star Graphics: An Object-Oriented Implementation

Dr. Daniel E. Lipkie
Xerox Corporation, El Segundo, California

Steven R. Evans, John K. Newlin, Robert L. Weissman
Xerox Corporation, Palo Alto, California

Abstract : The XEROX Star 8010 Information System features an integrated text and graphics editor. The Star hardware consists of a processor, a large bit-mapped display, a keyboard and a pointing device. Star's basic graphic elements are points, lines, rectangles, triangles, graphics frames, text frames and bar charts. The internal representation is in terms of idealized objects that are displayed or printed at resolutions determined by the output device. This paper describes the design and implementation of a graphics editor using an object-oriented technique based on a Star-wide subclassing method called the Trait Mechanism.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques - User interfaces; H.4.1 [**Information Systems Applications**]: Office Automation - Word processing; I.3.6 [**Computer Graphics**]: Methodology and Techniques - Interaction techniques; I.7.2 [**Text Processing**]: Document Preparation

General Terms: Design

Key Words: business graphics, subclassing

I. The Star Workstation

In 1975 Xerox started an effort to transfer research from the *Xerox Palo Alto Research Center* (PARC) into mainline office products. Central to this strategy was the development of a top-of-the-line professional workstation, subsequently named Star, that was to provide a major step forward in several different domains of office automation. A retrospective on the development of Star is presented in [2].

A unique aspect of Star is its user interface (UI) and the role it played in the development of Star [5, 6, 7]. About 30 work years of effort were expended in designing the UI *before* the functionality of the system was fully decided and before the computer hardware was even built.

The hardware that supports this UI (figure 1) consists of a microprogrammable 22-bit virtual, 18-bit real address space processor, an 808 by 1024 pixel (11" x 14") bit-mapped display, a keyboard, a pointing device called a *mouse* and a 10 or 29M byte disk. The workstation may be attached to a 10M bits-per-second Ethernet for access to remote printing, filing, communication and electronic mail services.

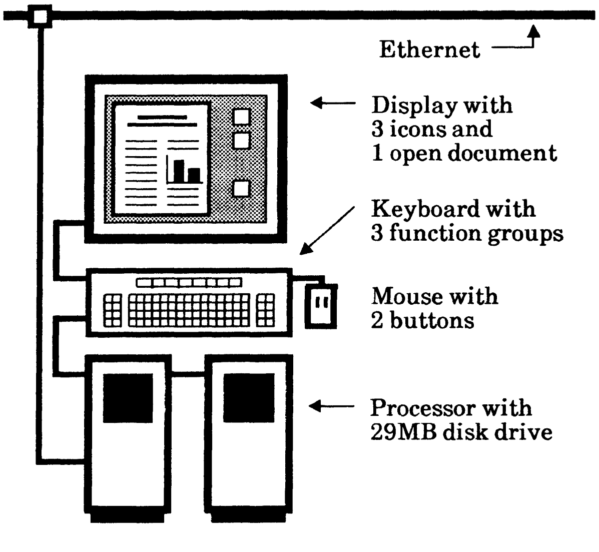

Ethernet
Display with 3 icons and 1 open document
Keyboard with 3 function groups
Mouse with 2 buttons
Processor with 29MB disk drive

Figure 1
Star Workstation Schematic

The mouse has a ball on the bottom that turns as the mouse slides over a flat surface. Electronics sense the ball rotation and displaces a cursor on the screen in

XEROX®, 8010 and Star are trademarks of XEROX CORP.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-076-1/82/007/0115

corresponding motions. There are two buttons on the mouse, called SELECT and ADJUST, used to make and adjust a selection as described below.

The keyboard has a conventional central part and three groups of function keys.

The left function group contains the *generic commands*: MOVE, COPY, DELETE, SHOW PROPERTIES, COPY PROPERTIES, and AGAIN. Their meaning is defined only in a generic sense; it is up to the currently selected element to further define them as explained below.

The keys in the upper function group are referred to as *soft keys*. Their meanings and use are discussed below.

The right function group includes the command KEYBOARD and others that are not of interest in this paper. When KEYBOARD is pressed, the soft keys allow the user to assign a new interpretation to the central keyboard and to display a window that shows the meaning of each keytop. Keyboards supported are Japanese, various European keyboards, Dvorak and keyboards with useful office and mathematical symbols.

Central concepts to the Star UI are *what you see is what you get*, *visibility* (don't hide things under CODE+key combinations) and a *physical office metaphor.*

One of the functional areas of the office addressed by Star is document creation, which encompasses text editing and formatting, figure editing (graphics), mathematical formula editing and page layout. These are all integrated. As an example of *what you see is what you get*, the Star user edits on the display both text and graphic figures, which appear exactly as they do when the document is printed. This document was prepared using Star; no special step was needed to merge the figures and text. Visibility and the office metaphor are discussed in the next section.

The design of the Star software began in the spring of 1978 and the first release, containing 255,000 lines of code, was completed in Oct. 1981. Over the 3.5 years approximately 93 work years of effort were expended and in excess of 400,000 lines of code were written. This effort was aided by the adoption of an object-oriented style of coding right from the start and by the use later of a *multiple-inheritance subclassing mechanism*, Traits [1], as the basis for defining and implementing objects. An object-oriented implementation was chosen because it corresponded closely to the UI model of interacting screen elements. In this paper we use the term *element* to refer to user perceived entities and reserve the term *object* for the corresponding internal implementations.

As explained below, there is no graphics editor per se, but of the 255,000 lines of code in the release about 28,000 are associated with editing figures in documents.

In Section II we describe the Star user interface. The Trait mechanism is presented in Section III. Its application in the Star implementation is discussed in Sections IV and V.

II. The Star User Interface

The Star UI differs from that of other computer systems through its heavy use of the graphics capabilities of a bitmap display, its adherence to a physical office metaphor, and its rigorous application of a small set of design principles [3]. The graphics capabilites reduce the amount of typing and remembering required to operate the system; the office metaphor makes the system familiar and friendly; the design principles unify the nearly two dozen functional areas of Star.

One important principle is to make objects and actions in the system visible. The system should not hide things under obscure CODE + key combinations or force the user to remember a lot of conventions. When a choice had to be made between easy novice use and efficient expert use, Alan Kay's maxim was followed: "Simple things should be simple; complex things should be possible".

As you make everything visible, the display becomes reality, and the user model becomes identical with what is on the screen.

Using the physical office metaphor Star creates electronic counterparts to the physical elements in an office: paper, folders, file cabinets, mail boxes and so on. The Star screen represents a *desktop* on which are placed small (~1" x 1") pictograms or *icons* that represent these elements, e.g. the document (paper) and file drawer (file cabinet) icons in figure 2.

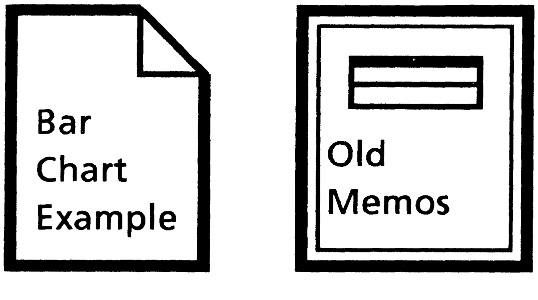


Figure 2
Document and File Drawer Icons

Within a illustration the currently implemented graphics elements are points, lines, rectangles, triangles, graphics frames, text frames and bar charts. Examples are shown in figure 3. A graphics frame is a

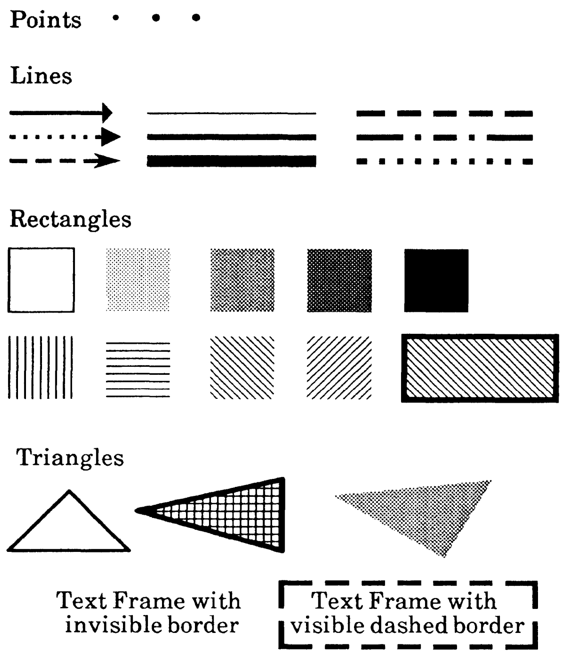

Points
Lines
Rectangles
Triangles
Text Frame with invisible border
Text Frame with visible dashed border

Figure 3
Examples of Graphic Elements

rectangular area reserved for figures. Text frames allow the user to put labels in figures.

Iconic, text and graphic elements are selected by pointing at them with the mouse and clicking (pressing and releasing) one of the buttons on the mouse.

Icons show that they are selected by *highlighting* (video reversing) their image. Character selections highlight themselves by inverting a rectangular region around the characters.

The user selects a graphics element by pointing anywhere along one of its lines or edges. When a graphic element is selected, it inverts a small square region around each of its *control points.* Lines have control points at each end, rectangles (figure 4) and frames at each corner and midpoint of each side and triangles at each vertex. The inverted region around the control point closest to the cursor is slightly larger. This control point is called the *guiding point* and becomes attached to the cursor when the element is moved, copied or stretched.

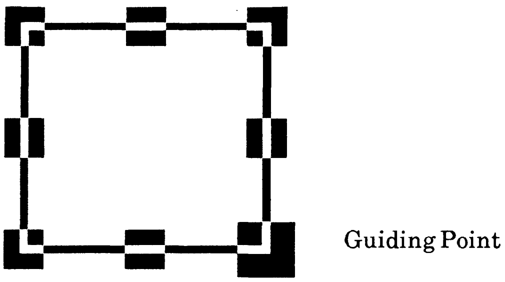


Figure 4
Rectangle with Inverted Control Points
(Expanded Scale)

An element highlights as if selected when the mouse button is depressed, but it is selected only when the button is released. The user may change the candidate selection by moving the mouse with the button still depressed until the desired element is highlighted.

After an icon, character or graphics element has been selected, it may be manipulated by one of the generic operations. To move a document to a file drawer, select the document icon, press MOVE, point at the file drawer icon and click the SELECT button. Elements may also be manipulated in other ways described below.

The meaning of the operation is determined by the selected element. Moving a document icon to a file drawer icon sends the document over the Ethernet and stores it on a file server; moving it to an out-basket icon sends the document via electronic mail; moving it to a printer icon makes a hardcopy of it.

Copying and deleting have similar straightforward semantics.

The OPEN command in the left function group may only be applied to an icon and creates a window through which the icon's contents are displayed and edited. Star has a *modeless* editing style; there are no *start edit* or *end edit* commands. The user merely selects a character in a window displaying a document and begins typing and the text is appended to the selected character. The page content is reformatted as the user edits. The generic operations also may be applied inside a window; e.g. text may be moved, copied or deleted by merely selecting, pressing the function key and pointing to the destination.

Before discussing the other editing actions, we will explain how graphics elements are entered into text.

To enter a figure into a document the user selects a character in the document and types a character that represents a graphics frame. (The character is found on a keyboard accessible through the KEYBOARD key.) This non-printing, but screen-displayable, character is inserted after the selected character. It looks like a boat anchor and represents an anchoring point for the graphics frame. The frame appears between two lines of text at the same time the anchor character is typed. As the textual content of the document is reformatted during subsequent editing this anchor character is shifted as any other character and in addition its associated graphics frame is also repositioned, e.g. the anchor character acts like a footnote reference mark, and the graphics frame moves from page to page as its reference mark is moved.

Once the user has a graphics frame in a document, other graphics elements may be moved or copied into the frame. Star graphics has only two *creation* actions, inserting a graphics frame as described above and MAKE LINE described below. All other graphics elements are made by copying. Every desktop has a *directory* icon that contains a blank document and a graphics document that has all the graphics elements. The directory's documents can only be copied, not moved or deleted, so the user always has a source of documents and graphics elements.

Pressing the SHOW PROPERTIES key opens a small window in which a *property sheet* appropriate to the current selection is displayed. The property sheet displays the property values for the currently selected element. The properties are changed by setting them to the desired values and clicking at the Done command which applies the new properties and closes the property sheet window. For each property the property sheet either displays an enumeratation of all possible values or provides a box into which the value is typed. The property sheet for a graphics line is shown in figure 5.

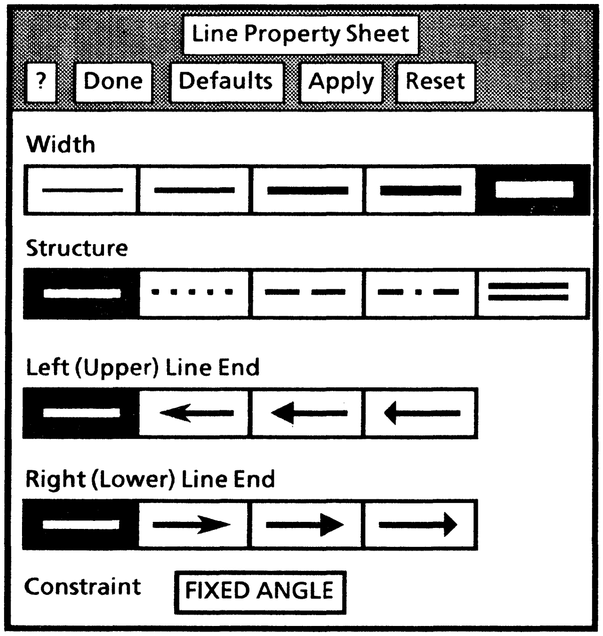


Figure 5
Line Property Sheet

The ? command provides access to online documentation about the line property sheet; Defaults sets the properties to system defined values; Apply applies the properties but does not close the property sheet; Reset sets the properties to the values they had when Show Properties was invoked.

There are three kinds of properties: *choice*, *state* and *text*.

A choice-type property displays a set of mutually exclusive values for the property which are shown immediately adjacent to each other; e.g. a line's width, structure and line endings are each choice properties. Exactly one is on at any one time and is video reversed. To change it the user points at the desired value and clicks a mouse button.

A state-type property may be either on or off. Pointing at and clicking a mouse button toggles its setting. A line may be constrained to be at a fixed angle so that its length but not its direction may be changed during the stretching action described below. An unconstrained line may have its length and/or direction changed.

A text-type property displays a box into which a text value is typed. None of the properties on the line property sheet are text-type. But an example is the text-type property on the document property sheet which determines the name of the document.

The properties of a rectangle are the width and style of its bounding box, its interior shading and a fixed shape constraint.

The properties of text and graphics frame include the width and style of the border. Text frames may be constrained to be *fixed* or *flexible*. A flexible text frame will change shape as its text contents are edited. Graphics frames may be positioned horizontally within a column (flush left, centered, flush right) and vertically within a column (top, centered, bottom or floating). Graphics frames also have a grid that may be displayed as dots or plus marks at each grid point or as ticks around the edge of the frame. Grid spacing is also variable.

Another way to change a selected element's properties is to press COPY PROPERTIES and then point at an element that is the source of the desired properties.

Associated with every text selection is a multi-click level: character, word, sentence or paragraph. Clicking at an unhighlighted character with the SELECT mouse button selects the character at the character level; clicking again with the SELECT mouse button at the same character selects the enclosing word; clicking at any character in a selected word selects the enclosing sentence; clicking at any character in a selected sentence selects the enclosing paragraph; clicking at a character in a selected paragraph brings the selection back to the character level and selects that character. Clicking at a character with the ADJUST mouse button expands or shrinks the selection at the current level to minimally span the pre-existing selection and the character pointed at.

There is no selection level associated with a graphics selection, but the ADJUST button has a graphics interpretation that is used to extend the selection to include multiple elements. Clicking the ADJUST button at a graphics element toggles it in/out of the current selection. The ADJUST button may also be used to extend the selection by adding all elements properly contained in a bounding box. The user presses the ADJUST button, which fixes one corner of the bounding box, and moves the mouse with the button depressed. The current mouse position defines the opposite corner of the bounding box. As long as the ADJUST button is depressed, a box is drawn on the screen from the fixed point to the current mouse postion and all elements properly contained are highlighted. When the button is released (at *button up*) these elements are added to the selection. An extended selection may be moved, copied, deleted, joined, stretched or the elements may have their properties changed.

The elements of an extended selection may be of different types, e.g. lines, rectangles and text frames.

Whenever there is a graphics selection the soft keys at the top of the keyboard take on graphics meanings: STRETCH, MAGNIFY, GRID, MAKE LINE, JOIN/SPLIT and TOP/BOTTOM. When the current selection is textual the soft keys take on meanings that allow the appearance of the charaters to be changed, e.g. bold, italic, underlined, superscript, subscript, larger font size and smaller font size.

When STRETCH is pressed the selection is de-highlighted and the control point furthest from the guiding point is replaced by an X and is considered pinned. The guiding point becomes attached to the mouse when a button is pressed. As the mouse is moved the selection is horizontally and vertically scaled to conform to the pinned and guiding points and redisplayed. On button up the element retains the new size, the X is removed, and the selection is rehighlighted. MAGNIFY is simlar to STRETCH except that the same scaling factor is applied in the horizontal and vertical directions, i.e. aspect is maintained.

The GRID soft key toggles the grid on/off for the frame containing the selection. If the grid is active, it controls the placement of the guiding point during move, copy, stretch and magnify.

MAKE LINE creates a line between two successive mouse click positions.

JOIN combines an extended selection of graphics elements into a *cluster* element. Once joined, all of the original elements behave as a single element for purposes of selection and editing. This allows users to define their own graphics symbols. The SPLIT function acts on a cluster and reverses the effect of JOIN.

Graphics and text frames are *opaque*, that is they obscure elements that are *under* them. In figure 6a the text frame is above the rectangle, while in figure 6b it is below. The soft keys TOP and BOTTOM allow the user to move the current selection to the top or bottom level in a frame.

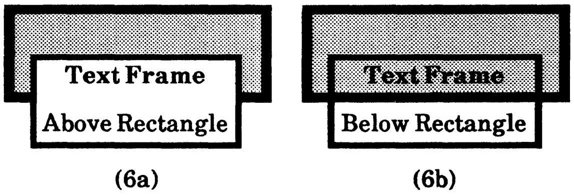


Figure 6
Overlapping Elements

In keeping with Star's style of modeless editing, the graphics editor is not invoked in the traditional sense. In fact, as we shall see later, there is no graphics editor in the traditional sense. All graphics editing capabilities are available whenever there is a document open. The Star user may pause during document editing and read incoming mail or use the records processing feature or any of the other Star functions. The responsibility for making the transition between these *editing environments* resides with the elements on the desktop, not the user. This is a major difference between Star and other information systems, including the Alto system [9] where the user explicitly invokes BRAVO for text editing, SIL or DRAW for figure editing, and LAUREL for electronic mail.

The full text editing capabilities are available for editing the contents of text frames within graphics frames, e.g. text frames may contain anchor characters and graphics frames. This means that Star must support the *virtual nested invocation* of editors.

III. Traits - The Star Subclassing Mechanism

Object-orientation is a method for organizing software such that, at any time, computation is performed under the aegis of a particular object, not a centralized program that handles every case from one place. The nature of the Star UI and the user model it fosters led to the adoption of an object-oriented method from the beginning of the software development.

Subclassing is a refinement of the basic object-oriented methodology that constructs objects out of more primitive behaviors. Initial Star subclassing efforts were in the SIMULA-67 and Smalltalk [8] style where the specialization relations form a tree. We found it necessary to generalize this concept to allow specialization relations that are represented by directed acyclic graphs. A full description of the Trait mechanism and the generalized concept of multiple-inheritance subclassing is beyond the scope of this discussion but may be found in [1].

Subclassing as a way of implementing objects was not used during initial development of Star. This was partly because the designers had had little experience with subclassing as a methodology for large production software systems where performance is a primary consideration. It was also believed, incorrectly, that an extensible design based on subclassing would necessitate a violation of the typing mechansim of the implementing language, Mesa [4]. But as implementation progressed, it became clear that significant code-sharing was possible since we were dealing with objects that were more similar than different and we re-examined the subclassng problem.

We first present some of the central concepts of the trait mechanism and then describe how it has been applied in graphics. The initial graphics implementation was about 17,000 lines of code and space does not permit a full presentation of the graphics traits and their interaction. During this description we will refer to trait definitions summarized in Section VI.

A *trait* is a characterization of a behavior and is the primitive abstraction used to define objects. A trait used to define an object is said to be *carried* by the object, e.g. the trait TreeElement is carried by objects that live in tree-like data structures. To implement a behavior, an object carrying a trait remembers information in a *state* defined by the trait. For example objects carrying TreeElement have 3 state variables, next, parent and eldest, that are pointers to the corresponding objects or are the special value *objectNil*, pointer to no object.

Trait definition:
trait name
state
component traits
fixed operations
replaceable operations

In a departure from SIMULA-67, traits may carry other *component traits* where the carry relationship is represented as an acyclic directed graph. This permits behaviors to be built on multiple lower-level abstractions. The basic imaging trait, Schema, carries TreeElement because all imaging objects are part of an *imaging tree* rooted at a Desktop Object that manages the Star display (see figure 9 below).

A trait defines *operations* as a means of presenting information to or extracting information from an object, e.g. the operations GetParent and SetParent for TreeElement. Operations also may be invoked for effect, e.g. the Schema operation Paint is a request to an object to paint its image on the display.

An operation is invoked on an object by specifying a trait carried by the object, an operation defined by the trait and the object. In Mesa, an operation invocation is implemented as a procedure call with the object handle as the first parameter and other parameters as needed, e.g.

Schema.Paint[object, ...]

Operations that extract information are implemented as procedures that return values.

A trait operation has a *specification* (name, parameters, return type) and a *realization* (an implementing piece of code).

Fixed operations are those for which the trait supplies the realization, e.g. the implementing code for GetParent in TreeElement is the same for all objects carrying the trait, it merely accesses the state variable parent and returns its value.

Replaceable operations are analogous to SIMULA-67 VIRTUAL procedures. The trait defines the specification and each class supplies its own realization that is used by all objects in the class, e.g. the Schema trait provides the specification for Paint but the classes Line and Rectangle each provide their own realizations that access the object's state to display the appropriate image.

A *class trait* is a trait that provides fixed operations for creating and destroying objects in the class. Associated with each class is a *replaceable operations vector* that is the composition of its own and its component trait's replaceable operations. The realizations of replaceable operations are assigned to the vector elements. The vector for the Line class is shown in figure 7.

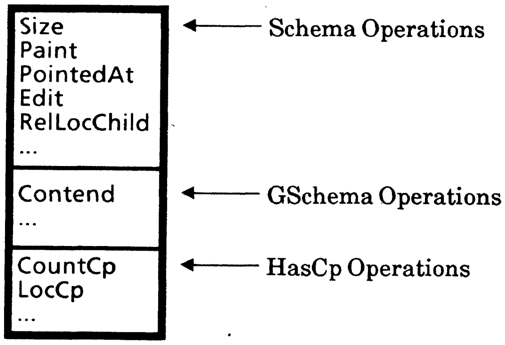


Figure 7
Replaceable Operations Vector for Line Class

Each object created by a class trait has an *object state vector* that is the composition of the class's state and the class's component trait's states. The vector for a Line object is shown in figure 8.

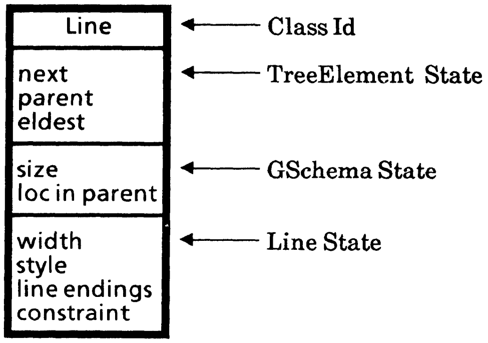


Figure 8
Object State Vector for a Line Object

IV. Applying the Trait Mechanism to Star

The first release of Star defined 169 traits, 129 of which were class traits. 99 traits required state variables and 39 had replaceable operations.

Non-class traits we will discuss are: TreeElement, Schema, GSchema, ListSchema and HasCp. Class traits we will discuss are AnchoredFrame, Line and TextBlock. Traits definitions for these traits are summarized in Section VI.

The **TreeElement** trait allows objects to be organized into tree data structures. The tree structure corresponding to a 3 page, 3 column document containing graphics and text is shown in figure 9. This structure will be explained more fully after we have introduced the Schema and ListSchema traits.

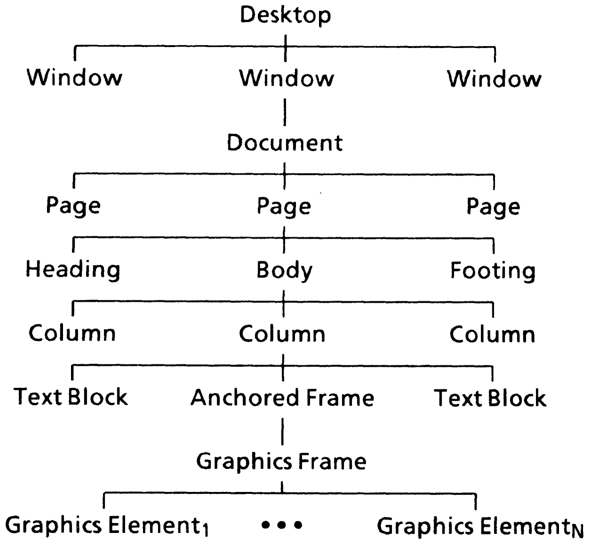


Figure 9
Desktop Imaging Tree

The **Schema** trait forms the basis for imaging, pointing resolution, selecting and editing within Star. It defines 22 replaceable operations but for the purposes of this discussion we are only be concerned with those shown in Section VI. These operations allow an object to be asked its size (Size), to honor a request that it paint its image (Paint), to handle a pointing action by the mouse (PointedAt), to respond to an editing action when it is selected (Edit), to return the location of a child relative to itself (RelLocChild) or relative to the upper left corner of the screen (ScreenLocChild).

GSchema is an extension to Schema to meet the needs of graphics objects and is the basic trait carried by all graphics objects. It provides state variables for its size and location within its parent. Of the 39 replaceable operations it defines we are concerned only with Contend which is used during pointing.

ListSchema is a trait carried by an object that has non-overlapping children that are arranged either vertically with left edges aligned (pages in a document) or horizontally with top edges aligned (columns on a page), see figure 10. These two arrangements are embodied in the ListSchema trait that is carried by an object that wishes to arrange its children in this manner. The state defines the inter-child spacing, the margin between the children and the parent carrying this trait and the color for the areas not covered by children. A list with color black and non-zero spacing and margin values is a common method for drawing lines around objects.

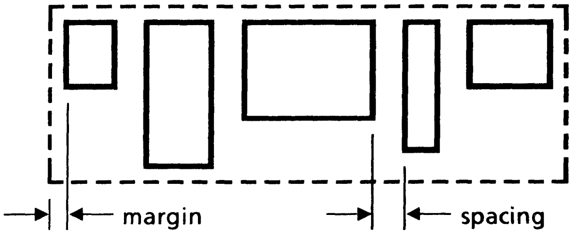


Figure 10
Horizontal ListSchema

HasCp is a trait carried by all graphics objects that have control points. The only graphics object that does not have control points is the cluster object created and destroyed by the JOIN and SPLIT functions. For a given class the number of control points is the same so a replaceable operation, CountCp, is defined to return this value, e.g. 2 for line objects, 8 for rectangles. The replaceable operation LocCp returns the object-relative location of a control point. The fixed operation HighlightCp highlights a control point in one of the styles: default (small square), guiding (larger square) or pinned (an X). ClosestCp and FurthestCp are fixed operations that enumerate the control points for an object and use LocCp to determine the control point closest and furthest from a particular coordinate. They are used to determine guiding and pinned control points. Rectangles, graphics frames and text frames have the same number and arrangement of control points and so use the same realizations for CountCp and LocCp. This increases code sharing so that only 2 realizations for 2 separate replaceable operations are needed to implement all the control point behaviors for 3 classes of objects. This is typical of the code sharing benefits of the trait mechanism.

AnchoredFrame is the class trait for the graphics frame that is associated with the anchor character. There is no screen-visible element for this object. The keyboard insert of an graphics frame actually creates two objects, an anchored frame with a graphics frame inside it. It is the graphics frame that is the visible box. Anchored frame objects are also used to anchor equations in text.

An anchored frame object forms the boundary between the non-graphics and graphics domains. A page column is a vertical list of left edge aligned text blocks and anchored frames. The one and only child of the anchored frame is a graphics frame that may be aligned flush left, centered or flush right within the anchored frame as determined by a property on the graphics frame property sheet. Within the graphics frame there are no restrictions on object arrangement.

Line is the class trait for graphics lines. Its state retains the properties shown in figure 5.

TextBlock is the class trait for objects that have textual content. Further details about this trait are beyond the scope of this discussion. Text blocks and anchored frames are the only objects that exist in a document column.

Note that the Schema trait defines operations for asking an object its size and location but does not define corresponding state variables. Also note that the replaceable location query is a request to a parent object for the location *within the parent* of a child object, i.e. a parent-relative location. This is done, as we discuss below, for flexibility and economy of storage and for performance.

It was felt best not to force all classes to store their size in the same manner at the Schema level because the trait is used as a component trait for a large number of classes each with possibly quite different behavior, e.g. a horizontal list-like object may determine its size by summing the widths of its children and use the height of its tallest child as its own height while a graphics object may store this information in its state. This judgement has been shown correct by the diversity of methods for determing size that now exist as the Star software has matured and new features, objects and behaviors have been implemented. It is quite common for a trait to define a behavior, such as Schema Size, that requires the cooperation of all objects that carry it in order to complete the behavior.

For performance reasons the fundamental location query is in terms of location within parent. Displaying an object or changing its location on the screen should not require changing its state.

For example, the Star workstation processor has instructions that support moving bits from one part of the screen to another. Scrolling a page upward is merely a matter of moving existing screen bits and painting new bits into the vacated portion of the window; none of the scrolled objects needs to be told to modify any of their state. This processor support also aids performance because it is not necessary to invoke the Paint operation for objects that already have their image on the screen.

Also, changing the size of an object or deleting an object near the front of a document does not require changing the state of all following objects in the document.

When the screen location of an object is needed for an operation the object is passed its screen location as a parameter or it invokes the operation ScreenLocChild on it parent.

V. Two Examples

Star graphics was the first major piece of Star software designed in terms of traits and that used the full generality of the mechanism. Pieces of software designed or implemented prior to graphics have subsequently been converted to the traits mechanism. In this section we will describe two interactions between the traits presented in section III. We first show how the GSchema trait *completes* the Schema size and location behaviors and second show how it *extends* the Schema trait for pointing behavior.

The GSchema state records a size and parent-relative location. Fixed GSchema operations allow this information to be accessed and changed. All GSchema objects use the same realization for the Schema replaceable operation RelLocChild which invokes the fixed GSchema operation GetRelLocSelf on the child object. Note that for a graphics object

```
        GSchema.GetRelLocSelf[ object ]
```

returns the same value as

```
    Schema.RelLocChild[
        TreeElement.GetParent[ object ], object ]
```

Objects that carry the Schema trait are responsible for a rectangular patch of the screen. Among sibling objects this may be sole responsibility, as is the case between page objects, or may be a shared responsibility, as is the case between overlapping graphics objects.

Sole vs shared responsibility has interesting implications for the implementation of imaging behaviors. We will look at the pointing and selecting behavior of objects.

The document object carries the ListSchema trait and is the parent of page objects. It is quite easy for a document to determine which page contains the cursor and then pass the buck for processing the pointing action to that page. As long as the cursor remains within the bounds of the window displaying the document and within the bounds of the page, the page object has sole responsibility for tracking the movement of the cursor, for providing user feedback in the form of highlighting and for making a selection when the user releases the mouse button. If the cursor leaves these bounds with the button still depressed the page passes responsibility back to the document object for continued processing. A page satisfies its obligation by passing the buck to the column containing the cursor, etc.

When the user releases the button, the currently pointed-at object registers itself as the current selection with a central mechanism. Subsequent user editing actions are sent to it via the Schema Edit operation. It is up to the object to decide how to respond to the editing action, e.g. graphics lines ignore typing.

This method for button processing is embodied in the Schema replaceable operation PointedAt. Parameters for the operations are the object being asked to process the pointing action, its screen location, a tracking region, the current cursor location and the state of the mouse buttons. The return value for the operation is an updated cursor location and an updated mouse button state. The object must track the cursor as long as it is in the tracking region and must return control when the cursor leaves the region.

The semantics of PointedAt were designed for the non-graphics domain where nested list-like arrangements predominate, e.g. pages in documents, columns in pages. List-like arrangements also predominate outside windows but a description of their uses there is beyond the scope of this discussion.

The ListSchema trait provides a buck-passing realization for PointedAt that is used by almost all objects carrying the ListSchema trait.

Sibling graphics objects must share responsibility for pointing, eg. pointing *inside* the boundary of a rectangle may really lead to the selection of some other object. If the user points at the letter "x" in "Text" in figure 6a the text frame does not allow the user to point through to the rectangle under it. If the user points at the upper left corner of the text frame in figure 6b the user is pointing *through* the rectangle. This sharing behavior is implemented by the GSchema operation Contend described below.

If the cursor is positioned inside a graphics frame and a mouse button is pushed, the list-like PointedAt realization behavior described above resolves the cursor to the window containing the cursor, the document within the window, the proper page, the proper column and finally to the anchored frame within the column.

The anchored frame's realization for PointedAt is to enumerate it descendants and ask each how much interest it has in the current cursor position. The child with the greatest interest is passed the buck for processing the pointing action by invoking its PointedAt realization. The tracking region it is passed is *very* small, a box about 1/8" square. This allows the anchored frame to regain control and re-poll its descendants if the user moves the cursor any significant amount. The GSchema operation Contend is the operation used to ask a graphics object how much interest it has in the current cursor position. The descendants are enumerated top-down and enumeration stops when all have been enumerated or one of the descendants says stop, e.g. the text frame in figure 6a when the cursor is pointing at the letter "x".

Rather than change the semantics of PointedAt for graphics objects, or replacing it completely with a new set of operations to do pointing resolution, we merely added a pre-processing phase by adding Contend. The extending of behaviors by addition, not replacement, of operations is a capability offered by the traits mechanism and used widely throughout Star.

Note also that the user is allowed to button down near a graphics element and see it highlight, move the mouse with the button still down out of the graphics frame and point to a letter in the main document text and see the graphics element de-highlight and the letter highlight, continue dragging the mouse out of the document window and point to an icon and see the letter de-highlight and see the icon highlight and then select the icon by releasing the button.

All this is possible as a single user action. In the traditional sense this may be thought of as automatically invoking three editors in succession, the *graphics editor*, the *text editor* and the *desktop editor*, and passing control between them when in reality we are traversing a tree of objects and asking each to exhibit its own behavior. The implementation corresponds to this model and for this reason there is no *graphics editor* per se that is invoked by the Star user; there are only graphics behaviors that are exhibited in response to user actions and these behaviors are available at all times.

VI. Trait Summary

The following trait summary is in the order they were introduced above. Additional state variables and operations beyond the scope of this discussion are represented as "...".

trait name: **TreeElement**
state: next, parent, eldest
component traits: none
fixed operations: GetNext, SetNext, GetParent, SetParent, GetEldest, SetEldest, ...
replaceable operations: none

trait name: **Schema**
state: none
component traits: TreeElement
fixed operations: ScreenLocChild, ...
replaceable operations: Size, Paint, PointedAt, Edit, RelLocChild, ...

trait name: **GSchema**
state: size, location in parent, ...
component traits: Schema
fixed operations: GetSize, SetSize, GetRelLocSelf, SetRelLocSelf, ...
replaceable operations: Contend, ...

trait name: **ListSchema**
state: margin, spacing, color
component traits: Schema
fixed operations: ...
replaceable operations: ...

trait name: **HasCp**
state: none
component traits: none
fixed operations: HighlightCp, ClosestCp, FurthestCp, ...
replaceable operations: CountCp, LocCp, ...

trait name: **Line**
state: width, style, line endings, constraint
component traits: GSchema, HasCp
fixed operations: none
replaceable operations: none

trait name: **TextBlock**
state: text contents, ...
component traits: Schema, ...
fixed operations: ...
replaceable operations: ...

VII. Conclusions

Adopting an object-oriented implementation and the traits mechanism has been a success.

The initial graphics design and implementation (without bar charts and text frames) was done in one work year by a new hire who knew nothing about the Mesa language or the Star object-oriented methodology or the traits mechansim. This was in large part due to the building block nature of the methodology. Also the graphics functional specification had already been written, and one of the authors had validated the graphics user interface by prototyping on the Xerox Alto using a different implementation technique.

Subsequent to graphics, most of Star has been converted to this methodology, and three other major pieces of software have been undertaken: an equations editing capability, a 3720 emulations window, and a table editing capability. All are having equally good results.

The trait mechanism has allowed a rather straightforward mapping of Star UI elements to internal implementing objects.

REFERENCES

1. G. Curry, L. Baer, D. Lipkie and B. Lee, "Traits - An Approach to Multiple-Inheritance Subclassing," *Conference on Office Automation Systems*, Philadelphia, Penn., June 1982.

2. E. Harslem and L.E. Nelson, "A Retrospective on the Development of Star," submitted to *6th International Conference on Software Engineering*; Tokyo, Japan, Sept, 1982.

3. C. Irby, L. Berginsteinsson, T. Moran, W. Newman and L. Tesler, "A Methodology for User Interface Design", Systems Development Division, Xerox Corporation, January 1977.

4. J. Mitchell, W. Maybury and R. Sweet, "Mesa Language manual," *Technical Report CSL-79-3*, Xerox Corp., Palo Alto Research Center, Palo Alto, CA, April 1979.

5. J. Seybold, "Xerox's 'Star'" in *The Seybold Report*, Media, Pennsylvania: Seybold Publications, v. 10, no. 16, 1981..

6. D. Smith, C. Irby, R. Kimball and E. Harslem, "The 7tar User Interface: An Overview," *NCC '82*.

7. D. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem, "Designing the Star User Interface," *Byte*, v. 7, no. 4, 1982.

8. L. Tesler, "The Smalltalk Environment", *Byte*, V. 6, no. 8, 1981.

9. C. Thacker, E. McCreight, B. Lampson, R. Sproull and D. Boggs, "Alto: A Personal Computer," *Computer Structures: Principles and Examples*, D. Siewiorek, C. Bell and A. Newell, editors, McGraw-Hill, 1982.

The Design of Star's Records Processing
Data Processing for the Non-Computer Professional

Robert Purvy, Jerry Farrell, Paul Klose

September 1982

© Copyright Association for Computing Machinery 1983. Printed with permission.

Abstract: Xerox' Star Professional Workstation is distinguished by a graphic user interface committed to the What-you-see-is-what-you-get design philosophy. The system promotes a see / point / push-a-button style of interaction with immediate feedback, in marked contrast to more familiar programming or command-language interfaces.

Star's Records Processing feature integrates traditional data processing functionality into this user model, using standard Star documents for data definition, entry, display, update and report generation. Benefits include an economy of concepts and effort for user and implementor alike, along with the synergy of a unified environment.

A version of this paper has been submitted to ACM's *Transactions on Office Information Systems,* to appear in the first quarter of 1983.

CR Categories: H.2.3, H.3.3, H.4.*

Key Words and phrases: office automation, office information systems, records processing, database, user interface.

XEROX

Xerox Corporation
Office Systems Division
3450 Hillview Av.
Palo Alto, California 94304

1. Background

The Records Processing (hereafter "RP") feature in the Xerox 8010 "Star" Professional Workstation is heavily influenced by the nature of the system it is associated with. Star's intended users are the so-called *knowledge workers*: financial analysts, research and development scientists and engineers, technical writers etc. These individuals work in relatively unstructured situations; they produce and communicate ideas. They may be contrasted with clerical personnel on the one hand, whose jobs tend to be much more structured, and defined in terms of some more concrete product; and managers on the other, who tend to concentrate on analyzing and deciding among already-presented alternatives, and exercising interpersonal skills to direct and motivate subordinates in implementing those decisions.

This model of Star's users has significant implications for the design of the system. Timely, effective communication is a paramount requirement for Star's user, both in collecting input and in presenting output. This means rich media, with the quality of traditional typographic and graphic communications. At the same time, the speed and comprehensiveness of electronic processing must be maintained. Facilities for automating large and routine systems are considered less important, as we expect our users to devote relatively less time to those processes, and to have diverse routines which are difficult to cover completely. Finally, while Star users are intelligent and capable people, they generally are *not* computer professionals; Star can not require computer sophistication from them, however advanced their needs become. This dictates a user interface which presents a familiar and uniform model to the user. To motivate a discussion of how Star's unified environment requires new and interesting approaches to these facilities, we first present a brief overview of the whole system.

1.1 Overview of Star

Star is a powerful personal computer and office automation system. Its hardware consists of one or two cabinets 12 inches wide by 25 inches high and deep, housing a processor of about .5 mips, 384KB or more of memory, and a 10 or 28MB hard disk. (The second cabinet is required if the larger disk is installed). This system is connected to an Ethernet [DEC / Intel / Xerox 80] with associated network services, such as filing, printing and electronic mail. The user faces a large, high-resolution (1024 x 808) bit-mapped display, an independent keyboard, and 2-key mouse for cursor control.

Star's user interface has been described in detail elsewhere ([Seybold 81], [Smith 82]); we sketch it briefly here for the reader's convenience. The user is presented an environment modeled on common office equipment and procedures. The screen displays a *desktop,* and on it, icons of elements of a physical office: stylized pictures of documents, file drawers and folders, in- and outbaskets, etc. These objects may be *selected*: the user moves the mouse until the cursor is over one, and presses one of the mouse keys. A selected icon may be *moved* and *copied* about on the Desktop, or *deleted*: it is selected, and the MOVE, COPY or DELETE key is pressed. When a destination is required, the user indicates it with the mouse. An icon which has contents (e.g. a document or a folder) may be *opened*: it is selected, and the OPEN key is pressed. A window opens on the screen, displaying the contents of that object.

Window contents, such as text in a document, may be selected and manipulated in the same fashion: Users point the cursor at text and press the mouse button to select it. They move, copy and delete selected text as they do icons on the Desktop. They add new text to the selection by simply typing it. Similarly, users may select and move a document from a folder to the Desktop or back, or select and open one folder nested inside another.

Some icons incorporate functions: File drawers, printers, and in- and outbaskets provide the interface to services provided on the Ethernet. Thus, moving a document to a printer icon causes it to be printed. A document is transferred to a file server by moving it to a file drawer; it is retrieved by opening the file drawer and moving or copying it back to the Desktop. Electronic mail is sent by moving an icon to the out-basket; and incoming mail accessed by opening the inbasket.

Objects in Star have *properties* appropriate to their type. For example

Object Type	Properties
documents	name, size, owner, create / read / write timestamps
characters	font, size, position, bold and italic faces
graphic rectangles	border weight and style, gray-scale

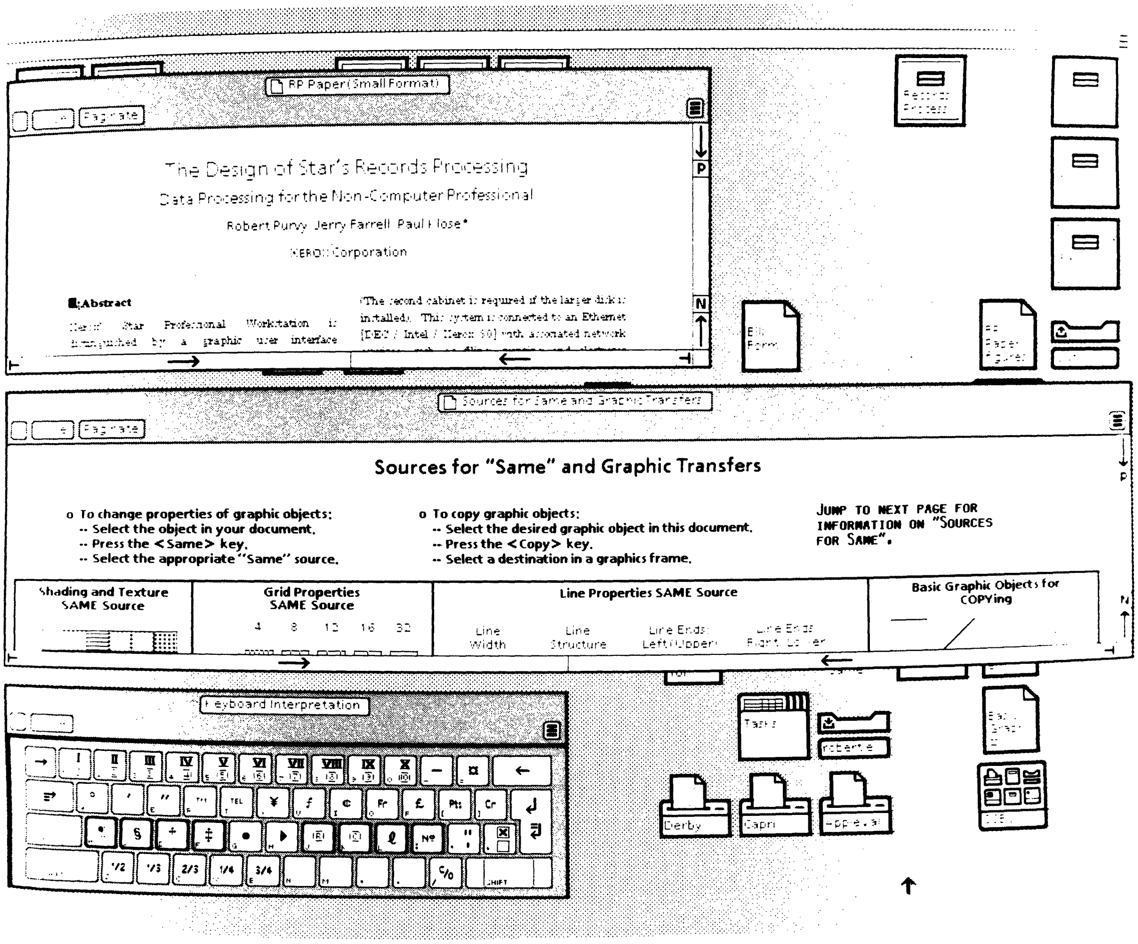
Users access an object's properties by selecting it and pressing the PROP'S key. A new window is opened on the screen to display the properties, which may then be adjusted as desired. Figure 1 shows a Star screen with many facilities in evidence (more than would generally be seen in real circumstances).

Since the screen is bit-mapped, it can display characters in a multitude of fonts and faces, accented letters appropriate to various European languages, non-Roman scripts, and special symbols for math, logic and office applications. To select from these various collections of characters, the keyboard may be remapped via the KEYBOARD key, with the current interpretation displayed on the screen whenever the user wishes, as shown in Figure 1.

1.2 Star User Interface and RP

Star is a unified and integrated environment. The operations invoked by selection and the keys MOVE, COPY, DELETE and PROP'S are considered *universal*: their semantics are expected to be uniform throughout the system, and the main operations a user wishes to perform in any domain are expressed in terms of them. There is no conventional command language, nor any executive to interpret one. In the usual case, the user does not write independent programs, and there is no mechanism for running them; for example, text editing is a function performed by the same actions throughout the system, rather than the special province of a text-editor program to be run as required.

Professionals require access to collections of structured data and facilities for processing them, sometimes in fairly repetitious ways. However, their requirements are quite changeable and personalized, and often small in volume; we contrast this domain to applications served by traditional data-processing, report-writing, and data management systems controlled by a programming language interface or command language. At the same time, the results of this processing appear commonly in other materials, and so require thorough integration with the remainder of the professional's system – document creation, electronic mail, and the like.



The Design of Star's Records Processing

Data Processing for the Non-Computer Professional

Robert Purvy Jerry Farrell Paul Klose*

XEROX Corporation

Abstract

Xerox' Star Professional Workstation is distinguished by a graphic user interface

(The second cabinet is required if the larger disk is installed.) This system is connected to an Ethernet [DEC / Intel / Xerox 80] with associated network

Sources for "Same" and Graphic Transfers

o To change properties of graphic objects:
-- Select the object in your document.
-- Press the <Same> key.
-- Select the appropriate "Same" source.

o To copy graphic objects:
-- Select the desired graphic object in this document.
-- Press the <Copy> key.
-- Select a destination in a graphics frame.

JUMP TO NEXT PAGE FOR INFORMATION ON "SOURCES FOR SAME".

Shading and Texture SAME Source	Grid Properties SAME Source	Line Properties SAME Source	Basic Graphic Objects for COPYing
	4 8 12 16 32	Line Width · Line Structure · Line Ends: Left/Upper · Line Ends: Right/Lower	

In keeping with this model, RP is fully integrated into the Desktop environment, using the same basic operations and paradigms that hold throughout Star. It makes especially heavy use of standard Star documents to define the structure of record files, display results of queries, format and generate reports, and accept data for additions and updates to stored data. It has extended the graphical interface to the process of specifying queries.

To some extent, RP has traded functional power for conceptual simplicity: Data are shared only in the sense that record files, like documents, may be mailed and stored in commonly accessible locations. There are no facilities for constructing virtual collections of data by specifying joins and other manipulations of independent record files. (This latter restriction is mitigated by RP's use of hierarchical structures, allowing related data to be stored together in a single file.)

1.3 Comparison of RP to Other Systems

A number of recently-developed commercial and academic systems share Star RP's departure from traditional data processing. Use of a *form*, generally taken to mean some kind of stylized document with a defined field structure, is particularly common, although details differ from system to system. Several systems are based on relational database systems; as such, they tend to provide richer database facilities than RP, but less integration into other aspects of office automation, such as print-quality text and graphics and electronic mail. We consider two systems in some detail: Zloof's Office-By-Example (an extension of his earlier, well-known Query-By-Example), and the University of Toronto's *OFS* office forms system. For discussions of other relevant work, see [Ellis 80], [Embley 81], and[Yao 81].

Star differs from Query-By-Example / Office-By-Example [Zloof 77, Zloof 81] primarily in its scope. QBE is an interface to a general relational data management system, while a Star record file is primarily a simpler, personal database. This less powerful design has enabled considerable simplication in the specification of queries and the relationship of Star documents to record files.

To a very large extent, we have made interactions with a record file the same as interactions with document structures, particularly Tables (see **2.2**). Thus, whereas in QBE the user indicates update, deletion, or insertion of a record with special operators (U, D, or I), in Star he might accomplish the same ends by normal editing of a table row's contents, deletion of a row, or insertion of a new row. Since hierarchically nested fields, if any, are actually stored with the record, rather than being linked to in a separate file, there is no ambiguity about what happens to them during update: if you delete a row in a *table*, you also delete all of its subrows, and the analogy carries across to record files.

Star maintains the basic query syntax of QBE. However, Star uses *views* (**4.1**) in a more fundamental and universal way than does QBE. In Star, certain functions that are performed by special operators in QBE are implicit in the user's definition of view properties, especially the choice of view document.

The University of Toronto's *OFS* [Tsichritzis 80] is explicitly aimed at a more structured environment than Star, the office conceived more in the sense of a bureau. Forms are associated with well-defined office procedures, and considerable emphasis is laid on authentication, authorization, and accountability. Interaction with forms is carried out through a command language at the user station, which may be either a personal computer or a terminal to a shared processor. Forms are communicated to or from a station via electronic mail. Alternatively, a collection of forms may be accessed as a relation in a database system, with the underlying data and indices shared between the two systems. The conception of the form file as the relation of data plus an associated form is similar in spirit to Star's association of display forms with a record, although Star considers the data more fundamental than the form in which it is rendered, and hence allows the association of multiple forms with the same collection of records. As with OBE, OFS exhibits the power of a full database system, which enables more and larger applications. This is particularly relevant for OFS' target environment, where collections of data may be expected to be larger than Star's, and required forms of access may be at once more established and more complex. The inclusion of office procedures is less clearly a distinction between the systems, since some of the RP icon-level manipulations embody simple office procedures (**5**), and more complicated procedures may be handled by the Star customer programming language (**2.3**). By its association with Star, RP derives a very high-quality graphical interface; in contrast, OFS is designed to be operable from a minimal terminal. This has effects on the capabilities of the system; display of repeating groups, for instance, is excluded from OFS, while Star requires the facility in many contexts besides RP.

Tsichritzis notes the conflict between providing enough power in a language to handle a broad selection of applications and the fear of overwhelming the user with the attendant complexity. Star's RP and customer programming designs have had to confront this same dilemma, with approximately the same result: a simple facility is provided to cover many interesting simple cases, with escape to a more general programming language for users with the need and ambition.

Star in general, and RP in particular, exhibit a sophistication about multi-national and multi-lingual applications which we have not seen in any comparable system. There are no deep theoretical issues here, but there are a great many practical details which must be dealt with. Texts can be stored in any of the various scripts supported by Star, including special characters in languages which basically use a Roman alphabet (currency symbols for pounds or yen), non-Roman alphabets (Greek, Katakana and Hiragana), and ideographic texts (Kanji). Ordering relations depend on the language (*ch* is a single letter in Spanish, falling between *c* and *d*; *ä* sorts the same as *a* in German, but is a separate letter which follows *z* in Swedish). Formats for dates and numbers differ among countries, affecting the interpretation of input and the form of output (123.456 is three orders of magnitude greater in France than in the US; Japanese may schedule a conference in the 6th month of the 58th year of the era of Shining Harmony).

2. Star Features Closely Related to RP

Three features of Star are particularly relevant to consideration of RP: Document Fields, Tables, and the customer programming language.

2.1 Fields

As described in the first section, the user has available several remappings of the standard keyboard. One such mapping is to a collection of special objects that may be inserted in text. These include equations, graphic frames and page breaks. Also included are *fields* and *tables*, which correspond approximately to the notion of variables in a programming language. In the simplest case, a field is a container for a single value. Structured data is represented in tables, discussed below; these correspond to programming language record definitions. Fields may occur in running text, as in a form letter, in which case their contents are formatted along with the surrounding characters. Alternatively, they may be placed in a frame with fixed size and position, as in a business form or report. Documents containing illustrations and/or fields are treated like any others on the Desktop and in the filing system.

The user fills in fields with the normal Star editing operations, augmented with the NEXT and SKIP keys. The fields in a document are arranged in an order (settable by the user), and the NEXT and SKIP keys on the keyboard will move the selection through the fields of the document in this order, ignoring intervening document contents. Field contents may be selected with the mouse, like any other document contents, and edited, moved or copied to other areas.

The mechanism for getting a new (empty) field in a document is, essentially, to type it. Star's alternate keyboard mappings are presented in response to the KEYBOARD key on the keyboard. Selecting the SPECIAL option sets the keyboard keys to the special objects that may be inserted in text, mentioned at the beginning of this section. In particular, the "Z" key is remapped to a field; pressing it now results in insertion (at the current type-in point) of a new field with default properties. Its position is marked with a pair of bracket characters: ⌜⌟. Once a field is inserted, it may be selected, and its properties set, as with normal characters or any other Star object. Thus, users create forms by straightforward extensions of other document operations.

2.1.1 Field properties

A field has a rich collection of properties, of which the most important are its *name*, its *type*, the *format* of its contents, its *range* of valid values, and optionally a *rule* for determining its value. Other properties include a description (which may be used as a prompt), and the language of the contents (which is required to deal with the multi-lingual issues mentioned in **1.3**).

A field's name is assigned by the system when it is inserted in the document; it may be changed by the user at any time. The name must be unique among fields in the containing document.

There are four field types: **Text**, **Amount**, **Date**, and **Any**. The first three have obvious constraints on their values. **Any** fields are allowed to contain anything legal in a document. The default is **Any**.

Formats may be used for data validation on input of Text fields, e.g. part number or social security number format. Date and Amount fields do not have input validation according to formats, but instead accept anything that "makes sense" according to the rules for those field types. Formats may be specified for output of Dates and Amounts to enforce uniformity of appearance. Date formats offer a choice among standard representations of dates, and are language-dependent. Format characters for Amounts and Text are similar to those in COBOL or PL/1 picture clauses, and appear on the SPECIAL keyboard.

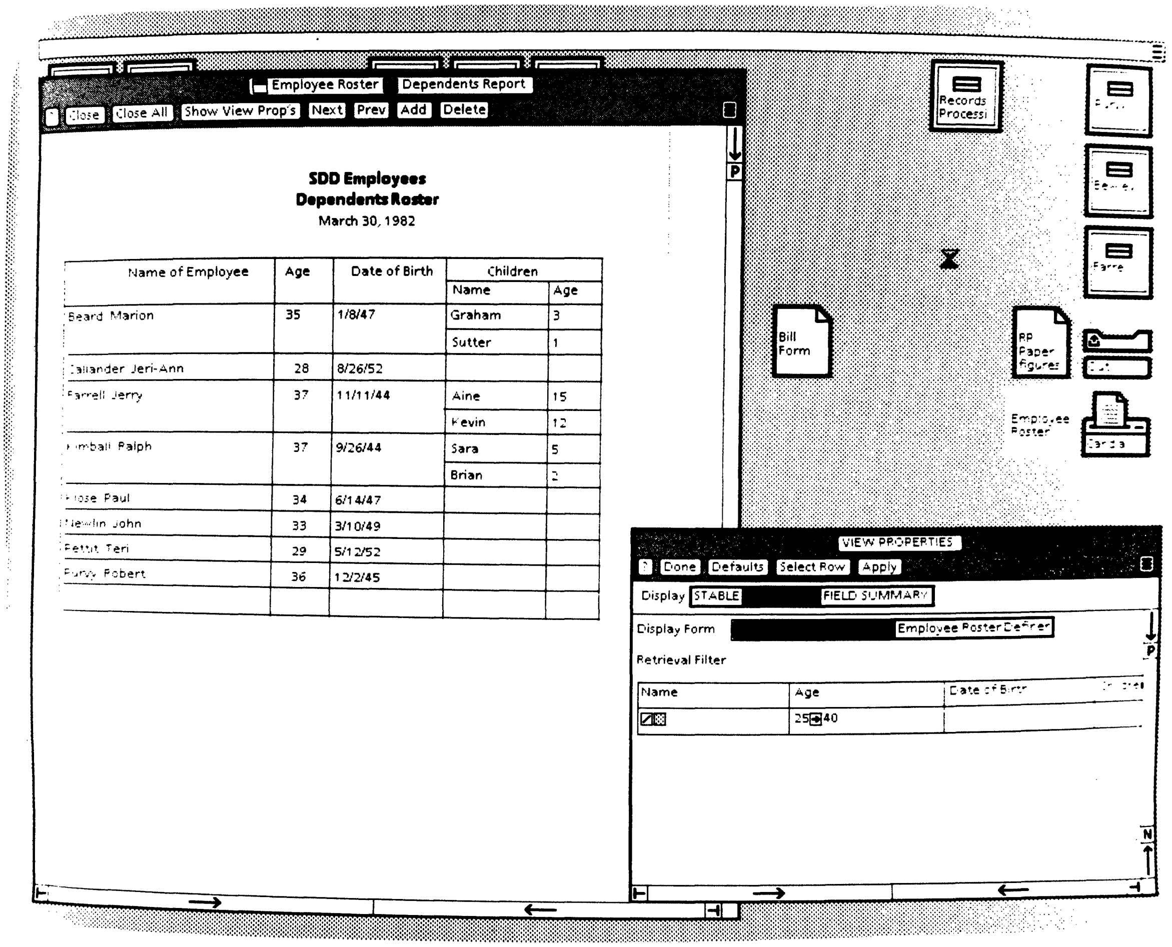
The Range property specifies acceptable values for the field. These involve more characters from the SPECIAL keyboard, indicating a closed interval and a textual ellipsis which matches 0 or more arbitrary characters. (These may be indicated by "→" and "..."; on the screen, they are given distinctive images which do not appear in text.) The range may be unbounded at either end: 0→, 1→10, →127. These same forms are used in specifying desired values in RP Filters (4).

The Fill-in Rule property is discussed in section 2.3. Figure 2 shows a form with an open property sheet for a field with a fill-in rule.

2.2 Tables*

A table is another of the special objects which may be inserted in a document. It is a rectangular frame with rich formatting characteristics: headings, footings, ruling lines, margins, captions, rows and columns which automatically adjust their extents and which may be selected, moved, copied, etc.

* We should note here that the implementation of Tables was not completed until after the first release of Star.



SDD Employees
Dependents Roster
March 30, 1982

Name of Employee	Age	Date of Birth	Children	
			Name	Age
Beard Marion	35	1/8/47	Graham	3
			Sutter	1
Callander Jeri-Ann	28	8/26/52		
Farrell Jerry	37	11/11/44	Aine	15
			Kevin	12
Kimball Ralph	37	9/26/44	Sara	5
			Brian	2
Rose Paul	34	6/14/47		
Newlin John	33	3/10/49		
Pettit Teri	29	5/12/52		
Purvy Robert	36	12/2/45		

Display STABLE FIELD SUMMARY

Display Form Employee Roster Definer

Retrieval Filter

Name	Age	Date of Birth
	25 → 40	

Of more interest here, a table is also a hierarchical structure of fields, arranged in rows and columns. A column may be *divided* (have sub-columns). A divided column may also be *repeating*, which allows for nested sub-rows within a row. (See, for instance, Figure 3b.) Conceptually, the table itself is a simply a higher-level repeating divided column. Thus, tables correspond to structured variables in standard programming languages.

Besides formatting control, the properties of a table column include the standard field properties. These apply to each of the fields in that column. Thus, all fields in a column bear the same name (they are distinguished by an index); they share the same format; and a single fill-in rule may be applied to each.

2.3 Fill-in Rules and CUSP

The use of fill-in rules on fields must qualify the statement above, that ". . . the user does not write independent programs, and there is no mechanism for running them." A user's day-to-day activities are not normally addressed by writing programs. Nonetheless, some user computations are best expressed by the user; Star's CUStomer Programming language responds to this requirement. In the first release of Star, CUSP appears only in a *fill-in rule*, which is a property of a field or table column.

A fill-in rule is an expression in a simple language with no side effects, and no control constructs except the conditional expression. It does include arithmetic, string concatenation, and aggregate operations like Sum and Count, comparison and boolean operators, and a conditional expression which selects a single value from a number of alternatives. There are built-in expressions for the current date, time and user identification. The value returned by the expression is stored in its field, properly converted and formatted. Simple fill-in rules include

```
CurrentDate

Taxable * 1.065              "Taxable" must be the name of another field in the same document

Choose
    Miles < 200 -> Miles * .20;
    Otherwise -> 40 + (Miles - 200)*.17
                             The Choice is simply a CASE statement, with a required Else.
```

The use of fill-in rules is extended to table columns, with provision for referencing the current row. Thus, a rule for computing one field as the sum of two others may be used to make one column in a table hold line totals for corresponding elements of the other columns.

A later release of Star includes a capability for users to program their own *Buttons*. These are parameterless procedures which may include iteration over sets, side-effects on fields and manipulation of objects on the Desktop, parallel to manual actions in the user interface. Buttons may appear in documents, and they mimic, in appearance and operation, the behavior of menu commands built into Star. Eventually, CUSP will become a full programming language, with procedures, variables, parameters and a programming environment. We are proceeding in this direction for two reasons:

1. The complexity of user applications is essentially unbounded, which makes some sort of programming language virtually mandatory.

2. As in the rest of Star, we believe we can layer the complexity of CUSP, presenting only as much as is relevant in a given situation. Non-programming users may content themselves with the facilities described in the rest of this paper; fill-in rules ignore flow-of-control and binding issues; buttons introduce restricted procedurality in a familiar context.

Taken together, these points echo Alan Kay's dictum "Simple things should be simple; hard things should be possible."

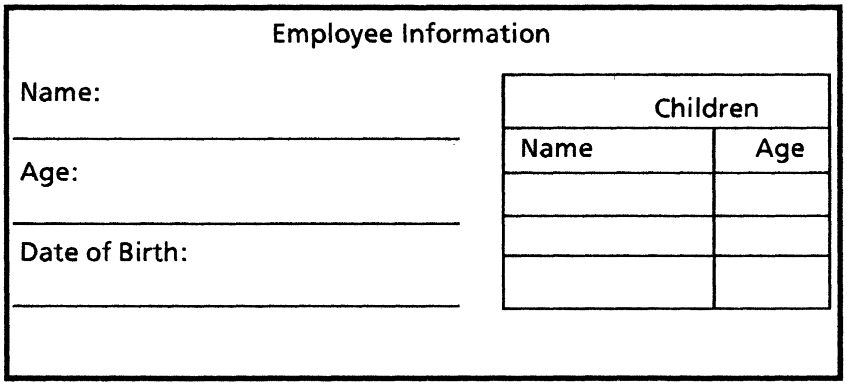
3. A Functional Description of Star RP

The next three sections of this paper review the functions of traditional data processing, with attention to how the Star user interface provides a graphical, non-procedural way of presenting them to the user.

3.1 Data definition

Data definition is the first function required of RP. The field structure of the record file must be indicated to the system (along with the types and constraints on the individual fields) before data can be entered or retrieved. This function is normally served by a data definition [sub]language. Star provides this function via the mechanisms already used to define fields in documents; in fact, the structure of a record file is set simply by indicating a document whose field structure is to be copied.

Each Star Desktop includes access to a collection of useful templates, e.g. an empty folder and a blank document. To create a new record file, the user copies the *empty record file*, and opens the copy. The window menu will include a command named Define Structure. The user selects a document which has the field structure desired for the new record file, and invokes the Define Structure command . Star reads through this *defining form*, copying to the record file the descriptions of fields and tables encountered. When this process is completed, the Define Structure command disappears from the window, and the record file is defined.



Employee Information

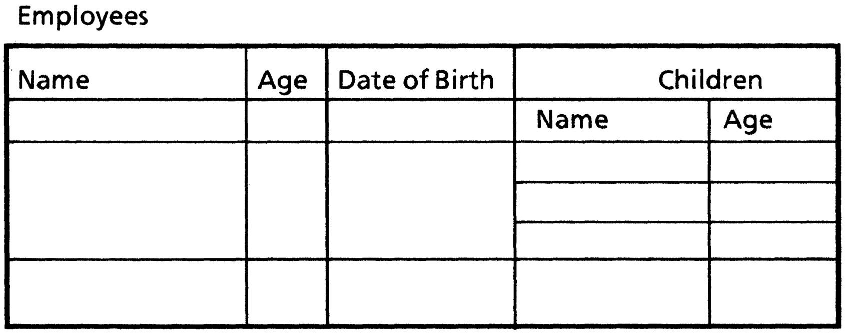
Name:

Age:

Date of Birth:

Children	
Name	Age

Figure 3a: Fields in a Form



Employees

Name	Age	Date of Birth	Children	
			Name	Age

Figure 3b: Corresponding Record Structure

The details of the definition process may be illustrated with an example: The personnel form in Figure 3a has a number of independent fields (Name, Age and Date of Birth), and a table of dependents named Children; the table's columns are Name and Age. If used as a defining form, this document would generate a record file structure as illustrated in Figure 3b. The independent fields and the table generate top-level fields in the record; the additional hierarchy of the Children table is reflected in a subdivided column in the record with repeating sub-records. All field properties (name, type, language, range constraints, ...) are carried over to the field in the record, except for any fill-in rule. (Since this is the definition of the stored data, it would be either redundant or inconsistent to leave a fill-in rule on the field. Therefore, the field is generated with all its properties except the rule.) A slightly anomalous case arises for documents which contain *only* a single table. By a strict adherence to the process we have described, we would expect a record with a single field; that field in turn would be sub-divided, with a sub-structure corresponding to the columns of the table. For convenience, such a document generates instead a record whose structure exactly matches the table's.

3.2 Display of data

The correspondence between the field structure of records and of documents carries over into all other access to record file data: Star documents containing fields and/or tables are used to add, display and modify records in record files. Multiple documents may be associated with a record file to provide varied forms of display of the data. Each such document is called a *display document.* A display document may contain only a subset of the fields in a record. It may contain additional fields which have fill-in rules to compute aggregate functions over the data. Its format may be that of a tabular report with data from many records gathered into a single document, or of a form whose multiple instances each correspond to a single record. Non-field text and formatting may include all the general facilities of Star documents, including arbitrary formatting and graphics. While these documents are referred to as display documents, it should be clear that any one may be used for both input and output; in fact there is no access to the data in a record file *except* through some document.

3.2.1 Lading

The process of establishing the correspondence between data in a document and in the records of a record file is called *lading.* Lading consists essentially of data transfer between fields that correspond by name. This covers both data input to the record file and output from it. The definition of corresponding names is generally straightforward, but must account for the capability of a single document to correspond to either a single record, or to a whole collection of them. (The two cases are very similar to the two varieties of defining document mentioned above.)

As mentioned in **2.1.1**, a field's name must be unique within its containing document or record. This is enforced immediately for independent fields and the top level of tables. In tables, only the fully qualified name (in the obvious pathname scheme) must be unique. Thus, in Figure 4, there is a field named Age, and a different one named Children.Age.

When lading between a record and a document which contains independent fields, the document and the record are considered to match; then any contained fields match if they have the same simple name, and their containers match. In this case, it will be seen there is one occurrence of a field in the document for each occurrence in a single record; multiple instances of the document must be generated for multiple records. Such a display document is called a *non-tabular form*; it would be appropriate for a form letter application, or form-style entry into the record file.

A document with a single table and no independent fields is treated somewhat differently. If the table has one or more columns whose names match record fields, then the table is considered to match the whole record, and rows of the table correspond to records. This is called a *tabular form,* and is typically used for reports and queries which may return several records. Independent fields which do not match record fields may occur in a tabular form; these typically have fill-in rules which compute summary data.

In either variety of display document, smaller tables may provide hierarchical structure with repeating sub-rows. The matching criterion must be refined to handle this case: fields do not match unless they share the same values for the Repeating and Divided properties. Figure 4 illustrates lading from a record into a tabular form. A new row will be generated in the Roster table for each record in the record file, and Children sub-rows will be generated in each row if there are corresponding children sub-records in that record of the record file. The defining form illustrated in Figure 3a could likewise be used as a display document for this record file; when laded, a new instance of the form would be generated for each record in the record file, each with its Children table filled out appropriately.

Field values are transferred between source and destination in the fill-in order of the destination. For output from the record file, fields with fill-in rules are computed as they are encountered in the fill-in order, on the basis of data already in the form. Fields which have neither computation rule nor matching source field are left unchanged. As each value is transferred, it is converted to the type and format of the destination field.

Figure 4 illustrates lading from the record structure of Figure 3 into a tabular form with a slightly different structure. In this case, the fields with the name Name match. The fields Date of Hire and Date of Birth do not match, despite the fact that they are both dates and are both at the same position. Therefore Date of Hire remains empty. The repeating divided field Children of the record file matches the column Children of the table. Their Age subcolumns therefore also match, and field values are transferred for each sub-row. No columns in Roster match either the other field named Age or the field Children.Name in the record, so their values are never accessed. The columns Salary and Grade do not match any field in the record and thus are not laded. Number will have a fill-in rule (Count [Roster[ThisRow].Children]), so its value is computed as the record is laded.

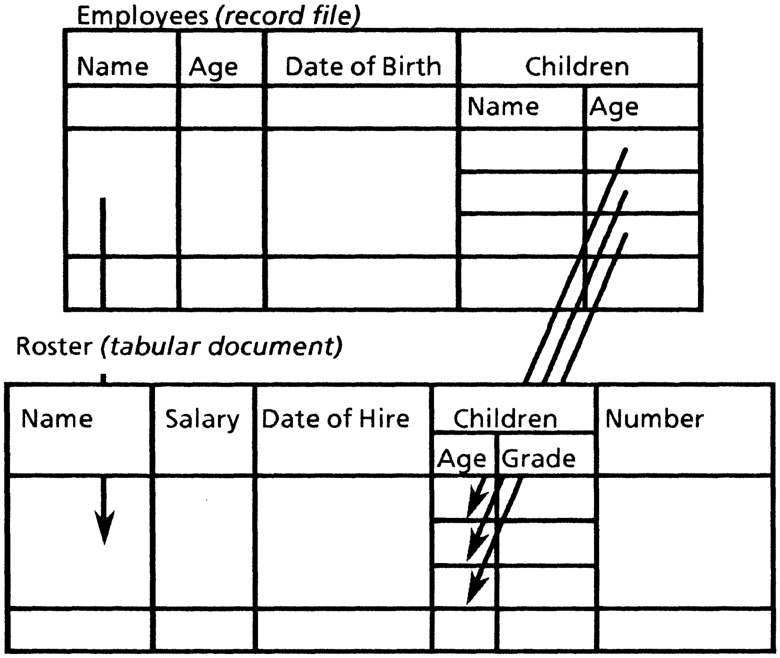


Figure 4: An Example of Lading

3.2.2 Scrolling and Thumbing

The portion of the record file displayed in the window is controlled in the same way as with documents and other long Star objects: by pointing with the mouse into a scroll bar on the right margin of the window. In a tabular form, there may be more records than can be displayed at once on the screen. The table is filled with rows which display a contiguous subset of the records in the record file. By thumbing, the user can jump to any point of the record file, causing the table to display a different set of records. The user can also scroll the records up or down one at a time.

In a non-tabular form, scrolling and thumbing cause the display document to be repositioned, since one record may be formatted into several document pages. The Next menu command displays the record following the one currently displayed; Prev backs up one record.

3.3 Inserting Records

To add a record in a tabular form, the user adds a new row to the table, using the standard table operations. The user then types the data for the fields of the new record. Star provides automatic confirmation during record insertion.

In a non-tabular form, the user is provided with an additional command in the auxiliary menu, Add Record. Invoking Add Record causes a new copy of the form to be displayed, with all of its fields empty and with all of its tables rowless. The user may now enter data by typing into the empty fields. When the user confirms his changes, the record is added.

Records may also be added to a record file in a batch; this process is invoked by user actions at the icon level, described in section 5.

3.4 Updating Records

A record is updated by editing its contents while it is being displayed through a document. Therefore, modifying the contents of a record involves exactly the same user actions as editing the contents of fields within a document.

RP uses a data validation scheme which minimizes the chance for user error: once the user begins editing a record, he is not permitted to edit any other record in the file; he must first confirm or cancel the changes already made. Until he confirms or cancels his edits, the user has only modified the display form, and not the record file. When confirmed, all fields of the updated record are validated according to both the record file's and the display document's field constraints. If any fields are invalid, then the record is not modified, and the user is notified as to which field is in error, so that he can can make the appropriate corrections. No changes are made to the record file; either all of the changes go through or none of them. If the user cancels his edits, then all changes are undone; the form is redisplayed so that it shows the original record contents.

3.5 MOVE / COPY / DELETE Record

One or more records displayed in a tabular form can be manipulated as a unit by selecting one or more rows and invoking MOVE, COPY or DELETE. These commands operate exactly as in documents and do not have to be confirmed. New records may also be added by selecting one or more table rows in another record file window and moving or copying them into the destination record file window. Source field values are copied into destination fields with matching field names, using the lading mechanism.

For records displayed through a non-tabular form, the user cannot select the whole document/record in the window; therefore menu commands are added to the window to Add and Delete records.

4. Querying using filters

Every database system provides some mechanism by which the user can cause a subset of the data to be extracted from the database and displayed, copied, printed, or otherwise made available for further operations. While the first database systems required some sort of programmer intervention to accomplish this, the current state of the art allows for direct query by non-data-processing personnel. Star has no query language as such; rather, it provides a facility called filtering, similar to Query-By-Example [Zloof 77].

Filtering is the process by which the user queries a Star record file. A filter is a predicate on the fields of the record file. When a user sets a filter, he is asking to see only those records that "pass" the filter. The filter appears to the user as a table; in fact, it looks exactly like the Full Tabular Form. All normal table operations (e.g. NEXT and selecting and adding rows) are available in the filter table. The filter acts as a template which defines the subset of records that the user is interested in. Each entry in the table may contain a field pattern, which specifies a condition that a corresponding record's field must satisfy in order to pass the filter. Field patterns have the same syntax and capabilities as the range specifications for fields in forms. Some examples of field patterns that might be specified for the example record file of employees used above are:

employee names starting with A thru M:	**A → M ...**	in the Name field
employees born in 1951:	**1951**	in Date of Birth
employees whose records have no entry for Age: (presumably an error condition)	**the Special "Empty" character**	in the Age field

Each row in the filter represents a simultaneous set of conditions that records must satisfy. In other words, the field patterns are AND'ed in a row. Thus, using the above examples, by filling in both the Name column and the Date of Birth column, the user may construct a filter passing only those employees whose names are between A and M, AND who were born in 1951.

To get an OR'ing of conditions, additional rows can be added to the filter using the normal table operations. If the user wanted to change the above example to pass employees whose names are between A and M, OR who were born in 1951, he would simply have two rows, one with the first condition and one with the second. To summarize, field predicates are AND'ed across columns and OR'ed down rows.

By using filters, the user is able to extract the subset of the records that he is interested in, merely by filling in a table, i.e. using the same operations that he already uses to interact with the records themselves. Figure 4 illustrates our example record file, with a filter selecting employees with non-null names and ages in the range 25 thru 40.

4.1 Views

All interaction with a Star record file is through views. A view consists of three attributes: a sort order, a View Filter, and a display form. It can be thought of as the encapsulation of one distinct use of the record file. For example, a large record file of employees might be used in a number of different ways within an organization: to input new employees; to print out monthly reports of all employees, alphabetically; to send form letters to special groups of employees; and perhaps for a wide range of querying by the personnel manager: who has been hired in the last month? how many employees are over 60? etc. It should be noted that our definition of "view" differs somewhat from a more common use of the term, namely a virtual, and usually read-only, table synthesized from multiple tables (e.g. [Zloof 77]). A view in RP is limited to displaying and editing data from a single record file, and all views allow updates.

Each distinct use of the record file may dictate that a view be created to support it. A record file can have arbitrarily many views, and views are moved, copied, deleted, and have their properties displayed and modified in exactly the same way as other objects in Star. It is important to note here that no matter how many views are defined, the actual records are stored only once.

Views have both static and dynamic properties. If the record file is used in the same way frequently, the user may choose to optimize that application by defining a view with a sort order and View Filter that are permanently maintained via an index (see **4.6**). On the other hand, the user may also specify the view properties interactively, without the overhead of permanent indices.

4.2 Sorting

The sort order for a view specifies the order in which the records in that view appear. Each view may have its own sort order, with the only cost being that indexes (see below) must be constructed. Thus, each view can display the records in the order that makes the most sense for its application. The sort order may either be maintained permanently via an index, or created dynamically each time the view is opened.

4.3 View Filters

Each view may have a view filter, which specifies a permanent subsetting of the records in the file. If, for example, one particular use of the record file required that notices be sent to those employees who have children, then it might be helpful to define a view whose filter passes only those employees. The effect of this is that whenever this view is opened, only those employees are displayed. The view filter functions, in effect, as a permanent query on the record file for those subsets of records that the user knows he will be accessing again and again. (There is also another level of filter, called the Retrieval Filter, for more transient queries; this is explained below.)

4.4 Display Form

The third important attribute of a view is the display form. Any Star document may be used as the display form of a view, although normally forms whose field structure has some correspondence to that of the record file are used. By changing the display form of a view, the user is able to control the format with which the data is displayed. Although in practice, some forms might be used predominantly for reports, others for interactive querying, and still others for updates, there is nothing that requires this. All display forms can be used for both input *and* output.

4.5 Current View

Each record file has a current view, which is the view that was last selected by the user. It is maintained after the record file is closed and selected automatically by Star when the record file is opened. The current view is important in printing record files and in moving or copying one record file icon to another.

4.6 Indices

The sort order and view filter together define an index, which is maintained across all record updates. The more distinct views that are defined for a given record file, the more indices have to be updated as records are added, deleted, or revised. This is the standard retrieval vs. update tradeoff of data management. If the record file is relatively stable, then the user would likely want to capture as many of his frequent queries as possible in views, but if the record file were in a constant state of flux, having this many indices might impose too high a cost on updates.

The View Property Sheet contains a parameter called Save Index, which can be used to specify whether the index is to be permanently maintained, or created dynamically each time the view is opened and deleted when it is closed. This allows the user to make the tradeoff referred to above. For example, a view that is used only once a month for reporting might be defined with Save Index off; this would allow its *definition* to be permanently stored, but the view would not require any overhead to maintain when not in use.

4.7 Retrieval Filter

Many of the queries that will be made against a record file cannot be predicted in advance, and they are often of a one-time-only nature. For such queries, it may not be appropriate to pay the cost of creating an index. The Retrieval Filter provides a low-cost alternative for this sort of application. The Retrieval Filter has exactly the same appearance and operations as the View Filter. It is applied in addition to the View Filter; that is, it further restricts the set of records that are displayed in a view.

5. Record File Manipulations at the Icon Level

Most of the operations described so far (filtering, adding, deleting and modifying records, even defining record files and views) are performed within a record file window, i.e. an opened record file icon. Icon-level operations are also used in RP, in a way analogous to their use in other Star domains.

5.1 Record File to Printer

The normal way of printing something in Star is to select its icon and move or copy it to a printer icon. The current view is what is printed when a record file is moved or copied to a printer. Thus, the user chooses the report format desired by selecting the appropriate view. The task of making regular reports from a Star record file now becomes simply that of defining the appropriate view once, and then printing it as needed. During the printing process, the records for the current view are laded into the display form, producing either a single document including a table with one row per record (with a tabular form), or multiple copies of the document, with one document per record (a non-tabular form). Repetitive mail (form letters) may be generated by using the form letter as the display form for a record file of names and addresses.

5.2 Document or Folder to Record File

New records can be inserted into a record file by moving or copying a document or folder to the record file icon. In this case, each document is matched to the record file structure, as described above in Lading (**3.2.1**). If any fields in the document have names matching fields in the record file, a new record is created, and the contents of those fields are copied over. This process is repeated for each item in the folder. Documents which are not accepted for some reason (e.g. failure to meet format or range constraints) are copied into another special folder, called the Error Folder, for the user's subsequent examination and editing. Using this facility, records can be created as forms and added to the record file whenever it is convenient.

5.3 Record File to Record File

By moving or copying one record file icon to another, records can be added to the destination *en masse*. This facility also provides a form of reorganization: the same process of matching on field names that is performed between documents and record files is also done between record files. Thus, fields can be added to a record file by creating a new record file with the same fields plus the new ones, and moving the old record file icon to the new one. In this case, the new fields are left empty in the destination record file. Similarly, fields can be deleted, reordered, or have their types, formats, or ranges changed by creating a new record file with the desired characteristics. When a record file is moved or copied, the current view is the source of records. By setting the appropriate filter on some view and making it current, the user can transfer only a subset of the records to the destination file.

5.4 Record File to Other Icons

A record file is transferred to a file server by moving or copying its icon to a file drawer icon on the user's Desktop. By opening the file drawer icon, the user can select and move or copy the record file icon back to his Desktop. Folders may hold record files as wells as simple documents and other folders.

A record file is mailed by moving or copying it to an outbasket icon, just like a document. In this case, the entire record file, including the Forms Folder and all its display forms, is transferred to the recipients' inbaskets.

5.5 Make Document

The Make Document command in the View window menu creates a Star document on the Desktop (or folder full of them, in the case of a non-tabular form) corresponding to all the records in the Current View. Such a document can now be edited and annotated, merged with other document contents, filed, mailed, printed, etc.

6. Review and outlook

6.1 Overall Appraisal

In general, we believe Star's RP feature has fulfilled its design goals. In the first place, RP objects and actions co-exist with the rest of Star; there is no more necessity to switch contexts to perform data retrieval or update functions than to draw a picture or to send electronic mail. Further, users remain within the standard Star paradigm. Intuitions about the general nature and behavior of icons extend naturally to record files; they behave in corresponding fashion, and the functions they share with other Star objects are invoked with the same user actions, particularly in the use of the universal commands. Data entry and update follows directly on the text processing model, and query specification by filters demonstrates an extension of the What-you-see-is-what-you-get principle to a new and powerful application.

More particularly, access to Star's document production facilities offers benefits in several areas. Our experience has been that report formatting constitutes a significant burden on computer professionals (from DBMS implementors to computing center personnel). All of the power of Star's text world is made available for the definition of output from RP; what has been a tedious and error-prone task for programmers becomes a straightforward matter for the end user to specify, with a final product that offers unsurpassed visual quality.

The lading paradigm has proved powerful in designing applications and extensions. The progression from a simple forms-processing model of an office application to a more sophisticated RP environment is eased by the use of forms for record file definition and data entry. Future extensions, such as graphic idioms (e.g. bar- and pie-charts) driven from record file data, appear natural and straightforward.

The success of the attempt to encapsulate stylized user applications in the View is less complete. Our experience to date indicates there is a significant conceptual hurdle in the concept of the view. One difficulty involves terminology: naive users often equate the view with its display document. Some users have found it difficult to understand what might be an appropriate use of the view mechanism in their own applications. Once comprehended, it seems to be enthusiastically accepted and effectively used, but the lack of immediacy is troublesome. Further research on sources of user confusion and means of obviating it seems appropriate.

6.2 Particular Risks

For all the benefits of unification with the rest of Star (the text world in particular), it also entails two major risks: one is the well-known tendency for performance to vary inversely with generality, and the other arises from the organizational difficulties attendant on increasing the size of any project.

The threat of diminished performance is not absolute for several reasons. Consistency in the user interface need not preclude recognizing and taking advantage of appropriate special cases in the implementation. The incentive to make effective optimizations is, if anything, increased in a more general system. And a global approach to implementing a system promotes application of talents to areas where they will produce best results. But the problem is real, and requires careful attention, in Star in particular as well as in the world in general.

The organizational difficulties in dealing with a system as large as Star may also be ameliorated, but they have had a real impact. The design task was painfully extended by the requirement to maintain consistency with the rest of Star, and that consistency sometimes was bought at the price of an "obvious" solution regarded strictly within the context of RP. A trivial example concerns the fact that records are "filtered" in a query; it would have been much closer to common usage to speak of "selecting" them, but the conflicts that would have introduced with the rest of Star would have been intolerable. In a more serious vein, support for the RP functions described here has laid an additional (and heavy) burden on the implementors of Star's document facilities. There have been a number of painful choices to make in the distribution of limited resources.

6.3 Contemplated extensions

Facilities for combining data in multiple record files are an obvious extension. Several approaches present themselves, ranging from providing sufficient power in CUSP for users to specify joins themselves, up to providing a graphical editor for constructing expressions in some version of a relational calculus. The database issues are reasonably well understood; selecting a user model and finding implementation resources present more difficulties.

Another extension would make the view closer to what goes under that name in database terminology, a virtual relation constructed by an established query. Such a step might involve distributing views to users, while a Database Administrator reserves access to the real record file. Benefits accrue in security (users can see only the records and fields in their own view), more effective data sharing, and database administration (centralized allocation and backup become feasible, for instance). But the issue of updates in virtual data also arises. This is a problem both in the semantics of the database (see e.g. [Bancilhon 81]), and in presenting an intelligible user model of those semantics. The current design of RP is intended to allow compatible growth into such a scheme.

7. Acknowledgements

Dave Smith, Derry Kabcenell, Ralph Kimball, Charles Irby, Eric Harslem, and Jim Reiley, as well as the authors, made major contributions to the RP user interface. The implementors of Star RP were: Fred Bulah, Dan DeSantis, Eric Harslem, Derry Kabcenell, Paul Klose, Robert Levine, Dimitri Moursellas, Chi Nguyen, Charlene Nohara, Robert Purvy, and Jim Reiley.

8. References

[Bancilhon 81] Bancilhon, F., and Spyratos, N., "Update Semantics of Relational Views," *ACM Transactions on Database Systems* **6**, 4 (December 1981), 557 - 575

[DEC / Intel / Xerox 80] Digital Equipment Corp., Intel Corp., and Xerox Corp., "The Ethernet: A Local Area Network", Version 1.0, September 1980

[Ellis 80] Ellis, C., and Nutt, G. "Computer Science and Office Automation," *Computing Surveys* **12**,1 (March 1980), 27-60

[Embley 80] Embley, David "A Forms-Based Nonprocedural Programming System," Dept. of Computer Science, University of Nebraska, Lincoln, NE 68588

[Seybold 81] Seybold, Jonathon, "Xerox's 'Star'" *The Seybold Report* **10**,16 (April 27 1981) Seybold Publications, Inc., Box 644, Media, PA 19063

[Smith 82] Smith, Harslem, Irby and Kimball, "The Star User Interface: An Overview," *AFIPS National Computer Conference* **51** (1982), AFIPS Press, Arlington, VA 22209.

[Tsichritzis 80] Tsichritzis, D., "OFS: An Integrated Form Management System," *Proceedings of the 6th Conference on Very Large Data Bases*, Montreal (1980), 161 - 166

[Yao 81] Yao, S. B. and Luo, D. "Form Operation by Example: A Language for Office Information Processing," *Proceedings of the ACM SIGMOD Conference on Mangement of Data* (1981), 212-223.

[Zloof 77] Zloof, Moshé, "Query-By-Example: A Database Language," *IBM Systems Journal* **16** (Fall 1977) 324-343

[Zloof 81] Zloof, Moshé, "QBE/OBE: A Language for Office and Business Automation," *Computer* **14**,5 (May 1981) 13-22

Evaluation of Text Editors

Teresa L. Roberts
Xerox Systems Development Department
3333 Coyote Hill Road
Palo Alto, CA 94304

Thomas P. Moran
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

This paper presents a methodology for evaluating computer text editors from the viewpoint of their users—from novices learning the editor to dedicated experts who have mastered the editor. The dimensions which this methodology addresses are:

—*Time* to perform edit tasks by experts.
—*Errors* made by experts.
—*Learning* of basic edit tasks by novices.
—*Functionality* over all possible edit tasks.

The methodology is objective and thorough, yet easy to use. The criterion of *objectivity* implies that the evaluation scheme not be biased in favor of any particular eaitor's conceptual model—its way of representing text and operations on the text. In addition, data is gathered by observing people who are equally familiar with each system. *Thoroughness* implies that several different aspects of editor usage be considered. *Ease-of-use* means that the methodology is usable by editor designers, managers of word processing centers, or other non-psychologists who need this kind of information, but have limited time and equipment resources.

In this paper, we explain the methodology first, then give some interesting empirical results from applying it to several editors.

THE METHODOLOGY

The methodology is based on a taxonomy of 212 editing tasks which could be performed by a text editor. These tasks are specified in terms of their effect on a document, independent of any specific editor's conceptual model. The tasks cover:

—modifying the content of the document,
—altering the appearance of paragraphs and characters and the page layout,
—creating and modifying special kinds of text (such as tables),
—specifying locations and text in the document in various ways,
—programming automatic repetition of edits,
—displaying the document in various ways,
—printing, filing, and other miscellaneous tasks.

The functionality dimension of an editor is measured with respect to this taxonomy. However, comparisons between editors on the performance dimensions (time, errors, and learning) must be done on tasks which all editors can do. For this purpose, a set of 32 *core tasks* was identified.

The core tasks were chosen to be those tasks that most editors perform and that are most common in everyday work. Most of the core tasks are generated by crossing a set of basic text editing operations with a set of basic text entities. Thus, a core task consists of one of the operations (*insert*, *delete*, *replace*, *move*, *copy*, *transpose*, *split*, *merge*) applied to one of (or a string of) the text entities (*character*, *word*, *number*, *sentence*, *paragraph*, *line*, *section*). The core tasks also include *locating a place* in the online document which corresponds to a place in a hardcopy document (using the editor's simplest addressing mechanism), *locating a string of text* according to its contents, *displaying* a continuous segment of the document, *saving* and *retrieving* copies of the document, *printing*, and *creating* a new document.

Time. The speed at which normal text modification can be done is measured by observing expert users as they perform a set of *benchmark tasks* from the core tasks. There are 50 editing tasks in the benchmark, embedded in four documents: a short inter-office memo, two two-page reports, and one chapter from a philosophy book. The locations and complexities of the benchmark tasks are randomly distributed. The distribution emphasizes small tasks because those are most common in normal work and tasks involving boundary conditions in order to identify special cases, such as insertion at the beginning of a paragraph, which editors may treat awkwardly. Four

This is a minor modification of a paper which appeared in the *Proceedings of the Conference on Human Factors in Computer Systems*, Gaithersburg, Md., 15-17 March 1982. Reprinted with permission from the Association for Computing Machinery.

experts are tested separately on the benchmarks. They are chosen to represent a spectrum of the user community: at least one user must be *non-technical* (i.e., does not have a programming background) and at least one must be *technical* (i.e., is very familiar with programming). The evaluator measures the performance in the test sessions with a stopwatch, timing the error-free performance of the tasks (errors are discussed below), and noting whether or not all tasks are performed correctly. This method of measurement is used because of the requirement that the test be easy for anyone to run (not everyone has an instrumented editor or a videotape setup, but anyone can acquire a stopwatch). That is also the reason for the limited number of subjects. The benchmark tasks typically take 30 minutes of steady work to complete. The score which results from this part of the test is the average error-free time to perform each task (the error-free time is the elapsed time minus time spent making and correcting errors). The overall time score is the average score for the four experts.

Additional information about the speed of use of a text editor may be obtained by applying the theoretical Keystroke-Level Model [1] to the benchmark tasks. This model predicts editing time by counting the number of physical and mental operations required to perform a task and by assigning a standard time to each operation. The operations counted include typing, pointing with the mouse, homing on the keyboard, mentally preparing for a group of physical operations, and waiting for system responses. In the present methodology, the evaluator must predict what methods a user would employ to perform the benchmark tasks; then the model is used to predict the amount of time to execute those methods. Differences between the conditions under which the Keystroke-Level Model was validated and the conditions here (e.g., small typographic errors are included, not all subjects use the same methods, etc.) lead to expected differences between predicted performance and the results of the experiments above. However, in addition to being a prediction of the benchmark time, the model also serves as a theoretical standard of expert performance.

Errors. The error-proneness of the editor is measured by recording the amount of time the expert users spend making and correcting errors on the benchmark tasks. Only those errors which take more than a few seconds to correct are noted (which is the best that can be done with a stopwatch). Thus, the time taken by simple typographical errors is not counted. Actually, this does not hurt the error time estimate too much, since the total amount of time in these kinds of small errors is relatively small. In addition to timing errors made and corrected while the user is working on the benchmarks, the evaluator also notes the tasks incorrectly performed; at the end of the experiment the user is asked to go back and complete those tasks correctly. The time to redo these tasks is added to the error time. Thus, the error score consists of all this error time as a percentage of the error-free time. The overall error score is the average for the four expert users.

Learning. The ease of learning to perform basic text modifications on the editor is tested by teaching four novices (with no previous computer or word processing experience) to perform the core tasks. The learning tests are performed in a one-on-one situation, i.e., by individually teaching each novice the editor. The evaluator orally teaches the novice how to do the core tasks in the editor, and the subject practices the tasks on the system. The methodology specifies the order in which to teach the tasks, but it is up to the evaluator to determine which specific editor commands to teach. Although all the teaching is oral, the evaluator supplies the novice with a one-page summary sheet listing all commands, so that the training is not hung up because of simple memory difficulties. After a set of tasks is taught, the novice is given a quiz, consisting of a document marked with changes to be made. Only a sample of possible tasks appears on each quiz, and not all tasks on the quiz have necessarily been taught up to that point. This allows for the novice to figure out, if possible, how to do tasks which haven't explicitly been taught. Referring to the summary sheet is permitted, but discouraged. The novice performs all of the tasks that he or she knows how to do, after which s/he is invited to take a short break if s/he wants it. Then another teaching period begins. In all, there are five training-plus-quiz cycles to teach all of the core tasks. Learning is evaluated by scoring the number of different tasks the subject has shown the ability to perform on the quizzes. The learning score is the total number of different tasks learned divided by the amount of time taken for the experiment, that is, the average time it takes to learn a task. The overall learning score is the average learning time for the four novices.

Functionality. The range of functionality available in the editor is tested by a checklist of tasks covering the full task taxonomy. Determining whether a task can be done or not with a given system isn't as trivial as it seems at first glance. Almost any task can be performed on almost any system, given enough effort. Consequently, the editor gets full credit for a task only if the task can be done efficiently with the system. It gets half credit if the task can be done clumsily (where clumsiness has several aspects: repetitiousness, requiring excessive text typing, limitations in the values of parameters to the task, interference with other functions, or a requirement of substantial planning by the user). The editor gets no credit for a task if either it can't be done at all (like use of italic typefaces on a system made for a line printer) or if doing the task requires as much work as retyping all affected text (such as manually inserting a heading on every page). The functionality checklist is filled out by a very experienced user of the editor, who may refer to a reference manual to ensure accuracy. The overall functionality score is the percentage of the total number of tasks that the editor can do. This percentage may be broken down by subclasses of tasks to show the strengths and weakness of the editor.

EMPIRICAL RESULTS

This methodology has been used to evaluate a diverse set of nine text editors: TECO [5], WYLBUR [9], a WANG word processor [10], NLS [3,4], EMACS [8], STAR [11], BRAVO [7], BRAVOX [6], and GYPSY (the last three editors are experimental systems developed within Xerox). The first two of these editors are made for teletype-like terminals, the rest are for display-based terminals. The intended users of these editors range from devoted system hackers to publishers and secretaries who have had little or no contact with computers. The results of these evaluations may be used in several ways: (1) as a comparison of the editors, (2) as a validation of the evaluation methodology itself, and (3) as general behavioral data on user performance.

Comparison of Editors. An editor's evaluation is a multi-dimensional score—a four-tuple of numbers, one from each performance dimension. A summary of the overall evaluation scores for the nine editors is given in Figure 1. Differences were found between the editors on all the evaluation dimensions (although only large differences were statistically significant, because of the large individual differences between the users tested). No editor was superior on all dimensions, indicating that tradeoffs must be made in deciding which editor is most appropriate for a given application.

	Evaluation Dimensions			
	Time	Errors	Learning	Functionality
Editor	*M* ± *CV*	*M* ± *CV*	*M* ± *CV*	
	(sec/task)	(% Time)	(min/task)	(% tasks)
TECO	49 ± .17	15% ± .70	19.5 ± .29	39%
WYLBUR	42 ± .15	18% ± .85	8.2 ± .24	42%
EMACS	37 ± .15	6% ± 1.2	6.6 ± .22	49%
NLS	29 ± .15	22% ± .71	7.7 ± .26	77%
BRAVOX	29 ± .29	8% ± 1.0	5.4 ± .08	70%
WANG	26 ± .21	11% ± 1.1	6.2 ± .45	50%
BRAVO	26 ± .32	8% ± .75	7.3 ± .14	59%
STAR	21 ± .16	19% ± .51	6.2 ± .42	62%
GYPSY	19 ± .11	4% ± 2.1	4.3 ± .26	37%
M(*M*) *M*(*CV*)	31 .19	12% .99	7.9 .26	54%
CV(*M*)	.31	.49	.53	.25

Figure 1. Overall evaluation scores for nine text-editors.
The Time score is the average error-free time per benchmark task. The Error score is the average time, as a percentage of the error-free, that time experts spend making and correcting errors. The Learning score is the average time to learn a core task. The Functionality score is the percentage of the tasks in the task taxonomy that can be accomplished with the editor. The *Coefficient of Variation* (*CV*) = *Standard Deviation / Mean* is a normalized measure of variability. The *CV*'s on the individual scores indicate the amount of between-user variability. The *M*(*CV*)'s give the mean between-user variability on each dimension, and the *CV*(*M*)'s give the mean between-editor variance on each dimension. The evaluations for TECO. NLS. WANG, and WYLBUR are from Roberts [2].

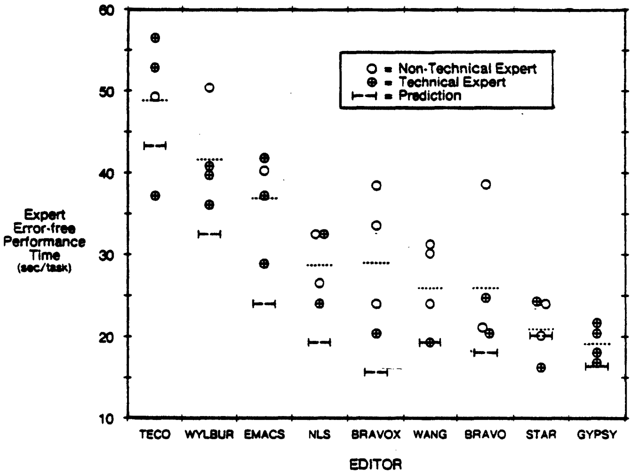


Figure 2. Time scores for individual expert users.

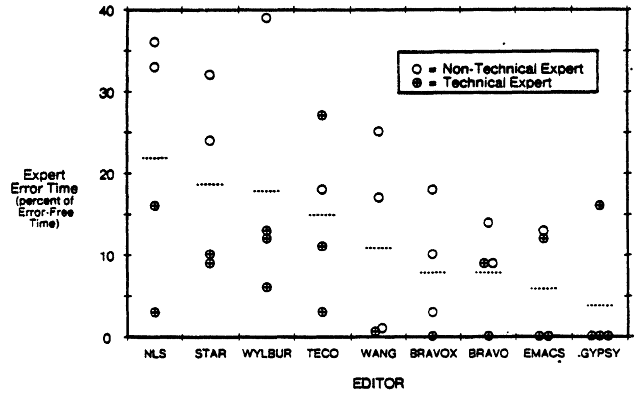


Figure 3. Error scores for individual expert users.

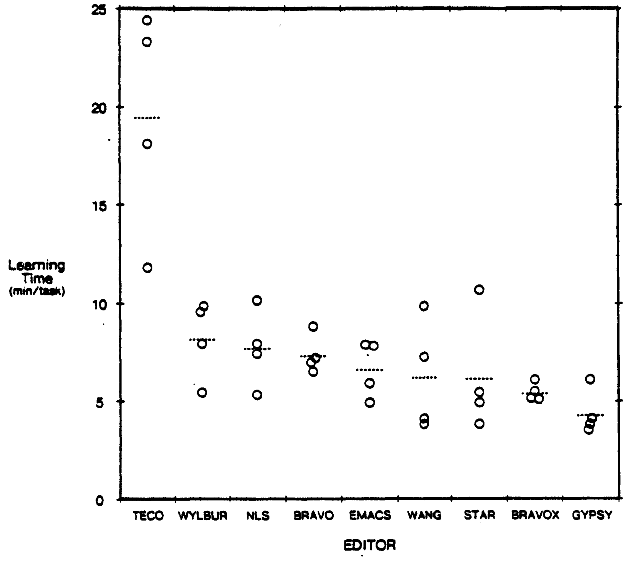


Figure 4. Learning scores for individual novice learners.

Time. The summary time data in Figure 1 is expanded in Figure 2 to show individual users' scores. These results show that TECO, WYLBUR, and EMACS are the slowest editors and that GYPSY and STAR are the fastest. Most of the display-based systems were used at about twice the speed of the non-display systems.

The times predicted by the Keystroke-Level Model, also shown in Figure 2, can be seen to be about 70% of the average error-free experimental time. But the data for individual users show that for most editors, one user comes very close to the "standard expert" performance that the Keystroke-Level Model predicts.

Errors. Individual users' error scores are shown in Figure 3. This data finds a factor of five difference between the best and the worst editors, but even so these differences are small compared to the differences between users. No conclusions about the error-proneness of editors can be drawn.

Learning. The overall learning scores are shown in Figure 1, and the scores for individual novices are shown in Figure 4. Large differences were found in the learnability of the different editors. TECO turned out to be an outlier, taking over twice as long to learn as the next editor (WYLBUR). The rest of the editors lie along a smooth progression which covers another factor of two in learning rate, with GYPSY being four times faster to learn than TECO.

Learning/Speed Tradeoff. The conventional wisdom is that there is a tradeoff between the speed of learning and the speed of expert use of a system. Combining the learning results with the time results, we see exactly the opposite. The data from this study shows a high positive correlation ($R = .79$) between the time and learning scores. The major difference was between the set of display editors and the set of non-display editors. Display editors are better for both expert time and novice learning.

Functionality. The functionality results showed that most of the editors could perform roughly half of the tasks in the task taxonomy. The reason for this was that, in general, each system had its areas of strength and weakness. In order to show this, the data are broken down by task categories in Figure 5. EMACS shows up well in programming capability, while NLS and BRAVOX shine in formatting and layout. Because the numbers of tasks in the taxonomy was more weighted toward text layout than editor programming, the document-oriented editors generally showed up as being somewhat better than EMACS. But NLS, which tries to cover all needs, was the clear winner in overall functionality.

Evaluation of the Methodology. The results above show that diverse editors can indeed be evaluated and compared. As a whole, the evaluation methodology seems to successfully provide an objective, multi-dimensional picture of text editors. This methodology is also quite practical. For an experienced evaluator, about one week of time is required to evaluate a new editor; thus it is quite accessible to a system designer or a potential buyer.

There are still many areas of editor use which are not covered by the methodology, for instance the needs of occasional users. In addition, use of the methodology has pointed out areas, such as error-proneness, where more reliable measures are needed to differentiate specific editors. Any further work will have to take into account the effect of large differences among the users.

Behavioral Results. The data gathered from these evaluations are also interesting for what they tell us about user behavior. It provides a pool of data on overall levels of user performance. We see from this data that typical core editing tasks require on the order of 30 seconds for experts to perform, and we see that a period of about two hours of one-on-one training is enough to get users started with the core tasks in most editors. Such data should be of interest to researchers in office productivity, e.g., to measure the cost-effectiveness of word processing.

Our data also provides some insight into individual differences in performance among users:

(1) By far the greatest individual differences are found in error rate (ranging from 0% to 39%), which reflects a wide variation in the style of using text editors.

(2) Comparing speed of expert, error-free use with error rate shows that it is the slower users who make more errors. So while there may be a speed/error tradeoff that an individual user can make, we don't see such a tradeoff between people.

(3) There is only moderate variation among experts in speed of editing—about a factor of 1.5 to 2 between the fastest and slowest user within each editor.

(4) A somewhat surprising result is that there is not much more variation among novice learners than among experts, i.e., about the same range of differences between the fastest and slowest learners.

Conclusion. This methodology has proven itself to be an effective tool for the empirical evaluation of text editors, helping us understand how people adapt to them as well as providing us with much-needed feedback on how they actually perform.

Task Category	# of Tasks	Editor									All Editors
		NLS	BRAVOX	STAR	BRAVO	EMACS	WANG	WYLBUR	TECO	GYPSY	*M±CV*
TOTAL	*212*	77	70	62	59	49	48	42	39	37	54±.25
Modification											
Content	*66*	94	89	93	90	74	87	63	88	80	84±.13
Text Layout	*19*	89	71	66	71	37	37	26	3	26	47±.56
Page Layout	*25*	74	62	56	40	2	34	6	4	4	31±.85
Characters	*21*	43	76	57	62	14	38	21	0	17	36±.66
Special Purpose	*16*	53	59	50	22	0	34	16	3	0	26±.84
Addressing	*22*	68	36	30	30	61	16	34	25	18	35±.48
Control	*23*	56	37	24	20	89	24	61	48	9	41±.58
Display	*8*	94	94	63	69	81	19	62	38	50	63±.42
Misc.	*12*	100	88	100	71	46	71	71	25	42	68±.38

Figure 5. Functionality sub-scores for eight text-editors.
Each functionality sub-score is given as a percentage of the total number of tasks in its task category (the italic numbers in the "# of Tasks" column). The numbers in the "All Editors" column summarize how well the task categories are handled by the whole collection of editors.

Acknowledgement. We would like to thank Betsey Summers for organizing and running many of these evaluation studies and for helping us analyze the data.

REFERENCES

[1] Card, S. K., Moran, T. P., and Newell A. The Keystroke-Level Model for user performance time with interactive systems. *Communications of the ACM*, 1980, *23*, 396-410.

[2] Roberts, T. L. *Evaluation of Computer Text Editors.* Ph.D. dissertation, Department of Computer Science, Stanford University, 1980. Available as Report AAD 80-11699 from University Microfilms, Ann Arbor, Michigan.

Editor Documentation References

[3] Augmentation Research Center. *NLS-8 Command Summary.* Menlo Park, California: Stanford Research Institute, May 1975.

[4] Augmentation Research Center. *NLS-8 Glossary.* Menlo Park, California: Stanford Research Institute, July 1975.

[5] Bolt, Beranek, and Newman, Inc. *TENEX Text Editor and Corrector* (Manual DEC10-NGZEB-D). Cambridge, Massachusetts: Author, 1973.

[6] Garcia, Karla. *Xerox Document System Reference Manual.* Palo Alto, California: Xerox Office Products Division, 1980.

[7] Palo Alto Research Center. *Alto User's Handbook.* Palo Alto, California: Xerox PARC, September 1979.

[8] Stallman, R. M. *EMACS Manual for ITS Users* MIT, AI Lab Memo 554, 1980.

[9] Stanford Center for Information Processing. *Wylbur/370 The Stanford Timesharing System Reference Manual, 3rd ed.* Stanford, California: Stanford University, November 1975.

[10] Wang Laboratories, Inc. *Wang Word Processor Operator's Guide, 3rd release.* Lowell, Mass., 1978.

[11] Xerox Corporation. *8010 Star Information System Reference Guide.* Dallas, Texas, 1981.

The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment

Derek C. Oppen[1] and Yogen K. Dalal[2]
Xerox Office Systems Division

April, 1983

Authors' Current Addresses:
[1]495 Arbor Road, Menlo Park, California 94025
[2] Metaphor Computer Systems, 2500 Garcia Avenue, Mountain View, California 94043

Abstract: We consider the problem of naming and locating objects in a distributed environment, and describe the *clearinghouse*, a decentralized agent for supporting the naming of these "network-visible" objects. The objects "known" to the clearinghouse are of many types, and include workstations, file servers, print servers, mail servers, clearinghouse servers, and human user. All objects known to the clearinghouse are named using the same convention, and the clearinghouse provides information about objects in a uniform fashion, regardless of their type. The clearinghouse also supports aliases.

The clearinghouse *binds* a name to a set of *properties* of various types. For instance, the name of a user may be associated with the location of his local workstation, mailbox, and non-location information such as password and comments.

The clearinghouse is decentralized and replicated. That is, instead of one *global* clearinghouse server, there are many *local* clearinghouse servers, each storing a copy of a portion of the global database. The totality of services supplied by these clearinghouse servers we call "the clearinghouse." Decentralization and replication increase efficiency, security, and reliability.

A request to the clearinghouse to bind a name to its set of properties may originate anywhere in the system and be directed to any clearinghouse server. A clearinghouse client need not be concerned with the question of which clearinghouse server actually contains the binding—the clearinghouse stub in the client in conjunction with distributed clearinghouse servers automatically find the mapping if it exists. Updates to the various copies of a mapping may occur asynchronously and be interleaved with requests for bindings of names to properties; updates to the various copies are not treated as indivisible transactions. Any resulting inconsistency between the various copies is only transient: the clearinghouse automatically arbitrates between conflicting updates to restore consistency.

CR Categories and Subject Descriptors: C.2.3 [**Computer-Communication Networks**]: Network Operations—*network management*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databaseses, network operating systems*; H.2.1 [**Database Management**]: Logical Design—*data models*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrievel—*search process*; H.4.3 [**Information Systems Applications**]: Communications Applications—*electronic mail.*

General Term: Design

Additional Key Words and Phrases: clearinghouse, names, locations, binding, network-visible objects, internetwork.

1. Introduction

We introduce the subject matter of this paper by considering the role of the information operator, the "White Pages" and the "Yellow Pages" in the telephone system.

Consider how we telephone a friend. First we find the person's telephone number, and then we dial the number. The fact that we consider these to be "steps" rather than "problems" is eloquent testimony to the success of the telephone system. But how do the two steps compare? The second—making the connection once we have the telephone number—is certainly the more mechanical and more predictable, from the user's point of view, and the more automated, from the telephone system's point of view. The first step—finding someone's telephone number given his or her name—is less automatic, less straightforward, and less reliable. We have to use the telephone system's information system, which we call the *telephone clearinghouse*. If the person lives locally, we telephone "information" or look up the telephone number in the White Pages. If the person's telephone is non-local, we telephone "information" for the appropriate city. We always have to treat whatever information we get from the telephone clearinghouse with a certain amount of suspicion, and treat it as a "hint." We have to accept the possibility that we have been given an incorrect number, perhaps because the person we wish to call has just moved. We are conditioned to this and automatically begin calls with "Is this ...?" to validate the hint.

In other words, although making the connection once we have the correct telephone number offers few surprises, *finding* the telephone number may be a time-consuming and frustrating task. The electrical and mechanical aspects of the telephone system have become so sophisticated that we can easily telephone almost anywhere in the world. The telephone clearinghouse remains unpredictable, and may require considerable interaction between us, as clients, and the information operator. As a result we all maintain our own personal database of telephone numbers (generally a combination of memory, little black books, and pieces of scrap paper) and rely on the telephone system's database only when necessary.

The telephone clearinghouse provides another service: the Yellow Pages. The Yellow Pages map generic names of services (such as "Automobile Dealers") into the names, addresses and telephone numbers of providers of these services.

In brief, there are three ways for objects in the telephone system to be found: by name, by number, or by subject. The telephone system prefers to use numbers, but its clients prefer subscriber and subject names. The telephone clearinghouse provides a means for mapping between these various ways of referring to objects in the telephone world.

We move from the telephone system to distributed systems and, in particular, to interconnections of local networks of computers. Suppose that we want to send a file to our local printer or to someone else's workstation, or we want to mail a message to someone elsewhere in the internetwork. The two steps we have to take remain the same: finding out where the printer, workstation, or mail server is (that is, what its network address is), and then using this network address to access it. The internetwork knows how to use a network address to route a *packet* to the appropriate machine in the internetwork. So the second step—accessing an object once we know its network address—has well-known solutions. It is the first step—finding the address of a distributed object given its name—that we consider here.

Why do we need names at all? Why not just refer to an object by its address? Why not just directly use the network address of our local file server, mail server, or printer? The reasons are much like those for using names in the telephone system or in a file system. The first is that locations are unappealingly unintuitive; we do not want to refer to our local printer by its network address 5#346#6745 any more than we want to refer to a colleague as 415-494-4763. The second is that distributed objects change locations much more frequently than they change names. We want a level of indirection between us and the object we wish to access, and that level of indirection is given by a name. (See also [Shoch 1978], [Abraham and Dalal 1980], [Saltzer 1982], [Pickens, Feinler, and Mathis 1979], and [Solomon, Landweber, Neuhengen 1982].)

When a network object is referred to by name, the name must be *bound* to the address of the object. The binding technique used greatly influences the ability of the system to react to changes in the environment. If client software binds names to addresses *statically* (for instance, if software supporting printing has the addresses of the print servers stored in it), the software must be updated if the environment changes (for instance, if new print servers are added or old servers are moved or removed). If client software binds names to addresses *dynamically*, the system reacts much more gracefully to changes in the environment (they are not necessarily even noticed by the client).

The problems we address in this paper are therefore the related problems of how to name objects in a distributed computer environment, how to find objects given their names, and how to find objects given their generic names. In other words, how to create an environment similar to the telephone system's with its notions of names, telephone numbers, White Pages and Yellow Pages. Before leaving this introduction, we see how the telephone clearinghouse works.

1.1. Locating Telephone Subscribers

The database used by the telephone clearinghouse—the "telephone book"—is highly decentralized. The decentralization is based on physical locality: each telephone book covers a specific part of the country. It is up to the the telephone clearinghouse client to know which telephone book is to be used.

This decentralization is partly motivated by size; there are just too many telephones for one common database. It is also motivated by the fact that people's names are ambiguous. Many people may share the same name, and corresponding to one name may be many telephone numbers. Decentralizing the telephone clearinghouse is one way to provide additional information for disambiguating a reference to a person—there may be many John Smiths in the country but hopefully not all are living in the same city as the John Smith whose telephone number we want. However, even by partitioning the database by city and by using other information such as street address, the telephone clearinghouse still may be confronted with a name for which it has several telephone numbers. When this happens it becomes the client's responsibility to disambiguate the reference, perhaps by trying each telephone number until he finds the one he wants. The telephone clearinghouse cannot assume that names are unambiguous, and leaves it to the client to resolve ambiguities.

1.2. Creating, Deleting and Changing Telephone Numbers

Responsibility for *initiating* updates rests with the telephone users. However, the actual updating of the database is done by the telephone company. Users of the telephone clearinghouse have read-only access to the clearinghouse's database. Allocation of telephone numbers is the responsibility of the telephone company; the telephone company provides a *naming authority* to allocate telephone numbers.

The updating process deserves scrutiny because it helps determine the accuracy of the information given out by the telephone clearinghouse. The information is not necessarily "correct." Offline "telephone books" are updated periodically and so do not contain recent updates. Even the online telephone directory used by information operators may give "old" information. One reason for this is that asking the operator for a telephone number and using it some time later to make a call are not treated by the telephone system as an indivisible operation: the directory may be updated between the two events. Another reason is that physically changing a telephone number and updating the database are asynchronous operations.

The partitioning of the telephone clearinghouse's database is not strict. The database is a replicated database. Copies of a directory may appear in different versions, and telephone directories for different cities may overlap in the telephone numbers they cover. Since the updating process is asynchronous, the database used by the telephone company may not be internally consistent.

The effect of this—information given out by the telephone clearinghouse does not necessarily reflect all existing updates—is that the information provided by the telephone clearinghouse can only be used as a *hint*. The user must accept the possibility that he is dialing a wrong number, and *validate* the hint by checking in some way that he has reached the right person. However, the telephone company does provide some mechanisms for helping a user who is relying on an out-of-date directory, memory, or little black book. For instance, if a person moves to another city, his old telephone number is not reassigned to another user for some time, and during that period callers of his old number are either referred to his new number, or are less informatively told that they have reached an out-of-service number.

1.3. Creating, Deleting and Changing Subscriber Names

What we said above about updating telephone numbers generally applies as well to updating names, with one exception. The *choice* of name appearing in the telephone clearinghouse database rests with the holder of the telephone being named, and only the holder can request an update. (That is, you are permitted to choose under what name you will appear in the telephone directory, even if the name is ambiguous.) This raises an interesting issue, that of nicknames, abbreviations and aliases. The above does not mean that we, as users of the telephone system, cannot choose our own name for you (a *nickname*), but only that the telephone company will not maintain the mapping of my name for you into your telephone number—it will only maintain the mapping of *your* name for yourself into your telephone number. We may have my own "little black book" containing our own relativized version of the telephone clearinghouse, but the telephone company does not try to maintain its accuracy. Similarly, the telephone clearinghouse does not necessarily respond to abbreviations of names. And, finally, the telephone clearinghouse will handle aliases only if they are entered in its database. That is, the telephone clearinghouse allows names to be non-unique: a person may have more than one name.

1.4. Passing Subscriber Names and Telephone Numbers

Giving someone else a telephone number cannot raise problems because telephone numbers are unambiguous. (Of course, the telephone number may be incorrect by the time that person uses it.)

Giving a name to someone else is trickier since names are ambiguous. For instance, because the telephone clearinghouse database is decentralized, giving a name to an information operator in one part of the country may elicit a different response from giving it to one in another part of the country. In the telephone clearinghouse, names are context-dependent. You can ensure that the person to whom you are giving a name will get exactly the same response only if you specify the appropriate telephone clearinghouse as well.

2. Naming Distributed Objects

With this background, we return to the problem of designing a distributed system clearinghouse. A central question in designing such a clearinghouse is how to name the objects known to the clearinghouse.

2.1. Naming Conventions

A *naming convention* describes how *clients* of the naming convention refer to the *objects* named using the convention. The set of clients may overlap with the set of named objects; for instance, people are both clients of, and objects named using the common firstname-middlename-surname naming convention.

Our basic model for describing naming conventions is a directed graph with *nodes* and *edges*. Nodes and edges may be labelled. If node *u* has edge labelled *i* leading from it, then *u[i]* denotes the node at the end of the edge. (Edges leading from any node must be unambiguously labelled.)

We assume that each named object and each client is represented by exactly one node in the graph. With these assumptions, we need not distinguish in the rest of this section between nodes in the name graph, named objects, and clients of the naming system, and our problem becomes: what is the name of one node (a named object) relative to another (a client)? There are two fundamental naming conventions, each of which we now describe.

2.2. Absolute Naming

Under the absolute naming convention, the graph has only unlabelled nodes. There is a distinguished node called the *directory* or *root node*. There is exactly one edge from the directory node to each other node in the graph; each such edge is uniquely and unambiguously labelled. There are no other edges in the graph. The name of a node is the label of the edge leading from the directory node to this node. This is defines what is usually meant by "choosing names from a flat name space." One obvious example of names using absolute naming conventions are Social Security numbers.

2.3. Relative Naming

Under the relative naming convention, the graph has unlabelled nodes but labelled edges. There is either zero or one uniquely-labelled edge from any node to any other. If there is an edge labelled i from u to v, then the *distinguished name* of v *relative to* u is i. Here, u is the client and v the named object. Names are ambiguous—a relative name is unambiguous only if qualified by some *source* node, the client node. Without additional disambiguating information, people's names are relative. One person's use of the name "John Smith" may well differ from another's.

2.4. Comparison of the Absolute and Relative Naming Conventions

Locating Named Objects. One important role of the clearinghouse is to maintain the mapping *LookUp* from names to objects. If i is the name of an object, then *LookUp*(i) is that object. Under the relative naming convention, *LookUp* is relative to each client node. That is, if the name of an object v *relative to* u is i, then $LookUp_u(i)$ is v. Under the absolute naming convention, *LookUp* is relative to the whole graph. That is, if the name of an object v is i, then *LookUp*(i) is v; we do not have to qualify *LookUp* with the source node. Thus, the database required by the absolute convention may be smaller than under the relative convention (where the number of names is on the order of the square of the number of nodes). However, since the relative convention does not require that every node to directly name every other node, the domain of each *LookUp* under the relative convention will typically be much smaller than the domain for *LookUp* under the absolute convention.

The relative convention encourages decentralization, since the mapping from names to objects is relative to each node. The absolute convention encourages centralization, since there is only one mapping for the whole system. Thus the relative convention allows more efficient implementation of the *LookUp* function. Of course, one can use efficient methods such as binary search or hashing with either convention, but these make use only of *syntactic* information in names, not *semantic* information.

Changing Locations or Names. The main considerations here are the size and degree of centralization of the databases. Consider, for instance, the allocation of names. The absolute naming convention requires a centralized naming authority, allocating names for the whole graph. The relative naming convention permits decentralized naming authorities, one for each node. The local data base handled by the naming authority under the relative convention will typically be much smaller than the global data base handled by the naming authority under the absolute convention.

Passing Names and Locations. A major advantage of the absolute naming convention is that there is a common way for clients to refer to named objects. It is possible for any client to hand any other client the name of any object in the environment and be guaranteed that the name will mean the same thing to the second client (that is, refer to the same object). This is not the case with the relative addressing convention; if u and v are nodes, $u[i]$ need not equal $v[i]$. Under the relative naming convention, the first client must give the second client the name of the object *relative to the second client*. In practice, this means that the first client has to understand how the second client names objects. This suggests excessive decentralization; it requires too much coordination when objects are to be shared or passed.

2.5. Hierarchical Naming

Neither the absolute nor the relative naming convention is obviously superior to the other. We can do better by adding another layer of structure to the basic naming model.

We partition the graph into subgraphs, consisting of subsets of the set of nodes. We assume that each node is in exactly one subgraph. The *distinguished name* of a node is *nodename:subgraphname* where *subgraphname* is the name of its containing subgraph and *nodename* is the name of the node in that subgraph. This definition is well-defined only if names are unambiguous within a subgraph; the absolute naming convention must be used within a subgraph. That is, within any subgraph, no two nodes can have the same name. Two different nodes may have names *A:B* and *A:C* however; names need be unambiguous only *within a subgraph*.

The name of a node consists of both its name within a subgraph and the name of the subgraph. As mentioned, the absolute naming convention must be used for naming nodes within any subgraph. Subgraphs are named using either the relative or the absolute naming convention.

If the absolute naming convention is used, each distinct subgraph has an unambiguous name. Since the absolute naming convention is also used for naming nodes within each subgraph, it follows that nodes have unambiguous distinguished names. Telephone numbers such as 494-4763 fit into this two-level absolute naming hierarchy. The local exchange is uniquely and unambiguously determined (within each area code) by the exchange number 494; within exchange 494, exactly one telephone has number 4763.

If the relative naming convention is used, each distinct subgraph has an unambiguous distinguished name relative to each other subgraph. And, since we are using the absolute naming convention within subgraphs, it follows that each node has an unambiguous distinguished name relative to each source. An example of this is the interface between the Xerox Grapevine mail transport mechanism [Birrell, Levin, Needham and Schroeder 1982] and the Arpanet mail transport system. A name may be *Jones.PA* within Xerox but *Jones@MAXC* outside—the subgraph name has changed.

In either case, the advantages of using a hierarchy is clear: absolute naming can be used without barring decentralization. A partitioned name suggests the search path to the object being named and encourages a decentralized naming authority.

One can imagine a hierarchy of graphs with corresponding names of the form $i_1:i_2:...:i_k$. Examples include telephone numbers fully expanded to include country and area codes (a four-level hierarchy), or network addresses (a three-level hierarchy of network number, host number, socket number), or booknaming conventions such as the Dewey Decimal System.

For the reasons cited above, the usual distinction made between "flat" and "hierarchical" is somewhat misleading. The distinctions should be "flat" versus "relative" and "hierarchical" versus "non-hierarchical."

2.6. Abbreviations

The notion of *abbreviation* arises naturally with hierarchical naming. Within subgraph *B*, the name *A:B* can be abbreviated to *A* without ambiguity, given the convention that abbreviations are expanded to include the name of the graph in which the client node exists. Abbreviation is a relative notion. (See, for example, [Daley and Neumann 1965] for another approach to abbreviations.)

2.7. Combining Networks

One major advantage of the hierarchical superstructure that we have not considered before, and which is independent of the absolute versus relative naming question, concerns combining networks. One feature that any clearinghouse should be able to handle gracefully is joining its database with the database of another clearinghouse, an event that happens when their respective networks are joined. For instance, consider the telephone model. When the various local telephone systems in North America combined, they did so by adding a superstructure above their existing numbering system, consisting of area codes. Area codes are the names of graphs encompassing various collections of local exchanges.

Adding new layers to names is one obvious way to combine networks. The major advantage is that if a name is unambiguous within one network then it is still unambiguous with its network name as prefix, even if the name also appears in some other network (because the latter name is prefixed by the name of *that* network). The major disadvantages are that the software or hardware has to be modified to admit the new level of naming, and that a centralized naming authority is needed to choose names for the new layer.

The alternative to adding a new layer is expanding the existing topmost layer. For instance, the North American area code numbering system is sufficiently flexible that another area code can be added if necessary. The advantage of this is that less change is required to existing software and hardware. The disadvantage is that the interaction with the centralized naming authority, to ensure that the new area code is unambiguous, is more intimate than in the case of adding a new layer.

2.8. Levels of Hierarchy

If one chooses to use a hierarchical naming convention, an obvious question is the following: should we agree on a constant number of levels (such as two levels in the Arpanet mailing system or four in the telephone system) or an arbitrary number of levels? If a name is a sequence of the form *i1:i2:...:ik*, should *k* be constant or arbitrary? There are pros and cons to either scheme. The advantage of the *arbitrary* scheme is that the naming system may evolve (acquire new levels as a result of combining networks) very easily. That is, if we have a network now with names of the form *A:B*, and combine this network (let us call it network *C*) with another network, then we can just change all our names to names of the form *A:B:C* without changing any of the algorithms manipulating names. Allowing arbitrary numbers of levels clearly has an advantage. It also has several non-trivial disadvantages. First, all software must be able to handle an arbitrary number of levels, so software manipulating names will tend to be more complicated than in the *constant* level scheme. Second, abbreviations become very difficult: does *A:B* mean exactly that (an object with a two-level name) or is it an abbreviation for some name *A:B:C*? The disadvantage with the *constant* scheme is that one has to choose a number, and if we later add new levels, we have to do considerably more work.

2.9. Aliases

Our basic model allows each node to have exactly one name under the absolute naming convention, and exactly one name relative to any other node under the relative naming convention. An obvious extension to this model is to allow *aliases* or alternative names for nodes. To do this, we define an equivalence relation on names; if two names are in the same equivalence class, they are names of the same node. Under the relative naming convention, there is one equivalence relation defined on names for each client node in the graph. Under the absolute naming convention, there is only one equivalence relation for the whole graph. Each equivalence class has a distinguished member, and this we designate the *distinguished name* of the node.

The notion of aliasing is easily confused with the notion of relative naming, since each introduces multiple names for objects. The difference lies in the distinction between ambiguity and non-uniqueness. Under the relative naming convention, a name can be *ambiguous* in that it can be the name of more than one node (relative to different source nodes). Under the absolute naming convention, names are unambiguous. In either case, without aliasing, names are *unique*: if a node knows another node by name, it knows that node by exactly one name. With aliasing, names are non-unique; one node may know another by several names. Another way of expressing the difference is to consider the mapping from names to nodes. Without aliasing, the mapping is either one-to-one (under the absolute naming convention: each object has exactly one name and no two objects have the same name) or one-to-many (under the relative naming convention: each object has exactly one name relative to any other, but many nodes may have the same name). With aliasing, the mappings become many-to-one or many-to-many.

3. Clearinghouse Naming Convention

We now describe the naming system supported by our clearinghouse. Recall first that we have a very general notion of the objects being named: an object is anything that has a name known to the clearinghouse and the intuitive property of "network visibility." We shall give some concrete examples in the following sections.

Objects are named in a uniform fashion. We use the same naming convention for every object, regardless of whether it is a user, a workstation, a server, a distribution list or whatever.

A *name* is a non-null character string of the form *<substring1>:<substring2>:<substring3>*, where *substring1* denotes the *localname*, *substring2* the *domain*, and *substring3* the *organization*. Thus names are of the form *L:D:O* where *L* is the localname, *D* the domain and *O* the organization. None of the substrings may contain occurrences of " " or "*" (the reason for the latter exclusion is given later).

Each object has a *distinguished name*. Distinguished names are absolute; no two objects may have the same distinguished name. In addition to its distinguished name, an object may have one or more aliases. Aliases are also absolute; no two objects may have the same alias. A name is either a distinguished name or an alias, but not both.

We have thus divided the world of objects into organizations, and subdivided organizations into domains: a three-level hierarchy. An object is *in organization O* if it has a name of the form

<anything>: <anything>:O. An object is *in domain D in organization O* or *in D:O* if it has a name of the form *<anything>:D:O*.

This division into organizations and, within them, domains is a logical rather than physical division. An organization will typically be a corporate entity such as Xerox Corporation. The names of all objects within Xerox will be of the form *<anything>: <anything>:Xerox*. Xerox will choose domain names to reflect administrative, geographical, functional or other divisions. Very large corporations may choose to use several organization names if their name space is very, very large. In any case, the fact that two addressable objects have names in the same domain or organization does not necessarily imply in any way that they are physically close.

3.1. Rationale

We use a uniform naming convention for all objects, regardless of their type.

Objects known to the clearinghouse have absolute distinguished names and aliases. Thus we favor an absolute naming convention over a relative naming convention. Most systems (including most mail transport systems) have opted for a relative naming convention. However, the advantages of an absolute convention are so clear that we are willing to put up with the burden of some centralization. By choosing the naming convention carefully, we can reduce the pain of this centralization to an acceptable level.

Names are hierarchical. We rejected a non-hierarchical system because, among their other advantages, hierarchical names can be used to help suggest the search path to the mapping.

We have chosen a three-level naming hierarchy, consisting of *organizations*, within them *domains*, and within them *localnames*. We did not choose the arbitrary level scheme because of the greater complexity of the software required to handle names, because we do not think that networks will be combined very often, and because (as with area codes) we will make the name space for organizations large enough so that combinations can generally be made within the three-level hierarchy by merging two sets of existing ones. We choose three levels rather than, say, two or four, for pragmatic reasons. A mail system such as Grapevine [Birrell, Levin, Needham and Schroeder 1982] works well with only a two-level hierarchy, combining networks across the company's divisional boundaries. We add the third level primarily to facilitate combining networks across company lines. However, the clearinghouse does not give any particular meaning to the partitions; this is why we chose the relatively innocuous names "organization" and "domain."

4. User Names

One important class of "objects" known to the clearinghouse is the set of users. For instance, the clearinghouse may be used to map a user's name into the network address of the mail server where his mailbox resides. To deliver a piece of mail to a user, an electronic mail system first asks the clearinghouse where the mailbox for that user is and then routes the piece of mail to that server.

A major design decision is how the localname of users are chosen. We describe our approach to naming users as this will provide further motivation for our naming convention. The following is not part of the design of our clearinghouse, but illustrates one of its important uses.

A *User Name* is a string of the form:

<firstname> <blanks> <middlename> <blanks> <lastname>: <domain>: <organization>.

Here, *<firstname>*, *<middlename>* and *<lastname>* are strings separated by blanks (they may themselves contain blanks, as in the last name *de Gaulle*). *<firstname>*, *<middlename>* and *<lastname>* are the first name, middle name and last name of the user being named. The following are examples of user names:

David Stephen Jones:SDD:Xerox
John D. Smith:SDD:Xerox

The basic scheme, therefore, is that a name consists of the user's three-part localname, domain and organization. The localname is the person's legal name. The reason for making the user name the complete three-part name (rather than just the last name) is to discourage clashes of names and encourage unambiguity. The chance of there being two people with the name *Derek Charles Jones* in domain *SDD* in organization *Xerox* is hopefully rather remote, and certainly more remote than their being two people with last name *Jones*.

Our convention for naming users differs from those used in most computer environments in requiring that names be absolute and in using full names to reduce the chance of ambiguity. We have discussed the issue of absolute versus relative naming conventions already, but the second topic deserves attention because it shows the advantages of having a consistent approach to aliases.

The most common way of choosing unambiguous user names in computer environments is to use last names prefixed with however many letters are needed to exclude ambiguity. Thus, if there are two *Jones's*, one might be *DJones* and the other *HJones*. This scheme we find unsatisfactory. It is difficult for users (who have to remember to map their name for the person into the system's name for the person) and difficult for system administrators (who have to manage this rather artificial scheme). Further, it requires users to occasionally change their system names: if a system name is presently *DJones* and another *D. Jones* becomes a user, the system name must be changed to avoid ambiguity.

Our convention is not cumbersome to the user, since we use the same firstname-middlename-lastname convention people are used to already. However, since users would find it very cumbersome to type in full names, various aliases for user names are stored in the clearinghouse. For instance, associated with the user name *David Stephen Jones* might be the aliases *David Jones, D Jones* and *Jones*. Since our naming convention requires that aliases be absolute, it follows that no two users can have the same alias.

4.1. Birthmarks

Even with our convention of using a user's full name, there is a possibility that there will be two users with exactly the same name in a domain. Our approach is to disallow this, and let the two users (or a system administrator) choose unambiguous names for each. Another approach is to add as a suffix to each full name a "birthmark." A *<birthmark>* is any string which, together with the user name, the domain name and the organization name, unambiguously identifies the user. The birthmark may be a universal identifier (perhaps the concatenation of the processor number of the workstation on which the name is being added together with the time of day). It might be the social security number of the individual (perhaps not a good idea on privacy grounds). It might be just a positive integer; the naming authority for each domain is responsible for handing out integers. In any case, the combination of the full name and the birthmark must be unambiguous so that no two users can have the same legal name. In the case that a birthmark is not meaningful to humans, ambiguities must be resolved by providing users of such names with additional information such as a "title." The mappings described next provides a mechanism for providing such disambiguating comments.

5. Mappings

Now that we know how to name the objects known to the clearinghouse, we treat the question of what names are mapped into.

The clearinghouse maps each name into a set of *properties* to be associated with that name. A *property* is an ordered tuple consisting of a *PropertyName*, a *PropertyType* and a *PropertyValue*. The clearinghouse maintains mappings of the form:

$$\begin{aligned} \mathit{name} \rightarrow \{ & <\mathit{PropertyName}_1, \mathit{PropertyType}_1, \mathit{PropertyValue}_1>, \\ & \ldots, \\ & <\mathit{PropertyName}_k, \mathit{PropertyType}_k, \mathit{PropertyValue}_k>\}. \end{aligned}$$

More precisely, to admit aliasing, the clearinghouse maps equivalence classes, rather than names, into sets of properties. Each equivalence class consists of a distinguished name and its aliases. The value of k is not fixed for any given name. A name may have associated with it any number of properties.

A *PropertyValue* is a datum of type *PropertyType*. There are only two types of property values. The first, of type *item*, is "uninterpreted block of data." The clearinghouse attaches no meaning to the contents of this datum, but treats it as just a sequence of bits. The second, of type *group*, is "set of names." A name may appear only once in the set, but the set may contain any number of different names (including aliases and names of other groups). The names "item" and "group" reflect the semantics attached by the clearinghouse, whether the property is an individual datum or a group of data.

Mapping a name into a network address is an example of a type *item* mapping, as in:

Daisy:SDD:Xerox → *{<Printer, item, network address of the printer named Daisy>}.*

A distribution list in electronic mail is an example of a mapping of type *group*, as in:

CHAuthors:SDD:Xerox → *{*
<Distribution List, group, {"Dalal:SDD:Xerox", "Oppen:SDD:Xerox"}>}.

Many properties may be associated with a name, as in:

> *John D. Smith:SDD:Xerox → {*
> *<User, item, descriptive comment such as V.P. Marketing>,*
> *<Password, item, password to be used for user authentication>,*
> *<File Server Name, item, name of file server containing user's files>,*
> *<Mailbox, item, name of mail server where user's mail is stored>,*
> *<Printer Names, group, set of names of local printers any of which may be used>}.*

In this example, the clearinghouse is used to store the user's "profile." Note that we choose to map the user's name into the *name* of his local file server (and mailbox and printer) rather than directly into its network address. The reason for this extra level of indirection is that the name of the file server will perhaps never change but its location certainly will occasionally change, and we do not want a change in a server's location to require a major update of the clearinghouse's database.

5.1. Rationale

Objects tend to fall into two broad categories: objects such as workstations, servers or people whose names are mapped into descriptions, and objects such as distribution lists whose names are mapped into sets of names.

We differentiate between properties of type item and properties of type group, but allow many properties of differing types to be associated with each name. The example given above showing the mapping for a user name shows why. Unlike the simpler telephone model where a single mapping from a user name into a telephone number suffices, we want to map a user's name into a richer collection of information. This applies even to non-user individuals. We may want to associate with a printer's name not only its location (so that files to be printed can be sent to it), but also information describing what fonts the printer supports, if it prints in color, and so on.

The main reason for having "set of names" as a distinct data type is to allow different clients to update the same set simultaneously. For instance, if the set represents an electronic mail distribution list, we allow two users to add themselves to this list asynchronously.

5.2. Generic Names

The set of property names known to the clearinghouse defines a set of generic names by which the clearinghouse provides a Yellow Pages-like facility. Such a capability can be used as follows:

Client software can request a service in a standardized fashion, and need not remember what named resources are available. For instance, each user workstation generally has a piece of software that replies to the user command "Help!" This software accesses some server to obtain the information needed to help the user. Suppose the generic name "Help Service" is agreed upon as the standard property name for such a service. To find the addresses of the servers providing help to users in *SDD:Xerox*, the workstation software calls asks to list all objects of name *"*:SDD:Xerox"* with propertyname *Help Service*. The wildcard character "*" matches zero or more characters. This piece of code can be used by any workstation, regardless of its location.

The "wildcard" feature allows clients to find valid names where they have only partial information on or can only guess the name. It is particularly useful in electronic mail and in other uses of user names. If looking up *"Smith"*, with propertyname *Mailbox* fails, because "Smith" is ambiguous, the electronic mail system may choose to list all names of the form *"*Smith*:SDD:Xerox"*, with propertyname *User* to find the set of user names matching this name. It presents this set to the sender of the mail and allows him to choose which unambiguous name is appropriate. A simple algorithm to use in general might be to take any string provided by the user, surround the string with *s, delete any periods, and replace any occurrence of <blank> by *<blank>. Thus *David S. Jones* becomes **David* S* Jones**, which matches *David Stephen Jones*, as desired.

6. The Client's Perspective

Recall first that the clients of the clearinghouse are pieces of software and hardware making use of the clearinghouse client interface. The fact that people are not clients of the clearinghouse (except indirectly by means of a software interface) immediately introduces an important difference between our clearinghouse and the telephone system's. The telephone system relies on human judgement and human interaction. The clients of our clearinghouse are machines, not people, and so all aspects of client-clearinghouse interaction, including fault-tolerance, must be fully automated.

The clearinghouse (and its associated database) is decentralized and replicated. That is, instead of one *global* clearinghouse, there are many *clearinghouse servers* scattered throughout the internetwork (perhaps, but not necessarily, one per local network), each storing a copy of a portion of the global database. Decentralization and replication increase efficiency (it is faster to access a clearinghouse server physically nearby), security (each organization can control access to its own clearinghouse servers) and reliability (if one clearinghouse server is down, perhaps another can respond to a request). However, we do assume that there is one *global* database (conceptually, that is; physically the database is decentralized). Each clearinghouse server contains a portion of this database. We make no assumptions about how much of the database any particular clearinghouse server stores. The union of all the local databases stored by the clearinghouse servers is the global database.

A client of the clearinghouse may refer by name to, and query the clearinghouse about, any named object in the distributed environment (subject to access control) regardless of the location of the object, the location of the client or the present distributed configuration of the clearinghouse. We make no assumptions about the physical proximity of clients of the clearinghouse to the objects whose names they present to the clearinghouse. A request to the clearinghouse to bind a name to its properties may originate anywhere in the internetwork. This makes the internal structure of our clearinghouse considerably more intricate than that of the telephone clearinghouse (where clients have to know which local telephone directory to access), but makes it much easier to use.

In order to provide a uniform way for clients to access the clearinghouse, we assume that all clients contain a (generally very small) clearinghouse component, which we call a *clearinghouse stub*. Clearinghouse stubs contain pointers to clearinghouse servers, and they provide a uniform way for clients to access the clearinghouse.

A client requests a binding from its stub clearinghouse. The stub communicates with clearinghouse servers to get the information. A client of the clearinghouse stub need not concern itself with the question of which clearinghouse server actually contains the binding—the client's stub in conjunction with the clearinghouse servers automatically find the mapping if it exists.

Updates to the various copies of a mapping may occur asynchronously and be interleaved with requests for bindings of names to properties. Therefore, clearinghouse server databases may occasionally have incorrect information or be mutually inconsistent. (In this respect, we follow the telephone system's model and not the various models for distributed databases in which there is a notion of "indivisible transaction." We find the latter too complicated for our needs.) Therefore, as in the telephone system, bindings given by clearinghouse servers should be considered by clients to be *hints*. If a client requests the address of a printer, it may wish to check with the server at that address to make sure it is in fact a printer. If not, it must be prepared to find the printer by other means (perhaps the printer will respond to a local broadcast of its name), wait for the clearinghouse to receive the update, or reject the printing request. If the information given out by the clearinghouse is incorrect, it cannot, of course, guarantee that the error in its database will be corrected. It can only hope that whoever has invalidated the information will send (or preferably already has sent) the appropriate update. However, the clearinghouse does guarantee that any inconsistencies between copies of the same portion of the database will be resolved, that any such inconsistency is transient. This guarantee holds even in the case of conflicting updates to the same piece of information; the clearinghouse arbitrates between conflicting updates in a uniform fashion.

Assuming this model of goodwill on the part of its clients—that they will quickly update any clearinghouse entry they have caused to become invalid—and assuming an automatic arbitration mechanism for quickly resolving in a predictable fashion any transient inconsistencies between clearinghouse servers, clients can assume that any information stored by the clearinghouse is either correct or, if not, will soon be corrected. Clients therefore may assume that the clearinghouse either contains the *truth* about any entry, or soon will contain it. It is very important that clients can trust the clearinghouse in this way, because the clearinghouse is often the only source of information available to the client on the locations of servers, on user profiles, and so on.

The fact that the information returned by the clearinghouse is treated by the clients as both the truth (the information is available only from the clearinghouse and so had better be correct) and a hint (the information may be temporarily incorrect) is not self-contradictory. It merely reflects the difference between the long-term and short-term properties of clearinghouse information.

6.1. Binding Strategies

An important consideration to be taken by the client is that of *when* to ask the clearinghouse for a binding. The binding technique used greatly influences the ability of the system to react to changes in the environment.

There are three possibilities: *static* binding, in which names are bound at the time of system generation; *early* binding, in which names are bound, say, at the time the system is initialized; and *late* binding, in which names are bound at the time their bindings are to be used. (The boundaries between the three possibilities are somewhat ill-defined; there is a continuum of choices.) The main tradeoff to be taken into consideration in choosing a binding strategy is *performance* versus *flexibility*.

The later a system binds names, the more gracefully it can react to changes in the environment. If client software binds names statically, the software must be updated whenever the environment changes. For instance, if software supporting printing directly stores the addresses of the print servers (that is, uses a static binding strategy), it must be updated whenever new print servers are added or existing servers are moved or removed. If the software uses a late binding strategy, it will automatically obtain the most up-to-date bindings known to the clearinghouse.

On the other hand, binding requires the resolution of one or more indirect references, and this takes time. Static or early binding increases runtime efficiency since, with either, names are already bound at runtime. Further, late binding requires interaction with the clearinghouse at runtime. Although we have designed the clearinghouse to be very reliable, the possibility exists that a client may occasionally be unable to find any clearinghouse server up and able to resolve a reference.

There are therefore advantages and disadvantages to any binding strategy. A useful compromise combines early and late binding, giving the performance and reliability of the former and the flexibility of the latter. The client uses early binding wherever possible, and uses late binding only if any of these (early) bindings becomes invalid. Thus, software supporting printing stores the addresses of print servers at initialization, and updates these addresses only if they become invalid. Of course, the client must be able to recognize if a stored address is invalid (just as it must accept the possibility that the information received from the clearinghouse is temporarily invalid). We discuss hint validation further in Appendix 1.

7. Client Interface

The clearinghouse provides a basic set of operations, some of which are exported operations which may be called by clients of the clearinghouse by means of the *clearinghouse stub* resident in the client, and some of which are internal operations used by clearinghouse components to communicate with each other. Strictly speaking, the clearinghouse requires only a very few commands, for reading, adding, and deleting entries. We provide many different operations, however, as described in detail in [Oppen and Dalal 1981].

We give different commands for different types (for instance, different commands to add an item and to add a group) to provide a primitive type-checking facility.

We give different operations for different levels of granularity (for instance, different commands for adding groups and adding elemenets to a group) for three reasons. First, it minimizes the data that must be transmitted by the clearinghouse or the client when reading or updating an entry. Second, it allows different clients to change different parts of the same entry at the same time. For instance, two clients may add different elements to the same group simultaneously; if each were required to update the whole entry, their two updates would conflict . Third, we make use of the different operations for different levels of granularity in our access control facility.

Finally, we provide separate operations for changing a propertyvalue although these operations are functionally equivalent to deleting the original and adding the new one. However, changing an entry constitutes one indivisible transaction; deleting and adding an entry constitute two transactions separated by a time period during which another client may try to read the incorrectly-empty entry.

8. Clearinghouse Structure

We now describe how the clearinghouse is structured internally.

8.1. Clearinghouse Servers

The database of mappings is decentralized. Copies of portions of the database are contained in *clearinghouse servers* which are servers spread throughout the internetwork. We refer to the union of all these clearinghouse servers as "the clearinghouse." Each clearinghouse server is a named object in the internetwork, and so has a distinguished name and possibly aliases as well.

Every client of the clearinghouse contains a clearinghouse component, called a *clearinghouse stub*. Clearinghouse stubs provide a uniform way for clients to access the clearinghouse. Stub clearinghouses do not have names (although they will typically be on machines containing named objects). Stubs are required to find at least one clearinghouse server (for example, by local or directed broadcast [Boggs 1981]).

8.2. Domain and Organization Clearinghouses

Corresponding to each domain *D* in each organization *O* are one or more clearinghouse servers each containing a copy of all mappings for every name of the form *<anything>:D:O*. Each such clearinghouse server is called a *domain clearinghouse* for *D:O*. (Each clearinghouse server that is a domain clearinghouse for *D:O* may contain other portions of the database other than just the database for this domain, and each of the domain clearinghouses for *D:O* may differ on what other portions of the global database, if any, they contain.) There is at least one domain clearinghouse for each domain in the distributed environment. Domain clearinghouses are addressable objects in the internetwork and hence have names. Each domain clearinghouse for each domain in organization *O* has a name of the form *<anything>:O:CHServers* which maps into the network address of the server, under property name *CH Location*. (*CHServers* is a reserved organization name.) Thus, if *L:O:CHServers* is the name of a domain clearinghouse for *D:O*, then there is a mapping of the form:

L:O:CHServers → *{..., <CH Location, item, network address>, ...}*.

For each domain *D:O*, we require that *D:O:CHServers* map into the set of names of domain clearinghouses for *D:O*, under property name *Distribution List*. For example, if the domain clearinghouses for domain *D:O* have names L_1*:O:CHServers*, ..., L_k*:O:CHServers*, then there is a mapping of the form:

D:O:CHServers → *{..., <Distribution List, group, {*L_1*:O:CHServers, ...,* L_k*:O:CHServers}>, ...}*.

For each *i* and *j*, L_i*:O:CHServers* is a *sibling* of L_j*:O:CHServers for domain D:O*. Thus, we have given a name to the set of sibling domain clearinghouses for each domain in organization *O*.

The names of all domain clearinghouses, and all sets of sibling domain clearinghouses, for domains in organization *O* are themselves names in the reserved domain *O:CHServers*. We will call each domain clearinghouse for this reserved domain an *organization clearinghouse for O* since it contains the name and address of every domain clearinghouse in the organization. In particular, if L_1*:O:CHServers*, ...,

L_k*:O:CHServers* are the domain clearinghouses for any domain *D:O*, then each organization clearinghouse for *O* contains the mappings:

> *D:O:CHServers* → *{..., <Distribution List, group, {*L_1*:O:CHServers, ...,* L_k*:O:CHServers}>, ...},*
> L_1*:O:CHServers* → *{..., <CH Location, item, network address>, ...},*
> ...,
> L_k*:O:CHServers* → *{..., <CH Location, item, network address>, ...}.*

Since *O:CHServers* is a domain, there is at least one domain clearinghouse for *O:CHServers* and hence at least one organization clearinghouse for *O*. Each such clearinghouse has a name of the form *<anything>:CHServers:CHServers* which maps into the network address of the server, under property name *CH Location*. Thus, if *L:CHServers:CHServers* is the name of a domain clearinghouse for *O:CHServers* (that is, an organization clearinghouse for *O*), then there is a mapping of the form:

> *L:CHServers:CHServers* → *{..., <CH Location, item, network address>, ...}.*

For each organization *O*, we require that *O:CHServers:CHServers* map into the set of names of organization clearinghouses for *O*, under property name *Distribution List*. For example, if the organization clearinghouses for *O* have names L_1*:CHServers:CHServers*, ..., L_k*:CHServers:CHServers*, then there is a mapping of the form:

> *O:CHServers:CHServers* → *{...,*
> *<Distribution List, group, {*L_1*:CHServers:CHServers, ...,* L_k*:CHServers:CHServers}>,*
> *...}.*

Each L_i*:CHServers:CHServers* is called a *sibling* of L_j*:CHServers:CHServers for organization O*. Thus, we have given names to the set of sibling organization clearinghouses for each organization *O*.

Note that each organization clearinghouse for *O* points directly to every domain clearinghouse for any domain in *O*, and hence indirectly to every object with a name in *O*.

Note the distinction between *clearinghouse servers* on the one hand and *domain* and *organization* clearinghouses on the other. The former are physical entities that run code, contain databases and are physically resident on network servers. The latter are logical entities, and are a convenience for referring to the clearinghouse servers which contain specific portions of the global database. A particular clearinghouse server may be a domain clearinghouse for zero or more domains, and an organization clearinghouse for one or more organizations.

8.3. Interconnections between Clearinghouse Components

Organization clearinghouses point "downwards" to domain clearinghouses, which point "downwards" to objects. Further interconnection structure is required so that stub clearinghouses can access clearinghouse servers, and clearinghouse servers can access each other.

Each clearinghouse server is required to be an organization clearinghouse for the reserved organization *CHServers*, and hence each clearinghouse server points "upwards" to *every* organization clearinghouse. In this way, each clearinghouse server knows the name and address of every organization clearinghouse.

We do not require a clearinghouse server to keep track of which clearinghouse stubs point to it, so these stubs will not be told if the server changes location, and must rely instead on other facilities, such as local or directed broadcast, to find a clearinghouse server if the stored address becomes invalid.

8.4. Summary

Each clearinghouse server contains mappings for a subset of the set of names. If it is a domain clearinghouse for domain *D* in organization *O*, it contains mappings for all names of the form *<anything>:D:O*. If it is an organization clearinghouse for organization *O*, it contains mappings for all names of the form *<anything>:O:CHServers* (names associated with domains in *O*). Each clearinghouse server contains the mappings for all names of the form *<anything>:CHServers:CHServers* (names associated with organizations); that is, the database associated with the reserved domain *CHServers:CHServers* is replicated in every clearinghouse server. Stubs point to any clearinghouse server.

For every domain *D* in an organization *O*, *D:O:CHServers* names the set of names of sibling domain clearinghouse servers for *D:O*. For every organization *O*, *O:CHServers:CHServers* names the set of names of sibling organization clearinghouse servers for *O*. The name *CHServers:CHServers:CHServers* contains the set of names of *all* clearinghouse servers. (We do not make use of this name in this paper).

This clearinghouse structure allows a relatively simple algorithm for managing the decentralized clearinghouse. However, it does require that copies of the mappings for all names of the form *<anything>:CHServers:CHServers* be stored in *all* clearinghouse servers.

9. Distributed Lookup Algorithm

Suppose that a stub clearinghouse receives the query to look up an item. The stub clearinghouse follows the following general protocol.

The stub clearinghouse contacts any clearinghouse server and passes it the query. If the clearinghouse server that receives the stub's query is a domain clearinghouse for *B:C*, it can immediately return the answer to the stub who in turn returns it to the client. Otherwise, the clearinghouse server returns the names and addresses of the organization clearinghouses for *C*, which it is guaranteed to have. The stub contacts any of these clearinghouse servers. If this clearinghouse server happens also to be a domain clearinghouse for *B:C*, it can immediately return the answer to the stub who in turn returns it to the client. Otherwise the clearinghouse server returns the names and addresses of the domain clearinghouses for *B:C*, which it is guaranteed to have. The stub contacts any of these, since any of them is guaranteed to have the answer. An algorithmic description of this search process can be found in [Oppen and Dalal 1981].

The domain clearinghouse for *B:C* that returns the answer to the query does so after authenticating the requestor and ascertaining that the requestor has appropriate access rights.

In the worst case, a query conceptually moves "upwards" to a domain clearinghouse to an organization clearinghouse, and then "downwards" to one of that organization's domain clearinghouse. The number of clearinghouse servers that a stub has to contact will never exceed three: the clearinghouse server whose address it knows, an organization clearinghouse for the organization containing the name in the query, and a domain clearinghouse in that organization.

However, before sending the query "upwards," each clearinghouse component *optionally* first sees if it can shortcut the process by sending the query "sideways," cutting out a level of the hierarchy. (This is similar to the shortcuts used in the routing structure of the telephone system.) These "sideways" pointers are cached pointers, maintained for efficiency. For instance, consider domain clearinghouses for *PARC* and for *SDD*, two logical domains within organization *Xerox*. Depending on the traffic, it may be appropriate for the *PARC* clearinghouse to keep a direct pointer to the *SDD* clearinghouse, and vice versa. This speeds queries that would otherwise go through the *Xerox* organization clearinghouse. (Cached entries are not kept up to date by the clearinghouse. This is not a serious problem; if a cached entry is wrong it is deleted, and the general algorithm described above is used instead.)

To increase the speed of response even further, each clearinghouse server could be a domain clearinghouse for both domains. Alternatively, if the number of domains in *Xerox* is relatively small, it may be appropriate to make each clearinghouse server in *Xerox* an organization clearinghouse for *Xerox*. In this way, each clearinghouse server in *Xerox* always points to every other clearinghouse server in *Xerox*.

Local queries (that is, queries about names "logically near" the clearinghouse stub) will typically be answered more quickly than non-local queries. That is appropriate. The caching mechanism (for storing of "sideways" pointers) can be used to fine-tune clearinghouse servers to respond faster to non-local but frequent queries.

10. Distributed Update Algorithm

The distributed update algorithm we use to alter the clearinghouse database is closely related to the distributed update algorithm used by Grapevine's registration service [Birrell, Levin, Needham, and Schroeder 1982].

The basic model is quite simple. Assume that a client wishes to update the clearinghouse database. The request is submitted via the stub resident in the client. The stub contacts any domain clearinghouse containing the mapping to be updated. The domain clearinghouse updates its own database and acknowledges that it has done so. The interaction with the client is now complete. The domain clearinghouse then propagates the update to its siblings if the database for this domain is replicated in more than one server.

The propagation of updates is not treated as an indivisible transaction. Therefore, sibling clearinghouse servers may have databases that are temporarily inconsistent; one server may have updated its database before another has received or acted on an update request. This requires mechanisms for choosing among conflicting updates and dealing with out-of-order requets. The manner in which the clearinghouse deals with these issues in the context of the services it provides can be found in [Oppen and Dalal 1981].

Since distributed office information systems usually have an electronic mail delivery facility that allows "messages" to be sent to recipients that are not ready to receive them (see [Birrell, Levin, Needham, and Schroeder 1982]), we make use of this facility to propagate updates. Clearinghouse servers send their siblings timestamped update messages (using the appropriate distribution lists reserved for clearinghouse servers) telling them of changes to the databases. There is a possible timelag between the sending and the receipt of a message. Since our clearinghouse design does not require that updating be an indivisible process, this is not a problem.

11. Security

We restrict ourselves to a brief discussion of two issues. The first concerns protecting the clearinghouse from unauthorized access or modification, and involves *authentication* (checking that you are who you say you are), the second concerns *access control* (checking that you have the right to do what you want to do). We do not discuss how the network ensures (or does not ensure) secure transmission of data, or how two mutually-suspicious organizations can allow their respective clearinghouse servers to interact and still keep them at arms' length.

11.1 Authentication

When a request is sent by a client to a clearinghouse server to read from or write onto a portion of the clearinghouse database, the request is accompanied by the credentials of the client. If the request is an internal one, from one clearinghouse server to another, the requestor is the name of the originating clearinghouse server and carries the credentials of the originating clearinghouse server. The clearinghouse thus makes sure that internally-generated updates come only from "trusted" clearinghouse servers. The clearinghouse uses a standard authentication service to handle authentication, and the authentication service uses the clearinghouse to store its authentication information [Needham and Schroeder 1979].

11.2. Access Control

Once a client has been authenticated, it is granted certain privileges. Access control is provided at the domain level and at the property level, and not at the mapping (set of properties) level nor at the level of element of a group. Associated with each domain and each property is an *access control list*, which is a set of the form $\{<set\ of\ names_1, set\ of\ operations_1 >, \ldots, <set\ of\ names_k, set\ of\ operations_k >\}$. Each tuple consists of a set of names and the set of operations each client in the set may call. Access control is described in [Oppen and Dalal 1981].

Certain operations that modify the clearinghouse database are protected only at the domain level. These are operations that are typically executed only by *domain system administrators* and by other clearinghouse servers. Examples of such operations are: adding or deleting a new name or alias, adding or deleting an item or group for a name. Other operations such as looking up an item, or adding a name to a group are protected at the property level.

12. Administration

An internetwork configuration of several thousand users and their associated workstations, printers, file servers, mail servers, etc., requires considerable management. Administrative tasks include managing the name and property name space; bringing up new networks; deciding how to split an organization into domains (reflecting administrative, geographical, functional or other divisional lines); deciding which objects (users, services, etc.) belong to which domains; adding, changing and deleting services (such as mail services, file services, and even clearinghouse services); adding and deleting users; maintaining users' passwords, the addresses of their chosen local printers, mail and file servers, and so on; and maintaining access lists and other security features of the network. We have designed the clearinghouse so that system administration can be decentralized as much as possible.

An algorithmic description of the process by which a new organization or domain clearinghouse server is added to the internetwork can be found in [Oppen and Dalal 1981].

13. Conclusions

A powerful binding mechanism that brings together the various network-visible objects of a distributed system is an essential component of any large network-based system. The clearinghouse provides such a mechanism.

Since we do not know how distributed systems will evolve, we have designed the clearinghouse to be as open-ended as possible. We did not design the clearinghouse to be a general-purpose, relational, distributed database nor a distributed file system, although the functions it provides are superficially similar. It is not clear that network binding agents, relational databases, and file systems should be thought of as manifestations of the same basic object; their implementations appear to require different properties. In any case, we certainly did not try to solve the "general database problem," but rather attempted to design a system which is implementable now within the existing technology yet which can evolve as distributed systems evolve.

A phased implementation of this design is currently under way. Xerox's Network Systems product may chose to deviate in minor ways from this design and enforce different administrative policies than described here. A future paper will describe the network protocols used in implementing the clearinghouse, and our experiences with them.

Acknowledgments

Many people have examined the requirements and structure of clearinghouse-like systems; and we have profited from the earlier efforts by Steven Abraham, Doug Brotz, Will Crowther, Bob Metcalfe, Hal Murray, and John Shoch. The organization of the clearinghouse, and, in particular, the lookup and updating algorithms, have been very heavily influenced by the work of the Grapevine project; we thank Andrew Birrell, Roy Levin, and Mike Schroeder for many stimulating discussions. The clearinghouse is a fundamental component of Xerox's Network Systems product line, and we are indebted to Marney Beard, Charles Irby, and Ralph Kimball for their considerable input on what the Star workstation needed from the clearinghouse. Dave Redell, who is responsible for the Network System electronic mail system, was a constant source of inspiration and provided us with many useful ideas. Bob Lyon and John Maloney implemented the clearinghouse; we thank them for turning the clearinghouse design into reality.

References

[Abraham and Dalal 1980] S. M. Abraham and Y. K. Dalal, Techniques for Decentralized Management of Distributed Systems, *20th IEEE Computer Society International Conference (Compcon)*, February 1980, pp. 430-436.

[Birrell, Levin, Needham, and Schroeder 1982] A. D. Birrell, R. Levin, R. M. Needham and M. D. Schroeder, Grapevine: an Exercise in Distributed Computing, *CACM*, vol 25 , no 4, April 1982, pp. 260-274.

[Boggs 1981] D. R. Boggs, Internet Broadcasting, Ph.D. Thesis, Stanford University, January 1982.

[Boggs, *et al.* 1980] D. R. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe, PUP: An internetwork architecture, *IEEE Transactions on Communications*, com-28:4, April 1980, pp. 612-624.

[Dalal 1982] Y. K. Dalal, Use of Multiple Networks in Xerox's Network System, *IEEE Computer Magazine*, vol 15, no 8, August 1982, pp. 10-27.

[Daley and Neumann 1965] R. C. Daley and P. G. Neumann, A general-purpose file system for secondary storage, *Proc. Fall Joint Computer Conf.*, 1965, AFIPS Press, pp. 213-228.

[Ethernet 1980] The Ethernet, A Local Area Network: Data Link Layer and Physical Link Layer Specifications, Version 1.0, September 30, 1980.

[Metcalfe and Boggs 1976] R. M. Metcalfe and D. R. Boggs, Ethernet: Distributed packet switching for local computer networks, *CACM*, 19:7, July 1976, pp. 395-404.

[Needham and Schroeder 1979] R. M. Needham and M. D. Schroeder, Using Encryption for Authentication in Large Networks of Computer, *CACM*, 21:12, December 1978, pp. 993-999.

[Oppen and Dalal 1981] D. C. Oppen and Y. K. Dalal, The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment, Xerox Office Systems Division, OPD-T8103, October 1981.

[Pickens, Feinler, and Mathis 1979] J. R. Pickens, E. J. Feinler, and J. E. Mathis, The NIC Name Server–A Datagram Based Information Utility, *Proceedings 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, August 1979.

[Saltzer 1982] J. H. Saltzer, On the Naming and Binding of Network Destinations, *Proc. IFIP TC 6 Symposium on Local Networks*, April 1982, pp. 311-317.

[Schickler] P. Schickler, Naming and Adressing in a Computer-Based Mail Environment, *IEEE Trans, Comm.*, vol. COM-30, no. 1, January 1982, pp. xxx.

[Shoch 1978] J. F. Shoch, Internetwork Naming Addressing and Routing, *17th IEEE Computer Society International Conference (Compcon)*, September 1978.

[Solomon, Landweber, and Neuhengen] M. Solomon, L. H. Landweber, and D. Neuhengen, The CSNET Name Server, *Computer Networks*, vol. 6, no. 3 July 1982. pp. xxx.

[Xerox 1982] *Office Systems Technology*, Xerox Office Systems Division, OSD-R8203, November 1982.

Appendix 1: Network Addresses and Address Verification

In Xerox's Network System. a *network address* is a triple consisting of a *network number*, a *host number*, and a *socket number* [Dalal 1982]. There is no clear correspondence between machines and the addressable objects known to the clearinghouse. One machine on an internetwork may contain many named objects; for instance, a machine may support a file service and a printer service (a *server* may contain many *services*). These different objects resident on the same machine may use the same network address even though they are separate objects logically and have different names. This introduces no problems since the clearinghouse does not check for uniqueness of addresses associated with names. Alternatively, different objects physically resident on one machine may have different network addresses, since a machine may have many different socket numbers. To allow both possibilities, we map the names of addressable objects into network addresses without worrying about the configurations of the machines in which they are resident.

However, it may be that one machine has more than one network address, since it may be physically part of more than one network. Therefore, the name of an addressable object such as printer or file server may be mapped into a set of network addresses, rather than a single address. However, these addresses may differ only in their network numbers: objects may be physically resident in one machine only.

Since the addresses given out by the clearinghouse may be momentatrily incorrect, clients need a way to check the accuracy of the network addresses given out by the clearinghouse. One way is to insist that each addressable object have a *uniqueid*, an absolute name which might, for example, consist of a unique processor number (hardwired into the processor at the factory) concatenated to the time of day. The uniqueid is used to check the accuracy of the network addresses supplied by the clearinghouse. This uniqueid is stored with the network addresses in the clearinghouse. When a client receives a set of addresses from the clearinghouse which are allegedly the addresses of some object, it checks with the object to make sure the uniqueid supplied by the clearinghouse agrees with the uniqueid stored by the object.

In summary, in the Xerox internetwork environment, the address of an object is stored as a tuple consisting of a set of network addresses and a uniqueid.

Authentication in Office System Internetworks

by Jay E. Israel and Theodore A. Linden

Abstract. In a distributed office system, authentication data (such as passwords) must be managed in such a way that users and machines from different organizations can easily authenticate themselves to each other. The authentication facility must be secure, but user convenience, decentralized administration, and a capability for smooth, long-term evolution are also important. In addition, the authentication arrangements must not permit failures at a single node to cause system-wide downtime. We describe the design used in the Xerox 8000 Series products. This design anticipates applications in an open network architecture where there are nodes from diverse sources and one node does not trust authentication checking done by other nodes. Furthermore, in some offices encryption will be requirec to authenticate data transmissions despite hostile intruders on the network. We discuss requirements and design constraints when applying encryption for authentication in office systems. We suggest that protocol standards for use in office systems should allow unencrypted authentication as well as two options for encrypted authentication; and we describe the issues that will arise as an office system evolves to deal with increasingly sophisticated threats from users of the system.

1. Introduction

In an office system, anyone who can impersonate someone else successfully will be able to avoid almost all security and accounting controls. Authentication deals with the verification of claimed identity. It is the foundation on which access controls, audit measures, and most other security and accounting controls are built.

Authentication became a challenging problem in the Xerox 8000 Series products because of the highly distributed nature of the systems for which these products are designed. Users and machines must be able to authenticate themselves to each other even when they are widely separated both by physical distance and by organizational and administrative boundaries. To ease the burden of maintaining and administering the authentication data (typically this data is a list of the users' passwords), some degree of specialized support is needed. But this support has to respect the territorial prerogatives of independent organizations and cannot become a performance bottleneck or a source of system-wide failure.

In this paper we use our experiences during development of the Xerox 8000 Series products to discuss practical issues concerning authentication in large, distributed office systems. In Sections 2 through 7 we describe the procedures for authentication in a distributed environment that are already largely implemented in the 8000 Series products. These sections describe when and where authentication is done and how the appropriate parties obtain the data needed to carry out the identity verification. We emphasize the tradeoffs between the conflicting goals of reliable authentication, ease of use, ease of administration, and system performance and availability.

Sections 8 through 13 discuss the tradeoffs in applying encryption technology to insure secure authentication even when users on the network deliberately attempt to subvert the authentication procedures. We recommend the development of protocol standards for use in office systems that recognize three levels of authentication requirements: 1) basic authentication in a distributed environment where there is no requirement for encryption techniques, 2) use of encryption to prevent someone from discovering information which can easily be used to impersonate someone else, and 3) full authentication of entire data interchanges in the face of hostile and sophisticated intruders.

2. The Problem to be solved.

This paper deals with techniques for maintaining, transmitting, and checking authentication data in a distributed system. It concerns authenticating both users and machines. It does not discuss specific identification techniques such as passwords or magnetically encoded cards or wired-in machine identifiers. The reader may see [11] for a discussion of such interfaces. The specific authentication is largely independent of the issues discussed here (except some adaptation would be needed for automatic signature analysis, fingerprint readers, and other techniques that involve checking large amounts of data with extensive processing). Since passwords are still the most common interface for authentication, we will use the more familiar term "password" even when the discussion applies equally well to checking other kinds of authentication data.

The problem of maintaining, transmitting, and checking passwords is challenging because a good solution needs to take into account the practical constraints and conflicting goals that apply to most office systems:

> *Limited advance planning.* Office systems grow incrementally from small systems to large internetworks. It is generally impossible to foresee future needs with any clarity. The authentication must be compatible with this unplanned growth. The most stringent test of a design in this regard is to ask how much administrative effort is involved in taking two office networks that have been developed independently – but are otherwise compatible – and making them work together. To allow communication between the networks, users from one network must be able to authenticate themselves to machines of the other network.
>
> *Minimal administrative complexity.* The administrative effort to support authentication must be small. Users come and go all the time. It must be simple to enforce desired controls reliably despite these constant updates to the authentication data base. For example, when a change occurs an administrator should not have to enter the update at many locations or be responsible for manually maintaining consistency of multiple copies. In the early internal Xerox experimental internetwork, each file server contained a list of its authorized users and their passwords. As the number of servers increased, the maintenance of these partially overlapping lists across geographical and organizational boundaries rapidly became unmanageable.
>
> *User convenience.* Users of office systems will be under pressure to get a job done. They are likely to resent any complicated actions that are required for authentication. For example, when the file servers on the Xerox experimental internetwork each maintained their own passwords, users trying to retrieve a document from a server that they had not used recently would find they had forgetten which password to use.
>
> *Heterogeneity.* Devices with varying characteristics in their ability to support security measures should co-exist without reducing all security to that of the weakest device.
>
> *Level of protection.* The strength of the protection required and the price one is willing to pay for it should vary with the situation.

Availability. Useful work should remain possible in the face of many types of failures, including failures in the hardware that supports the authentication.

Naming. Entities in an internetwork - users and shared resources - have textual human-sensible names. To make it easy to use, the naming scheme must be flexible and should build on people's everyday experiences. Authentication techniques must mesh with this naming, since the identity of a communication partner is the issue to be determined.

3. Xerox Distributed Office System Architecture

Our context is that of an integrated, distributed office system designed to support incremental growth. For communication within a facility or campus we use an Ethernet local area network [DEC 80]. This provides high bandwidth local communications using an underlying broadcast medium. Other communication media can be used to link geographically dispersed local networks into a large internetwork.

In our architecture, nodes are divided into two general categories: *workstations* and *servers*. A workstation is the physical device that the user sees. It is at a person's desk, providing direct interactive functions. Xerox provides different models of workstations with different levels of functionality. This paper deals primarily with the most powerful of these, the Xerox 8010 Information Processing System, and its interactions with servers. This workstation provides an integrated user interface to a wide variety of operations on text, business graphics, and records. Some details of its capabilities are documented in [9, 14, 16, 17].

In contrast with workstations, a server is generally an unattended device that manages resources that are shared among many users. At a user's instigation, a workstation communicates with a server to operate on the shared resource. Sometimes, one server calls on another in a sub-contractor arrangement to do part of the work. Generally, servers are small processors (they use the same processor as the 8010 workstation) with limited roles. A *service* is a program that runs on a server to provide a specific set of capabilities. A server is a device and a service is a program running on it. Several services may reside in the same server, with limitations dictated by device capacities and level of usage. Different servers within an internetwork can provide instances of the same generic service.

The Internet Transport Protocols [19] are designed so that workstations can easily interact with any instance of a service, local or remote. For example, the same workstation software that sends information to a local printer can equally well send the information to a printer in another city, as long as that printer is on a network linked to the internetwork. Courier, a Remote Procedure Call Protocol [18] allows application-level protocols to be specified by modeling a service interface as a set of procedure calls. Application-level protocols that complete the definition of interfaces to various services are being prepared for publication.

Current services include the following:

Print service. Produces paper copy of documents and records transmitted to it.

File service. Stores and retrieves documents and record files. It has larger storage capacity than workstations, and provides a mechanism for sharing files among users.

Electronic mail service. Routes information to named users or to lists of users.

Internetwork routing service. Combines multiple dispersed Ethernets into a single logical internetwork with a uniform address space.

Gateway service and *interactive terminal service.* Allow network services to be extended over an external telephone line (rather than by direct connection to an Ethernet cable).

External communication service. A lower-level service that controls external telephone lines. It supports some of the previously-mentioned services, as well as workstations that can interact with data processing systems by emulating terminals.

Clearinghouse service. A repository of information about users, services, and other resources accessible in the internetwork.

Figure 1 depicts a typical, small internetwork.

4. The Clearinghouse Service

The clearinghouse service deserves further discussion here because of its central role in user authentication. Its philosophy and design are elucidated in [13]. The clearinghouse is a distributed service, maintaining a distributed data base of information about network resources and users. For example, it keeps a record of the names, types, and network addresses of all instances of services available in an internetwork. This information is available to other nodes, whenever they need to discover something about their environment. A clearinghouse service also keeps a record of the users who are registered in an internetwork. In what follows, we use the term "clearinghouse" by itself as an informal way of referring to a clearinghouse service. Keep in mind that it is not a machine, but a distributed service with instances in (potentially) many servers.

The design of the clearinghouse illustrates several issues in the implementation of distributed office systems. In particular, there is a trade-off between centralized and distributed operation. A centralized design would have led to a simpler development task. The arguments for logical and physical decentralization were decisive, however. With the service residing on several machines, failure of one is not serious, since another can assume the load of both (perhaps with some performance degradation). Duplication of the data storage on more than one server means that all parts of the database remain accessible. It also means that damage to one copy of the data is not serious, since it can be recovered from another service instance.

Logical decentralization means that administration of different parts of the database is in the hands of different people. This permits access control of a resource to remain under the control of the organization owning the resource. It also avoids the administrative bottleneck of a single data administration entity. Each organization is responsible for registering its own users and services.

A distributed clearinghouse presents certain design challenges. The various service instances must coordinate with each other for several reasons. For example, a distribution list maintained by one

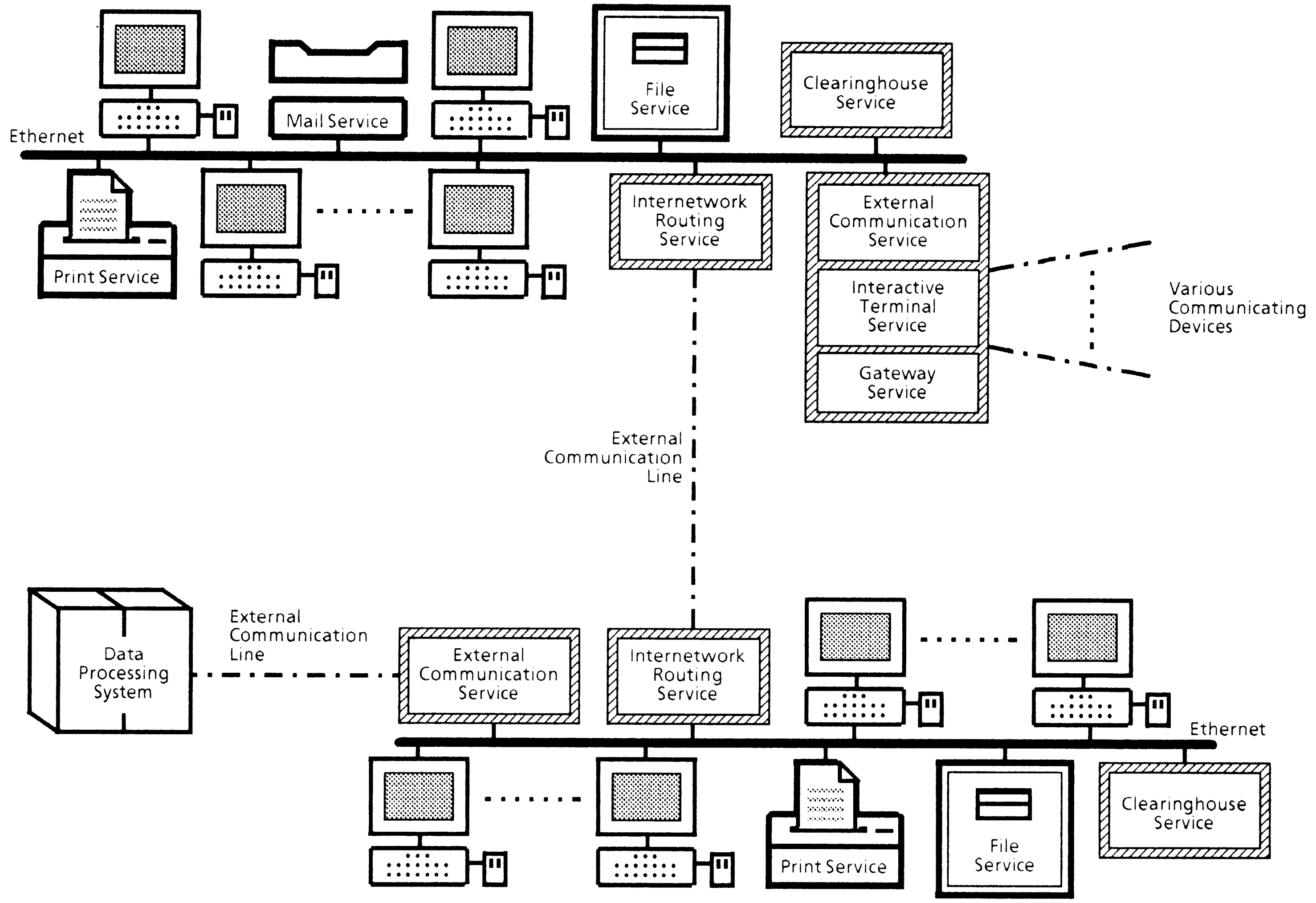

Ethernet
Mail Service
File Service
Clearinghouse Service
Print Service
Internetwork Routing Service
External Communication Service
Interactive Terminal Service
Gateway Service
Various Communicating Devices
External Communication Line
Data Processing System
External Communication Line
External Communication Service
Internetwork Routing Service
Ethernet
Print Service
File Service
Clearinghouse Service

Figure 1. Schematic of typical small internetwork.

organization may contain users from several organizations. Also, updates directed by the administrator to one node must propogate automatically to other nodes holding copies of the affected data.

5. Users, Desktops, and Authentication

This section gives more detail on the naming of entities, mobility of users and their data, and the two contexts under which authentication is required. The theme is that the distributed system must provide flexibility of usage, a characteristic that must be supported by the authentication design.

Each person authorized to use workstations and network services is a "user," represented by a record of information in the internetwork's clearinghouse. Each user has a textual name (as does, in fact, each instance of a service in the internetwork). Some of the challenges in implementing authentication arise not from security requirements, but from the flexible way by which entities can be named.

> Fully qualified names are unique throughout the world, so different internetworks can join without global renaming. (This is analogous to Ethernet's global assignment of physical host numbers.) If names were not unique, connecting two internetworks that grew independently would likely result in the same names being used for different entities. The renaming would be time-consuming for administrators and disruptive to users.
>
> Administration of name assignment can be decentralized in the user community. This is accomplished by making names hierarchical. Groups in the organizational hierarchy are delegated the responsibility for administering their part of the name space. This is how name uniqueness is achieved while ceding to each group autonomous control over its own resources.
>
> Users' network names are predictable from their ordinary names. Part of the hierarchical name is the user's full legal name. This permits users to refer to one another by the familiar names they use in daily discourse. People are adverse to learning cryptic account names or acronyms to identify their colleagues.
>
> Users can be identified with short, informal names when there is no ambiguity. Informal "aliases" are supported, and levels of the hierarchical name may be omitted when reasonable default values can be inferred from the context. This again lends an air of familiarity to the naming scheme for the benefit of the ordinary user.

These last two features of the naming system add some subtlety to the authentication algorithms, since numerous text strings can be alternative names for the same entity.

Any workstation or service desiring to authenticate a user relies on the common clearinghouse facility, vastly simplifying administrative procedures (compared with a design requiring different services to keep track of their users separately). For each user, the clearinghouse keeps a record of interesting information about him or her, including the password employed during authentication. It also keeps named lists of users, employed (for example) as electronic mail distribution lists. Assignment of names and aliases is the responsibility of system administrators, users with special privileges.

Each user of the 8010 system has a desktop. This is his or her private working environment. When someone is using a desktop, it is kept entirely at the local workstation. It is not shared; no one else may gain access to any object stored in the desktop unless the user explicitly moves or copies that object to some other node that allows shared access. The desktop is portrayed graphically to the user as a collection of documents, folders, record files, text, graphics, etc. Several desktops are allowed to reside on a workstation concurrently, though only one may be in use at any given time. Each bears the name of its user.

Users often need to take their work environments with them when visiting another location, or when using any available machine in a pool of workstations. To facilitate this, a desktop is allowed to migrate from one workstation to another in the internetwork. At the end of a session, the user has the option of moving the desktop to a file service. At a later time, the act of logging on retrieves the desktop to a workstation – either the same one as before or a different one. The file service chosen is the user's "home" file service, the one identified for this purpose in that user's clearinghouse entry. For the most part, however, it is expected that a desktop remains on a particular workstation. In fact, the distribution of functionality is designed in such a way that a great deal of a user's work can be accomplished on the local workstation alone, relying on services only when access is needed to their shared resources. Contributing to this autonomy is selective caching of information in the desktop data structure. Caching permits certain redundant accesses to services to be bypassed, enhancing both reliability and performance.

An object on a user's desktop is private – others do not have access to it. To support this model, a workstation does not respond to communication requests from elsewhere. In any communication interchange, it is the initiator. Consequently, a user who has left his or her desktop on a workstation (i.e., has not stored it on a file server when ending a session) must return to that workstation to resume work with that desktop. While this could be inconvenient in some situations, it has an important security advantage: if a desktop is left permanently on one workstation, the objects on it are secure from intruders on the network. As long as the physical integrity of the workstation's hardware and software is maintained, objects on the desktop can be accessed only by somebody who goes to that workstation and is successfully authenticated as that user.

Authentication is necessary in two places: when a user desires access to a workstation, and when some requestor wants access to a service. (The requestor may be a workstation acting on behalf of a user, or it may be another service.) In the next two sections, we describe the initial designs of first the workstation authentication mechanism, then the services procedures.

6. Workstation Logon

A user begins a session at a workstation by presenting a name and password. Access to a desktop is granted if they are found to be valid. The design of the initial logon mechanism was intended to meet certain objectives:

> *Verify user identity.* Check that the name and password are those of a user registered in the clearinghouse. The name may be abbreviated and may be an alias, as discussed above.
>
> *Ascertain user characteristics.* Some workstation features require information about the user that is maintained by the clearinghouse. For example, electronic mail must know where to look for incoming messages. This information is obtained at logon and cached locally.

Tolerate equipment failures. If the services normally involved are inaccessible, or if a workstation fails, users should still be able to carry out much of their work.

Control proliferation of desktops. In general, a policy of one desktop per user is desired, to simplify the user's model of the environment. To do otherwise would require coping with still another naming scheme (one for desktops).

As is readily apparent, these objectives are partially in conflict with one another, and some compromise is necessary. For example, if a user's desktop is temporarily trapped on a broken workstation or file server, the last two objectives conflict. Our approach is to permit an additional desktop for a user under such unusual conditions. Here, system availability outweighs naming simplicity. Having made this decision, though, we must ensure that software can handle the multiple-desktop situation.

One objective not addressed by the initial design is protection against unauthorized access by an intruder to the underlying communication media. Later in this paper, we will discuss the implications of adding encryption to the design of authentication in distributed systems, so that passwords will not be transmitted in the clear. First, however, we sketch the initial implementation.

The first step is to check the name and password. There are two sources of information on which this test can be made: the clearinghouse service and data cached locally. The former is tried first, since the latter may be out of date. In the rare event that the clearinghouse is inaccessible, the logon program attempts to find a local desktop for the user. To permit this, each desktop contains its user's complete *fully qualified name*, most recently used alias, and most recently validated password (the latter is protected by one-way encryption). Consequently, authentication is possible using local data alone.

The second step during logon is to locate the user's desktop (unless it has already been found locally). The clearinghouse provided some additional information about the user. For example, if an alias or abbreviated name was used, it supplied the corresponding fully qualified name. Using this, we can look on the workstation and determine whether or not the desktop being sought is there. If it is not, we employ a second piece of information that was obtained from the clearinghouse: the identity of the user's "home" file service. If the desktop exists, that is where it must be (unless it is at another workstation, and thus is inaccessible). The attempt to retrieve the desktop from that service could fail if the desktop is not there, or the server is broken, or communication with it is severed, or there is inadequate space on the workstation. In these situations, the user is given the option of having a new desktop created.

As a final step, the logon program caches in the desktop the information about the user obtained from the clearinghouse, overwriting whatever version of this information was there previously. Note that in all the logon processing, this cache was never used as a source of information if the clearinghouse was accessible. Thus, no problem arises if any of the cached information is out of date. At worst, the clearinghouse contains a new password for the user, changed since the last session. If the clearinghouse is inaccessible, the user may have to employ the old password temporarily. It is important to note, however, that validity of the old password is indeed temporary; it becomes invalid the first time a logon occurs while the clearinghouse is accessible. This design was deemed to be a proper trade-off between security and availability.

7. Logon for Services

In the course of providing its services, a server receives requests over the communication network from workstations (or other services). The service-provider has an option: it can accede to requests indiscriminately (trusting workstations to have performed the authentication), or it can (re)authenticate the user. How one approaches this decision depends on one's view of the network and the trustworthiness of its various nodes. In a closed local network (one in which the nodes and their software are from a single source and are reasonably immune to tampering), one is tempted to think of the local area network as an internal bus in a multi-processor system. The image is enhanced by the speed of a local area network, though not by its physical dispersal. From this point of view, it seems superfluous for a service to do authentication. An open local network is a different situation. Here, the architecture supports devices from a variety of manufacturers, and nodes may be widely dispersed. It is less reasonable for a service to assume that the user's identity is always authenticated adequately by the workstation.

Of course, there are associated costs if servers re-authenticate. First, there is the performance cost: it takes time. Second, there is an availability cost. If a node crucial to authentication is inaccessible, the user may be unable to use a file service, for example, even if the workstation and file service are both operational. The file service could be designed with a cache to help in this situation, but it is not clear how the server is to decide what to cache – far less clear, certainly, than in the case of the workstation. If it caches information on all potential users, that is a large storage burden. One must also consider the processing burden to keep it all up to date. If it caches less, there will be situations in which some users can do remote filing but not other users. This may be undesirable from the point of view of the user community as a whole, especially if the discrimination among users appeared to be arbitrary.

In the initial 8000 Series implementation, some services require no access controls, so they do not authenticate users. For example, the external communication service is a low-level service, and assumes that any authentication necessary will be performed by the application software employing it.

A file service or clearinghouse is much more discriminating. When a workstation contacts a file service, the user's name and password are presented. (Of course, the user does not have to retype them; they are already known by the workstation software.) The service checks the name and password against the clearinghouse data base, and accepts the connection attempt only if it is satisfied that the user is who he or she claims to be. Note that a three-way interchange results, involving the workstation, file service, and clearinghouse service. In validating a user, therefore, the initial design places a server in much the same role as a workstation in its relationship to the clearinghouse.

8. Requirements for Encryption during Authentication

The authentication procedures discussed thus far have proven to be quite workable for distributed office systems, and they provide a level of authentication that is adequate in offices with ordinary security requirements. In the typical office, the communication subsystem is not the weakest link in the office's security procedures. For example, passwords are transmitted in plaintext between terminals and most on-line computers, and this is seldom considered the most serious security concern. On the other hand, when one has an open network where protocols are public and where the connection of products from different vendors is encouraged, there are offices where it will be important to authenticate network communications in the face of hostile activity from other users of the network.

The workstations and servers in the Xerox 8000 Series products are programmed so that a user of these machines cannot read or modify information being sent to a different node. However, as an open network grows and becomes more diverse, it becomes less realistic to trust that administrative and software controls will prevent a malicious user from looking at information in transit and exploiting the information in some way. The greatest exposure occurs when there are nodes on the network where users can bypass the basic software with which the machine was delivered. In principle, such users can read any data communicated on the local area network, and can inject arbitrary packets. In this environment, a concerted effort is required to insure that data transmissions are authentic.

The first step in this direction is to protect passwords to insure that an intruder on the network cannot watch for some privileged user's password and then later impersonate that user. To protect the user authenticator, merely encrypting it under some stable key would do little good. An intruder could simply read the encrypted string, then later deceive the same service by replaying the encrypted string. Note also that protecting only the passwords may accomplish very little. For example, the 8000 Series file service requires a password only to establish a "session." Thereafter, a session identifier is used to associate individual commands with the user. Clearly one must also prevent an intruder from reading and replaying a session identifier.

Full authentication of network communications involves not only verifying the identity of the source of the communications but also verifying that the content of the communication has not been modified. One must also detect an intruder who attempts to replay a previous valid message. Full authentication also requires authentication of communications in both directions – an intruder should not be able to imitate a legitimate service and have users believe that their attempts to interact with that service have been successful.

There is little one can do to safeguard authentication against intruders on the network without a fairly sophisticated application of encryption. The remainder of this paper discusses the tradeoffs involved in applying encryption technology to provide secure authentication in an office system. After describing some of the practical constraints, we identify three separate options for authentication that seem to make sense in different office environments:

1. The first option is simple checking of user passwords. In a distributed system, this checking must be reasonably effective and easily administered, but does not protect against intruders reading or modifying information during transmission. This level of authentication was described in the first part of this paper.
2. The second level of authentication is robust in the face of relatively passive efforts to discover information that can later be used to impersonate someone. It uses encryption techniques in a minimal way.
3. The third level fully authenticates the entire data interchange in the face of active and sophisticated efforts by a hostile party injecting information into the network.

We recommend that future protocol standards should recognize the distinct requirements for these three levels of authentication. Furthermore, these protocols should handle the difficult problems that occur when devices supporting different levels of authentication co-exist and interact on the same network.

This paper does not discuss uses of encryption to protect data confidentiality only. If one is only concerned with confidentiality, then a simple facility for encrypting and decrypting documents at the user's workstation may be adequate. This requires the user to remember the encryption key – with potentially disasterous results if it is forgotten. Once a document has been encrypted, it can be safely transmitted over the network. However, if it is to be read by someone else, the problem of distributing and managing different encryption keys can easily become a nightmare for the users. If one wants software to support the key management, then that software must have a very secure way of authenticating its users. Encryption used for full authentication of data interchanges can easily be designed so that it also prevents intruders from reading the data during transimission.

9. Constraints on the Use of Encryption in Office Applications

Encryption has long been used for military and diplomatic communications. Recently, it has come into widespread use by financial institutions to protect electronic funds transfers. There have been expectations that other applications of encryption would spread rapidly through the business world. These expectations increased when the National Bureau of Standards (NBS) established a standard data encryption algorithm [10] and when low-cost hardware implementing the standard became available. Unfortunately, successful applications of encryption have not spread as rapidly as expected. One reason is that there are many system design problems that must be solved to make encryption an effective and practical part of a full security system.

The constraints of an office system mean that few existing applications of encryption serve as a model for applying encryption in office systems. Few, if any, current applications of encryption have all the following characteristics:

(1) The communication system is a complex, frequently changing internetwork.

(2) Management of the encryption keys is not a serious burden for the system administrator or the individual user.

(3) Users do not have to be aware of details about how the encryption works, and the system is robust in the face of many user oversights and errors.

(4) The devices attached to the network are diverse, and not all of them can be modified to participate in the plan for using encryption.

(5) There is a low tolerance for increased costs – including added system costs attributable to the encryption.

There is a substantial body of recent literature about system designs for authenticated communications that promise to eliminate or reduce at least the first three of the above impediments. The design that is most relevant to our environment is documented in [12] with related work in [1, 2, 3, 4] and elsewhere. This design is oriented toward distributed communications. It calls for a network service that automatically manages encryption keys, and it hides most of the encryption details from the end user. The key features of the design are summarized in the following section.

10. Design Issues for Secure Authentication

Full authentication of communication between a workstation A and a service B can be achieved if A and B encrypt their communication using a key that is known only by A and B.[1] The problem is to get the same key to both A and B without revealing it to anyone else and without creating too much extra work for the user or the system administrator. This is accomplished by introducing a trusted authentication service whose role is to generate and distribute encryption keys that can be used for a limited time by A and B to carry on a "conversation." Like the clearinghouse, the authentication service should be a distributed service that supports redundant storage of its data and automatically propagates updates of those data.

When workstation A wants to set up authenticated communications with service B, it first asks the authentication service for a conversation key that A and B can use to encrypt their communications with each other. Of course, the conversation key itself has to be encrypted when it is sent to A and to B, using keys known only by A and B respectively (and known by the the authentication service, of course). The authentication service need not communicate directly with B but can rely on A to forward the right information to B (since it is encrypted so only B can decrypt it). We refer to the information that A forwards to B as A's *credentials*. These credentials contain at least A's user name, the conversation key, and a time stamp - all encrypted so only B can decrypt them.[2] When fully implemented as described in [12], this scheme guarantees to B that subsequent communications encrypted under the conversation key did originate from A and guarantees to A that only B will decrypt the communications successfully. Corresponding guarantees apply for B's responses to A.

Automatic key distribution using an authentication service goes a long way toward making fully authenticated communications practical for office systems. However, further discussion is needed regarding the last two of the practical design constraints listed in the preceding section.

The constraint that not all devices on the network will be able to participate in encrypted communications is especially troublesome. It would be much better to have one scheme for authentication, not two or three. Unfortunately, when one has an open network and wants to encourage participation by others in the network architecture, one simply cannot require an encryption capability at every device. Some nodes may have encryption implemented in hardware; others, in software; still others, not at all. Standardizing on a level low enough so that inexpensive devices can be used implies a rather weak security system. Choosing a higher level increases the cost

[1] This discussion assumes the use of the NBS data encryption standard where the encryption and decrpytion keys are identical. A scheme fairly similar to this that uses a public key encryption algorithm is also discussed in [12].

[2] For a limited time A can cache the credentials it receives from the authentication service and reuse them in several sessions with B (e.g., the workstation could use the cache mentioned in section 5). This would allow workstations to interact with the services they use most frequently even if the authentication service is unavailable for a time. It also speeds the establishment of communication.

of the entire system. Not only the backbone equipment cost is involved; there is also a substantial user acceptance factor. For example, many of our users value the ability to dial in for their electronic mail using ordinary teletypewriter terminals or home computers. With voice mail systems, even the telephone has to be considered. Some day perhaps all such devices will be equipped with encryption chips, but that is certainly many years away. Our systems and protocols must deal with the mixed environment of today.

The cost constraints of most office system applications also make it especially difficult to introduce an encryption capability. It is true that low cost encryption chips are available. On the other hand, one must consider the costs for software, testing, documentation, user training, and customer support. The sum of all the costs attributable to security must remain a small fraction of the total system cost.

11. The Need for Three Authentication Options

Changes in the authentication arrangement propagate as changes in the protocols of all services that authenticate their users. Clearly, once encryption is introduced into the authentication arrangements, one has yet another source of incompatibility between different office systems. Incompatible uses of encryption will make it much harder to build devices that support the protocols of two or more different office systems, and it will greatly complicate the construction of gateways between systems. It is clearly desirable to work toward industry standard protocols for any use of encryption in office systems. The NBS Data Encryption Standard provides an industry standard algorithm; however, many design choices remain about how to use this algorithm in the context of office systems.

Ideally, one would like to have a single fully secure protocol for authentication that is used by all devices in the internetwork. This would avoid the system complexities that arise when multiple authentication options have to co-exist. Unfortunately, a fully secure protocol for authentication implies that every device that participates in the the office system would have to have hardware for encryption. Despite the decreasing costs of such hardware, this is just not feasible in the foreseeable future.

We believe that it is a reasonable goal to develop protocols for authentication that provide three options:

1. The lowest level option would not require any use of encryption and would support authentication in the form that was discussed in the first half of this paper.
2. An intermediate option would provide a migration path between the first and third options and would reduce the number of cases where the first and third options must co-exist. This option is based on the observation that it is much easier for an intruder on the network to watch for a password (or other data that can be used later to impersonate someone) than it is to actively and intelligently modify information as it is being transmitted. This observation is interesting because one can protect against the more likely threats with a very minimal use of encryption.
3. The most secure option would support full authentication of entire data transmissions. This requires an encryption of all the data transmitted and thus is unwieldy without special hardware for encryption. A secondary option in this case might allow either

encrypted transmission of the data or plaintext transmission of the data with an added cryptographic checksum (digital signature).

Under the second option, users still verify their identity during a secure interaction with an authentication service and they still receive a conversation key and credentials for use in interacting with a service. However, instead of encrypting the entire communication with the service, only one small block of data is encrypted to accompany each communication. The block of data to be encrypted must change for each communication - it could contain a sequence number and/or a date and time. This authentication option avoids any plaintext transmission of passwords and is safe against threats other than jamming and real time retransmission after partial modification of the transmission.[3] This option allows devices that do not have hardware for encryption to participate in a reasonably secure authentication arrangement - the amount of encryption required for this option can be implemented efficiently in software on most processors.

Technically, a software implementation does not conform with the NBS Data Encryption Standard; however, for existing devices that do not have encryption hardware, the alternative is plaintext transmission of passwords.

12. Issues Concerning Co-existence of Authentication Options

While network protocols need to allow several options for authentication, management of a particular installation may choose to use only one of these options. In many offices, there is little point to going beyond the first option until other security procedures have been implemented. Another management option is to maintain strong security by excluding low-capability devices from the network and forcing all devices to use full encryption. However, many offices, can be characterized as having a relatively small amount of information that is very sensitive, and that information has to coexist in the same internetwork with a vast sea of information that does not need as much security. In this environment, workstations with different capabilities may all need to communicate with a common set of services. The weakness of one workstation should not compromise the security of other nodes.

This set of design objectives has several implications:

(1) Someone who alternates between different workstations may need to have two passwords for use with different kinds of workstations. It does little good for one workstation to use the stronger authentication options if the user sometimes exposes the same password by typing it into a device which transmits it in the clear.

[3] This scheme might be further improved if the block of data to be encrypted contained an effective checksum computed over the remainder of the data transmission; however, we have failed to find any checksumming function which can be computed a couple orders of magnitude faster than is required for a cryptographic checksum and still provides some demonstrable increase in security. Note that the use of "exclusive or" as the checksumming function would provide no increase in security since it would be easy to make compensating changes in the data with an assurance that the encrypted checksum would still be valid.

(2) Servers should have a notion of the different levels of protection and support the protocol variations this implies. They must remember which level is in use throughout a user's session.

(3) The third implication affects access control. The new goal is for access control to take into consideration the strength of the authentication arrangement being used in a particular session, so that sensitive documents can be protected from being accessed by anyone having only a "weak" password (one that is exposed to potential intruders on the net).

(4) Finally, the authentication service must support the notion of varying levels of security strength in the conversations it arranges. Its responsibilities include insuring that each party knows the security strength of its partner's authentication. It should support multiple passwords for each user (forcing a user to adopt multiple identities is an unpleasant alternative). Basing protection on workstation type alone is inadequate: a user having only a "weak" password should be able to use a "strong" workstation and be granted only "weak" privileges by servers.

When users have two passwords to use on different machines, one can anticipate that they will sometimes mistakenly use the wrong one. To make security robust in the face of likely user errors, one should try to confine the security exposure from this error. The problem occurs when a "strong" password (intended to be used at workstations that support options 2 or 3) is entered at a "weak" workstation (one which exposes passwords to intruders on the net). A partial solution is to have nodes without encryption perform a simple hashing transform on passwords before any transmission. This transform would not make option 1 any more secure since it would be easy to find one password or another that transforms into any given hash value, and with option 1 any such password will do. However, if the password was really intended for use with option 2 or 3, then the hash value does not provide enough information to determine the user's exact password. For this to be effective, strong passwords must be selected randomly from a large space of possible values – a characteristic which would be highly desirable in any case.

Carrying this idea one step further gives the option of returning to a single password per user, while still having more than one level of authentication. A mechanism along these lines was designed for the protocol used by the experimental file service described in [6]. The scheme depends on the password having far more bits of randomness in it than the hash code does; it gives away a few bits of protection in order to achieve the convenience of a single password.

13. Other Authentication Design Issues

One issue is exactly what entity is to be authenticated. Above, we assumed that the user's password was to be authenticated. It is equally possible to have a secret key associated with a workstation and authenticate the workstation. By combining the user's password with a key from the workstation in some fixed way, it is possible to authenticate both the user and the workstation in use (see [15]). The benefit in authenticating workstations is that one can physically secure a room containing privileged machines. For example, an authorized user who could access sensitive information only in an open room in front of colleagues might be less likely to do something irregular.

The authentication service design must handle authentication during user logon at a workstation. A protocol is needed to convince the software running in the workstation that the user is authentic. Note that if spoofing is to be countered, the protocol must deal with the situation in which the person presenting the password also controls a node pretending to provide an authentication service. One way to accomplish this is for a legitimate authentication service to know about certain information embedded in the workstation, information that users of the workstation cannot uncover. This information can be the basis of authenticated communication between the authentication service and software on the workstation.

Another set of design decisions involves one service calling on another in order to do work on behalf of an end user. The chain of calls can involve several intermediary services between the user and the final service. The issue is: what privileges does the last service grant to the request? That of the end user? That of the next-to-last service? Do the intermediate services even have their own sets of privileges ascribed to them? If so, do we want to take their minimum? Does one have to grant the intermediary services all one's privileges, or can the activities they may do on one's behalf be circumscribed? Reliable, secure protocols must be devised to implement whatever policy emerges from answering these questions.

Another issue that we have not dealt with here concerns the extent to which one must trust the software residing in various nodes on the network. For example, software at a workstation could, in principal, remember user passwords and make them available to a subsequent user of the same workstation. Problems like this are more likely to result from deliberate, sophisticated efforts rather than from accidental bugs. See [7,8] for two surveys of techniques that deal with such problems.

14. Conclusion

There are many possible approaches to authenticating users in a small office system. However, as the system grows larger and more diverse, many of the possible authentication schemes either will become a large administrative burden or will sometimes be the cause of the entire system being unavailable. The approach chosen in the Xerox 8000 Series products was designed to avoid these two problems, and to have an evolutionary path open to ever increasing levels of security.

Acknowledgement

Many colleagues contributed to the design concepts discussed in this paper and to the initial implementation. Dorothy Andrews, Bob Ayers, Marney Beard, Andrew Birrell, Yogen Dalal, Bob Lyon, John Maloney, Dave Redell, and Michael Schroeder deserve special mention.

References

1. D. Branstad, "Encryption Protection in Computer Data Communications." *Proc. Fourth Data Communications Symp.*, ACM, October, 1975.

2. D. Branstad, "Security Aspects of Computer Networks." *Proc. AIAA Comptr. Network Syst. Conf.*, April, 1973.

3. G.D. Cole, "Design Alternatives for Computer Network Security." *NBS SP-500-21, vol. 1*, January, 1978.

4. D.E. Denning and G.M. Sacco, "Timestamps in Key Distribution Protocols." *Comm. ACM,* 24, 8, August , 1981.

5. Digital Equipment Corp., Intel Corp, and Xerox Corp., "The Ethernet, a Local Area Network: Data Link Layer and Physical Layer Specifications." Version 1.0, September, 1980.

6. J. Israel, J. Mitchell, and H. Sturgis, "Separating Data from Function in a Distributed File System." *Operating Systems* (Proceedings of the Second International Symposium on Operating Systems Theory and Practice), D. Lanciaux, ed.; North-Holland, 1979.

7. T. A. Linden, "Operating System Structures to Support Security and Reliable Software," *ACM Computing Surveys*, 8,4, Dec. 1976.

8. T. A. Linden, "Protection for Reliable Software," *System Reliability and Integrity,* Infotech State of the Art Report, Infotech International Ltd., Maidenhead, UK, 1978.

9. D. E. Lipkie, Steven R. Evans, John K. Newlin, and Robert L. Weissman, "Star Graphics: An Object-Oriented Implementation." *Computer Graphics*. 16,3: pp. 115-124; July, 1982.

10. National Bureau of Standards, *Data Encryption Standard. FIPS Pub. 46,* NBS, Washington, D.C., January, 1977.

11. National Bureau of Standards, *Guidelines on User Authentication Techniques for Computer Network Access Control, FIPS Pub. 83,* NBS, Washington, D.C. Sept. 1980.

12. R.M. Needham and M.D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers." *Comm. ACM,* 21, 12, December, 1978.

13. D.C. Oppen and Y.K. Dalal, "The Clearinghouse: a Decentralized Agent for Locating Named Objects in a Distributed Environment." *OPD-T8103,* Xerox, Palo Alto, Cal., October, 1981.

14. R. Purvy, J. Farrell, and P. Klose, "The Design of Star's Records Processing," *Proc. ACM-SIGOA Conf. on Office Automation Systems,* June 1982.

15. M.E. Smid, "A Key Notarization System for Computer Networks." *NBS SP-500-54, vol. 1,* October, 1979.

16. D.C. Smith, E. Harslem, C. Irby, and R. Kimball, "The Star User Interface, An Overview," *AFIPS Conf. Proc. of NCC,* June, 1982.

17. D.C. Smith, C. Irby, R. Kimball, and W. Verplank, "Designing the Star User Interface," *Byte,* April, 1982.

18. Xerox Corp., "Courier: the Remote Procedure Call Protocol." *XSIS 038112,* Xerox, Stamford, Conn., December, 1981.

19. Xerox Corp., "Internet Transport Protocols." *XSIS 028112,* Xerox, Stamford, Conn., December, 1981.

Grapevine: An Exercise in Distributed Computing

Andrew D. Birrell, Roy Levin,
Roger M. Needham, and Michael D. Schroeder

Xerox Palo Alto Research Center

Grapevine is a multicomputer system on the Xerox research internet. It provides facilities for the delivery of digital messages such as computer mail; for naming people, machines, and services; for authenticating people and machines; and for locating services on the internet. This paper has two goals: to describe the system itself and to serve as a case study of a real application of distributed computing. Part I describes the set of services provided by Grapevine and how its data and function are divided among computers on the internet. Part II presents in more detail selected aspects of Grapevine that illustrate novel facilities or implementation techniques, or that provide insight into the structure of a distributed system. Part III summarizes the current state of the system and the lessons learned from it so far.

CR Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications, distributed databases*; C.4 [**Performance of Systems**]—*reliability, availability and serviceability*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; H.2.4 [**Database Management**]: Systems—*distributed systems*; H.2.7 [**Database Management**]: Database Administration; H.4.3 [**Information Systems Applications**]: Communications Applications—*electronic mail*

General Terms: Design, Experimentation, Reliability

Part I. Description of Grapevine

1. Introduction

Grapevine is a system that provides message delivery, resource location, authentication, and access control ser-

Authors' Present Addresses: Andrew D. Birrell, Roy Levin, and Michael D. Schroeder, Xerox Palo Alto Research Center, Computer Science Laboratory, 3333 Coyote Hill Road, Palo Alto, CA 94304; Roger M. Needham, University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge, CB2 3QG, United Kingdom.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0001-0782/82/0400-0260 $00.75.

vices in a computer internet. The implementation of Grapevine is distributed and replicated. By *distributed* we mean that some of the services provided by Grapevine involve the use of multiple computers communicating through an internet; by *replicated* we mean that some of the services are provided equally well by any of several distinct computers. The primary use of Grapevine is delivering computer mail, but Grapevine is used in many other ways as well. The Grapevine project was motivated by our desire to do research into the structure of distributed systems and to provide our community with better computer mail service.

Plans for the system were presented in an earlier paper [5]. This paper describes the completed system. The mechanisms discussed below are in service supporting more than 1500 users. Designing and building Grapevine took about three years by a team that averaged two to three persons.

1.1 Environment for Grapevine

Figure 1 illustrates the kind of computing environment in which Grapevine was constructed and operates. A large internet of this style exists within the Xerox Corporation research and development community. This internet extends from coast-to-coast in the U.S.A. to Canada, and to England. It contains over 1500 computers on more than 50 local networks.

Most computing is done in personal *workstation* computers [12]; typically each workstation has a modest amount of local disk storage. These workstations may be used at different times for different tasks, although generally each is used only by a single individual. The internet connecting these workstations is a collection of Ethernet local networks [6], gateways, and long distance links (typically telephone lines at data rates of 9.6 to 56 Kbps). Also connected to the internet are *server* computers that provide shared services to the community, such as file storage or printing.

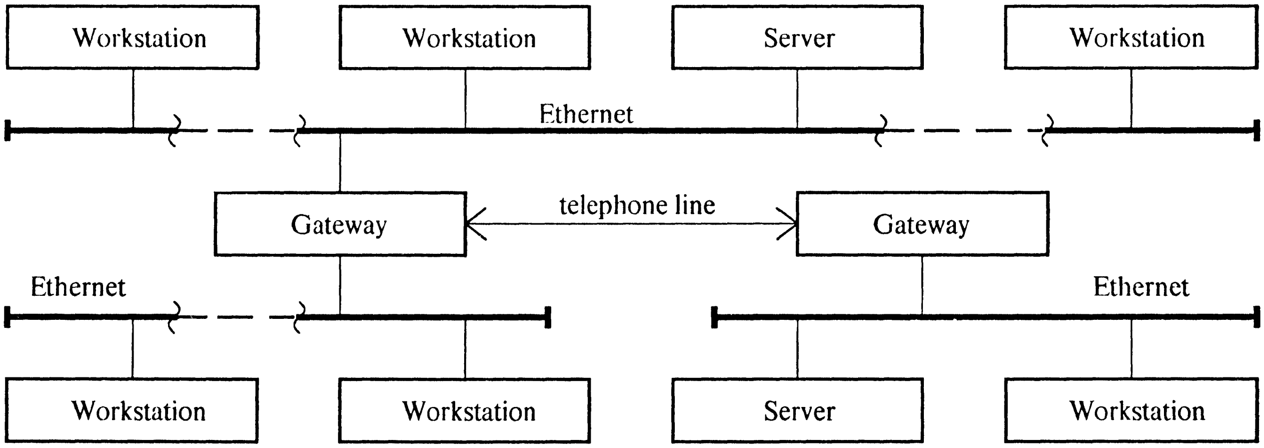
Protocols already exist for communicating between computers attached to the internet [11]. These protocols provide a uniform means for addressing any computer attached to any local network in order to send individual packets or to establish and use byte streams. The individual packets are typically small (up to 532 bytes), and are sent unreliably (though with high probability of success) with no acknowledgment. The byte stream protocols provide reliable, acknowledged, transmission of unlimited amounts of data [1].

1.2 Services and Clients

Our primary consideration when designing and implementing Grapevine was its use as the delivery mechanism for a large, dispersed computer mail system. A computer mail system allows a group of human users to exchange messages of digital text. The sender prepares a message using some sort of text editing facility and names a set of recipients. He then presents the message to a delivery mechanism. The delivery mechanism moves the message from the sender to an internal buffer for each recipient, where it is stored along with other messages for that recipient until he wants to receive them. We call the buffer for a recipient's messages an *inbox*. When ready, the recipient can read and process the messages in his inbox with an appropriate text display program. The recipient names supplied by the sender may identify *distribution lists*: named sets of recipients, each of whom is to receive the message. We feel that computer mail is both an important application of distributed computing and a good test bed for ideas about how to structure distributed systems.

Buffered delivery of a digital message from a sender to one or more recipients is a mechanism that is useful in many contexts: it may be thought of as a general communication protocol, with the distinctive property that the recipient of the data need not be available at the time the sender wishes to transmit the data. Grapevine separates this message delivery function from message creation and interpretation, and makes the delivery function available for a wider range of uses. Grapevine does not interpret the contents of the messages it transports. Interpretation is up to the various message manipulation programs that are software *clients* of Grapevine. A client

Fig. 1. An Example of a Small Internet.



program ımplementing a computer mail user ınterface will interpret messages as interpersonal, textual memos. Other clients might interpret messages as print files, digital audio, software, capabilities, or data base updates.

Grapevine also offers *authentication*, *access control*, and *resource location* services to clients. For example, a document preparation system might use Grapevine's resource location service to find a suitable printing server attached to the internet (and then the message delivery service to transfer a document there for printing) or a file server might use Grapevine's authentication and access control services to decide if a read request for a particular file should be honored.

Grapevine's clients run on various workstations and server computers attached to the internet. Grapevine itself is implemented as programs running on server computers dedicated to Grapevine. A client accesses the services provided by Grapevine through the mediation of a software package running on the client's computer. The Grapevine computers cooperate to provide services that are distributed and replicated.

2. Design Goals

We view distributed implementation of Grapevine both as a design goal and as the implementation technique that best meets the other design goals. A primary motivation for the Grapevine project was implementing a useful distributed system in order to understand some system structures that met a real set of requirements. Once we chose message delivery as the functional domain for the project, the following specific design goals played a significant role in determining system structure.

Grapevine makes its services available to many different clients. Thus, it should make no assumptions about message content. Also, the integrity of these services should not in any way depend on correctness of the clients. Though the use of an unsatisfactory client program will affect the service given to its user, it should not affect the service given to others. These two goals help determine the distribution of function between Grapevine and its clients.

Two goals relate to Grapevine's reliability properties. First, a user or client implementor should feel confident that if a message is accepted for delivery then it will either be made available to its intended recipients or returned with an indication of what went wrong. The delivery mechanism should meet this goal in the face of user errors (such as invalid names), client errors (such as protocol violations), server problems (such as disk space congestion or hardware failures), or communication difficulties (such as internet link severance or gateway crashes). Second, failure of a single Grapevine server computer should not mean the unavailability of the Grapevine services to any client.

The typical interval from sending a message to its arrival in a recipient's inbox should be a few minutes at most. The typical interactive delay perceived by a client program when delivering or receiving a message should be a few seconds at most. Since small additions to delivery times are not likely to be noticed by users, it is permissible to improve interactive behavior at the expense of delivery time.

Grapevine should allow decentralized administration. The users of a widespread internet naturally belong to different organizations. Such activities as admission of users, control of the names by which they are known, and their inclusion in distribution lists should not require an unnatural degree of cooperation and shared conventions among administrations. An administrator should be able to implement his decisions by interacting directly with Grapevine rather than by sending requests to a central agency.

Grapevine should work well in a large size range of user communities. Administrators should be able to implement decentralized decisions to adjust storage and computing resources in convenient increments when the shape, size, or load patterns of the internet change.

Grapevine should provide authentication of senders and recipients, message delivery secure from eavesdropping or content alteration, and control on use and modification of its data bases.

3. Overview

3.1 Registration Data Base

Grapevine maintains a *registration data base* that maps names to information about the users, machines, services, distribution lists, and access control lists that those names signify. This data base is used in controlling the message delivery service; is accessed directly for the resource location, access control, and authentication services; and is used to configure Grapevine itself. Grapevine also makes the values in the data base available to clients to apply their own semantics.

There are two types of entries in the registration data base: *individual* and *group*. We call the name of an entry in the registration data base an *RName*.

A group entry contains a set of RNames of other data base entries, as well as additional information that will be discussed later. Groups are a way of naming collections of RNames. The groups form a naming network with no structural constraints. Groups are used primarily as distribution lists: specifying a group RName as a recipient for a message causes that message to be sent to all RNames in that group, and in contained groups. Groups also are used to represent access control lists and collections of like resources.

An individual entry contains an *authenticator* (a password), a list of *inbox sites*, and a *connect site*, as well as additional information that will be discussed later. The inbox site list indicates, in order of preference, the Grapevine computers where the individual's messages may be buffered. The way these multiple inboxes are used is discussed in Sec. 4.2. The connect site is an internet address for making a connection to the individual. Thus, an individual entry specifies ways of authenticating the identity of and communicating with—by message delivery or internet connection—the named entity. Individuals are used to represent human users and servers, in particular the servers that implement Grapevine. Usually the connect site is used only for individuals that represent servers. Specifying an individual RName (either a human or a server) as a recipient of a message causes the message to be forwarded to and buffered in an inbox for that RName.

3.2 Functions

Following is a list of the functions that Grapevine makes available to its clients. Responses to error conditions are omitted from this description. The first three functions constitute Grapevine's *delivery service.*

Accept message:

[sender, password, recipients, message-body] → ok

The client presents a message body from the sender for delivery to the recipients. The sender must be RName of an individual and the password must authenticate that individual (see below). The recipients are individual and group RNames. The individuals correspond directly to message recipients while the groups name distribution lists. After Grapevine acknowledges acceptance of the message the client can go about its other business. Grapevine then expands any groups specified as recipients to produce the complete set of individuals that are to receive the message and delivers the message to an inbox for each.

Message polling:

[individual] → {empty, nonempty}

Message polling is used to determine whether an individual's inboxes contain messages that can be retrieved. We chose not to authenticate this function so it would respond faster and load the Grapevine computers less.

Retrieve messages:

[name, password] → sequence of messages → ok

The client presents an individual's name and password. If the password authenticates the individual then Grapevine returns all messages from the corresponding inboxes. When the client indicates "ok," Grapevine erases these messages from those inboxes.

Grapevine's authentication, access control, and resource location services are implemented by the remaining functions. These are called the *registration service,* because they are all based on the registration data base.

Authenticate:

[individual, password] → {authentic, bogus}

The authentication function allows any client to determine the authenticity of an individual. An individual/password combination is authentic if the password matches the one in the individual's registration data base entry.[1]

Membership:

[name, group] → {in, out}

Grapevine returns an indication of whether the name is included in the group. Usually the client is interpreting the group as an access control list. There are two forms of the membership function. One indicates direct membership in the named group; the other indicates membership in its closure.

Resource location:

[group] → members
[individual] → connect site
[individual] → ordered list of inbox sites

The first resource location function returns a group's membership set. If the group is interpreted as a distribution list, this function yields the individual recipients of a message sent to the distribution list; if the group is interpreted as the name of some service, this function yields the names of the servers that offer the service. For a group representing a service, combining the first function with the second enables a client to discover the internet addresses of machines offering the service, as described in Sec. 5. The third function is used for message delivery and retrieval as described in Sec. 4.

Registration data base update and inquiry:

There are various functions for adding and deleting names in the registration data base, and for inspecting and changing the associated values.

3.3 Registries

We use a partitioned naming scheme for RNames. The partitions serve as the basis for dividing the administrative responsibility, and for distributing the data base among the Grapevine computers. We structure the name space of RNames as a two-level hierarchy. An RName is a character string of the form *F.R* where *R* is a *registry* name and *F* is a name within that registry. Registries can correspond to organizational, geographic, or other arbitrary partitions that exist within the user community. A two-level hierarchy is appropriate for the size and organizational complexity of our user community, but a larger community or one with more organizational diversity would cause us to use a three-level scheme. Using more levels would not be a fundamental change to Grapevine.

[1] This password-based authentication scheme is intrinsically weak. Passwords are transmitted over the internet as clear-text and clients of the authentication service see individuals' passwords. It also does not provide two-way authentication: clients cannot authenticate servers. The Grapevine design includes proper encryption-based authentication and security facilities that use Needham and Schroeder's protocols [9] and the Federal Data Encryption Standard [8]. These better facilities, however, are not implemented yet.

3.4 Distribution of Function

As indicated earlier, Grapevine is implemented by code that runs in dedicated Grapevine computers, and by code that runs in clients' computers. The code running in a Grapevine computer is partitioned into two parts, called the *registration server* and the *message server.* Although one registration server and one message server cohabit each Grapevine computer, they should be thought of as separate entities. (Message servers and registration servers communicate with one another purely by internet protocols.) Several Grapevine computers are scattered around the internet, their placement being dictated by load and topology. Their registration servers work together to implement the registration service. Their message servers work together to implement the delivery service. As we will see in Secs. 4 and 5, message and registration services are each clients of the other.

The registration data base is distributed and replicated. Distribution is at the grain of a registry; that is, each registration server contains either entries for all RNames in a registry or no entries for that registry. Typically no registration server contains all registries. Also, each registry is replicated in several different registration servers. Each registration server supports, by publicly available internet protocols, the registration functions described above for names in the registries that it contains. Any server that contains the data for a registry can accept a change to that registry. That server takes the responsibility for propagating the change to the other relevant servers.

Any message server is willing to accept any message for delivery, thus providing a replicated mail submission service. Each message server will accept message polling and retrieval requests for inboxes on that server. An individual may have inboxes on several message servers, thus replicating the delivery path for the individual.

If an increase in Grapevine's capacity is required to meet expanding load, then another Grapevine computer can be added easily without disrupting the operation of existing servers or clients. If usage patterns change, then the distribution of function among the Grapevine computers can be changed for a particular individual, or for an entire registry. As we shall see later this redistribution is facilitated by using the registration data base to describe the configuration of Grapevine itself.

The code that runs in clients' machines is called the *GrapevineUser package.* There are several versions of the GrapevineUser package: one for each language or operating environment. Their function and characteristics are sufficiently similar, however, that they may be thought of as a single package. This package has two roles: it implements the internet protocols for communicating with particular Grapevine servers; and it performs the resource location required to choose which server to contact for a particular function, given the data distribution and server availability situation of the moment. GrapevineUser thus makes the multiple Grapevine servers look like a single service. A client using the GrapevineUser package never has to mention the name or internet address of a particular Grapevine server. The GrapevineUser package is not trusted by the rest of Grapevine. Although an incorrect package could affect the services provided to any client that uses it, it cannot affect the use of Grapevine by other clients. The implementation of Grapevine, however, includes engineering decisions based on the known behavior of the GrapevineUser package, on the assumption that most clients will use it or equivalent packages.

3.5 Examples of How Grapevine Works

With Fig. 2 we consider examples of how Grapevine works. If a user named *P.Q* were using workstation 1 to send a message to *X.Y.*, then events would proceed as follows. After the user had prepared the message using a suitable client program, the client program would call the delivery function of the GrapevineUser package on workstation 1. GrapevineUser would contact some registration server such as *A* and use the Grapevine resource location functions to locate any message server such as *B*; it would then submit the message to *B*. For each recipient, *B* would use the resource location facilities, and suitable registration servers (such as *A*) to determine that recipient's best inbox site. For the recipient *X.Y*, this might be message server *C*, in which case *B* would forward the message to *C*. *C* would buffer this message locally in the inbox for *X.Y*. If the message had more recipients, the message server *B* might consult other registration servers and forward the message to multiple message servers. If some of the recipients were distribution lists, *B* would use the registration servers to obtain the members of the appropriate groups.

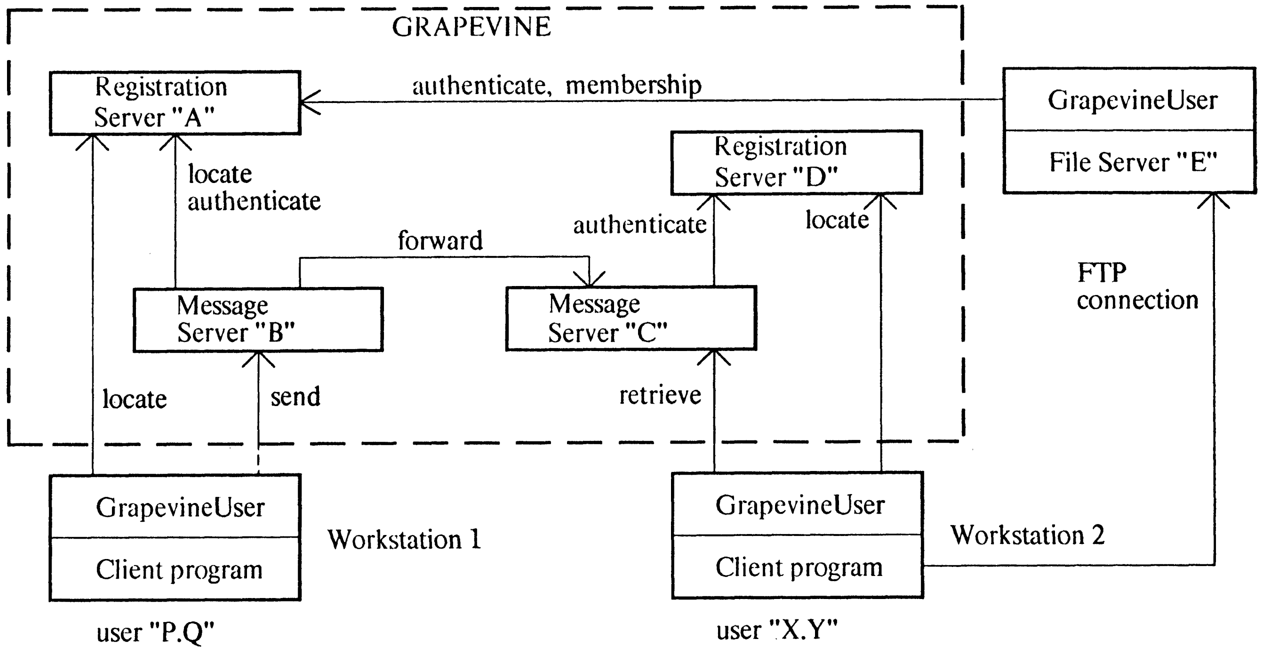
When *X.Y* wishes to use workstation 2 to read his mail, his client program calls the retrieval function of the GrapevineUser package in workstation 2. GrapevineUser uses some registration server (such as *D*) that contains the *Y* registry to locate inbox sites for *X.Y*, then connects to each of these inbox sites to retrieve his messages. Before allowing this retrieval, *C* uses a registration server to authenticate *X.Y*.

If *X.Y* wanted to access a file on the file server *E* through some file transfer program (FTP) the file server might authenticate his identity and check access control lists by communicating with some registration server (such as *A*).

3.6 Choice of Functions

The particular facilities provided by Grapevine were chosen because they are required to support computer mail. The functions were generalized and separated so other applications also could make use of them. If they want to, the designers of other systems are invited to use the Grapevine facilities. Two important benefits occur, however, if Grapevine becomes the *only* mechanism for authentication and for grouping individuals by organization, interest, and function. First, if Grapevine per-

Fig. 2. Distribution of Function.



forms all authentications, then users have the same name and password everywhere, thus simplifying many administrative operations. Second, if Grapevine is used everywhere for grouping, then the same group structure can be used for many different purposes. For example, a single group can be an access control list for several different file servers and also be a distribution list for message delivery. The groups in the registration data base can capture the structure of the user community in one place to be used in many ways.

4. Message Delivery

We now consider the message delivery service in more detail.

4.1 Acceptance

To submit a message for delivery a client must establish an internet connection to a message server; any operational server will do. This resource location step, done by the GrapevineUser package, is described in Sec. 5. Once such a connection is established, the GrapevineUser package simply translates client procedure calls into the corresponding server protocol actions. If that particular message server crashes or otherwise becomes inaccessible during the message submission, then the GrapevineUser package locates another message server (if possible) and allows the client to restart the message submission.

The client next presents the RName and password of the sender, a *returnTo* RName, and a list of recipient RNames. The message server authenticates the sender by using the registration service. If the authentication fails, the server refuses to accept the message for delivery. Each recipient RName is then checked to see if it matches an RName in the registration data base. All invalid recipient names are reported back to the client. In the infrequent case that no registration server for a registry is accessible, all RNames in that registry are presumed for the time being to be valid. The server constructs a *property list* for the message containing the sender name, returnTo name, recipient list, and a *postmark*. The postmark is a unique identification of the message, and consists of the server's clock reading at the time the message was presented for delivery together with the server's internet address. Next, the client machine presents the *message body* to the server. The server puts the property list and message body in reliable storage, indicates that the message is accepted for delivery, and closes the connection. The client may cancel delivery anytime prior to sending the final packet of the message body, for example, after being informed of invalid recipients.

Only the property list is used to direct delivery. A client might obtain the property values by parsing a text message body and require that the parsed text be syntactically separated as a "header," but this happens before Grapevine is involved in the delivery. The property list stays with the message body throughout the delivery process and is available to the receiving client. Grapevine guarantees that the recipient names in the property list were used to control the delivery of the message, and that the sender RName and postmark are accurate.

4.2 Transport and Buffering

Once a message is accepted for delivery, the client may go about its other business. The message server, however, has more to do. It first determines the complete list of individuals that should receive the message by recursively enumerating groups in the property list. It obtains from the registration service each individual's inbox site list. It chooses a destination message server for each on the basis of the inbox site list ordering and its opinion of the present accessibility of the other message servers. The individual names are accumulated in *steering lists*, one for each message server to which the message should be forwarded and one for local recipients. The message server then forwards the message and appropriate steering list to each of the other servers, and places the message in the inboxes for local recipients. Upon receiving a forwarded message from another server, the same algorithm is performed using the individuals in the incoming steering list as the recipients, all of which will have local inboxes unless the registration data base has changed. The message server stores the property list and body just once on its local disk and places references to the disk object in the individual's inboxes. This sharing of messages that appear in more that one local inbox saves a considerable amount of storage in the server.[2]

With this delivery algorithm, messages for an individual tend to accumulate at the server that is first on the inbox site list. Duplicate elimination, required because distribution lists can overlap, is achieved while adding the message into the inboxes by being sure never to add a message if that same message, as identified by its postmark, was the one previously added to that inbox. This duplicate elimination mechanism fails under certain unusual circumstances such as servers crashing or the data base changing during the delivery process, but requires less computation than the alternative of sorting the list of recipient individuals.

In some circumstances delivery must be delayed, for example, all of an individual's inbox sites or a registry's registration servers may be inaccessible. In such cases the message is queued for later delivery.

In some circumstances delivery will be impossible: for example, a recipient RName may be removed from the registration data base between validation and delivery, or a valid distribution list may contain invalid RNames. Occasionally delivery may not occur within a reasonable time, for example, a network link may be down for several days. In such cases the message server mails a copy of the message to an appropriate RName with a text explanation of what the problem was and who did not get the message. The appropriate RName for this error notification may be the returnTo name recorded in the message's property list or the owner of the distribution list that contained the invalid name, as recorded in a group entry in the registration data base. Even this error notification can fail, however, and ultimately such messages end up in a *dead letter* inbox for consideration by a human administrator.

4.3 Retrieval

To retrieve new messages for an individual, a client invokes the GrapevineUser package to determine the internet addresses of all inbox sites for the individual, and to poll each site for new messages by sending it a single *inbox check* packet containing the individual's RName. For each positive response, GrapevineUser connects to the message server and presents the individual's name and password. If these are authentic, then the message server permits the client to inspect waiting messages one at a time, obtaining first the property list and then the body. When a client has safely stored the messages, it may send an acknowledgment to the message server. On receipt of this acknowledgment, the server discards all record of the retrieved messages. Closing the retrieval connection without acknowledgment causes the message server to retain these messages. For the benefit of users who want to inspect new messages when away from their personal workstation, the message server also allows the client to specify that some messages from the inbox be retained and some be discarded.

There is no guarantee that messages will be retrieved in the order they were presented for delivery. Since the inbox is read first-in, first-out and messages tend to accumulate in the first inbox of an individual's inbox site list, however, this order is highly likely to be preserved. The postmark allows clients who care to sort their messages into approximate chronological order. The order is approximate because the postmarks are based on the time as perceived by individual message servers, not on any universal time.

4.4 Use of Replication in Message Delivery

Replication is used to achieve a highly available message delivery service. Any message server can accept any message for delivery. Complete replication of this acceptance function is important because the human user of a computer mail client may be severely inconvenienced if he cannot present a message for delivery when he wants to. He would have to put the message somewhere and remember to present it later. Fortunately, complete replication of the acceptance function is cheap and simple to provide. Message transport and buffering, however, are not completely replicated. Once accepted for delivery, the crash of a single message server can delay delivery of a particular message until the server is operational again, by temporarily trapping the message in a forwarding queue or an inbox.[3] Allowing multiple inboxes for an individual replicates the delivery path. Unless all servers containing an individual's inbox sites

[2] As another measure to conserve disk storage, messages from an inbox not emptied within seven days are copied to a file server and the references in the inbox are changed to point at these archived copies. Archiving is transparent to clients: archived messages are transferred back through the message server when messages from the inbox are retrieved.

[3] The servers are programmed so any crash short of a physical disk catastrophe will not lose information. Writing a single page to the disk is used as the primitive atomic action.

are inaccessible at once, new messages for that individual can get through. We could have replicated messages in several of an individual's inboxes, but the expense and complexity of doing so does not seem to be justified by the extra availability it would provide. If the immediate delivery of a message is important then its failure to arrive is likely to be noticed outside the system; it can be sent again because a delivery path for new messages still exists.

5. The Registration Data Base

The registration data base is used by Grapevine to name registration servers, message servers, and indeed, registries themselves. This recursive use of the registration data base to represent itself results in an implementation that is quite compact.

5.1 Implementing Registries

One registry in the data base is of particular importance, the registry named GV (for Grapevine). The GV registry is replicated in every registration server; all names of the form **.gv* exist in every registration server. The GV registry controls the distribution and replication of the registration data base, and allows clients to locate appropriate registration servers for particular RNames.

Each registration server is represented as an individual in the GV registry. The connect site for this individual is the internet address where clients of this registration server can connect to it. (The authenticator and inbox site list in the entry are used also, as we will see later.)

The groups of the GV registry are the registries themselves; *reg* is a registry if and only if there exists a group *reg.gv*. The members of this group are the RNames of the registration servers that contain the registry. The GV registry is represented this way too. Since the GV registry is in every registration server, the membership set for *gv.gv* includes the RNames of all registration servers.

5.2 Message Server Names

Each message server is represented as an individual in the MS registry (for message servers). The connect site in this entry is the internet address where clients of this message server can connect to it. (The authenticator and inbox site list in the entry are used also, as we will see later.) It is message server RNames that appear in individuals' inbox site lists.

A group in the MS registry, *Maildrop.ms*, contains as members some subset (usually, but not necessarily, all) of the message server RNames. This group is used to find a message server that will accept a message for delivery.

5.3 Resource Location

The registration data base is used to locate resources. In general, a service is represented as a group in the data base; servers are individuals. The members of the group are the RNames of the servers offering the service; the connect sites of the individuals are the internet addresses for the servers. To contact an instance of the service, a client uses the GrapevineUser package to obtain the membership of the group and then to obtain the connect site of each member. The client then may choose among these addresses, for example, on the basis of closeness and availability.

The GrapevineUser package employs such a resource location strategy to find things in the distributed registration data base. Assume for a moment that there is a way of getting the internet address of some operational registration server, say *Cabernet.gv*. GrapevineUser can find the internet addresses of those registration servers that contain the entry for RName *f.r* by connecting to *Cabernet.gv* and asking it to produce the membership of *r.gv*. GrapevineUser can pick a particular registration server to use by asking *Cabernet.gv* to produce the connect site for each server in *r.gv* and attempting to make a connection until one responds. If *f.r* is a valid name, then any registration server in *r.gv* has the entry for it. At this point GrapevineUser can extract any needed information from the entry of *f.r*, for example, the inbox site list.

Similarly, GrapevineUser can obtain the internet addresses of message servers that are willing to accept messages for delivery by using this resource location mechanism to locate the servers in the group *MailDrop.ms*. Any available server on this list will do.

In practice, these resource location algorithms are streamlined so that although the general algorithms are very flexible, the commonly occurring cases are handled with acceptable efficiency. For example, a client may assume initially that *any* registration server contains the data base entry for a particular name; the registration server will return the requested information or a *name not found* error if this registration server knows the registry, and otherwise will return a *wrong server* error. To obtain a value from the registration data base a client can try any registration server; only in the case of a *wrong server* response does the client need to perform the full resource location algorithm.

We are left with the problem of determining the internet address of some registration server in order to get started. Here it is necessary to depend on some more primitive resource location protocol. The appropriate mechanism depends on what primitive facilities are available in the internet. We use two mechanisms. First, on each local network is a primitive *name lookup server*, which can be contacted by a broadcast protocol. The name lookup server contains an infrequently updated data base that maps character strings to internet addresses. We arrange for the fixed character string *GrapevineRServer* to be entered in this data base and mapped to the internet addresses of some subset of the registration servers in the internet. The GrapevineUser package can get a set of addresses of registration servers using the broadcast name lookup protocol, and send a distinctive packet to each of these addresses. Any accessible registration server will respond to such packets, and the client may then attempt to connect to whichever server responds. Second, we broadcast a distinctive packet on the directly connected local network. Again, any accessible registration server will respond. This second mechanism is used in addition to the first because, when there is a registration server on the local network, the second method gives response faster and allows a client to find a local registration server when the name lookup server is down.

Part II. Grapevine as a Distributed System

6. Updating the Registration Data Base

The choice of methods for managing the distributed registration data base was largely determined by the requirement that Grapevine provide highly available, decentralized administrative functions. Administrative functions are performed by changing the registration data base. Replication of this data base makes high availability of administrative functions possible. An inappropriate choice of the method for ensuring the consistency of copies of the data, however, might limit this potential high availability. In particular, if we demanded that data base updates be atomic across all servers, then most servers would have to be accessible before any update could be started. For Grapevine, the nature of the services dependent on the registration data allows a looser definition of consistency that results in higher availability of the update function. Grapevine guarantees only that the copies of a registration data base entry eventually will have the same new value following an update to one of them. If all servers containing copies are up and can communicate with one another, convergence will occur within a few minutes at most. While an update is converging, clients may detect inconsistency by reading the value of an entry from several servers.

6.1 Representation

The value for each entry in the registration data base is represented mainly as a collection of lists. The membership set of a group is one such list. Each list is represented as two sublists of items, called the *active* sublist and the *deleted* sublist. An item consists of a string and a timestamp. A particular string can appear only once in a list, either in the active or the deleted sublist. A timestamp is a unique identifier whose most significant bits are a time and least significant bits an internet address. The time is that perceived by the server that placed the item in the list; the address is that server's. Because a particular server never includes the same time in two different timestamps, all timestamps from all servers are totally ordered.[4]

Fig. 3. A Group from the Registration Data Base.

Prefix: [1-Apr-81 12:46:45, 3#14], type = group, LaurelImp↑.pa

Remark: (stamp=[22-Aug-80 23:42:14, 3#22]) Laurel Team

Members: Birrell.pa Brotz.pa, Horning.pa, Levin.pa, Schroeder.pa
Stamp-list: [23-Aug-80 17:27:45, 3#22], [23-Aug-80 17:42:35, 3#22], [23-Aug-80 19:04:54, 3#22], [23-Aug-80 19:31:01, 3#22], [23-Aug-80 20:50:23, 3#22]
DelMembers: Butterfield.pa
Stamp-list: [25-Mar-81 14:15:12, 3#14]

Owners: Brotz.pa
Stamp-list: [22-Aug-80 23:43:09, 3#14]
DelOwners: none
Stamp-list: null

Friends: LaurelImp↑.pa
Stamp-list: [1-Apr-81 12:46:45, 3#14]
DelFriends: none
Stamp-list: null

For example, Fig. 3 presents the complete entry for a group named "LaurelImp↑.pa" from the registration data base as it appeared in early April 1981. There are three such lists in this entry: the membership set labeled *members* and two access control lists labeled *owners* and *friends* (see Sec. 6.5 for the semantics of these). There are five current members followed by the corresponding five timestamps, and one deleted member followed by the corresponding timestamp. The owners and friends lists each contain one name and no deletions are recorded from either.

A registration data base entry also contains a version timestamp. This timestamp, which has the same form as an item timestamp, functions as an entry's version number. Whenever anything in an entry changes the version timestamp increases in value, usually to the maximum of the other timestamps in the entry. When interrogating the data base, a client can compare the version timestamp on which it based some cached information with that in the data base. If the cached timestamp matches then the client is saved the expense of obtaining the data base value again and recomputing the cached information. The version timestamp appears in the prefix line in Fig. 3.

6.2 Primitive Operations

Grapevine uses two primitive operations on the lists in a registration data base entry. An *update* operation can add or delete a list item. To add/delete the string *s* to/from a list, any item with the matching string in either of the sublists first is removed. Then a timestamp *t* is produced from the server's internet address and clock. Finally the item (*s*, *t*) is added to the active/deleted sublist. A *merge* operation combines two versions of a complete list to produce a new list with the most recent information from both. Each string that appears in either

[4] The item timestamps in the active sublist are used to imply the preference order for the inbox site list in an individual's entry; older items are preferred. Thus, deleting then adding a site name moves it to the end of the preference ordering.

version will appear precisely once in the result. Each string will be in the active or deleted sublist of the result according to the largest timestamp value associated with that string in either version. That largest timestamp value also provides the timestamp for the string in the result. Keeping the sublists sorted by string value greatly increases the speed with which the merge can be performed. The update and merge operations are atomic in each particular server.

6.3 Propagation

The administrative interface to Grapevine is provided by client software running in an administrator's computer. To make a change to the data of any registry, a client machine uses the resource location facilities of the GrapevineUser package to find and connect to some registration server that knows about that registry. That registration server performs an update operation on the local copy of an entry. Once this update has been completed the client can go about its other business. The server propagates the change to the replicas of the entry in other servers. The means used to propagate the change is Grapevine's delivery service itself, since it gives a guarantee of delivery and provides buffering when other servers are temporarily inaccessible. As described in Sec. 5.1, the members of the group that represent a registry are the registration servers that contain a copy of the data for that registry. Thus, if the change is to an entry in the *reg* registry, the accepting server sends a *change message* to the members, other than itself, of the distribution list *reg.gv*. A change message contains the name of the affected entry and the entire new value for the entry. Registration servers poll their inboxes for new messages every 30 seconds. When a change message is received by a server it uses merge operations to combine the entry from the change message with its own copy.

With this propagation algorithm, the same final state eventually prevails everywhere. When a client makes multiple updates to an entry at the same server, a compatible sequence of entry values will occur everywhere, even if the resulting change messages are processed in different orders by different servers. If two administrators perform conflicting updates to the data base such as adding and removing the same member of a group, initiating the updates at different servers at nearly the same time, it is hard to predict which one of them will prevail; this appears to be acceptable, since the administrators presumably are not communicating with each other outside the system. Also, since copies will be out of step until the change messages are received and acted upon, clients must be prepared to cope with transient inconsistencies. The algorithms used by clients have to be *convergent* in the sense that an acceptable result will eventually ensue even if different and inconsistent versions of the registration data appear at various stages in a computation. The message delivery algorithms have this property. Similar update propagation techniques have been proposed by others who have encountered situations that do not demand instantaneous consistency [10, 13].

If deleted items were never removed from an entry, continued updates would cause the data base to grow. Deleted items are kept in an entry so that out-of-order arrival of change messages involving addition followed by deletion of the same string will not cause the wrong final state. Deleted items also provide a record of recent events for use by human administrators. We declare an upper bound of 14 days upon the clock asynchrony among the registration servers, on message delivery delay, and on administrative hindsight. The Grapevine servers each scan their local data base once a day during inactive periods and purge all deleted items older than the bound.

If a change message gets destroyed because of a software bug or equipment failure, there is a danger that a permanent inconsistency will result. Since a few destroyed messages over the life of the system are inevitable, we must provide some way to resynchronize the data base. At one point we dealt with this problem by detecting during the merge operation whether the local copy of the entry contained information that was missing from the incoming copy. Missing information caused the server to send the result of the merge in a change message to all servers for the registry. While this "anti-entropy" mechanism tended to push the data base back into a consistent state, the effect was too haphazard to be useful; errors were not corrected until the next change to an entry. Our present plan for handling long-term inconsistencies is for each registration server periodically, say once a night, to compare its copy of the data base for a registry with another and to use merges to resolve any inconsistencies that are discovered. The version timestamp in each entry makes this comparison efficient: if two version timestamps are equal then the entries match. Care must be taken that the comparisons span all registration servers for a registry, or else disconnected regions of inconsistency can survive.

6.4 Creating and Deleting Names

The rule that the latest timestamp wins does not deal adequately with the creation of new names. If two administrators connect to two different registration servers at about the same time and try to create a new data base entry with the same name, it is likely that both will succeed. When this data base change propagates, the entry with the latest time timestamp will prevail. The losing administrator may be very surprised, if he ever finds out. Because the later creation could be trapped in a crashed registration server for some time, an administrator could never be sure that his creation had won. For name creation we want the *earlier* creation to prevail. To achieve this effect, we faced the possibility of having to implement one of the known and substantial algorithms for atomic updates to replicated databases [3], which seemed excessive, or of working out a way to make all names unique by appending a hidden timestamp, which seemed complex. We instead fell back on observations about the way in which systems of this nature are used. For each registry there is usually some human-level centralization of name creation, if only to deal with questions of suitability of RNames (not having a junior clerk preempt the RName which everyone would associate with the company president). We consider this centralization enough to solve the problem. Note that there is no requirement that a particular *server* be used for name creation: there is no centralization at the machine level.

Deleting names is straightforward. A deleted entry is marked as such and retained in the data base with a version timestamp. Further updates to a deleted entry are not allowed. Recreation of a deleted entry is not allowed. Sufficiently old deleted entries are removed from the data base by the purging process described in Sec. 6.3.

6.5 Access Controls

An important aspect of system administration is control of who can make which administrative changes. To address this need we associate two access control lists with each group: the *owners list* and the *friends list*. These lists appear in the example entry in Fig. 3. The interpretation of these access lists is the responsibility of the registration server. For ordinary groups the conventions are as follows: membership in the owners list confers permission to add or remove any group member, owner, or friend; membership in the friends list confers permission to add or remove oneself. The names in the owners and friends lists may themselves be the names of groups. Quite separately, clients of the registration server have freedom to use membership in groups for access control purposes about which the registration server itself knows nothing at all. The owners and friends lists on the groups that represent registries are used to control name creation and deletion within registries; these lists also provide the default access controls on groups whose owners list is empty. While we have spent some time adjusting the specific semantics of the Grapevine access controls, we do not present further details here.

6.6 Other Consequences of Changes

The registration servers and message servers are normal clients of one another's services, with no special relationship. Registration servers use message server delivery functions and message servers use the registration service to authenticate clients, locate inboxes, etc. This view, however, is not quite complete. If a change is made to the inbox locations of any individual, notice has to be given to all message servers that are removed, so they can redeliver any messages for that individual buffered in local inboxes. Notice is given by the registration server delivering a message to the message servers in question informing them of the change. Correctness requires that the last registration server that changes its copy of the entry emit the message; we achieve this effect by having each registration server emit such a message as the change is made. A message server receiving an inbox removal message simply redelivers all messages in the affected inbox. Redelivery is sufficient to rebuffer the messages in the proper server. In the system as implemented a simplification is made; inbox removal messages are sent to all inbox sites for the affected individual, not just to removed sites. While this may appear to be wasteful, it is most unusual for any site other than the primary one to have anything to redeliver.

Other registration service clients that use the registration data base to control resource bindings may also desire notification of changes to certain entries. A general notification facility would require allowing a notification list to be associated with any data base entry. Any change to an entry would result in a message being sent to the RNames on its notification list. We have not provided this general facility in the present implementation, but would do so if the system were reimplemented.

7. Finding an Inbox Site

The structure and distribution of the Grapevine registration data base are quite complex, with many indirections. Algorithms for performing actions based on this data base should execute reliably in the face of administrative changes to the registration data base (including those which cause dynamic reconfiguration of the system) and multiple servers that can crash independently. In their full generality such algorithms are expensive to execute. To counter this, we have adopted a technique of using caches and hints to optimize these algorithms. By *cache* we mean a record of the parameters and results of previous calculations. A cache is useful if accessing it is much faster than repeating the calculation and frequently produces the required value. By *hint* we mean a value that is highly likely to be correct and that is faster to check than to recalculate. To illustrate how caches and hints can work, we describe here in some detail how the message server caches hints about individuals' inbox sites.

The key step in the delivery process is mapping the name of an individual receiving a message to the preferred inbox site. The mapping depends upon the current state of the registration data base and the availability of particular message servers. To make this mapping process as efficient as possible, each message server maintains an *inbox site cache* that maps RNames of individuals to a hint for the currently preferred inbox site. Each message server also maintains a *down server list* containing the names of message servers that it believes to be inaccessible at present. A message server is placed on this list when it does not accept connections or fails during a connection. The rules for using the inbox site cache to determine the preferred message server for a recipient *I* are:

1. If an entry for *I* is in the cache and the site indicated for *I* in the cache is not on the down server list, then use that site;
2. Otherwise get the inbox site list for *I* from the registration service; cache and return for use the first site not on the down server list; if the selected site is not first on the list, mark the entry as "secondary."

There has to be a rule for removing message servers from the down server list; this happens when the server shows signs of life by responding to a periodic single packet poll.

When a message server is removed from the down server list, the inbox site cache must be brought up to date. Any entry that is marked as "secondary" and that is not the revived site could be there as a substitute for the revived site; all such entries are removed from the cache. This heuristic removes from the cache a superset of the entries whose preferred inbox site has changed (but not all entries in the cache) and will cause recalculation of the preferred inbox site for those entries the next time they are needed.

We noted earlier that changing an individual's inbox site list may require a message server to redeliver all messages in that individual's inbox, and that this redelivery is triggered by messages from registration servers to the affected message servers. The same changes also can cause site caches to become out-of-date. Part of this problem is solved by having the inbox redelivery messages also trigger appropriate site cache flushing in the servers that had an affected inbox. Unfortunately any message server potentially has a site cache entry made out-of-date by the change. Instead of sending a message to *all* message servers, we correct the remaining obsolete caches by providing feedback from one message server to another when incorrect forwarding occurs as a result of an out-of-date cache. Thus, the site cache really does contain hints.

To summarize the cache flushing and redelivery arrangements, then, registration servers remove servers from an inbox site list and send messages to all servers originally on the list. Each responds by removing any entry for the subject individual from its site cache and redelivering any messages found in that individual's inbox. During this redelivery process, the cache entry will naturally be refreshed. Other message servers with out-of-date caches may continue to forward messages here for the subject individual. Upon receiving any message forwarded from another server, then, the target message server repeats the inbox site mapping for each name in the steering list. If the preferred site is indeed this target message server, then the message is added to the corresponding inbox. If not, then the target site does the following:

1. Forwards the message according to the new mapping result;
2. Sends a cache flush notification for the subject individual back to the server that incorrectly forwarded the message here.

The cache flush notification is a single packet sent unreliably: if it fails to arrive, another one will be provoked in due course. This strategy results in the minimum of cache flush notifications being sent—one to each message server whose cache actually needs attention, sent when the need for attention has become obvious. This mechanism is more economical than the alternative of sending cache flush notifications to all message servers, and even if that were done it would still be necessary to cope with the arrival of messages at old inbox sites.

8. System Configuration

As described in Sec. 5, the configuration of the Grapevine system is controlled by its registration data base. Various entries in the data base define the servers available to Grapevine and the ways in which the data and functions of Grapevine are distributed among them. We now consider procedures for reconfiguring Grapevine.

8.1 Adding and Deleting Registry Replicas

The set of registration servers that contain some registry is defined by the membership set for the corresponding group in the GV registry. When a change occurs to this membership set, the affected server(s) need to acquire or discard a copy of the registry data. To discover such changes, each registration server simply monitors all change messages for groups in the GV registry, watching for additions or deletions of its own name. A registration server responds to being deleted by discarding the local replica of the registry. With the present implementation, a registration server ignores being added to a registry site list. Responding to a registry addition in the obvious way—by connecting to another registration server for the registry and retrieving the registry data—is not sufficient. Synchronization problems arise that can lead to the failure to send change messages to the added server. Solving these problems may require the use of global locks, but we would prefer a solution more compatible with the looser synchronization philosophy of Grapevine. For the present obtaining a registry replica is triggered manually, after waiting for the updates to the GV registry to propagate and after ensuring that other such reconfigurations are not in progress.

8.2 Creating Servers

Installing a new Grapevine computer requires creating a new registration server and a new message server. To create the new registration server named, say, *Zinfandel.gv*, a system administrator first creates that individual (with password) in the registration data base, and gives it a connect site that is the internet address of the new computer. Next, *Zinfandel.gv* is added to the membership set of all registries that are to be recorded in this new registration server. To create the new message server named, say, *Zinfandel.ms*, the administrator creates that individual with the same connect site, then adds *Zinfandel.ms* to *MailDrop.ms.* Both servers are assigned inbox sites.

Once the data base changes have been made, the registration and message servers are started on the new computer. The first task for each is to determine its own name and password so that it may authenticate itself to the other Grapevine servers. A server obtains its name by noting its own internet address, which is always available to a machine, then consulting the data base in a different registration server to determine which server is specified to be at that address: the registration server looks for a name in the group *gv.gv*, the message server looks for a name in the group *MailDrop.ms.* Having found its name, the server asks a human operator to type its password; the operator being able to do this correctly is the fundamental source of the server's authority. The server verifies its password by the authentication protocol, again using a registration server that is already in operation, and then records its name and password on its own disk. The new registration server then consults some other registration server to obtain the contents of the GV registry in order to determine which groups in the GV registry contain its name: these specify which registries the new server should contain. It then contacts appropriate other servers to obtain copies of the data base for these registries. Because the new server can authenticate itself as an individual in the GV registry, other registration servers are willing to give it entire data base entries, including individuals' passwords.

Obtaining the registry replicas for the new registration server suffers from the same synchronization problems as adding a registry replica to an existing server. We solve them the same way, by waiting for the administrative updates to the GV registry to propagate before starting the new computer and avoiding other simultaneous reconfigurations.

8.3 Stopping and Restarting Servers

Stopping a server is very easy. Grapevine computers can be stopped without disturbing any disk write in progress. The message and registration servers are programmed so that, when interrupted between disk page writes, they can be restarted without losing any permanent information. While a message or registration server is not running, messages for it accumulate in its inboxes in message servers elsewhere, to be read after it restarts.

Whenever a message and registration server restart, each verifies its name and password by consulting other servers, and verifies that its internet address corresponds to the connect site recorded for it in the data base; if necessarry it changes the connect site recorded in the data base. Updating the connect site allows a server to be moved to a new machine just by moving the contents of the disk. After restarting, a registration server acts on all accumulated data base change messages before declaring itself open for business.

Using the internet, it is possible, subject to suitable access controls, to load a new software version into a remote running Grapevine computer, stop it, and restart it with the new version.

8.4 Other Reconfigurations

One form of reconfiguration of the system requires great care: changing the location of inbox sites for a registration server. Unless special precautions are taken, the registration server may never encounter the change message telling it about a new inbox site, because that message is waiting for it at the new site. A similar problem arises when we change the internet address of a message server that contains a registration server's inbox. Restrictions on where such data base changes can be initiated appear to be sufficient to solve these problems, but we have not automated them. Although this resolution of this problem is somewhat inelegant, the problem is not common enough to justify special mechanisms.

Part III. Conclusions

9. Present State

The Grapevine system was first made available to a limited number of clients during 1980. At present (Fall 1981) it is responsible for most of the mail traffic and distribution lists on the Xerox research internet. There are five dedicated Grapevine computers, each containing a registration server and a message server. The computers are physically distributed among northern and southern California and New York. The registration data base contains about 1500 individuals and 500 groups, divided mainly into four major registries; there are two other registries used by nonmail clients of the registration service, plus the GV and MS registries. The total message traffic amounts to some 2500 messages each working day, with an average of 4 recipients each; the messages average about 500 characters, and are almost exclusively text.

The registration data base also is used for authentication and configuration of various file servers, for authentication and access control in connection with maintenance of the basic software and data bases that support our internet gateways, and for resource location associated with remote procedure call binding. The registration data base is administered almost exclusively by nontechnical staff. There are at least three separate computer mail interface programs in use for human-readable mail. Most mail system users add and delete themselves from various distribution lists, removing this tiresome job from administrative staff.

The Grapevine registration and message servers are programmed in Mesa [7]. They contain some 33,000 lines of custom written code, together with standard packages for runtime support and PUP-level communications. The Grapevine computers are Altos [12] with 128K bytes of main memory and 5M bytes of disk storage. A running Grapevine computer has between 40 and 70 Mesa processes [4], and can handle 12 simultaneous connections. The peak load of messages handled by a single message server so far exceeds 150 per hour and 1000 messages per day. One server handled 30,000 messages while running for 1000 hours. The maximum number of primary inboxes that have been assigned to a server is 380.

10. Discussion

The fundamental design decision to use a distributed data base as the basis for Grapevine's message delivery services has worked out well. The distributed data base allowed us to meet the design goals specified in Sec. 2, and has not generated operational difficulties. The distributed update algorithms that trade atomic update for increased availability have had the desired effect. The temporary inconsistencies do not bother the users or administrators and the ability to continue data base changes while the internet is partitioned by failed long-distance links is exercised enough to be appreciated.

In retrospect, our particular implementation of the data base for Grapevine was too inflexible. As the use of the system grew, the need for various extensions to the values recorded in individual and group entries has become apparent. Reformatting the existing distributed data base to include space for the new values is difficult operationally. In a new implementation we would consider providing facilities for dynamic extension of the value set in each entry. With value set extension, however, we would keep the present update algorithm and its loose consistency guarantees. These guarantees are sufficient for Grapevine's functional domain, and their simplicity and efficiency are compelling. There is a requirement in a message system for some data base which allows more flexible descriptions of recipients or distribution lists to be mapped onto message system RNames (such as the white or yellow page services of the telephone system), but in our view that service falls outside of Grapevine's domain. A system which provides more flexibility in this direction is described in [2].

Providing all naming semantics by indirection through the registration data base has been very powerful. It has allowed us to separate the concept of *naming* a recipient from that of *addressing* the recipient. For example, the fact that a recipient is named *Birrell.pa* says nothing about where his messages should be sent. This is in contrast to many previous message systems. Indirections also provide us with flexibility in configuring the system.

One feature which recurs in descriptions of Grapevine is the concept of a "group" as a generalization of a distribution list. Our experience with use of the system confirms the utility of use of the single "group" mechanism for distribution lists, access control lists, services, and administrative purposes.

Clients other than computer mail interfaces are beginning to use Grapevine's naming, authentication, and resource location facilities. Their experience suggests that these are an important set of primitives to provide in an internet for constructing other distributed applications. Message transport as a communication protocol for data other than textual messages is a useful addition to our set of communication protocols. The firm separation between Grapevine and its clients was a good decision; it allows us to serve a wide variety of clients and to give useful guarantees to our clients, even if the clients operate in different languages and in different computing environments.

At several points in Grapevine, we have defined and implemented mechanisms of substantial versatility. As a consequence, the algorithms to implement these mechanisms in their full generality are expensive. The techniques of caches and hints are powerful tools that allow us to regain acceptable efficiency without sacrificing "correct" structure. The technique of adding caches and hints to a general mechanism is preferable to the alternative style of using special case short cut mechanisms whose existence complicates algorithmic invariants.

Grapevine was built partly to demonstrate the assertion that a properly designed replicated system can provide a very robust service. The chance of all replicas being unavailable at the same time seems low. Our experience suggests that unavailability due to hardware failure follows this pattern. No more than one Grapevine computer at a time has ever been down because of a hardware problem. On the other hand, some software bugs do not exhibit this independence. Generally all servers are running the same software version. If a client's action provokes a bug that causes a particular server to fail, then in taking advantage of the service replication that client may cause many servers to fail. A client once provoked a protocol bug when attempting to present a message for delivery. By systematically trying again at each server in *MailDrop.ms*, that client soon crashed all the Grapevine computers. Another widespread failure occurred as a result of a malformed registration data base update propagating to all servers for a particular registry. We conclude that it is hard to design a replicated system that is immune from such coordinated software unreliability.

Our experience with Grapevine has reinforced our belief in the value of producing "real" implementations of systems to test ideas. At several points in the implementation, reality forced us to rethink initial design proposals: for example, the arrangements to ensure long-term consistency of the data base in the presence of lost messages. There is no alternative to a substantial user community when investigating how the design performs under heavy load and incremental expansion.

Acknowledgments. Many people have contributed to the success of the Grapevine project. Bob Taylor and Bob Metcalfe recognized early the need for work on computer mail systems and encouraged us to develop Grapevine. Ben Wegbreit participated in the initial system design effort. Many colleagues have helped the project in various ways: Dave Boggs, Doug Brotz, Jeremy Dion, Jim Horning, Robert Kierr, and Ed Taft deserve special mention. Jerry Saltzer and several anonymous referees have made valuable commentaries on earlier drafts of the paper.

References

1. Boggs, D.R., Shoch, J.F., Taft, E.A., and Metcalfe, R.M. PUP: An internetwork architecture. *IEEE Trans. on Communications 28*, 4 (April 1980), 612–634.

2. Dawes, N., Harris, S., Magoon, M., Maveety, S., and Petty, D. The design and service impact of COCOS—An electronic office system. In *Computer Message Systems.* R.P. Uhlig (Ed.) North-Holland, New York, 1981, pp 373–384.

3. Gifford, D.K. Weighted voting for replicated data. In *Proc. 7th Symposium on Operating Systems Principles.* (Dec. 1979), ACM Order No. 534 790, pp 150–162.

4. Lampson, B.W., and Redell, D.D. Experience with processes and monitors in Mesa. *Comm. ACM 23*, 2 (Feb. 1980), 105–117.

5. Levin, R., and Schroeder, M.D. Transport of electronic messages through a network. *TeleInformatics 79*, North Holland, 1979, pp. 29–33; also available as Xerox Palo Alto Research Center Technical Report CSL-79-4.

6. Metcalfe, R.M., and Boggs, D.R. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM 19*, 7 (July 1976), 395–404.

7. Mitchell, J.G., Maybury, W., and Sweet, R. Mesa language manual (Version 5.0) Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1979.

8. National Bureau of Standards, Data encryption standard. *Federal Information Processing Standards 46*, Jan. 1977.

9. Needham, R.M., and Schroeder, M.D. Using encryption for authentication in large networks of computers. *Comm. ACM 21*, 12 (Dec. 1978), 993–999.

10. Rothnie, J.B., Goodman, N., and Bernstein, P.A. The redundant update methodology of SDD-1: A system for distributed databases (The fully redundant case). Computer Corporation of America, June 1977.

11. Shoch, J.F. Internetwork naming, addressing and routing. In *Proc. 17th IEEE Computer Society International Conference*, Sept. 1978, IEEE Cat. No. 78 CH 1388-8C, pp 72–79.

12. Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., and Boggs, D.R. Alto: A personal computer. In D.P. Siewiorek, C.G. Bell, and A. Newell, *Computer Structures: Principles and Examples.* (2nd Ed.) McGraw-Hill, New York 1981.

13. Thomas, R.H. A solution to the update problem for multiple copy data base which used distributed control. Bolt, Beranek and Newman Technical Report #3340, July 1976.

The Information Outlet: A new tool for office organization

by Yogen K. Dalal

OPD-T8104 October 1981

Abstract: Today's office can be better organized by using tools that help in managing information. Distributed office information systems permit an organization to control their conversion to "the office of the future" by reducing the initial purchase cost, and by permitting the system to evolve according to the needs and structure of the organization. Within an organization one finds a natural partitioning of activity and interaction, which can be preserved and exploited by local computer networks such as the Ethernet system. Although local computer networks are the foundation of office information systems, they should still be viewed as one component of an internetwork communication system.

The architecture of the system must permit growth both in size and types of office services. It must also permit interconnection with systems from other vendors through protocol translation gateways that capture the incompatibilities, rather than forcing each application to handle the incompatibilities

CR Categories: 3.81, 4.32.

Key words and phrases: office information systems, local networks, internetworks, distributed systems.

An earlier version of this paper was presented at the Online Conference on Local Networks & Distributed Office Systems, London, 11-13 May, 1981.

© Copyright 1981 by Xerox Corporation

XEROX
OFFICE PRODUCTS DIVISION
SYSTEMS DEVELOPMENT DEPARTMENT
3333 Coyote Hill Road / Palo Alto / California 94304

Introduction

Managing information is an integral part of today's office.

Organizations and businesses are becoming more complex, both in the way they function and evolve, and in the services and products they offer. Information exists in many forms, such as on paper, as moving video images and as voice, and is constantly being generated, used and exchanged. Executives and managers constantly process information that determines the future of their organization, professionals examine vast amounts of information that help them provide new services and products, marketeers distribute information describing these services and products, and the administrative staff records information on the daily progress of their organization.

Advances in technology, particularly in the communications and computer industry, are making it possible to build new tools that help manage information in ways that are natural to the operation of an office. These tools make it possible to create, store, retrieve, display, modify reproduce and share information in ways that encourage creativity and increase the productivity of the office worker. Inexpensive, yet powerful workstations simplify creating, modifying and displaying information. Electronic filing, printing, and database systems will simplify storing, retrieving, reproducing, and selectively extracting information. Communication networks will permit exchanging and sharing information.

The *Information Outlet*, which Xerox Corporation describes as a "plug in the wall" to an Ethernet local computer network, is the conduit to tools that manage this information. A sophisticated communication and distributed systems architecture is necessary to provide meaning to the electronic signals as they go in and out of this "plug."

This paper describes how local computer networks like the Ethernet system [Metcalfe76, Ethernet80, Shoch80a, Shoch81a, Shoch81b] form the backbone of a distributed communication system on which many automated office services can be built.

Distributed Architectures

With the continuing improvement in the price/performance ratio of computing and communications, the structure of computerized office information systems is beginning to change. It is no longer necessary to have large centralized systems in order to realize economies of scale. By pushing intelligence back into the terminal or workstation, and decentralizing resources by function into dedicated servers, an office information system becomes a collection of *loosely-coupled* system elements tied together by a communication network. System elements communicate (1) for the economic sharing of expensive resources like electronic printing and filing systems, and (2) for the exchange of information among users, as in the case of electronic mail.

The inherent flexibility of distributed systems permits an office information system to be closely tied to the needs of the surrounding user community. The overall system may be reconfigured to satisfy immediate and future requirements. This flexibility will prove invaluable in the business environment, since a system will be able to evolve and adapt to changes necessitated by alterations in an organization's requirements.

In general terms, a distributed system requires (1) a set of standards or protocols that define the structure of data, and the rules by which it is exchanged, and (2) a binding mechanism that brings together the relatively autonomous system elements.

It is a fortunate property of communication systems that functions can be layered one on top of another. Standards for the following levels are necessary:

1) *Data formats* that describe files, records, documents, forms, images, voice, etc. They describe objects that an end-user is familiar with.
2) *Control protocols* that define mechanisms by which files are exchanged, documents sent to printers, and electronic mail delivered to recipients.
3) *Transport protocols* that provide media-, processor- and application-independent delivery of data.
4) *Digital transmission systems* that specify conventions for signalling and line control.

These levels may be refined into a number of layers using the ISO Open Systems Interconnection Reference Model [Zimmermann80, OSI81].

Binding mechanisms are necessary for providing resource directories analogous to the telephone system's "white" and "yellow" pages [Oppen81]. By decoupling the many objects in a system correctly it is possible to reconfigure it easily.

Local computer networks like the Ethernet system provide digital transmission of data. They form the very foundation upon which office information systems are built, but in terms of the functions and standards necessary to build such an integrated system they represent only about 1 to 2% of the complexity [Metcalfe81].

We now describe various features of an architecture which make it possible to build the remaining 98% of the system in stages, as and when they are required.

Communication Systems

Within an organization one finds natural localities of activity and interaction. This usually decreases as one moves geographically further away. While the nature and characteristics of interaction between geographically close and geographically distant stations are different, they are both essential to the functioning of an organization.

Communication technologies have evolved to provide both local and long-haul networks. We postulate that for a given cost the bandwidth-distance product is constant. That is. for a given cost a local network will cover a small area and provide high bandwidth, while a long-haul network will cover a wider area and provide lower bandwidth. Such price/performance structures are exactly what is needed for office information systems where we expect that on the average most of the bits transmitted will be within the natural locality of activity.

To meet the communication needs of a large organization, the design of any local network *must* be considered in the context of an overall network architecture. A local network is one component of an internetwork system that provides communications services to many diverse devices connected to many different kinds networks (see for example [Boggs80, Cerf78]).

There are many different kinds of local computer networks, like Ethernet, Mitrenet, Primenet, LocalNet, Cambridge Ring, SDLC loop, etc. [Shoch80b]. They differ along the following axes: technology, media, topology, speed, moaulation, control, and applications. The Ethernet system satisfies most of the requirements for local office communications.

An *internetwork* is simply an interconnection of networks. An additional protocol layer must be interposed between the application-specific layer and the layer that transmits information across a network. This layer, is called the internet layer, and permits the addressing of system elements on any network and the delivery of data to them. Internetwork transport protocols are network-independent and define a communication system one level higher up from local networks.

Networks are interconnected by *internetwork routers*. as illustrated in the figure. There are many ways to view and build internetwork systems. Internetworks should provide store-and-forward delivery of *datagram* packets. *Virtual circuit*-like connections may then be easily built on top wherever necessary. Such a strategy is adopted by the Advanced Research Projects Agency's (ARPA) Internet and Transmission Control Protocols [IP80, TCP80], and Xerox's internal, research Pup Protocols [Boggs80]. Other schemes like X.75 assume that each of the constituent networks provides X.25 virtual circuits that may be concatenated to provide an end-to-end virtual circuit [X25. X75, Grossman79].

A well-designed network architecture must permit interconnection to systems from other vendors, obeying different protocols. This is achieved by providing *protocol translation gateways* at different levels. as required in the system, rather than having each application aware of all possible protocols. Incompatibilities between different vendors (and the different products of a single vendor) is a fact of life that must be accounted for from the very start to permit customers to integrate their existing tools into a new system.

The Ethernet system underlies Xerox' distributed systems architecture much the same way that SDLC underlies IBM's SNA. It is important to note, however, that the Ethernet and the

Information Outlet are not alternatives to IBM's SNA. It is possible for our internetworks to use SDLC links or broadband communication satellites or X.25 networks internally, and conversely SNA systems may use the Ethernet local network as a communications link. Both systems will surely interconnect through appropriate protocol translation gateways, thereby providing users access to resources on both sides.

Network Management

Distributed systems permit users to tailor the system to meet their needs, rather than change their operating procedures to meet the system's structure. The Ethernet local network uses distributed algorithms to control access to the communications channel, thereby doing away with any centralized component. This should encourage office information systems designers to use similar mechanisms at higher levels whenever possible.

In general, it snould be possible to:

1) Incrementally add or remove new system elements, services, and resources as necessary.
2) Migrate services and resources to other system elements should the one on which they reside need repair or maintenance.
3) Move workstations easily when, for example, users change offices.
4) Modify the topology of the communication system to better meet the traffic flow patterns of a particular set of users.
5) Isolate malfunctions, thereby permitting the rest of the system to continue functioning.

In order to achieve these goals, certain functions in a distributed system should be decoupled. In particular, it is necessary to differentiate between aliases, names, addresses, and routes [Shoch78, Abraham80]. At run time an alias must be resolved into a name, an address located for a name, and a route determined for an address.

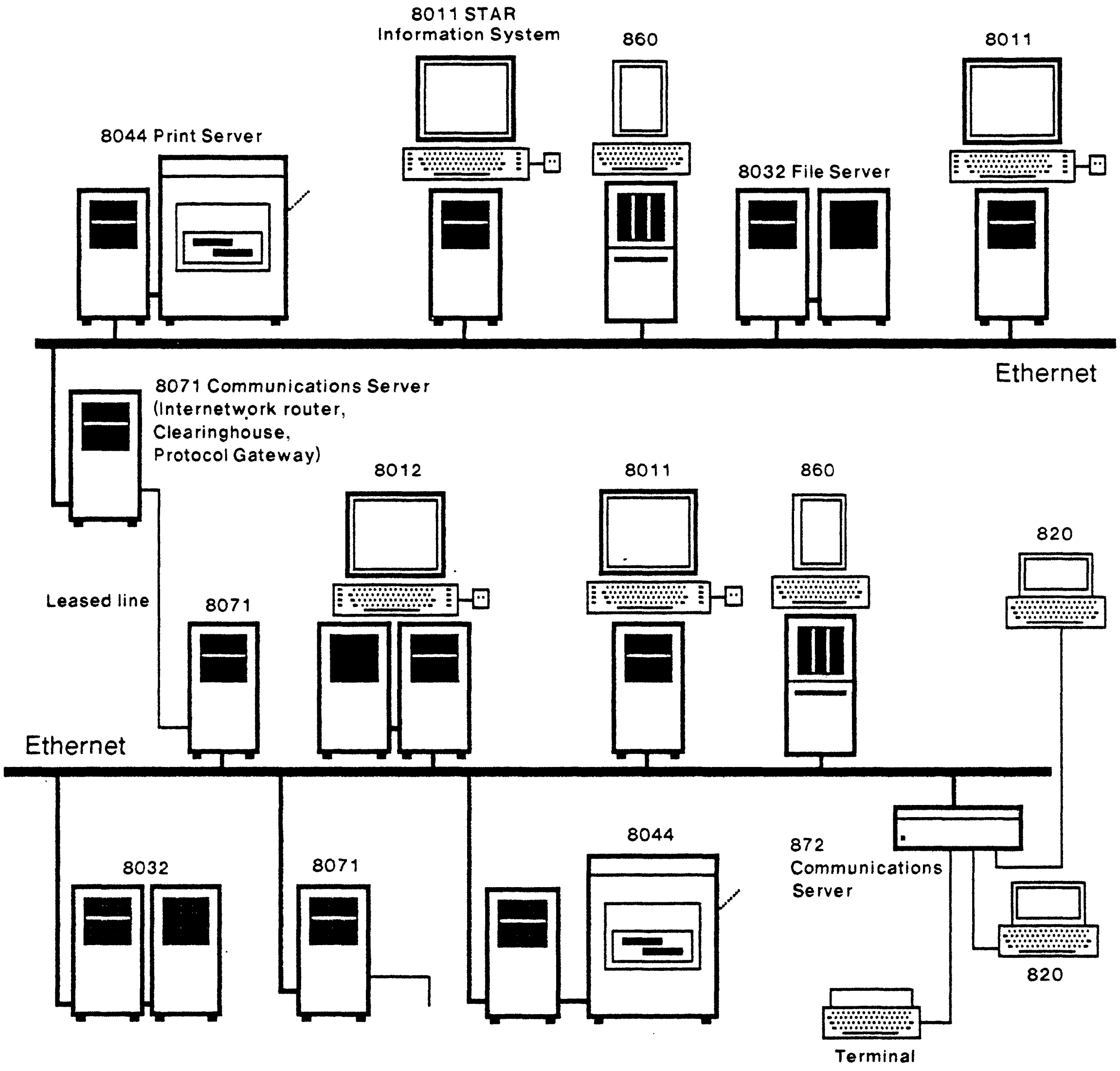
An online directory or registry service, that we call the *clearinghouse*, resolves aliases into names, and maps names into addresses [Oppen81]. This is similar to the telephone system's "white" and "yellow" pages, and permits services and system elements to be moved, added or removed. An internetwork communication system that uses adaptive routing algorithms permits the topology to be easily modified to meet changing traffic patterns, and permits graceful degradation of service in the event of line failures by using alternate and possibly less efficient routes.

One of the major advantages of decentralized network management techniques is that the system structure can be made to complement organizational structures, thus reducing the burden on the customer. Such systems can nevertheless be managed in a centralized fashion should customers so desire; they have the choice.

Office Services

So far, all we have done is describe the architecture of a distributed computer and communication system, and said very little about the design of specific office services and tools. That is precisely the point—a well-designed system permits all kinds of office services to be added as and when their need arise. This permits an organization to grow their office information system in a controlled manner, while minimizing the initial purchase cost.

We expect that higher-level protocols and data formats will be designed for many kinds of office services, and distributed office management procedures [Ellis80]. In particular, those that permit arbitrarily complex text, graphics and images to be printed, documents stored in and retrieved from electronic files, database queries, delivery of electronic mail [Levin79, Birrell81], terminal emulation to timesharing systems, voice communication, teleconferencing, etc. The list is endless. The figure shows Xerox' Network System.



The Xerox Network System

Mutual Suspicion

While an office information system should provide the right tools for manipulating information, it must also provide mechanisms for protecting information. The system should be designed with hooks to provide access control, authentication and security, should the need arise [Needham78].

Organizations are usually suspicious of one another, and would like to control the manner in which they interact. Building ultra-secure, yet very general systems is not always cost-effective for many commercial organizations. We believe that in many cases mutually suspicious organizations will resort to secure electronic document distribution as the vehicle for interaction. This is very similar to the way the postal system currently carries mail among organizations.

Conclusions

Local computer networks provide the very foundation with which to "plug one's office into the future." Such a network must, however, be viewed as one component of an internetwork communication system, and represents only about 1 to 2% of the complexity of an office information system. The protocol architecture must be layered and open-ended to permit evolution and growth, and to minimize initial purchase cost. Decentralized management of office information systems compliment organizational structures, thus reducing the burden on the customer.

Designers of local computer networks, in turn, should be influenced by some of the broader architectural considerations that go into building distributed systems.

The distributed systems architecture underlying the Ethernet local computer network, and Xerox' 8000 Network System adhere to these principles, thus permitting the Information Outlet to provide new tools for organizing the office.

Acknowledgements

The design and development of office information systems underlying the Information Outlet have involved many people from Xerox' Office Products Division and Palo Alto Research Center. Our network architecture embodies principles that evolved from experiences gained from research on the Pup Internetwork; David Boggs, John Shoch, Ed Taft, Bob Metcalfe, Hal Murray and Jim White contributed to this effort.

References

[Abraham80]
S. M. Abraham and Y. K. Dalal, "Techniques for Decentralized Management of Distributed Systems." *20th IEEE Computer Society International Conference (Compcon)*, February 1980, pp. 430-436.

[Birrell81]
A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, "Grapevine," to appear in *CACM.*

[Boggs80]
D. R. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe, "PUP: An internetwork architecture," *IEEE Transactions on Communications*, com-28:4, April 1980, pp. 612-624.

[Cerf78]
V. G. Cerf and P. K. Kirstein, "Issues in Packet-Network Interconnection," *Proceedings of the IEEE*, vol 66, no 11, November 1978, pp. 1386-1408.

[Ellis80]
C. A. Ellis and G. J. Nutt, "Computer Science and Office Information System," *ACM Computing Surveys*, vol 12, no 1, March 1980, pp. 27-60.

[Ethernet80]
The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications, Version 1.0, September 30, 1980. Available from Intel, Digital Equipment and Xerox Corporations.

[Grossman79]
G. R. Grossman, A. Hinchley, and C. A. Sunshine, "Issues in International Public Data Networking," *Computer Networks*, vol 3, no 4, September 1979, pp. 259-266.

[IP80]
DoD Standard Internet Protocol, January 1980, J. Postel editor, NTIS No. ADA079730, also in *ACM Computer Communication Review*, vol 10, no 4, October 80, pp. 2-51.

[Levin79]
R. Levin and M. D. Schroeder, "Transport of Electronic Messages Through a Network," Xerox PARC Technical Report CSL-79-4, April 1979.

[Metcalfe76]
R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM*, 19:7, July 1976, pp. 395-404.

[Metcalfe81]
R. M. Metcalfe, "A strategic overview of local computer networks," *Proceedings of the Online Conference on Local Networks & Distributed Office Systems*, London, 11-13 May, 1981, pp. 1-10.

[Oppen81]
D. C. Oppen and Y. K. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," Xerox Office Products Division, Palo Alto, OPD-T8103, October, 1981.

[OSI81]
ISO Open Systems Interconnection—Basic Reference Model, ISO/TC 97/SC 16 N 719, August, 1981.

[Shoch78]
J. F. Shoch, "Internetwork Naming, Addressing, and Routing," *17th IEEE Computer Society International Conference (Compcon)*, September 1978, pp. 430-437.

[Shoch80a]
J. F. Shoch and J. A. Hupp, "Measured performance of an Ethernet local network," *Communications of the ACM*, 23:12, December 1980, pp. 711-721.

[Shoch80b]
J. F. Shoch, "An Annotated Bibliography on Local Computer Networks," Third Edition, Xerox PARC Technical Report, SSL-80-2, April 1980.

[Shoch81a]
J. F. Shoch, Y. K. Dalal, R. C. Crane, and D. D. Redell "Evolution of the Ethernet Local Computer Network," Xerox Office Products Division, Palo Alto, OPD-T8102, September, 1981; and to appear in IEEE *Computer* Magazine.

[Shoch81b]
J. F. Shoch, *Local Computer Networks*, McGraw-Hill, in press.

[TCP80]
DoD Standard Transmission Control Protocol, January 1980, J. Postel editor, NTIS No. ADA082609, also in *ACM Computer Communication Review*, vol 10, no 4, October 80, pp. 52-132.

[X25]
Recommendation X.25/Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for Terminals Operating in the Packet Mode on Public Data Networks, CCITT Orange Book, vol 7, International Telephone and Telegraph Consultative Committee, Geneva.

[X75]
Proposal for Provisional Recommendation X.75 on International Interworking between Packet Switched Data Networks, in CCITT Study Group VII Contribution No. 207, International Telephone and Telegraph Consultative Committee, Geneva, May 1978.

[Zimmermann80]
H. Zimmermann, "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, com-28:4, April 1980, pp. 425-432.

Evolution of the Ethernet Local Computer Network

As it evolved from a research prototype to the specification of a multi–company standard, Ethernet compellbd designers to consider numerous trade-offs among alternative implementations and design strategies.

John F. Shoch, Yogen K. Dalal, and David D. Redell, Xerox
Ronald C. Crane, 3Com

With the continuing decline in the cost of computing, we have witnessed a dramatic increase in the number of independent computer systems used for scientific computing, business, process control, word processing, and personal computing. These machines do not compute in isolation, and with their proliferation comes a need for suitable communication networks—particularly local computer networks that can interconnect locally distributed computing systems. While there is no single definition of a local computer network, there is a broad set of requirements:

- relatively high data rates (typically 1 to 10M bits per second);
- geographic distance spanning about one kilometer (typically within a building or a small set of buildings);
- ability to support several hundred independent devices;
- simplicity, or the ability "to provide the simplest possible mechanisms that have the required functionality and performance";[1]
- good error characteristics, good reliability, and minimal dependence upon any centralized components or control;
- efficient use of shared resources, particularly the communications network itself;
- stability under high load;
- fair access to the system by all devices;
- easy installation of a small system, with graceful growth as the system evolves;
- ease of reconfiguration and maintenance; and
- low cost.

One of the more successful designs for a system of this kind is the Ethernet local computer network.[2,3] Ethernet installations have been in use for many years. They support hundreds of stations and meet the requirements listed above.

In general terms, Ethernet is a multi-access, packet-switched communications system for carrying digital data among locally distributed computing systems. The shared communications channel in an Ethernet is a passive broadcast medium with no central control; packet address recognition in each station is used to take packets from the channel. Access to the channel by stations wishing to transmit is coordinated in a distributed fashion by the stations themselves, using a statistical arbitration scheme.

The Ethernet strategy can be used on many different broadcast media, but our major focus has been on the use of coaxial cable as the shared transmission medium. The Experimental Ethernet system was developed at the Xerox Palo Alto Research Center starting in 1972. Since then, numerous other organizations have developed and built "Ethernet-like" local networks.[4] More recently, a cooperative effort involving Digital Equipment Corporation, Intel, and Xerox has produced an updated version of the Ethernet design, generally known as the Ethernet Specification.[5]

One of the primary goals of the Ethernet Specification is compatibility—providing enough information for different manufacturers to build widely differing machines in such a way that they can directly communicate with one another. It might be tempting to view the Specification as simply a design handbook that will allow designers to develop their own Ethernet-like network, perhaps cus-

0018-9162/82/0800-0010$00.75 © 1982 IEEE

tomized for some specific requirements or local constraints. But this would miss the major point: Successful interconnection of heterogeneous machines requires equipment that precisely matches a single specification.

Meeting the Specification is only one of the necessary conditions for intermachine communication at all levels of the network architecture. There are many levels of protocol, such as transport, name binding, and file transfer, that must also be agreed upon and implemented in order to provide useful services.[6-8] This is analogous to the telephone system: The common low-level specifications for telephony make it possible to dial from the US to France, but this is not of much use if the caller speaks only English while the person who answers the phone speaks only French. Specification of these additional protocols is an important area for further work.

The design of any local network must be considered in the context of a distributed system architecture. Although the Ethernet Specification does not directly address issues of high-level network architecture, we view the local network as one component in an *internetwork* system, providing communication services to many diverse devices connected to different networks.[6,9] The services provided by the Ethernet are influenced by these broader architectural considerations.

As we highlight important design considerations and trace the evolution of the Ethernet from research prototype to multicompany standard, we use the term Experimental Ethernet for the former and Ethernet or Ethernet Specification for the latter. The term Ethernet is also used to describe design principles common to both systems.

General description of Ethernet-class systems

Theory of operation. The general Ethernet approach uses a shared communications channel managed with a distributed control policy known as *carrier sense multiple access with collision detection*, or CSMA/CD. With this approach, there is no central controller managing access to the channel, and there is no preallocation of time slots or frequency bands. A station wishing to transmit is said to "contend" for use of the common shared communications channel (sometimes called the Ether) until it "acquires" the channel; once the channel is acquired the station uses it to transmit a packet.

To acquire the channel, stations check whether the network is busy (that is, use *carrier sense*) and defer transmission of their packet until the Ether is quiet (no other transmissions occurring). When quiet is detected, the deferring station immediately begins to transmit. During transmission, the transmitting station listens for a collision (other transmitters attempting to use the channel simultaneously). In a correctly functioning system, collisions occur only within a short time interval following the start of transmission, since after this interval all stations will detect carrier and defer transmission. This time interval is called the *collision window* or the *collision interval* and is a function of the end-to-end propagation delay. If no collisions occur during this time, a transmitter has acquired the Ether and continues transmission of the packet. If a station detects collision, the transmission of the rest of the packet is immediately aborted. To ensure that all parties to the collision have properly detected it, any station that detects a collision invokes a *collision consensus enforcement procedure* that briefly jams the channel. Each transmitter involved in the collision then schedules its packet for retransmission at some later time.

To minimize repeated collisions, each station involved in a collision tries to retransmit at a different time by scheduling the retransmission to take place after a random delay period. In order to achieve channel stability under overload conditions, a controlled retransmission strategy is used whereby the mean of the random retransmission delay is increased as a function of the channel load. An estimate of the channel load can be derived by monitoring the number of collisions experienced by any one packet. This has been shown to be the optimal strategy among the options available for decentralized decision and control problems of this class.[10]

Stations accept packets addressed to them and discard any that are found to be in error. Deference reduces the probability of collision, and collision detection allows the timely retransmission of a packet. It is impossible, however, to guarantee that all packets transmitted will be delivered successfully. For example, if a receiver is not enabled, an error-free packet addressed to it will not be delivered; higher levels of protocol must detect these situations and retransmit.

Under very high load, short periods of time on the channel may be lost due to collisions, but the collision resolution procedure operates quickly.[2,11-13] Channel utilization under these conditions will remain high, particularly if packets are large with respect to the collision interval. One of the fundamental parameters of any Ethernet implementation is the length of this collision interval, which is based on the round-trip propagation time between the farthest two points in the system.

Basic components. The CSMA/CD access procedure can use any broadcast multi-access channel, including radio, twisted pair, coaxial cable, diffuse infrared, and fiber optics.[14] Figure 1 illustrates a typical Ethernet system using coaxial cable. There are four components.

Station. A station makes use of the communication system and is the basic addressable device connected to an Ethernet; in general, it is a computer. We do not expect that "simple" terminals will be connected directly to an Ethernet. Terminals can be connected to some form of terminal controller, however, which provides access to the network. In the future, as the level of sophistication in terminals increases, many terminals will support direct connection to the network. Furthermore, specialized I/O devices, such as magnetic tapes or disk drives, may incorporate sufficient computing resources to function as stations on the network.

Within the station there is some interface between the operating system environment and the Ethernet controller. The nature of this interface (often in software) depends upon the particular implementation of the controller functions in the station.

Controller. A controller for a station is really the set of functions and algorithms needed to manage access to the channel. These include signaling conventions, encoding and decoding, serial-to-parallel conversion, address recognition, error detection, buffering, the basic CSMA/CD channel management, and packetization. These functions can be grouped into two logically independent sections of each controller: the transmitter and the receiver.

The controller functions are generally implemented using a combination of hardware, microcode, and software, depending on the nature of the station. It would be possible, for example, for a very capable station to have a minimal hardware connection to the transmission system and perform most of these functions in software. Alternatively, a station might implement all the controller functions in hardware, or perhaps in a controller-specific microprocessor. Most controller implementations fall somewhere in between. With the continuing advances in LSI development, many of these functions will be packaged in a single chip, and several semiconductor manufacturers have already announced plans to build Ethernet controllers. The precise boundary between functions performed on the chip and those in the station is implementation-dependent, but the nature of that interface is of great importance. As many of the functions as possible should be moved into the chip, provided that this preserves all of the flexibility needed in the construction and use of system interfaces and higher level software.

The description of the controller in this article is functional in nature and indicates how the controller must behave independent of particular implementations. There is some flexibility in implementing a correct controller, and we will make several recommendations concerning efficient operation of the system.

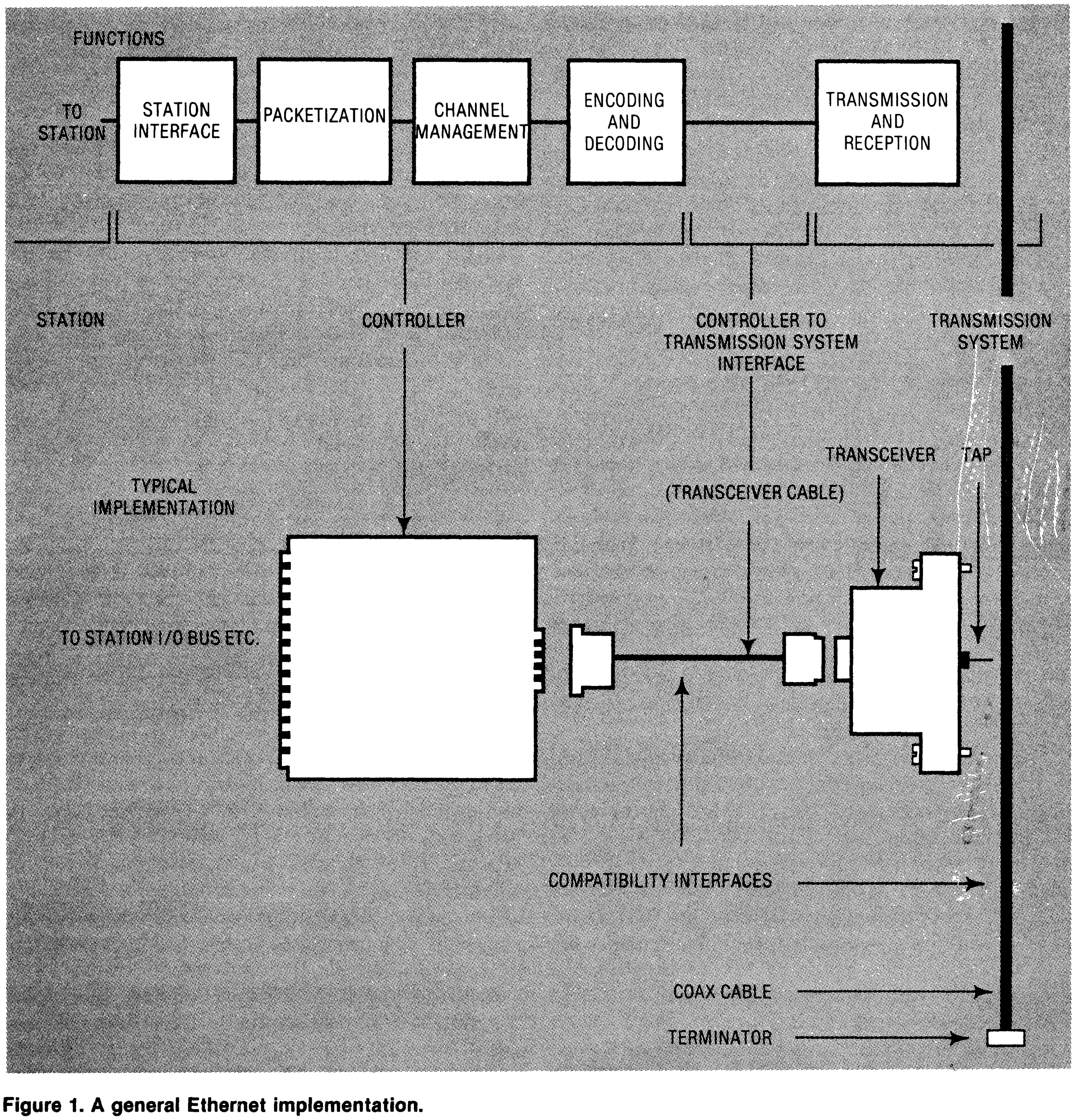


Figure 1. A general Ethernet implementation.

Transmission system. The transmission system includes all the components used to establish a communications path among the controllers. In general, this includes a suitable broadcast transmission medium, the appropriate transmitting and receiving devices—transceivers—and, optionally, repeaters to extend the range of the medium. The protocol for managing access to the transmission system is implemented in the controller; the transmission system does not attempt to interpret any of the bits transmitted on the channel.

The broadcast transmission medium contains those components that provide a physical communication path. In the case of coaxial cable, this includes the cable plus any essential hardware—connectors, terminators, and taps.

Transceivers contain the necessary electronics to transmit and receive signals on the channel and recognize the presence of a signal when another station transmits. They also recognize a collision that takes place when two or more stations transmit simultaneously.

Repeaters are used to extend the length of the transmission system beyond the physical limits imposed by the transmission medium. A repeater uses two transceivers to connect to two different Ethernet segments and combines them into one logical channel, amplifying and regenerating signals as they pass through in either direction.[15] Repeaters are transparent to the rest of the system, and stations on different segments can still collide. Thus, the repeater must propagate a collision detected on one segment through to the other segment, and it must do so without becoming unstable. A repeater makes an Ethernet channel longer and as a result increases the maximum propagation delay of the system, meaning delay through the repeater and propagation delay through the additional segments. To avoid multipath interference in an Ethernet installation, there must be only one path between any two stations through the network. (The higher level internetwork architecture can support alternate paths between stations through different communications channels.)

Controller-to-transmission-system interface. One of the major interfaces in an Ethernet system is the point at which the controller in a station connects to the transmission system. The controller does much of the work in managing the communications process, so this is a fairly simple interface. It includes paths for data going to and from the transmission system. The data received can be used by the controller to sense carrier, but the transmission system normally includes a medium-specific mechanism for detecting collisions on the channel; this must also be communicated through the interface to the controller. It is possible to power a transceiver from a separate power source, but power is usually taken from the controller interface. In most transmission systems, the connection from the controller is made to a transceiver, and this interface is called the transceiver cable interface.

Two generations of Ethernet designs. The Experimental Ethernet circa 1972 confirmed the feasibility of the design, and dozens of installations have been in regular use since then. A typical installation supports hundreds of stations and a wide-ranging set of applications: file transfer, mail distribution, document printing, terminal access to timesharing systems, data-base access, copying disks, multimachine programs, and more. Stations include the Alto workstation,[16] the Dorado (an internal research machine),[17] the Digital Equipment PDP-11, and the Data General Nova. The system has been the subject of extensive performance measurements confirming its predicted behavior.[12,13]

Based upon that experience, a second-generation system was designed at Xerox in the late 1970's. That effort subsequently led to the joint development of the Ethernet Specification. Stations built by Xerox for this network include the Xerox 860, the Xerox 8000 Network System Processor, and the Xerox 1100 Scientific Information Processor (the "Dolphin").

The two systems are very similar: they both use coaxial cable, Manchester signal encoding, and CSMA/CD with dynamic control. Some changes were made based on experience with the experimental system or in an effort to enhance the characteristics of the network. Some of the differences between the systems are summarized in Table 1.

An "Ethernet Technical Summary," which brings together the important features of Version 1 of the joint specification on two pages, is included for reference (pp. 14-15). (In building a compatible device or component, the full Ethernet Specification[5] remains the controlling document. In describing the Ethernet Specification, this article corresponds to Version 1.0; Version 2.0, including extensions and some minor revisions, will be completed later this year.)

Figure 2 is a photograph of some typical components from the Experimental Ethernet, including a transceiver and tap, transceiver cable, and an Alto controller board. Figure 3 is a photograph of similar components based on the Ethernet Specification. Note that both controller boards have been implemented with standard MSI circuits.

Transmission system design

A number of design issues and trade-offs emerged in the development of the Ethernet transmission system, and several lessons were learned from that experience.

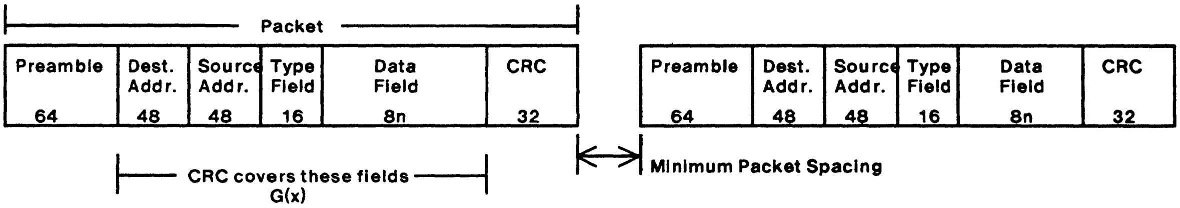
Coaxial cable subsystem. In addition to having favorable signaling characteristics and the ability to handle multimegabit transmission rates, a single coaxial

Table 1.
Comparison of Ethernet systems.

	Experimental Ethernet	Ethernet Specification
Data rate	2.94M bps	10M bps
Maximum end-to-end length	1 km	2.5 km
Maximum segment length	1 km	500 m
Encoding	Manchester	Manchester
Coax cable impedance	75 ohms	50 ohms
Coax cable signal levels	0 to +3V	0 to −2V
Transceiver cable connectors	25- and 15-pin D series	15-pin D series
Length of preamble	1 bit	64 bits
Length of CRC	16 bits	32 bits
Length of address fields	8 bits	48 bits

Ethernet 1.0 Technical Summary

Packet Format



Preamble	Dest. Addr.	Source Addr.	Type Field	Data Field	CRC
64	48	48	16	8n	32

Stations must be able to transmit and receive packets on the common coaxial cable with the indicated packet format and spacing. Each packet should be viewed as a sequence of 8-bit bytes; the least significant bit of each byte (starting with the preamble) is transmitted first.

Maximum Packet Size: 1526 bytes (8 byte preamble + 14 byte header + 1500 data bytes + 4 byte CRC)

Minimum Packet Size: 72 bytes (8 byte preamble + 14 byte header + 46 data bytes + 4 byte CRC)

Preamble: This 64-bit synchronization pattern contains alternating 1's and 0's, ending with two consecutive 1's.
The preamble is: 10101010 10101010 10101010 10101010 10101010 10101010 10101010 10101011.

Destination Address: This 48-bit field specifies the station(s) to which the packet is being transmitted. Each station examines this field to determine whether it should accept the packet. The first bit transmitted indicates the type of address. If it is a 0, the field contains the unique address of the one destination station. If it is a 1, the field specifies a logical group of recipients; a special case is the broadcast (all stations) address, which is all 1's.

Source Address: This 48-bit field contains the unique address of the station that is transmitting the packet.

Type Field: This 16-bit field is used to identify the higher-level protocol type associated with the packet. It determines how the data field is interpreted.

Data Field: This field contains an integral number of bytes ranging from 46 to 1500. (The minimum ensures that valid packets will be distinguishable from collision fragments.)

Packet Check Sequence: This 32-bit field contains a redundancy check (CRC) code, defined by the generating polynomial:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

The CRC covers the address (destination/source), type, and data fields. The first transmitted bit of the destination field is the high-order term of the message polynomial to be divided by G(x) producing remainder R(x). The high-order term of R(x) is the first transmitted bit of the Packet Check Sequence field. The algorithm uses a linear feedback register which is initially preset to all 1's. After the last data bit is transmitted, the contents of this register (the remainder) are inverted and transmitted as the CRC field. After receiving a good packet, the receiver's shift register contains 11000111 00000100 11011101 01111011 (x^{31}, ... ,x^{0}).

Minimum Packet Spacing: This spacing is 9.6 usec, the minimum time that must elapse after one transmission before another transmission may begin.

Round-trip Delay: The maximum end-to-end, round-trip delay for a bit is 51.2 usec.

Collision Filtering: Any received bit sequence smaller than the minimum valid packet (with minimum data field) is discarded as a collision fragment.

Control Procedure

The control procedure defines how and when a station may transmit packets into the common cable. The key purpose is fair resolution of occasional contention among transmitting stations.

Defer: A station must not transmit into the coaxial cable when carrier is present or within the minimum packet spacing time after carrier has ended.

Transmit: A station may transmit if it is not deferring. It may continue to transmit until either the end of the packet is reached or a collision is detected.

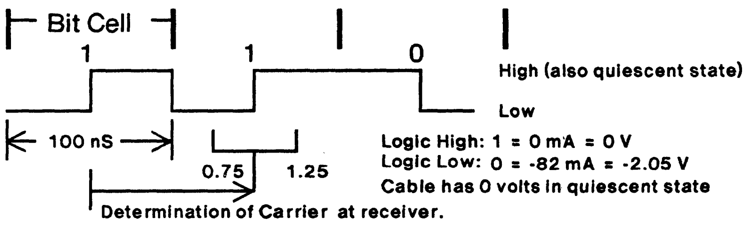
Abort: If a collision is detected, transmission of the packet must terminate, and a *jam* (4-6 bytes of arbitrary data) is transmitted to ensure that all other participants in the collision also recognize its occurrence.

Retransmit: After a station has detected a collision and aborted, it must wait for a random *retransmission delay*, defer as usual, and then attempt to retransmit the packet. The random time interval is computed using the backoff algorithm (below). After 16 transmission attempts, a higher level (e.g. software) decision is made to determine whether to continue or abandon the effort.

Backoff: Retransmission delays are computed using the *Truncated Binary Exponential Backoff* algorithm, with the aim of fairly resolving contention among up to 1024 stations. The delay (the number of time units) before the n^{th} attempt is a uniformly distributed random number from $[0 \text{ to } 2^{n}-1]$ for $0 < n \leq 10$ ($n = 0$ is the original attempt). For attempts 11-15, the interval is *truncated* and remains at [0 to 1023]. The unit of time for the retransmission delay is 512 bit times (51.2 usec).

Channel Encoding

Manchester encoding is used on the coaxial cable. It has a 50% duty cycle, and insures a transition in the middle of every bit cell ("data transition"). The first half of the bit cell contains the complement of the bit value, and the second half contains the true value of the bit.

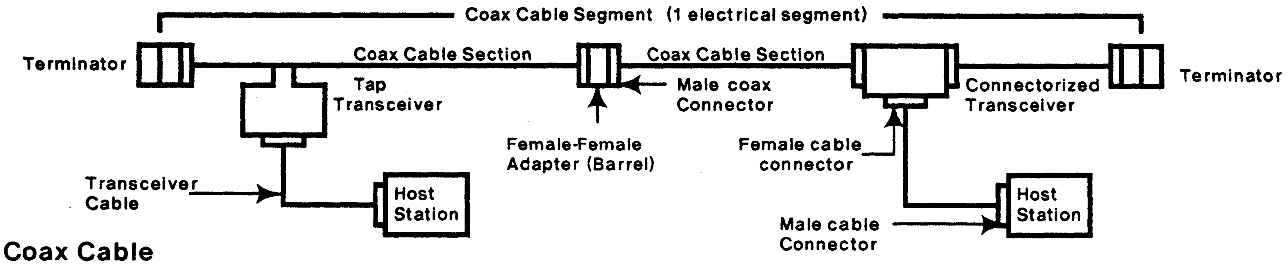

Bit Cell
1
1
0
High (also quiescent state)
Low
100 nS
0.75
1.25
Logic High: 1 = 0 mA = 0 V
Logic Low: 0 = -82 mA = -2.05 V
Cable has 0 volts in quiescent state
Determination of Carrier at receiver.

Data Rate

Data rate is 10 M bits/sec = 100 nsec bit cell ± 0.01%.

Carrier

The presence of data transitions indicates that carrier is present. If a transition is not seen between 0.75 and 1.25 bit times since the center of the last bit cell, then carrier has been lost, indicating the end of a packet. For purposes of deferring, carrier means any activity on the cable, independent of being properly formed. Specifically, it is any activity on either receive or collision detect signals in the last 160 nsec.



Coax Cable

Impedance: 50 ohms ± 2 ohms (Mil Std. C17-E). This impedance variation includes batch-to-batch variations. Periodic variations in impedance of up to ± 3 ohms are permitted along a single piece of cable.

Cable Loss: The maximum loss from one end of a cable segment to the other end is 8.5 db at 10 MHz (equivalent to ~500 meters of low loss cable).

Shielding: The physical channel hardware must operate in an ambient field of 2 volts per meter from 10 KHz to 30 MHz and 5 V/meter from 30 MHz to 1 GHz. The shield has a transfer impedance of less than 1 milliohm per meter over the frequency range of 0.1 MHz to 20 MHz (exact value is a function of frequency).

Ground Connections: The coax cable shield shall not be connected to any building or AC ground along its length. If for safety reasons a ground connection of the shield is necessary, it must be in only one place.

Physical Dimensions: This specifies the dimensions of a cable which can be used with the *standard tap*. Other cables may also be used, if they are not to be used with a tap-type transceiver (such as use with connectorized transceivers, or as a section between sections to which standard taps are connected).

Center Conductor:	0.0855" diameter solid tinned copper
Core Material:	Foam polyethylene or foam teflon FEP
Core O.D.:	0.242 " minimum
Shield:	0.326" maximum shield O.D. (>90% coverage for outer braid shield)
Jacket:	PVC or teflon FEP
Jacket O.D.:	0.405"

Coax Connectors and Terminators

Coax cables must be terminated with male N-series connectors, and cable sections will be joined with female-female adapters. Connector shells shall be insulated such that the coax shield is protected from contact to building grounds. A sleeve or boot is acceptable. Cable segments should be terminated with a female N-series connector (can be made up of a barrel connector and a male terminator) having an impedance of 50 ohms ± 1%, and able to dissipate 1 watt. The outside surface of the terminator should also be insulated.

Transceiver

CONNECTION RULES

Up to 100 transceivers may be placed on a cable segment no closer together than 2.5 meters. Following this placement rule reduces to a very low (but not zero) probability the chance that objectionable standing waves will result.

COAX CABLE INTERFACE

Input Impedance: The resistive component of the impedance must be greater then 50 Kohms. The total capacitance must be less than 4 picofarads.

Nominal Transmit Level: The important parameter is average DC level with 50% duty cycle waveform input. It must be -1.025 V (41 mA) nominal with a range of -0.9 V to -1.2 V (36 to 48 mA). The peak-to-peak AC waveform must be centered on the average DC level and its value can range from 1.4 V P-P to twice the average DC level. The voltage must never go positive on the coax. The quiescent state of the coax is logic high (0 V). Voltage measurements are made on the coax near the transceiver with the shield as reference. Positive current is current flowing out of the center conductor of the coax.

Rise and Fall Time: 25 nSec ± 5 nSec with a maximum of 1 nSec difference between rise time and fall time in a given unit. The intent is that dV/dt should not significantly exceed that present in a 10 MHz sine wave of same peak-to-peak amplitude.

Signal Symmetry: Asymmetry on output should not exceed 2 nSec for a 50-50 square wave input to either transmit or receive section of transceiver.

TRANSCEIVER CABLE INTERFACE

Signal Pairs: Both transceiver and station shall drive and present at the receiving end a 78 ohm balanced load. The differential signal voltage shall be 0.7 volts nominal peak with a common mode voltage between 0 and +5 volts using power return as reference. (This amounts to shifted ECL levels operating between Gnd and +5 volts. A 10116 with suitable pulldown resistor may be used). The quiescent state of a line corresponds to logic high, which occurs when the + line is more positive than the - line of a pair.

Collision Signal: The active state of this line is a 10 MHz waveform and its quiescent state is logic high. It is active if the transceiver is transmitting and another transmission is detected, or if two or more other stations are transmitting, independent of the state of the local transmit signal.

Power: +11.4 volts to +16 volts DC at controller. Maximum current available to transceiver is 0.5 ampere. Actual voltage at transceiver is determined by the interface cable resistance (max 4 ohms loop resistance) and current drain.

ISOLATION

The impedance between the coax connection and the transceiver cable connection must exceed 250 Kohms at 60 Hz and withstand 250 VRMS at 60 Hz.

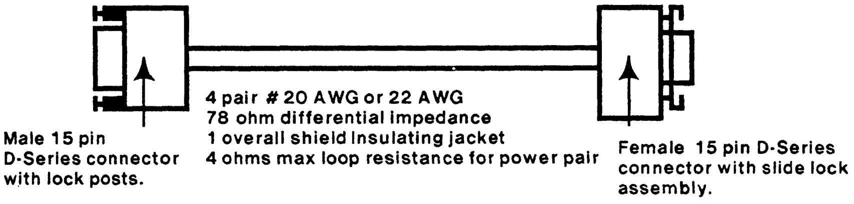
Transceiver Cable and Connectors

Maximum signal loss = 3 db @ 10 MHz. (equivalent to ~50 meters of either 20 or 22 AWG twisted pair).

Transceiver Cable Connector Pin Assignment

1.	Shield*		
2.	Collision +	9.	Collision -
3.	Transmit +	10.	Transmit -
4.	Reserved	11.	Reserved
5.	Receive +	12.	Receive -
6.	Power Return	13.	+ Power
7.	Reserved	14.	Reserved
8.	Reserved	15.	Reserved

*Shield must be terminated to connector shell.


4 pair # 20 AWG or 22 AWG
78 ohm differential impedance
1 overall shield insulating jacket
4 ohms max loop resistance for power pair
Male 15 pin D-Series connector with lock posts.
Female 15 pin D-Series connector with slide lock assembly.

cable can support communication among many different stations. The mechanical aspects of coaxial cable make it feasible to tap in at any point without severing the cable or producing excessive RF leakage; such considerations relating to installation, maintenance, and reconfigurability are important aspects in any local network design.

There are reflections and attenuation in a cable, however, and these combine to impose some limits on the system design. Engineering the shared channel entails tradeoffs involving the data rate on the cable, the length of the cable, electrical characteristics of the transceiver, and the number of stations. For example, it is possible to operate at very high data rates over short distances, but the rate must be reduced to support a greater maximum length. Also, if each transceiver introduces significant reflections, it may be necessary to limit the placement and possibly the number of transceivers.

The characteristics of the coaxial cable fix the maximum data rate, but the actual clock is generated in the controller. Thus, the station interface and controller must be designed to match the data rates used over the cable. Selection of coaxial cable as the transmission medium has no other direct impact on either the station or the controller.

Cable. The Experimental Ethernet used 75-ohm, RG-11-type foam cable. The Ethernet Specification uses a 50-ohm, solid-center-conductor, double-shield, foam dielectric cable in order to provide some reduction in the magnitude of reflections from insertion capacitance (introduced by tapping into the cable) and to provide better immunity against environmental electromagnetic noise. Belden Number 9880 Ethernet Coax meets the Ethernet Specification.

Terminators and connectors. A small terminator is attached to the cable at each end to provide a termination impedance for the cable equal to its characteristic impedance, thereby eliminating reflection from the ends of the cable. For convenience, the cable can be divided into a number of sections using simple connectors between sections to produce one electrically continuous segment.

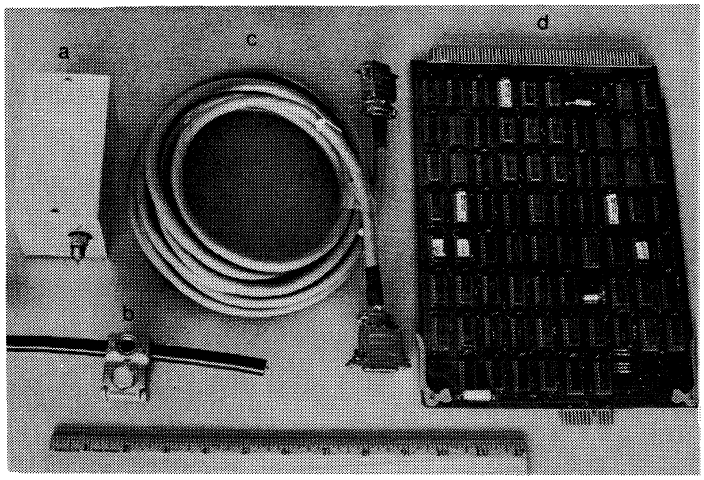


Figure 2. Experimental Ethernet components: (a) transceiver and tap, (b) tap-block, (c) transceiver cable, and (d) Alto controller board.

Segment length and the use of repeaters. The Experimental Ethernet was designed to accommodate a maximum end-to-end length of 1 km, implemented as a single electrically continuous segment. Active repeaters could be used with that system to create complex topologies that would cover a wider area in a building (or complex of buildings) within the end-to-end length limit. With the use of those repeaters, however, the maximum end-to-end length between any two stations was still meant to be approximately 1 km. Thus, the segment length and the maximum end-to-end length were the same, and repeaters were used to provide additional flexibility.

In developing the Ethernet Specification, the strong desire to support a 10M-bps data rate—with reasonable transceiver cost—led to a maximum segment length of 500 meters. We expect that this length will be sufficient to support many installations and applications with a single Ethernet segment. In some cases, however, we recognized a requirement for greater maximum end-to-end length in one network. In these cases, repeaters may now be used not just for additional flexibility but also to extend the overall length of an Ethernet. The Ethernet Specification permits the concatenation of up to three segments; the maximum end-to-end delay between two stations measured as a distance is 2.5 km, including the delay through repeaters containing a point-to-point link.[5]

Taps. Transceivers can connect to a coax cable with the use of a *pressure tap,* borrowed from CATV technology. Such a tap allows connection to the cable without cutting it to insert a connector and avoids the need to interrupt network service while installing a new station. One design uses a tap-block that is clamped on the cable and uses a special tool to penetrate the outer jacket and shield. The tool is removed and the separate tap is screwed into the block. Another design has the tap and tap-block integrated into one unit, with the tap puncturing the cable to make contact with the center conductor as the tap-block is being clamped on.

Alternatively, the cable can be cut and connectors fastened to each piece of cable. This unfortunately disrupts the network during the installation process. After the connectors are installed at the break in the cable, a T-connector can be inserted in between and then connected to a transceiver. Another option, a connectorized transceiver, has two connectors built into it for direct attachment to the cable ends without a T-connector.

Experimental Ethernet installations have used pressure taps where the tap and tap-block are separate, as illustrated in Figure 2. Installations conforming to the Ethernet Specification have used all the options. Figure 3 illustrates a connectorized transceiver and a pressure tap with separate tap and tap-block.

Transceiver. The transceiver couples the station to the cable and is the most important part of the transmission system.

The controller-to-transmission-system interface is very simple, and functionally it has not changed between the two Ethernet designs. It performs four functions: (1) transferring transmit data from the controller to the transmission system, (2) transferring receive data from the transmission system to the controller, (3) indicating to the controller that a collision is taking place, and (4) providing power to the transmission system.

It is important that the two ground references in the system—the common coaxial cable shield and the local ground associated with each station—not be tied together, since one local ground typically may differ from another local ground by several volts. Connection of several local grounds to the common cable could cause a large current to flow through the cable's shield, introducing noise and creating a potential safety hazard. For this reason, the cable shield should be grounded in only one place.

It is the transceiver that provides this ground isolation between signals from the controller and signals on the cable. Several isolation techniques are possible: transformer isolation, optical isolation, and capacitive isolation. Transformer isolation provides both power and signal isolation; it has low differential impedance for signals and power, and a high common-mode impedance for isolation. It is also relatively inexpensive to implement. Optical isolators that preserve tight signal symmetry at a competitive price are not readily available. Capacitive coupling is inexpensive and preserves signal symmetry but has poor common-mode rejection. For these reasons transformer isolation is used in Ethernet Specification transceivers. In addition, the mechanical design and installation of the transceiver must preserve this isolation. For example, cable shield connections should not come in contact with a building ground (e.g., a cable tray, conduit, or ceiling hanger).

The transceiver provides a high-impedance connection to the cable in both the power-on and power-off states. In addition, it should protect the network from possible internal circuit failures that could cause it to disrupt the network as a whole. It is also important for the transceiver to withstand transient voltages on the coax between the center conductor and shield. While such voltages should not occur if the coax shield is grounded in only one place, such isolation may not exist during installation.[1]

Negative transmit levels were selected for the Ethernet Specification to permit use of fast and more easily integrated NPN transistors for the output current source. A current source output was chosen over the voltage source used in the Experimental Ethernet to facilitate collision detection.

The key factor affecting the maximum number of transceivers on a segment in the Ethernet Specification is the input bias current for the transceivers. With easily achievable bias currents and collision threshold tolerances, the maximum number was conservatively set at 100 per segment. If the only factors taken into consideration were signal attenuation and reflections, then the number would have been larger.

Controller design

The transmitter and receiver sections of the controller perform signal conversion, encoding and decoding, serial-to-parallel conversion, address recognition, error detection, CSMA/CD channel management, buffering, and packetization. Postponing for now a discussion of buffering and packetization, we will first deal with the various functions that the controller needs to perform and then show how they are coordinated into an effective CSMA/CD channel management policy.

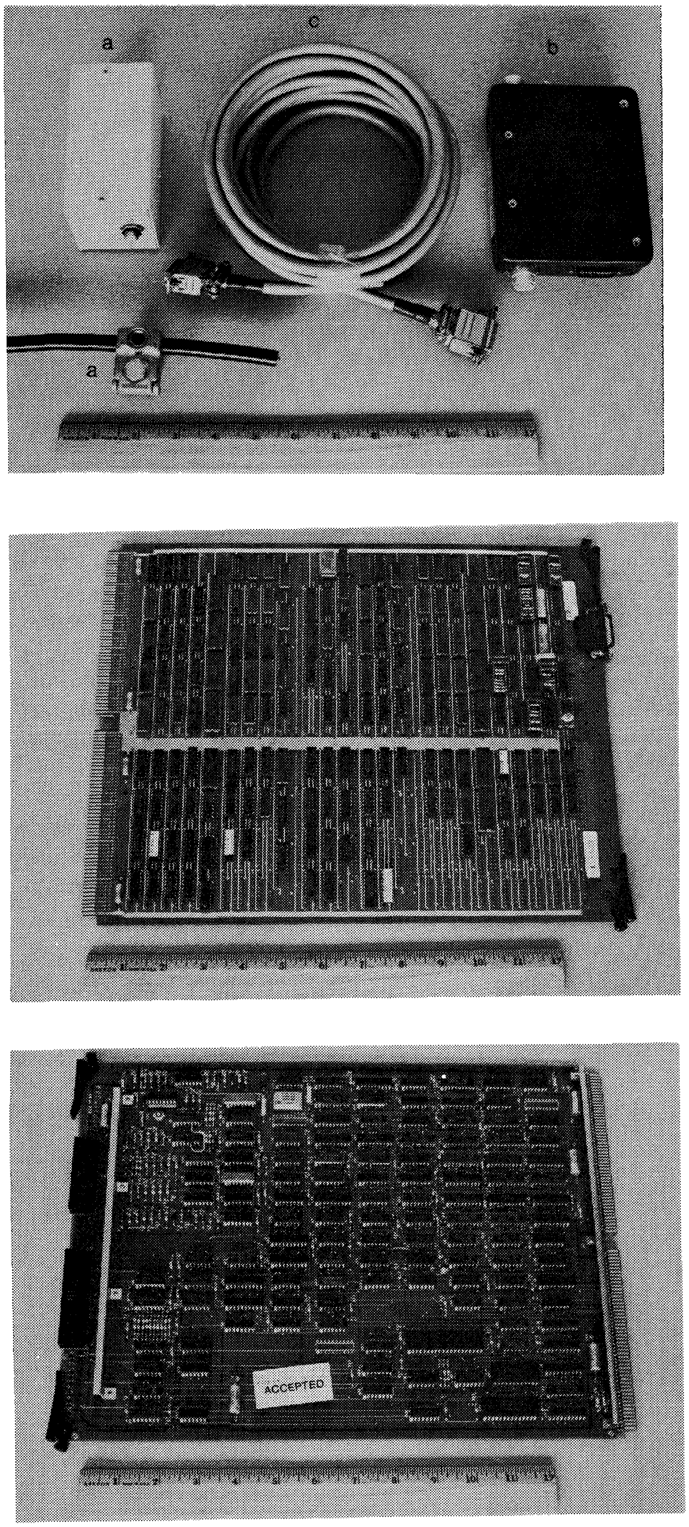


Figure 3. Ethernet Specification components: (a) transceiver, tap, and tap-block, (b) connectorized transceiver, (c) transceiver cable, (d) Dolphin controller board, and (e) Xerox 8000 controller board.

Signaling, data rate, and framing. The transmitter generates the serial bit stream inserted into the transmission system. Clock and data are combined into one signal using a suitable encoding scheme. Because of its simplicity, Manchester encoding was used in the Experimental Ethernet. In Manchester encoding, each bit cell has two parts: the first half of the cell is the complement of the bit value and the second half *is* the bit value. Thus, there is always a transition in the middle of every bit cell, and this is used by the receiver to extract the data.

For the Ethernet Specification, MFM encoding (used in double-density disk recording) was considered, but it was rejected because decoding was more sensitive to phase distortions from the transmission system and required more components to implement. Compensation is not as easy as in the disk situation because a station must receive signals from both nearby and distant stations. Thus, Manchester encoding is retained in the Ethernet Specification.

In the Experimental Ethernet, any data rate in the range of 1M to 5M bps might have been chosen. The particular rate of 2.94M bps was convenient for working with the first Altos. For the Ethernet Specification, we wanted a data rate as high as possible; very high data rates, however, limit the effective length of the system and require more precise electronics. The data rate of 10M bps represents a trade-off among these considerations.

Packet framing on the Ethernet is simple. The presence of a packet is indicated by the presence of carrier, or transitions. In addition, all packets begin with a known pattern of bits called the *preamble.* This is used by the receiver to establish bit synchronization and then to locate the first bit of the packet. The preamble is inserted by the controller at the sending station and stripped off by the controller at the receiving station. Packets may be of variable length, and absence of carrier marks the end of a packet. Hence, there is no need to have framing flags and "bit stuffing" in the packet as in other data-link protocols such as SDLC or HDLC.

The Experimental Ethernet used a one-bit preamble. While this worked very well, we have, on rare occasions, seen some receivers that could not synchronize with this very short preamble.[18] The Ethernet Specification uses a 64-bit preamble to ensure synchronization of phase-lock loop receivers often used at the higher data rate. It is necessary to specify 64 bits to allow for (1) worst-case tolerances on phase-lock loop components, (2) maximum times to reach steady-state conditions through transceivers, and (3) loss of preamble bits owing to squelch on input and output within the transceivers. Note that the presence of repeaters can add up to four extra transceivers between a source and destination.

Additional conventions can be imposed upon the frame structure. Requiring that all packets be a multiple of some particular byte or word size simplifies controller design and provides an additional consistency check. All packets on the Experimental Ethernet are viewed as a sequence of 16-bit words with the most significant bit of each word transmitted first. The Ethernet Specification requires all packets to be an integral number of eight-bit bytes (exclusive of the preamble, of course) with the least significant bit of each byte transmitted first. The order in which the bytes of an Ethernet packet are stored in the memory of a particular station is part of the controller-to-station interface.

Encoding and decoding. The transmitter is responsible for taking a serial bit stream from the station and encoding it into the Manchester format. The receiver is responsible for decoding an incoming signal and converting it into a serial bit stream for the station. The process of encoding is fairly straightforward, but decoding is more difficult and is realized in a *phase decoder.* The known preamble pattern can be used to help initialize the phase decoder, which can employ any of several techniques including an analog timing circuit, a phase-locked loop, or a digital phase decoder (which rapidly samples the input and performs a pattern match). The particular decoding technique selected can be a function of the data rate, since some decoder designs may not run as fast as others. Some phase decoding techniques—particularly the digital one—have the added advantage of being able to recognize certain phase violations as collisions on the transmission medium. This is one way to implement collision detection, although it does not work with all transmission systems.

During transmission a controller must recognize that another station is also transmitting.

The phase decoders used by stations on the Experimental Ethernet included an analog timing circuit in the form of a delay line on the PDP-11, an analog timing circuit in the form of a simple one-shot-based timer on the Alto, and a digital decoder on the Dorado. All stations built by Xerox for the Ethernet Specification use phase-locked loops.

Carrier sense. Recognizing packets passing by is one of the important requirements of the Ethernet access procedure. Although transmission is baseband, we have borrowed the term "sensing carrier" from radio terminology to describe the detection of signals on the channel. Carrier sense is used for two purposes: (1) in the receiver to delimit the beginning and end of the packet, and (2) in the transmitter to tell when it is permissible to send. With the use of Manchester phase encoding, carrier is conveniently indicated by the presence of transitions on the channel. Thus, the basic phase decoding mechanism can produce a signal indicating the presence of carrier independent of the data being extracted. The Ethernet Specification requires a slightly subtle carrier sense technique owing to the possibility of a saturated collision.

Collision detection. The ability to detect collisions and shut down the transmitter promptly is an important feature in minimizing the channel time lost to collisions. The general requirement is that during transmission a controller must recognize that another station is also transmitting. There are two approaches:

(1) *Collision detection in the transmission system.* It is usually possible for the transmission system itself to recognize a collision. This allows any medium-dependent technique to be used and is usually implemented by comparing the injected signal with the received signal. Comparing the transmitted and received signals is best done in the transceiver where there is a known relationship between the two signals. It is the controller, however, which needs to know that a collision is taking place.

(2) *Collision detection in the controller.* Alternatively, the controller itself can recognize a collision by comparing the transmitted signal with the received signal, or the receiver section can attempt to unilaterally recognize collisions, since they often appear as phase violations.

Both generations of Ethernet detect collisions within the transceiver and generate the collision signal in the controller-to-transmission-system interface. Where feasible, this can be supplemented with a collision detection facility in the controller. Collision detection may not be absolutely foolproof. Some transmission schemes can recognize all collisions, but other combinations of transmission scheme and collision detection may not provide 100-percent recognition. For example, the Experimental Ethernet system functions, in principle, as a wired OR. It is remotely possible for one station to transmit while another station sends a packet whose waveform, at the first station, exactly matches the signal sent by the first station; thus, no collision is recognized there. Unfortunately, the intended recipient might be located between the two stations, and the two signals would indeed interfere.

There is another possible scenario in which collision detection breaks down. One station begins transmitting and its signal propagates down the channel. Another station still senses the channel idle, begins to transmit, gets out a bit or two, and then detects a collision. If the colliding station shuts down immediately, it leaves a very small collision moving through the channel. In some approaches (e.g., DC threshold collision detection) this may be attenuated and simply not make it back to the transmitting station to trigger its collision detection circuitry.

The probability of such occurrences is small. Actual measurements in the Experimental Ethernet system indicate that the collision detection mechanism works very well. Yet it is important to remember that an Ethernet system delivers packets only with high probability—not certainty.

To help ensure proper detection of collisions, each transmitter adopts a *collision consensus enforcement* procedure. This makes sure that all other parties to the collision will recognize that a collision has taken place. In spite of its lengthy name, this is a simple procedure. After detecting a collision, a controller transmits a *jam* that every operating transmitter should detect as a collision. In the Experimental Ethernet the jam is a phase violation, while in the Ethernet Specification it is the transmission of four to six bytes of random data.

Another possible collision scenario arises in the context of the Ethernet Specification. It is possible for a collision to involve so many participants that a transceiver is incapable of injecting any more current into the cable. During such a collision, one cannot guarantee that the waveform on the cable will exhibit any transitions. (In the extreme case, it simply sits at a constant DC level equal to the saturation voltage.) This is called a *saturated collision.* In this situation, the simple notion of sensing carrier by detecting transitions would not work anymore. In particular, a station that deferred only when seeing transitions would think the Ether was idle and jump right in, becoming another participant in the collision. Of course, it would immediately detect the collision and back off, but in the extreme case (everyone wanting to transmit), such jumping-in could theoretically cause the saturated collision to snowball and go on for a very long time. While we recognized that this form of instability was highly unlikely to occur in practice, we included a simple enhancement to the carrier sense mechanism in the Ethernet Specification to prevent the problem.

We have focused on collision detection by the transmitter of a packet and have seen that the transmitter may depend on a collision detect signal generated unilaterally by its receiving phase decoder. Can this receiver-based collision detection be used just by a receiver (that is, a station that is not trying to transmit)? A receiver with this capability could immediately abort an input operation and could even generate a jam signal to help ensure that the collision came to a prompt termination. With a reasonable transmitter-based collision detection scheme, however, the collision is recognized by the transmitters and the damaged packet would come to an end very shortly. Receiver-based collision detection could provide an early warning of a collision for use by the receiver, but this is not a necessary function and we have not used it in either generation of Ethernet design.

CRC generation and checking. The transmitter generates a cyclic redundancy check, or CRC, of each transmitted packet and appends it to a packet before transmission. The receiver checks the CRC on packets it receives and strips it off before giving the packet to the station. If the CRC is incorrect, there are two options: either discard the packet or deliver the damaged packet with an appropriate status indicating a CRC error.

While most CRC algorithms are quite good, they are not infallible. There is a small probability that undetected errors may slip through. More importantly, the CRC only protects a packet from the point at which the CRC is generated to the point at which it is checked. Thus, the CRC cannot protect a packet from damage that occurs in parts of the controller, as, for example, in a FIFO in the parallel path to the memory of a station (the DMA), or in the memory itself. If error detection at a higher level is required, then an end-to-end software checksum can be added to the protocol architecture.

In measuring the Experimental Ethernet system, we have seen packets whose CRC was reported as correct but whose software checksum was incorrect.[18] These did not necessarily represent an undetected Ethernet error; they usually resulted from an external malfunction such as a broken interface, a bad CRC checker, or even an incorrect software checksum algorithm.

Selection of the CRC algorithm is guided by several concerns. It should have sufficient strength to properly detect virtually all packet errors. Unfortunately, only a limited set of CRC algorithms are currently implemented in LSI chips. The Experimental Ethernet used a 16-bit CRC, taking advantage of a single-chip CRC generator/checker. The Ethernet Specification provides better error detection by using a 32-bit CRC.[19,20] This function will be easily implemented in an Ethernet chip.

Addressing. The packet format includes both a source and destination address. A local network design can adopt either of two basic addressing structures: *network-specific* station addresses or *unique* station addresses.[21] In the first case, stations are assigned network addresses that must be unique on *their* network but may be the same as the address held by a station on another network. Such addresses are sometimes called *network relative* addresses, since they depend upon the particular network to which the station is attached. In the second case, each station is assigned an address that is unique over all space and time. Such addresses are also known as absolute or universal addresses, drawn from a flat address space.

To permit internetwork communication, the network-specific address of a station must usually be combined with a unique network number in order to produce an unambiguous address at the next level of protocol. On the other hand, there is no need to combine an absolute station address with a unique network number to produce an unambiguous address. However, it is possible that internetwork systems based on flat (internetwork and local network) absolute addresses will include a unique network number at the internetwork layer as a "very strong hint" for the routing machinery.

If network-specific addressing is adopted, Ethernet address fields need only be large enough to accommodate the maximum number of stations that will be connected to one local network. In addition, there must be a suitable administrative procedure for assigning addresses to stations. Some installations will have more than one Ethernet, and if a station is moved from one network to another it may be necessary to change its network-specific address, since its former address may be in use on the new network. This was the approach used on the Experimental Ethernet, with an eight-bit field for the source and the destination addresses.

We anticipate that there will be a large number of stations and many local networks in an internetwork. Thus, the management of network-specific station addresses can represent a severe problem. The use of a flat address space provides for reliable and manageable operation as a system grows, as machines move, and as the overall topology changes. A flat internet address space requires that the address space be large enough to ensure uniqueness while providing adequate room for growth. It is most convenient if the local network can directly support these fairly large address fields.

For these reasons the Ethernet Specification uses 48-bit addresses.[22] Note that these are station addresses and are not associated with a particular network interface or controller. In particular, we believe that higher level routing and addressing procedures are simplified if a station connected to multiple networks has only one identity which is unique over all networks. The address should not be hard-wired into a particular interface or controller but should be able to be set from the station. It may be very useful, however, to allow a station to read a unique station identifier from the controller. The station can then choose whether to return this identifier to the controller as its address.

In addition to single-station addressing, several enhanced addressing modes are also desirable. *Multicast* addressing is a mechanism by which packets may be targeted to more than one destination. This kind of service is particularly valuable in certain kinds of distributed applications, for instance the access and update of distributed data bases, teleconferencing, and the distributed algorithms that are used to manage the network and the internetwork. We believe that multicast should be supported by allowing the destination address to specify either a physical or logical address. A logical address is known as a *multicast ID. Broadcast* is a special case of multicast in which a packet is intended for all active stations. Both generations of Ethernet support broadcast, while only the Ethernet Specification directly supports multicast.

Stations supporting multicast must recognize multicast IDs of interest. Because of the anticipated growth in the use of multicast service, serious consideration should be given to aspects of the station and controller design that reduce the system load required to filter unwanted multicast packets. Broadcast should be used with discretion, since all nodes incur the overhead of processing every broadcast packet.

Controllers capable of accepting packets regardless of destination address provide *promiscuous* address recognition. On such stations one can develop software to observe all of the channel's traffic, construct traffic matrices, perform load analysis, (potentially) perform fault isolation, and debug protocol implementations. While such a station is able to read packets not addressed to it, we expect that sensitive data will be encrypted by higher levels of software.

CSMA/CD channel management

A major portion of the controller is devoted to Ethernet channel management. These conventions specify procedures by which packets are transmitted and received on the multi-access channel.

Transmitter. The transmitter is invoked when the station has a packet to send. If a collision occurs, the controller enforces the collision with a suitable jam, shuts down the transmitter, and schedules a retransmission.

Retransmission policies have two conflicting goals: (1) scheduling a retransmission quickly to get the packet out and maintain use of the channel, and (2) voluntarily backing off to reduce the station's load on a busy channel. Both generations of Ethernet use the *binary exponential back-off algorithm* described below. After some maximum number of collisions the transmitter gives up and reports a suitable error back to the station; both generations of Ethernet give up after 15 collisions.

The binary exponential back-off algorithm is used to calculate the delay before retransmission. After a collision takes place the objective is to obtain delay periods that will reschedule each station at times quantized in steps at least as large as a collision interval. This time quantization is called the *retransmission slot time.* To guarantee quick use of the channel, this slot time should be short; yet to avoid collisions it should be larger than a collision interval. Therefore, the slot time is usually set to be a little longer than the round-trip time of the channel. The real-time delay is the product of some retransmission delay (a positive integer) and the retransmission slot time.

Collisions on the channel can produce collision fragments, which can be eliminated with a fragment filter in the controller.

To minimize the probability of repeated collisions, each retransmission delay is selected as a random number from a particular retransmission interval between zero and some upper limit. In order to control the channel and keep it stable under high load, the interval is doubled with each successive collision, thus extending the range of possible retransmission delays. This algorithm has very short retransmission delays at the beginning but will back off quickly, preventing the channel from becoming overloaded. After some number of back-offs, the retransmission interval becomes large. To avoid undue delays and slow response to improved channel characteristics, the doubling can be stopped at some point, with additional retransmissions still being drawn from this interval, before the transmission is finally aborted. This is referred to as *truncated binary exponential back-off.*

The truncated binary exponential back-off algorithm approximates the ideal algorithm where the probability of transmission of a packet is 1/Q, with Q representing the number of stations attempting to transmit.[23] The retransmission interval is truncated when Q becomes equal to the maximum number of stations.

In the Experimental Ethernet, the very first transmission attempt proceeds with no delay (i.e., the retransmission interval is [0-0]). The retransmission interval is doubled after each of the first eight transmission attempts. Thus, the retransmission delays should be uniformly distributed between 0 and $2^{\min(\text{retransmission attempt, } 8)} - 1$. After the first transmission attempt, the next eight intervals will be [0-1], [0-3], [0-7], [0-15], [0-31], [0-63], [0-127], and [0-255]. The retransmission interval remains at [0-255] on any subsequent attempt, as the maximum number of stations is 256. The Ethernet Specification has the same algorithm with ten intervals, since the network permits up to 1024 stations; the maximum interval is therefore [0-1023]. The back-off algorithm restarts with a zero retransmission interval for the transmission of every new packet.

This particular algorithm was chosen because it has the proper basic behavior and because it allows a very simple implementation. The algorithm is now supported by empirical data verifying the stability of the system under heavy load.[12,13] Additional attempts to explore more sophisticated algorithms resulted in negligible performance improvement.

Receiver. The receiver section of the controller is activated when the carrier appears on the channel. The receiver processes the incoming bit stream in the following manner:

The remaining preamble is first removed. If the bit stream ends before the preamble completes, it is assumed to be the result of a short collision, and the receiver is restarted.

The receiver next determines whether the packet is addressed to it. The controller will accept a packet in any of the following circumstances:

(1) The destination address matches the specific address of the station.
(2) The destination address has the distinguished broadcast destination.
(3) The destination address is a multicast group of which the station is a member.
(4) The station has set the controller in promiscuous mode and receives all packets.

Some controller designs might choose to receive the entire packet before invoking the address recognition procedure. This is feasible but consumes both memory and processing resources in the controller. More typically, address recognition takes place at a fairly low level in the controller, and if the packet is not to be accepted the controller can ignore the rest of it.

Assuming that the address is recognized, the receiver now accepts the entire packet. Before the packet is actually delivered to the station, the CRC is verified and other consistency checks are performed. For example, the packet should end on an appropriate byte or word boundary and be of appropriate minimum length; a minimum packet would have to include at least a destination and source address, a packet type, and a CRC. Collisions on the channel, however, can produce short, damaged packets called collision fragments. It is generally unnecessary to report these errors to the station, since they can be eliminated with a fragment filter in the controller. It is important, however, for the receiver to be restarted promptly after a collision fragment is received, since the sender of the packet may be about to retransmit.

Packet length. One important goal of the Ethernet is data transparency. In principle, this means that the data field of a packet can contain any bit pattern and be of any length, from zero to arbitrarily large. In practice, while it is easy to allow any bit pattern to appear in the data field, there are some practical considerations that suggest imposing upper and lower bounds on its length.

At one extreme, an empty packet (one with a zero-length data field) would consist of just a preamble, source and destination addresses, a type field, and a CRC. The Experimental Ethernet permitted empty packets. However, in some situations it is desirable to enforce a minimum overall packet size by mandating a minimum-length data field, as in the Ethernet Specification. Higher level protocols wishing to transmit shorter packets must then pad out the data field to reach the minimum.

At the other extreme, one could imagine sending many thousands or even millions of bytes in a single packet. There are, however, several factors that tend to limit packet size, including (1) the desire to limit the size of the buffers in the station for sending and receiving packets, (2) similar considerations concerning the packet buffers that are sometimes built into the Ethernet controller itself, and (3) the need to avoid tying up the channel and increasing average channel latency for other stations. Buffer management tends to be the dominant consideration. The maximum requirement for buffers in the station is usually a parameter of higher level software determined by the overall network architecture; it is typically on the order of 500 to 2000 bytes. The size of any packet buffers in the controller, on the other hand, is usually a design parameter of the controller hardware and thus represents a more rigid limitation. To insure compatibility among buffered controllers, the Ethernet Specification mandates a maximum packet length of 1526 bytes (1500 data bytes plus overhead).

Note that the upper and lower bounds on packet length are of more than passing interest, since observed distributions are typically quite bimodal. Packets tend to be either very short (control packets or packets carrying a small amount of data) or maximum length (usually some form of bulk data transfer).[12,13]

The efficiency of an Ethernet system is largely dependent on the size of the packets being sent and can be very high when large packets are used. Measurements have shown total utilization as high as 98 percent. A small quantum of channel capacity is lost whenever there is a collision, but the carrier sense and collision detection mechanisms combine to minimize this loss. Carrier sense reduces the likelihood of a collision, since the acquisition effect renders a given transmission immune to collisions once it has continued for longer than a collision interval. Collision detection limits the duration of a collision to a single collision interval. If packets are long compared with the collision interval, then the network is vulnerable to collisions only a small fraction of the time and total utilization will remain high. If the average packet size is reduced, however, both carrier sense and collision detection become less effective. Ultimately, as the packet size approaches the collision interval, system performance degrades to that of a straight CSMA channel without collision detection. This condition only occurs under a heavy load consisting predominantly of very small packets; with a typical mix of applications this is not a practical problem.

If the packet size is reduced still further until it is less than the collision interval, some new problems appear. Of course, if an empty packet is already longer than the collision interval, as in the Experimental Ethernet, this case cannot arise. As the channel length and/or the data rate are increased, however, the length (in bits) of the collision interval also increases. When it becomes larger than an empty packet, one must decide whether stations are allowed to send tiny packets that are smaller than the collision interval. If so, two more problems arise, one affecting the transmitter and one the receiver.

The transmitter's problem is that it can complete the entire transmission of a tiny packet before network acquisition has occurred. If the packet subsequently experiences a collision farther down the channel, it is too late for the transmitter to detect the collision and promptly schedule a retransmission. In this situation, the probability of a collision has not increased, nor has any additional channel capacity been sacrificed; the problem is simply that the transmitter will occasionally fail to recognize and handle a collision. To deal with such failures, the sender of tiny packets must rely on retransmissions invoked by a higher level protocol and thus suffer reduced throughput and increased delay. This occasional performance reduction is generally not a serious problem, however. Note that only the sender of tiny packets encounters this behavior; there is no unusual impact on other stations sending larger packets.

While occasional collisions should be viewed as a normal part of the CSMA/CD access procedure, line errors should not. One would therefore like to accumulate information about the two classes of events separately.

The receiver's problem with tiny packets concerns its ability to recognize collision fragments by their small size and discard them. If the receiver can assume that packets smaller than the collision interval are collision fragments, it can use this to implement a simple and inexpensive fragment filter. It is important for the receiver to discard collision fragments, both to reduce the processing load at the station and to ensure that it is ready to receive the impending retransmission from the transmitter involved in the collision. The fragment filter approach is automatically valid in a network in which there are no tiny packets, such as the Experimental Ethernet. If tiny packets can occur, however, the receiver cannot reliably distinguish them from collision fragments purely on the basis of size. This means that at least the longer collision fragments must be rejected on the basis of some other error detection mechanism such as the CRC check or a byte or word alignment check. One disadvantage of this approach is that it increases the load on the CRC mechanism, which, while strong, is not infallible. Another problem is that the CRC error condition will now be indicating two kinds of faults: long collisions and genuine line errors. While occasional collisions should be viewed as a normal part of the CSMA/CD access procedure, line errors should not. One would therefore like to accumulate information about the two classes of events separately.

The problems caused by tiny packets are not insurmountable, but they do increase the attractiveness of simply legislating the problem out of existence by forbidding the sending of packets smaller than the collision interval. Thus, in a network whose collision interval is longer than an empty packet, the alternatives are

(1) *Allow tiny packets.* In this case, the transmitter will sometimes fail to detect collisions, requiring retransmis-

sion at a higher level and impacting performance. The receiver can use a partial fragment filter to discard collision fragments shorter than an empty packet, but longer collision fragments will make it through this filter and must be rejected on the basis of other error checks, such as the CRC check, with the resultant jumbling of the error statistics.

(2) *Forbid tiny packets.* In this case, the transmitter can always detect a collision and perform prompt retransmission. The receiver can use a fragment filter to automatically discard all packets shorter than the collision interval. The disadvantage is the imposition of a minimum packet size.

Unlike the Experimental Ethernet, the Ethernet Specification defines a collision interval longer than an empty packet and must therefore choose between these alternatives. The choice is to forbid tiny packets by requiring a minimum data field size of 46 bytes. Since we expect that Ethernet packets will typically contain internetwork packet headers and other overhead, this is not viewed as a significant disadvantage.

Controller-to-station interface design

The properties of the controller-to-station interface can dramatically affect the reliability and efficiency of systems based on Ethernet.

Turning the controller on and off. A well-designed controller must be able to (1) keep the receiver on in order to catch back-to-back packets (those separated by some minimum packet spacing), and (2) receive packets a station transmits to itself. We will now look in detail at these requirements and the techniques for satisfying them.

Keeping the receiver on. The most frequent cause of a lost packet has nothing to do with collision or bad CRCs. Packets are usually missed simply because the receiver was not listening. The Ethernet is an asynchronous device that can present a packet at any time, and it is important that higher level software keep the receiver enabled.

The problem is even more subtle, however, for even when operating normally there can be periods during which the receiver is not listening. There may, for instance, be turnaround times between certain operations when the receiver is left turned off. For example, a receive-to-receive turnaround takes place after one packet is received and before the receiver is again enabled. If the design of the interface, controller, or station software keeps the receiver off for too long, arriving packets can be lost during this turnaround. This occurs most frequently in servers on a network, which may be receiving packets from several sources in rapid succession. If back-to-back packets come down the wire, the second one will be lost in the receive-to-receive turnaround time. The same problem can occur within a normal workstation, for example, if a desired packet immediately follows a broadcast packet; the workstation gets the broadcast but misses the packet specifically addressed to it. Higher level protocol software will presumably recover from these situations, but the performance penalty may be severe.

Similarly, there may be a transmit-to-receive turnaround time when the receiver is deaf. This is determined by how long it takes to enable the receiver after sending a packet. If, for example, a workstation with a slow transmit-to-receive turnaround sends a packet to a well-tuned server, the answer may come back before the receiver is enabled again. No amount of retransmission by higher levels will ever solve this problem!

It is important to minimize the length of any turnaround times when the receiver might be off. There can also be receive-to-transmit and transmit-to-transmit turnaround times, but their impact on performance is not as critical.

Sending to itself. A good diagnostic tool for a network interface is the ability of a station to send packets to itself. While an internal loop-back in the controller provides a partial test, actual transmission and simultaneous reception provide more complete verification.

The Ethernet channel is, in some sense, half duplex: there is normally only one station transmitting at a time. There is a temptation, therefore, to also make the controller half duplex—that is, unable to send and receive at the same time. If possible, however, the design of the interface, controller, and station software should allow a station to send packets to itself.

Recommendations. The Ethernet Specification includes one specific requirement that helps to solve the first of these problems: There must be a minimal interpacket spacing on the cable of 9.6 microseconds. This requirement applies to a transmitter getting ready to send a packet and does not necessarily mean that all receivers conforming to the Specification must receive two adjacent packets. This requirement at least makes it possible to build a controller that can receive adjacent packets on the cable.

Satisfying the two requirements described earlier involves the use of two related features in the design of a controller: full-duplex interfaces and back-to-back receivers. A full-duplex interface allows the receiver and the transmitter to be started independently. A back-to-back receiver has facilities to automatically restart the receiver upon completion of a reception. Limited back-to-back reception can be done with two buffers; the first catches a packet and then the second catches the next without requiring the receiver to wait. Generalized back-to-back reception can be accomplished by using chained I/O commands; the receiver is driven by a list of free input buffers, taking one when needed. These two notions can be combined to build any of the following four interfaces: (1) half-duplex interface, (2) full-duplex interface, (3) half-duplex interface with back-to-back receive, and (4) full-duplex interface with back-to-back receive.

The Experimental Ethernet controller for the Alto is half duplex, runs only in a transmit or receive mode, and must be explicitly started in each mode. The need to explicitly start the receiver (there is no automatic hardware turnaround) means that there may be lengthy turnaround times in which packets may be missed. This approach allows sharing certain components, like the CRC function and the FIFO.

Experimental Ethernet controllers built for the PDP-11 and the Nova are full-duplex interfaces. The transmit-to-receive turnaround has been minimized, but there is no provision for back-to-back packets.

The Ethernet controller for the Xerox 8000 processor is a half-duplex interface with back-to-back receive. Although it cannot send to itself, the transmit-to-receive turnaround delay has been avoided by having the hardware automatically revert to the receive state when a transmission is completed.

The Experimental Ethernet and Ethernet Specification controllers for the Dolphin are full-duplex interfaces with back-to-back receivers. They are the ultimate in interface organization.

Our experience shows that any one of the four alternatives will work. However, we strongly recommend that all interface and controller designs support full-duplex operation and provide for reception of back-to-back packets (chained I/O).

The controller-to-station interface defines the manner in which data received from the cable is stored in memory and, conversely, how data stored in memory is transmitted on the cable.

Buffering. Depending upon the particular data rate of the channel and the characteristics of the station, the controller may have to provide suitable buffering of packets. If the station can keep up with the data rate of the channel, only a small FIFO may be needed to deal with station latency. If the station cannot sustain the channel data rate, it may be necessary to include a full-packet buffer as part of the controller. For this reason, full compatibility across different stations necessitates the specification of a maximum packet length.

If a single-packet buffer is provided in the controller (a buffer that has no marker mechanism to distinguish boundaries between packets), it will generally be impossible to catch back-to-back packets, and in such cases it is preferable to have at least two input buffers.

Packets in memory. The controller-to-station interface defines the manner in which data received from the cable is stored in memory and, conversely, how data stored in memory is transmitted on the cable. There are many ways in which this parallel-to-serial transformation can be defined.[24] The Ethernet Specification defines a packet on the cable to be a sequence of eight-bit bytes, with the least significant bit of each byte transmitted first. Higher level protocols will in most cases, however, define data types that are multiples of eight bits. The parallel-to-serial transformations will be influenced by the programming conventions of the station and by the higher level protocols. Stations with different parallel-to-serial transformations that use the same higher level protocol must make sure that all data types are viewed consistently.

Type field. An Ethernet packet can encapsulate many kinds of client-defined packets. Thus, the packet format includes only a data field, two addresses, and a type field. The type field identifies the special client-level protocol that will interpret the data encapsulated within the packet. The type field is never processed by the Ethernet system itself but can be thought of as an escape, providing a consistent way to specify the interpretation of the rest of the packet.

Low-level system services such as diagnostics, bootstrap, loading, or specialized network management functions can take advantage of the identification provided by this field. In fact, it is possible to use the type field to identify all the different packets in a protocol architecture. In general, however, we recommend that the Ethernet packet encapsulate higher level internetwork packets. Internetwork router stations might concurrently support a number of different internetwork protocols, and the use of the type field allows the internetwork router to encapsulate different kinds of internetwork packets for a local network transmission.[25] The use of a type field in the Ethernet packet is an instance of a principle we apply to all layers in a protocol architecture. A type field is used at each level of the hierarchy to identify the protocol used at the next higher level; it is the bridge between adjacent levels. This results in an architecture that defines a layered tree of protocols.

The Experimental Ethernet design uses a 16-bit type field. This has proved to be a very useful feature and has been carried over into the Ethernet Specification.

Summary and conclusions

We have highlighted a number of important considerations that affect the design of an Ethernet local computer network and have traced the evolution of the system from a research prototype to a multicompany standard by discussing strategies and trade-offs between alternative implementations.

The Ethernet is intended primarily for use in such areas as office automation, distributed data processing, terminal access, and other situations requiring economical connection to a local communication medium carrying bursts of traffic at high peak data rates. Experience with the Experimental Ethernet in building distributed systems that support electronic mail, distributed filing, calendar systems, and other applications has confirmed many of our design goals and decisions.[26-29]

Questions sometimes arise concerning the ways in which the Ethernet design addresses (or chooses not to address) the following considerations: reliability, addressing, priority, encryption, and compatibility. It is important to note that some functions are better left out of the Ethernet itself for implementation at higher levels in the architecture.

All systems should be reliable, and network-based systems are no exception. We believe that reliability must be addessed at each level in the protocol hierarchy; each level should provide only what it can guarantee at a reasonable price. Our model for internetworking is one in which reliability and sequencing are performed using end-to-end transport protocols. Thus, the Ethernet provides a "best effort" datagram service. The Ethernet has been designed to have very good error characteristics, and, without promising to deliver all packets, it will deliver a very large percentage of offered packets without error. It includes error detection procedures but provides no error correction.

We expect internetworks to be very large. Many of the problems in managing them can be simplified by using absolute station addresses that are directly supported within the local network. Thus, address fields in the Ethernet Specification seem to be very generous—well beyond the number of stations that might connect to one local network but meant to efficiently support large internetwork systems.

Our experience indicates that for practically all applications falling into the category "loosely coupled distributed system," the average utilization of the communications network is low. The Ethernet has been designed to have excess bandwidth, not all of which must be utilized. Systems should be engineered to run with a sustained load of no more than 50 percent. As a consequence, the network will generally provide high throughput of data with low delay, and there are no priority levels associated with particular packets. Designers of individual devices, network servers, and higher level protocols are free to develop priority schemes for accessing particular resources.

Protection, security, and access control are all system-wide functions that require a comprehensive strategy. The Ethernet system itself is not designed to provide encryption or other mechanisms for security, since these techniques by themselves do not provide the kind of protection most users require. Security in the form of encryption, where required, is the responsibility of the end-user processes.

Higher level protocols raise their own issues of compatibility over and above those addressed by the Ethernet and other link-level facilities. While the compatibility provided by the Ethernet does not guarantee solutions to higher level compatibility problems, it does provide a context within which such problems can be addressed by avoiding low-level incompatibilities that would make direct communication impossible. We expect to see standards for higher level protocols emerge during the next few years.

Within an overall distributed systems architecture, the two generations of Ethernet systems have proven to be very effective local computer networks.■

Acknowledgments

Many people have contributed to the success and evolution of the Ethernet local computer network. Bob Metcalfe and David Boggs built the Experimental Ethernet at the Xerox Palo Alto Research Center, and Tat Lam built and supplied the many transceivers. Since then, Ed Taft, Hal Murray, Will Crowther, Roy Ogus, Bob Garner, Ed Markowski, Bob Printis, Bob Belleville, Bill Gunning, and Juan Bulnes have contributed to the design and implementation of the Ethernet. Cooperation among Digital Equipment Corporation, Intel, and Xerox also produced many important contributions to the Ethernet Specification.

References

1. R. C. Crane and E. A. Taft, "Practical Considerations in Ethernet Local Network Design," *Proc. 13th Hawaii Int'l Conf. Systems Sciences,* Jan. 1980, pp. 166-174.
2. R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM,* 19:7, July 1976, pp. 395-404.
3. R. M. Metcalfe, D. R. Boggs, C. P. Thacker, and B. W. Lampson, "Multipoint Data Communication System with Collision Detection," US Patent No. 4,063,220, Dec. 13, 1977.
4. J. F. Shoch, *"An Annotated Bibliography on Local Computer Networks"* (3rd ed.), Xerox Parc Technical Report SSL-80-2, and IFIP Working Group 6.4 Working Paper 80-12, Apr. 1980.
5. *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications,* Version 1.0, Digital Equipment Corporation, Intel, Xerox, Sept. 30, 1980.
6. D. R. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe, "PUP: An Internetwork Architecture," *IEEE Trans. Comm.,* Apr. 1980, pp. 612-624.
7. H. Zimmermann, "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Comm.,* Apr. 1980, pp. 425-432.
8. Y. K. Dalal, "The Information Outlet: A New Tool for Office Organization," *Proc. On-line Conf. Local Networks and Distributed Office Systems,* London, May 1981, pp. 11-19.
9. V. G. Cerf and P. K. Kirstein, "Issues in Packet-Network Interconnection," *Proc. IEEE,* Vol. 66, No. 11, Nov. 1978, pp. 1386-1408.
10. F. C. Shoute, "Decentralized Control in Computer Communication," Technical Report No. 667, Division of Engineering and Applied Physics, Harvard University, Apr. 1977.
11. R. M. Metcalfe, "Packet Communication," Thesis Harvard University, Project MAC Report MAC TR-114, Massachusetts Institute of Technology, Dec. 1973.
12. J. F. Shoch and J. A. Hupp, "Performance of an Ethernet Local Network—A Preliminary Report," *Local Area Comm. Network Symp.,* Boston, May 1979, pp. 113-125. Revised version *Proc. Compcon Spring 80,* San Francisco, pp. 318-322.
13. J. F. Shoch and J. A. Hupp, "Measured Performance of an Ethernet Local Network," *Comm. ACM,* Vol. 23, No. 12, Dec. 1980, pp. 711-721.
14. E. G. Rawson and R. M. Metcalfe, "Fibernet: Multimode Optical Fibers for Local Computer Networks," *IEEE Trans. Comm.,* July 1978, pp. 983-990.
15. D. R. Boggs and R. M. Metcalfe, Communications network repeater, US Patent No. 4,099,024, July 4, 1978.
16. C. P. Thacker et al., "Alto: A Personal Computer," Xerox Palo Alto Research Center Technical Report CSL-79-11, Aug. 1979.
17. "The Dorado: A High-Performance Personal Computer," Three Reports, Xerox Palo Alto Research Center, CSL-81-1, Jan. 1981.

18. J. F. Shoch, *Local Computer Networks,* McGraw-Hill, in press.

19. J. L. Hammond, J. E. Brown, and S. S. Liu, "Development of a Transmission Error Model and an Error Control Model," Technical Report RADC-TR-75-138, Rome Air Development Center, 1975.

20. R. Bittel, "On Frame Check Sequence (FCS) Generation and Checking," ANSI working paper X3-S34-77-43, 1977.

21. J. F. Shoch, "Internetwork Naming, Addressing, and Routing," *Proc. Compcon Fall 78,* pp. 430-437.

22. Y. K. Dalal and R. S. Printis, "48-bit Internet and Ethernet Host Numbers," *Proc. Seventh Data Comm. Symp.,* Oct. 1981.

23. R. M. Metcalfe, "Steady-State Analysis of a Slotted and Controlled Aloha System with Blocking," *Proc. Sixth Hawaii Conf. System Sciences,* Jan. 1973. Reprinted in *Sigcom Review,* Jan. 1975.

24. D. Cohen, "On Holy Wars and a Plea for Peace," *Computer,* Vol. 14, No. 10, Oct. 1981, pp. 48-54.

25. J. F. Shoch, D. Cohen, and E. A. Taft, "Mutual Encapsulation of Internetwork Protocols," *Computer Networks,* Vol. 5, No. 4, July 1981, pp. 287-301.

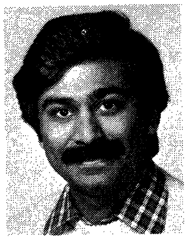
26. A. D. Birrell et al., "Grapevine: An Exercise in Distributed Computing," *Comm. ACM,* Vol. 25, No. 4, Apr. 1982, pp. 260-274.

27. H. Sturgis, J. Mitchell, and J. Israel, "Issues in the Design and Use of a Distributed File System," *ACM Operating Systems Rev.,* Vol. 14, No. 3, July 1980, pp. 55-69.

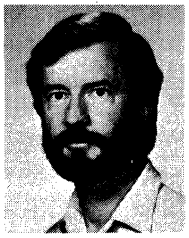
28. D. K. Gifford, "Violet, an Experimental Decentralized System," Xerox Palo Alto Research Center, CSL-79-12, Sept. 1979.

29. J. F. Shoch and J. A. Hupp, "Notes on the 'Worm' Programs—Some Early Experiences with a Distributed Computation," *Comm. ACM,* Vol. 25, No. 3, Mar. 1982, pp. 172-180.

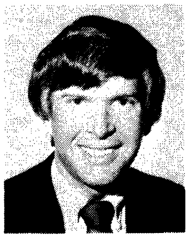
John F. Shoch is deputy general manager for office systems in the Office Products Division of Xerox Corporation. From 1980 to 1982, he served as assistant to the president of Xerox and director of the corporate policy committee. He joined the research staff at the Xerox Palo Alto Research Center in 1971. His research interests have included local computer networks (such as the Ethernet), internetwork protocols, packet radio, and other aspects of distributed systems. In addition, he has taught at Stanford University, is a member of the ACM and the IEEE, and serves as vice-chairman (US) of IFIP Working Group 6.4 on local computer networks. Shoch received the BA degree in political science and the MS and PhD degrees in computer science from Stanford University.



Yogen K. Dalal is manager of services and architecture for office systems in the Office Products Division of Xerox Corporation. He has been with the company in Palo Alto since 1977. His research interests include local computer networks, internetwork protocols, distributed systems architecture, broadcast protocols, and operating systems. He is a member of the ACM and the IEEE. He received the B. Tech. degree in electrical engineering from the Indian Institute of Technology, Bombay, in 1972, and the MS and PhD degrees in electrical engineering and computer science from Stanford University in 1973 and 1977, respectively.



David D. Redell is a staff scientist in the Office Products Division of Xerox Corporation. He was previously on the faculty of the Massachusetts Institute of Technology. His research interests include computer networks, distributed systems, information security, and computer architecture. He received his BA, MS, and PhD degrees in computer science from the University of California at Berkeley.



Ronald C. Crane, a founder of 3Com Corporation in Mountain View, California, now heads advanced engineering for the firm. From 1977 to 1980 he served as a technical staff member and subsequently a consultant to Xerox's Office Products Division in Palo Alto where he was a principal designer of the Digital, Intel, Xerox Ethernet system. His research interests have included adaptive topology packet networks, digital broadcasting systems (Digicast), and baseband transmission systems. He is a member of the ACM and IEEE. He received the BS degree in electrical engineering from the Massachusetts Institute of Technology in 1972 and the MS degree in electrical engineering from Stanford University in 1974.

In this distributed office information system, Ethernets can be interconnected either directly or via public data networks. Systems from other vendors are connectable through protocol conversion gateways.

Special Feature
Use of Multiple Networks in the Xerox Network System

Yogen K. Dalal, Xerox

Managing information is an integral part of today's office, and Xerox's Network System is a distributed office information system that provides tools for doing this. With these tools, office personnel can create, store, retrieve, display, modify, reproduce, and share information in ways that encourage creativity and increase productivity. Workstations like Star help to simplify creation, modifiying, and displaying information.[1] Electronic filing, printing, database, and mail systems simplify storing, retrieving, reproducing, and sharing information.

With the continuing improvements in the price/performance ratio of computing and communications, the structure of computerized office information systems is changing. We no longer need large centralized systems to realize economies of scale. Instead, we can push intelligence back into the workstation, and decentralize resources by function into dedicated servers to create a system that is a collection of loosely coupled elements tied together by a communication network. The Network System is just such a system, in which expensive resources are shared and information is exchanged among users.

Within an organization, we typically find natural localities of activity and interaction. Interaction between localities generally decreases as they are farther apart. While the nature and characteristics of the interaction between close and distant stations are different, both are essential to the functioning of an organization. The Ethernet local computer network[2-4] provides digital transmission of data, and satisfies most of the requirements for local office communications. The Ethernet, however, was designed in the context of an overall network architecture and is viewed as one component of an internetwork communication system that serves many diverse devices connected to many different kinds of networks.[5-8]

An internetwork architecture allows the communication system to be reconfigured to satisfy the immediate and future requirements of the user. For example, the Network System may have only one Ethernet initially and then be expanded (without software modification) to contain two or more Ethernets, which are interconnected directly or via other communication media, whose choice depends on the volume, frequency, and dispersion of communications. Public and private packet-switching facilities can be used to carry higher dispersions of low-volume office communication, and as facilities for lower cost, higher rate, modemless digital transmission become available, they can be used to carry higher volumes of data.

Of concern in this article are the major features of the Network System's internetwork communication system, in particular, its ability to use different kinds of networks (Figure 1). Protocol layers above the internetwork communication system permit different kinds of office services to be added as the need arises, thereby allowing an organization to minimize the initial purchase cost and to control any system expansion. The article also describes how the Network System can be connected to systems (network-based or stand-alone) from other vendors that obey different protocols by using protocol conversion gateways at different levels.

Internetwork model

An internetwork, or internet as it is often called, is simply an interconnection of networks. The communication system underlying the Network System embodies the fundamental principles associated with store-and-forward, packet-switching, datagram, internetwork communication, and is modeled after the experimental Pup internetwork system developed at the Xerox Palo Alto Research Centers.[7] This approach is similar to that adopted by the Advanced Research Projects Agency's Internet Protocols.[9,10] But other techniques can be used in interconnecting communication systems, such as the X.75 virtual-circuit model.[11-14]

The fundamental unit of information flow in the Xerox internet is the media-, processor-, and application-independent internet packet. An internet packet contains control information, source and destination network addresses, and data. Data may range from a few bits to several thousand bits. Internet packets are routed through the internetwork as datagrams via store-and-forward system elements called internetwork routers (also called internetwork gateways). Internetwork routers connect networks together. Internet packets are routed from a source machine to a destination machine, potentially through a variety of networks, each encapsulating and decapsulating the internet packet according to its rules. The internet gives its best effort to deliver an internet packet; that is, Xerox does not mandate that the internet always deliver a packet once and only once or that it deliver packets in the order submitted.

Fortunately the functions provided by a communication system can be layered, and an internet is defined by dividing the network layer specified by the ISO open

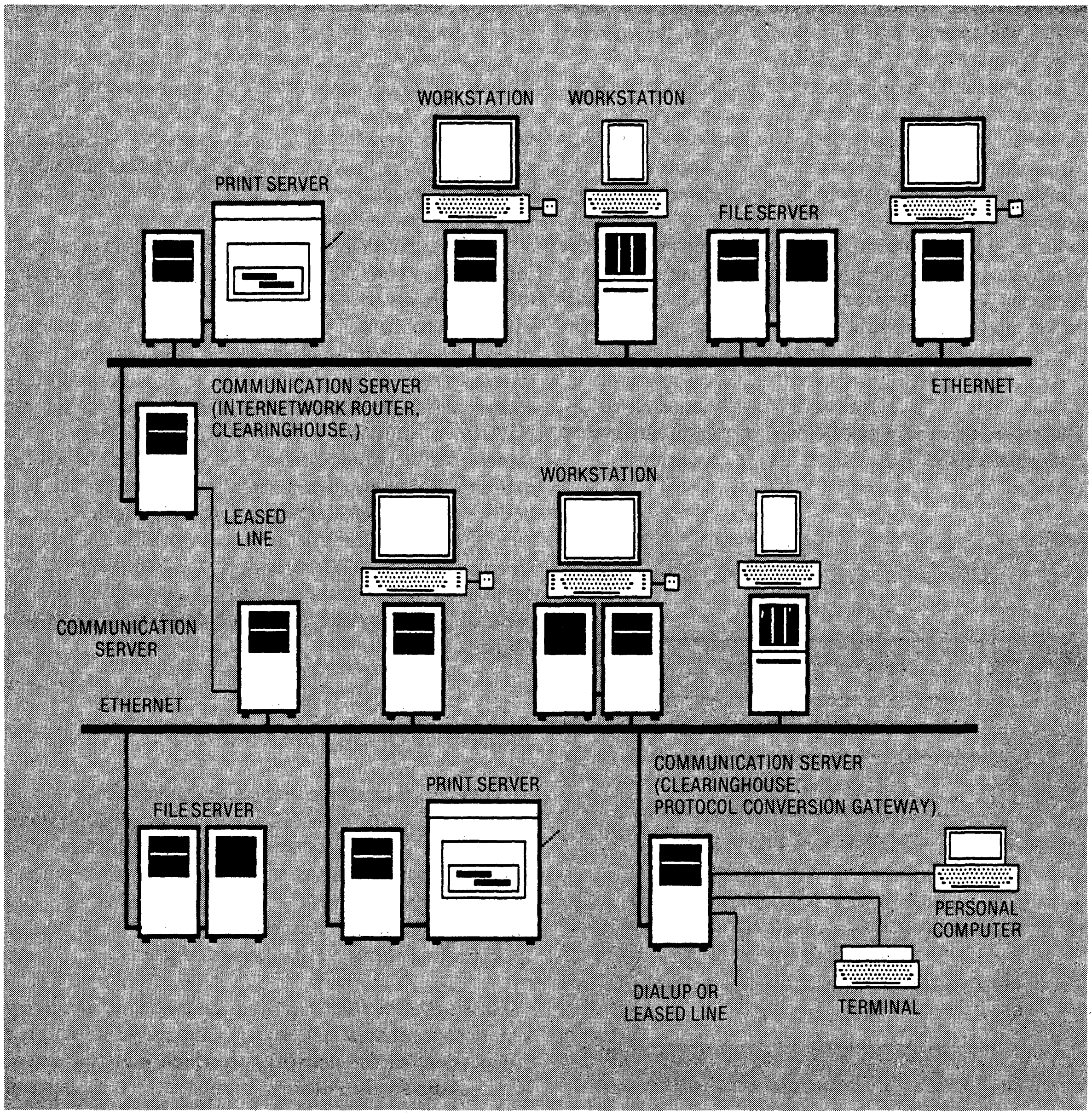


Figure 1. The Xerox Network System, a distributed office information system.

systems interconnection reference model[15,16] into two sublayers: the *network-specific* sublayer and the *internet* sublayer (Figure 2). The internet sublayer is concerned with addressing a source and a destination in the internet and routing the internet packet through one or more networks toward its goal. Other standards groups are now including this sublayer in their architecture.[17]

A transport protocol sufficient for the needs of all possible clients cannot really be defined, so Xerox did not attempt such a task; instead a number of transport protocols based on the internet packet were specified[18] (as was done in the Pup architecture). These protocols accomplish general communication functions, such as delivery of sequenced data, notification of errors, and exchange of routing information. The protocol architecture permits implementers with special needs to build their own transport protocols. Since both the incorrect delivery of data and the delivery of incorrect data are serious problems, steps are taken at different layers, as appropriate, to greatly reduce the probability that these errors will occur. But bear in mind that the internet sublayer gives only its best effort.

An office information system should provide the right tools for manipulating information, but it must also provide mechanisms for protecting information. Access control, authentication, and security, where appropriate and needed, are provided at protocol layers above the internet sublayer.[19-21]

Xerox specifies one internet protocol, and assumes that an internet packet may be delayed in its delivery for periods averaging a few hundred milliseconds; an exceptional packet may be delayed about a minute. The maximum internet packet lifetime is therefore assumed to be about a minute. Packet lifetimes within the internet are bounded by this value in the majority of internet configurations. Therefore, this value can be used in fine-tuning system performance and designing transport protocols.

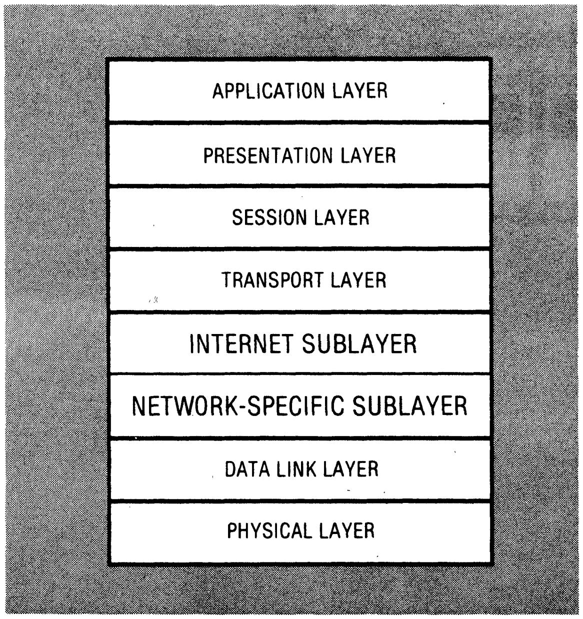


Figure 2. Enhanced ISO open systems interconnection reference model with seven layers.

The layers up to and including the network-specific sublayer depend heavily on the particular transmission medium involved, but they must perform certain special functions to transport an internet packet. In the Network System, a network is a transmission medium configured to carry internet packets, and a transmission medium is any communication equipment configured to carry data. The transmissions media now under consideration are expected to operate at bandwidths from low kilobits to megabits. A network can be a broadcast network, a multicast network, or a point-to-point network. A broadcast network is one in which the routing algorithms of the network allow a packet to be transmitted to all hosts connected to the network. A multicast network is one in which the routing algorithms of the network allow a packet to be transmitted to some subset of the hosts connected to the network. Point-to-point networks permit the routing of packets only from one host to another. The Ethernet is an example of a broadcast and multicast network, while a phone line and Telenet[22] are examples of a point-to-point network.

A host is any system element that supports the Network System communication protocols and is connected to a network. A socket is a uniquely identified data structure within a host, to which internet packets can be delivered, and from which internet packets can be transmitted. A socket is inherently a bidirectional structure, able to both send and receive packets.

The internet delivers packets as datagrams among sockets in much the same way that the post office transfers letters between post office boxes. The sockets may be on the same host, on hosts on the same network, or on hosts on different networks. A host that receives an internet packet first delivers the packet to the appropriate socket, and then the client of the socket demultiplexes the packet according to its transport protocol type. In this respect, the Network System approach differs from that used in other internetwork architectures such as the one defined by the ARPA Internet Protocol, which does not include a socket number in its network address—a host receiving an ARPA internet packet demultiplexes it according to its protocol type, and the next higher level of protocol then has the option of defining a socketlike object.

Properties of network addresses

A network address is a sequence of three fields *n, h,* and *s,* where *n* is 32-bit network number that uniquely identifies a network in an internet, *h* is a 48-bit host number that uniquely identifies a host across all Network Systems processors ever manufactured (by Xerox and others), and *s* is a 16-bit socket number that uniquely identifies a socket within the operating system of a host.

Host number. Host numbers are absolute, and every system element must be assigned a unique 48-bit number independent of the networks to which it is connected. Xerox chose an absolute host numbering scheme instead of the more conventional network-specific host numbering scheme.[23,24] Absolute host numbers have many advantages when building large distributed systems. Operating systems and application software can use this number in generating unique identifiers.[25] Also, when a host is moved from one network to another, its host number does not change, making alterations to the hardware or special bookkeeping unnecessary. Since such alterations are required when using network-specific host numbers, use of absolute host numbers substantially reduces field service overhead. However, when a host is moved to another network, the network addresses of resources in it will change, thereby requiring the update of resource directories, etc. This higher-level operation must be performed often but can be automated through software procedures.

The host number will probably be hard-wired (using jumper wires, dip-switches, or PROMs) to some part of the machine, for example, the backplane or one of the processor boards. If this piece of hardware is replaced, the host number of the machine and its associated software could change. Reinitializing the software may then be necessary but would typically have to be done anyway if other characteristics of the hardware changed. Note that a machine's host number can change, but no two machines can have the same number at the same time.

Xerox internets consist, for the most part, of Ethernets, which is the main reason that Ethernet addresses are identical to 48-bit host numbers.[24] This structure is strictly for convenience, and in no way compromises the generality of the architecture. When a host is connected to more than one Ethernet, its 48-bit Ethernet address on all Ethernets is equal to its 48-bit host number.

Network number. Since a host number uniquely identifies a specific host, the network number field would seem redundant, but it is needed for internetwork routing. When the network number is included in the network address, each host has to know only the (partial) path to each network rather than that to each host—significantly reducing the amount of information that must be retained. A host may be connected to more than one network but still has a unique identity, even though a socket within it has multiple network addresses. In other words, sources or destinations of internet packets can have more than one network address, but no two sources or destinations can have the same network address.

An internet packet addressed to a host contains the identity of the network to which the source believes the host is connected. Internetwork routers attempt to route the internet packet to the host via this network. If no route to the specified network exists, the packet is not delivered and client software must use another network address. A higher level binding agent called the clearinghouse supplies all network addresses for resources such as file servers, print servers, or a user's mailbox.[20] All networks within an internet have unique network numbers.

Socket number. A socket is inherently a bidirectional structure, able to both send and receive internet packets at the same address. Certain socket numbers are considered well known; that is, the service performed by software using them is statically defined. Each host supplying a specific well-known service does so at the same well-known socket. All other socket numbers can be reused.

Multidestination addressing. Multicast is the delivery of a packet to more than one destination, and it can be performed at either the internetwork or intranetwork level if the transmission medium supports the concept. The Network System supports internetwork multicast. Broadcast is a special case of multicast in which a packet is delivered to all hosts in the internet. The need for a generalized multicast capability arises from the anticipated need for a more general addressing capability.[26-28] Broadcast is used in many situations to search for an object or to inform all hosts of an event.[29] Although all applications can be designed without this capability, multicast provides some performance improvements.

A multicast group is the list of intended recipients of the packet and can be specified by explicitly enumerating the destinations or by using an identifier that has a suitable encoding. Before accepting a multicast packet every system element uses an acceptance filter to determine whether or not the packet is intended for that particular system element.

Multicast groups are indentified at the internet sublayer by logical host numbers. Each host must have exactly one physical host number, but a simple extension to the host numbering scheme permits a group of hosts to be identified as a logical host. One bit in the host number field indicates whether the number is a physical host number or a multicast identifier. The logical host number, "all hosts," is 48 ones and is reserved for broadcast.

Physical host numbers are assigned to system elements at manufacture time or later using carefully controlled administrative procedures.[24] Multicast identifiers are assigned by some dynamic resource allocation mechanism, by administrative procedure, or by both, depending on whether or not multicast identifiers can be reused.

Since internetwork multicast involves communication between one sender and many recipients, the recipients must use the same network address: a multicast network address. The socket number in all multicast network addresses could be the same, but then all packets for all multicast groups must be received at one socket and demultiplexed using a multicast transport protocol. Having one socket number for all multicast network addresses is not necessary. In fact, a multicast group can have a number of multicast network addresses, each with a different socket number so that semantically different packets can be easily demultiplexed.

Since all recipients of a multicast packet must receive the packet on the same socket, a set of well-known sockets are reserved for multicast. Assigning multicast well-known socket numbers requires the same form of allocation as that for multicast identifiers. A somewhat centralized mechanism is a reasonable choice for the assignment of multicast identifiers and associated well-known socket numbers. The way they are assigned—by administrative procedure or through a dynamic resource-allocation strategy—is a matter of style and a function of the rate at which new multicast network addresses are created, the number of multicast network addresses available, and the need to reuse them.

In general, the three fields that define a network address of an internet packet can take on the values unknown, all, or specific (Table 1). Zero is reserved for unknown, one for all, and any other value for specific, but a value from each class may not be appropriate for each field of a network address. For network addresses containing physical host numbers, for example, the network number can be unknown or specific, while the host and socket numbers must be specific and cannot have the value unknown. A network number of unknown implies that the packet should be transmitted on the locally connected networks. The network number unknown stands for the networks to which the host is connected until the host discovers the specific values. The notion of directing a packet to all sockets within a host has not proved useful and is therefore disallowed.

For multicast network addresses, the network number can be unknown, specific, or all; the host number is specific or all; and the socket number is specific. A multicast group, by its very definition at the internet layer, can span many individual networks. Directed multicast is the delivery of an internet packet to the members of the multicast group on the network contained in the destination multicast network address. The network number in such directed multicast internet packets is unknown or specific. A network number of unknown implies that the packet should be transmitted on the locally connected networks. The network number unknown again stands for the networks to which the host is connected until the host discovers the specific values. Directed broadcast is the delivery of an internet packet to all machines on the specified network.

In global multicast or global broadcast the internet packet is delivered to all members of the group within the entire internet. The internet does not support global multicast or broadcast, but this and other forms of multicast (such as expanding search rings[29]) are implemented by a series of directed multicasts.

Since multicast refers to the delivery of a packet to multiple destinations, the internet protocol permits multicast identifiers to appear only in the destination network address field of an internet packet. The internet layer gives its best effort to deliver multicast packets to their destinations.

Table 1.
Combination of network, host, and socket numbers in network addresses.

NETWORK ADDRESS	NETWORK NO.	HOST NO.	SOCKET NO.
PHYSICAL	SPECIFIC/ UNKNOWN	SPECIFIC	SPECIFIC
MULTICAST			
DIRECTED	SPECIFIC/ UNKNOWN	SPECIFIC	SPECIFIC
GLOBAL	ALL	SPECIFIC	SPECIFIC
BROADCAST			
DIRECTED	SPECIFIC/ UNKNOWN	ALL	SPECIFIC
GLOBAL	ALL	ALL	SPECIFIC

Internetwork delivery

In the delivery of internet packets, packets are routed from source to destination through zero or more internetwork routers using the network address of the destination host. The internet packet is usually encapsulated for transmission through the various communication networks; the encapsulation specifies the addressing and delivery mechanisms specific to that network. Each communication network may have a different form of internal addressing. When an internetwork packet is to be transported over a communication medium, the immediate destination of the packet must be determined and specified in the encapsulation. The immediate destination is determined directly from the network address if it is the final destination or through a routing table if it is an intermediate destination.

Encapsulation and decapsulation. The absolute host number may have no relation to the internal addressing schemes used by the communication networks, and during encapsulation, the absolute host number is merely a name that must be translated to an address on the network. Translation involves consulting a translation table, possibly in conjunction with the routing table if we assume that the routing table supplies the absolute host number of the next internetwork router rather than its network-specific address. (Internetwork routers will then have other network addresses, if direct communication is needed to exchange routing information or statistics.)

In a very general internet, the overhead resulting from the translation of an absolute host number to an internal address can be large in both space and time and requires the maintenance of translation tables in all hosts. Since Network System internets are expected to contain many Ethernets, Xerox chose to support the absolute host numbering scheme directly on the Ethernet to avoid translation. Therefore, for Ethernets the absolute host number is an address, not a name.

When internet packets traverse other communication networks that do not support 48-bit absolute host numbers—like the Bell Telephone DDD network, Telenet, or other public or private data networks—hosts and internetwork routers must have translation tables to translate absolute host numbers into internal addresses. These tables do not cause many operational problems, other than setup and maintenance at a limited number of hosts and internetwork routers. Absolute internet host numbers are not widely used because designers of internetworks have little or no control over the design of the constituent communication networks and are thus forced to use network-specific host numbers instead.

Figure 3 illustrates a typical internet and shows how an internet packet is routed from a source host *s* to a destination host *d* through an internet router host *r*. When the internet packet is transmitted through network *B*, the absolute host number *d* is translated into the network-specific host number *m*.

By decoupling a system element's host numbers from the connected network, we can solve many hard internetwork routing problems in situations involving network partitioning, multihoming, mobile hosts, etc. (Sunshine describes these situations and the new problems they create.[30]) With this decoupling, stand-alone workstations that implement Network System protocols can connect to an internetwork router using any communication media.

Since multicast groups are identified by logical host numbers, the routing of directed multicast packets by the internet is no different from the delivery of an internet packet addressed to a physical host. If the target network is not a broadcast network and does not support multicast the way the Ethernet does, then the 48-bit multicast group identifier must be translated upon entering the target network into one or more 48-bit physical host numbers. A separate copy of the packet is then delivered to each host on the target network. As we have just seen, if the target network does not support delivery of 48-bit addressed packets, then another host number translation is necessary.

Routing tables. Internet packets are routed as datagrams through the internet using a store-and-forward algorithm that relies on a routing table in each host. This database directs packets toward other internetwork routers on their way to the final destination, and routing information is maintained in a manner very similar to the

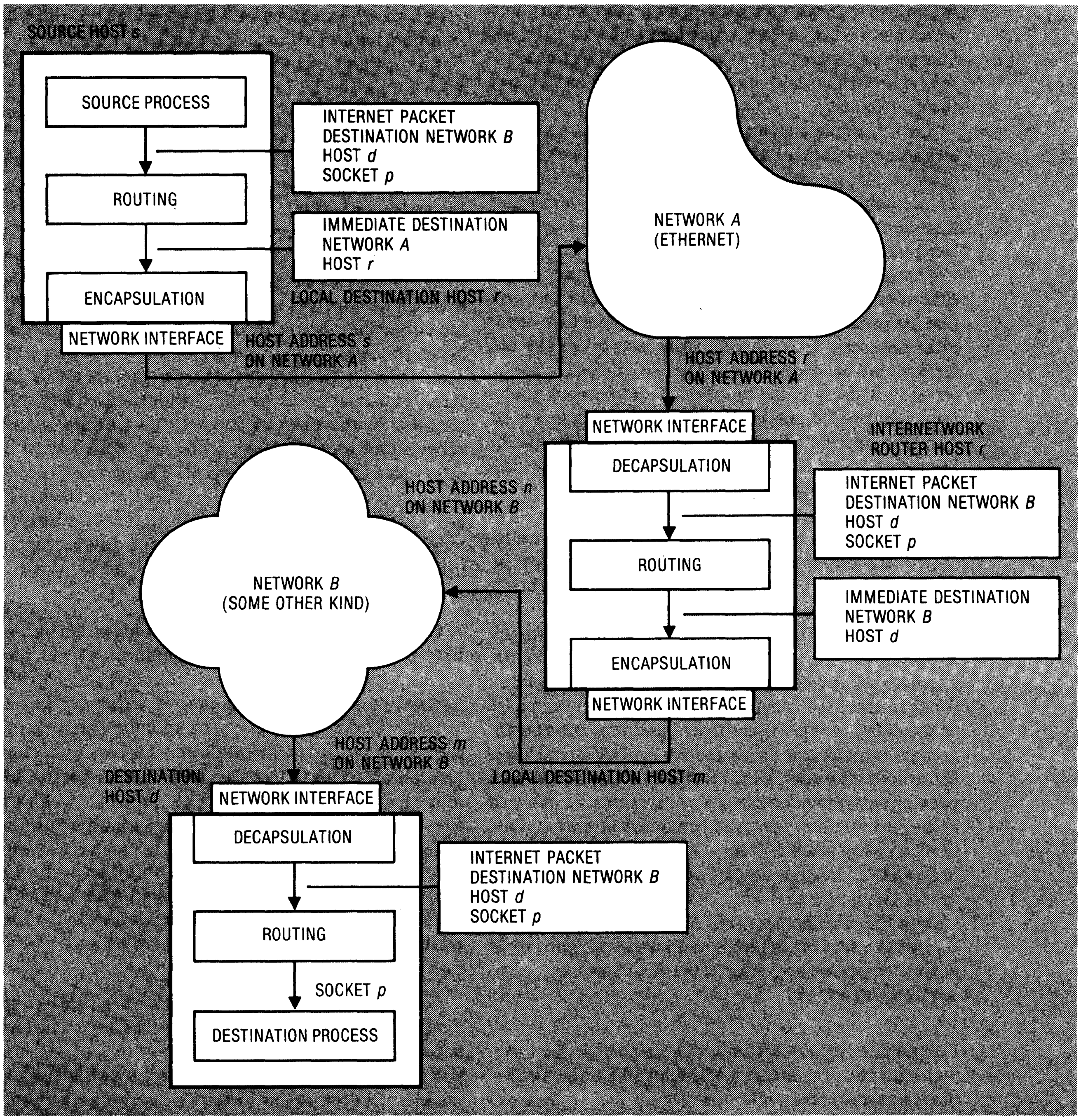


Figure 3. Internetwork store-and-forward delivery.

old Arpanet adaptive procedures.[31] Neighboring internetwork routers periodically exchange routing information using a connectionless transport protocol.[29] Changes in internetwork topology may cause the routing tables in different internetwork routers to become momentarily inconsistent, but the algorithm is stable, since routing tables rapidly converge to a consistent state and remain that way until the topology changes again.

A host that is not an internetwork router obtains routing information by polling internetwork routers on its directly connected networks. The host may obtain updates periodically if it receives the broadcast packets that other internetwork routers are exchanging; if not, then it may periodically poll internetwork routers for updates with which to manage its cache. If more than one internetwork router is providing paths to other networks, then an internetwork router or host can merge the information it receives and thus select the best route for packets directed to any network.

The fundamental assumption made for routing is that an internet contains at most a few hundred networks. As a consequence, the network number space can be flat, and all internetwork routers can maintain a complete image of the internet in their routing tables. Area routing is not being used.

Another assumption is that an internet may contain Ethernets, leased lines, public or private data networks that use packet or circuit switching, and other high-speed local networks. Even though these networks have different delay-bandwidth characteristics, the metric for internetwork delay is the number of internetwork router hops. The predominant network in Xerox internets is expected to be Ethernet, and hence the algorithms use the broadcast capabilities of the internet protocol and Ethernet to exchange routing information. Public data networks, however, may not support broadcast, or broadcast may be expensive; in these configurations, internetwork routers must know the identity of the other internetwork routers with which they periodically exchange routing information.

These assumptions and, therefore, the protocol for updating routing tables may not be suitable in large complex internetwork topologies for a number of reasons. First, for all internetwork routers to hold and update an image of the entire internet is inefficient and time consuming. Second, large internets have networks of different delay-bandwidth characteristics, which, in conjunction with queuing delays in internetwork routers, must be reflected in the information exchanged between routers if they are to pick nearly optimal routes. Mismatches in the delay-bandwidth characteristics of the constituent networks can result in congestion.

Both the routing algorithm and routing table update algorithm and their implementations are designed to be flexible to permit upgrades in the field when new algorithms are developed.

Maximum packet lifetime. To ensure that the maximum lifetime of a packet is less than about one minute, the internet layer in each internetwork router must discard an internet packet that has been forwarded through more than 15 networks, and the network layer must discard packets that have remained on its "transmit queue" for longer than a tenth of the maximum packet lifetime. Packets may, therefore, be discarded; this activity is consistent with the previously described expectation that the internet gives only its best effort to deliver a packet.

Fragmentation. We assume that all hosts can process internet packets up to a certain length, and that all internetwork routers are capable of forwarding such packets. Networks incapable of transmitting such packets must perform internetwork fragmentation and reassembly at the network layer.[32,33]

Recursive encapsulation. Protocol hierarchies are traditionally defined and implemented in a strict "top-down" fashion, with higher layers calling lower layers. Such a structure is also used in many operating systems and is desirable when dealing with data abstraction and proof of correctness. However, when products from different vendors subject to different protocol hierarchies (each consistent with the ISO OSI model) coexist, one hierarchy must often refer to the other. For example, when the Pup internetwork was connected to the ARPA internetwork, ARPA internet packets were encapsulated in Pup internet packets for transmission over the Pup internetwork and vice versa.[34] The important observation is that the interface (often called a service interface) between the internet and network-specific sublayers is the same for all networks whether they offer datagram services, circuit-oriented services, or host-based transport services. In the Network System, the network-specific sublayer for a particular network may transport an internet packet using any protocol. The network-specific sublayer is, therefore, said to be implemented "recursively."[17] Care must be taken to avoid deadlocks, since protocol modules interact in much the same way as program modules in a large software system.

Circuit-oriented communication systems. So far, we have used a rather intuitive definition of the term network—a transmission medium configured to carry internet packets. This definition of a network can be applied without modification to any datagram communication system, but further refinement allows us to easily incorporate circuit-oriented communication systems (like those available from the phone company, or PTTs), which can provide dedicated or dynamically set-up circuits. Stand-alone system elements can now be connected to an internetwork router through phone lines or other commercially available circuit-oriented communication systems to become remote system elements (with the same privileges as any other system element in the internet) during the time they are part of the internet.

A communication system (a virtual circuit or leased line in many cases) that just connects internetwork routers is not considered a network with its own network number unless no other way exists to create a network address for the internetwork router. The two internetwork routers can be thought of as "half internetwork" routers connected by a "line." In computing routing metrics, the delay-bandwidth characteristic of the line is absorbed by the internetwork routers. Algorithms or heuristics that set up and take down short-lived circuits are needed.

A communication system that just connects stations (workstations and servers) is also not considered a network with its own network number (it could simply be zero), since no internetworking is taking place. However, we specifically require that stand-alone Ethernets have a network number, since the probability that they will remain stand-alone is very low.

An entire circuit-oriented communication system (such as the Bell Telephone's DDD system) that connects stations to an internetwork router is not considered a network either. Rather, the collection of circuit ports at an internetwork router is thought of as a network with its own network number. (A circuit port can be a physical "plug" identified by a modem or a logical channel as in X.25, for example.) We call such networks cluster networks. The network number in a station on such a circuit-oriented communication system corresponds to that of the cluster network where the circuit terminates during the time it is active. The internetwork router at which its circuit terminates can change from call to call without any problems.

Stand-alone workstations therefore are not assigned a network number of their own and instead acquire one when they connect to the internet. Stand-alone servers, however, must register with the clearinghouse all network addresses corresponding to all internetwork routers from which they can be called. Further, each internetwork router associates a local address (analogous to a phone number or an X.25 address) with the server's host number, so the server can be called when a packet with its destination first arrives. No resulting operational problem is expected, since the number of stand-alone servers is small, while the number of stand-alone workstations is very large.

In this internetwork configuration the definition of network has been stretched to be a collection of circuit ports (possibly of different types) at an internetwork router. The circuit-handling drivers can be thought of as a "switch" connecting the stations and the internetwork router. A station that communicates with another on the same network sends its packets to the switch (resident as a "star network" on the internetwork router) that loops the packet back onto another circuit.

Hence, if hosts are classified into two groups, internetwork routers and stations, where stations may be either workstations or servers, then a network becomes a communication system configured to carry internet packets, such that it connects stations together and is connected to an internetwork router. A network is assigned a network number. Stations on the same network can intercommunicate without the explicit aid of the internetwork router's store-and-forward services.

In general, techniques to initialize and manage the internet are optimized for datagram and broadcast networks and permit decentralized management of the internet.[35] When the internet includes nonbroadcast datagram networks and circuit-oriented communication systems, additional specialized initialization and binding is necessary.

Protocol conversion gateways

The set of protocols provided by the Network System architecture enables the design of many office services and permits any two hosts implementing the protocols to communicate. However, foreign devices (devices obeying other protocols) must also be able to access services provided by the Network System, and the Network System hosts need to access services provided by foreign devices. Since these devices use their own protocols to communicate instead of Network System protocols, protocol conversion gateways are needed to handle any incompatibilities. These gateways communicate with the foreign devices using their protocol and convert to Network System protocol at any of the necessary layers.

In general, seven kinds of incompatibilities are possible when communicating with foreign devices—one corresponding to each of the seven ISO open systems interconnection layers. If the incompatibility occurs only at the network-specific sublayer, the internetwork architecture described earlier accommodates it. Problems arise when incompatibilities exist through higher layers. On the basis of experiences with protocol design and implementation, we conclude that the seven-layer protocol hierarchy performs two important functions: (1) it provides a reliable data-transparent session with the communicating device and (2) it interprets the data transmitted to achieve some application. Therefore, instead of seven different types of incompatibility we have only two: transport incompatibility and application incompatibility. A gateway transport function deals with the first, and a gateway application function deals with the second.

The gateway transport function communicates with a foreign device using its protocols, up to and including the transport and session layer, and provides a generic "stream" interface (much like the Bell Laboratories Unix pipe),[25] by which data and control are transported to and from the foreign device. This stream interface preserves all the semantics associated with the foreign device but couches them in a general form, thereby allowing communication with many different kinds of devices using the same interface. This gateway stream interface can be accessed by software in the host connected to the foreign device and by software in other Network System hosts through use of a Network System application protocol. The gateway transport function can then be thought of as "converting" the foreign device's transport protocol into a Network System application protocol.

The gateway application function communicates with the foreign device at its application level and uses the gateway transport function as a communication vehicle. The foreign device may be connected to the hosts implementing the gateway transport function directly or through phone lines or public data networks. The gateway application functions convert to and from a Network System service, such as filing, printing, or electronic mail.

Figure 4 illustrates how users at simple terminals access services in the Network System and manipulate resources. An interactive terminal service, which is a gateway application function, manipulates Network System resources on behalf of the user at an interactive terminal and uses an external communication service, which is a gateway transport function, to transport characters to and from the terminal.

Figure 5 illustrates how a user at a Star workstation accesses services in an IBM mainframe, and manipulates resources. The workstation software creates the illusion that it is an IBM 3270 terminal by implementing a gateway application function and uses an external communication service to transport data to and from an IBM mainframe.

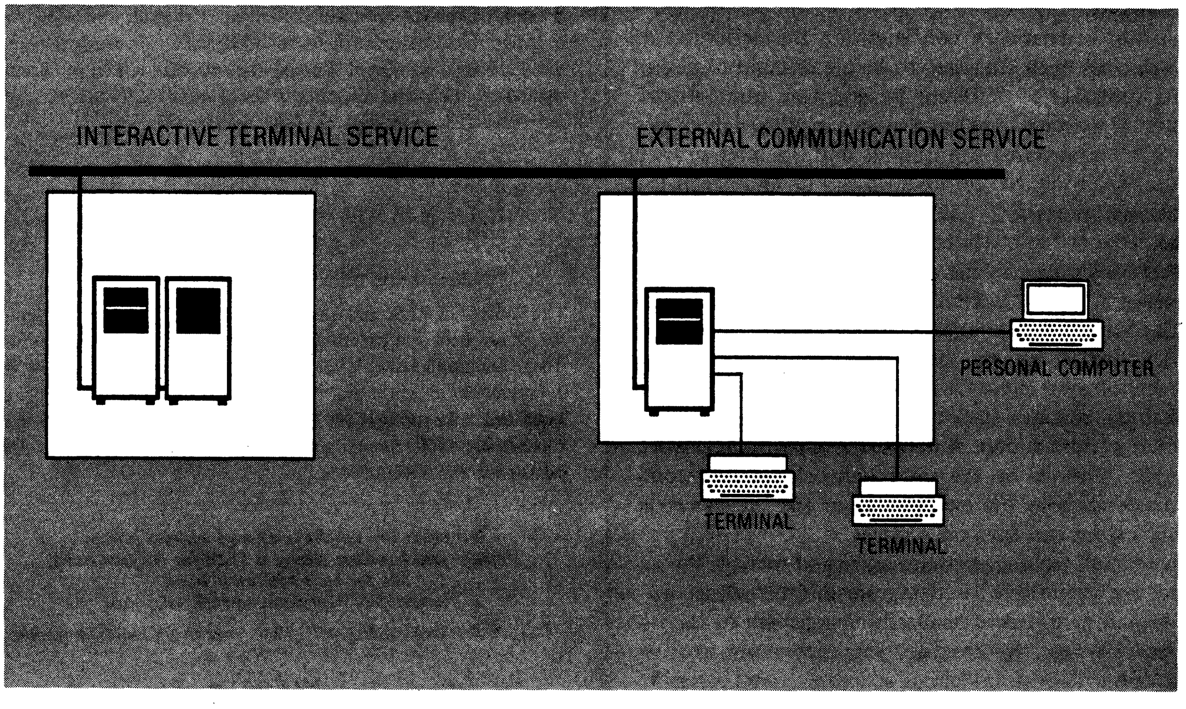


Figure 4. An interactive terminal service.

These simple examples illustrate the architecture for protocol conversion gateways, in particular, how a foreign device's transport protocol is converted into a Network System application protocol. The gateway application and transport functions may be in the same host or in two different hosts. With this architecture, gateways can be built to exchange electronic mail with systems like Telex or Teletex or to exchange documents between different filing systems.

The protocol architecture underlying the Network System is layered and meets the goals of the ISO open systems interconnection reference model. The model has been generalized, however, by introducing an internet sublayer and by permitting any layer to be recursively defined in terms of other layers.

The internet delivers packets from any host connected to it to any other connected host, and access control is performed by higher levels of protocol. The internet architecture permits complex topologies and the use of different communication media and public data networks. The network-specific sublayer supporting Xerox's internet protocol must, in addition, perform certain functions such as intranetwork fragmentation, if necessary. The internet sublayer defines one protocol and supports the use of many different protocols at the transport and network layers. The protocol hierarchy has an hourglass shape, with the internet protocol at the narrow point.

The protocol conversion gateway architecture permits the design of any number of gateway functions. The gateway transport function communicates with foreign devices, which may be connected to the Network System through various communication systems using their protocols. Gateway application functions deal with the hard problem of converting one service into another. ■

Acknowledgments

The design and development of the Network System involved many people from Xerox's Office Products Division and Palo Alto Research Centers. The internetwork architecture embodies principles that evolved from experience with the Pup internetwork and ARPA Internet Protocol. Members of the Systems Development Department designed and implemented this communication system.

References

1. D. C. Smith et al., "The Star User Interface: An Overview," *Proc. NCC*, May 1982, pp. 515-528.
2. R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM,* Vol. 19, No. 7, July 1976, pp. 395-404.
3. *The Ethernet, a Local Area Network: Data Link Layer and Physical Layer Specifications,* Digital Equipment, Intel, and Xerox Corporations, Version 1.0, Sept., 1980.
4. J. F. Shoch et al., "Evolution of the Ethernet Local Computer Network," Xerox Office Products Division, Palo Alto, OPD-T8102, Sept. 1981, and *Computer*, Vol. 15, No. 8, Aug. 1982, pp. 10-26.
5. C. A. Sunshine, "Interconnection of Computer Networks," *Computer Network*, Vol. 1, No. 3, Jan. 1977, pp. 175-195.
6. V. G. Cerf and P. K. Kirstein, "Issues in Packet-Network Interconnection," *Proc. IEEE,* Vol. 66, No. 11, Nov. 1978, pp. 1386-1408.
7. D. R. Boggs et al., "Pup: An Internetwork Architecture," *IEEE Trans. Comm.* Vol. COM-28, No. 4, Apr. 1980, pp. 612-624.

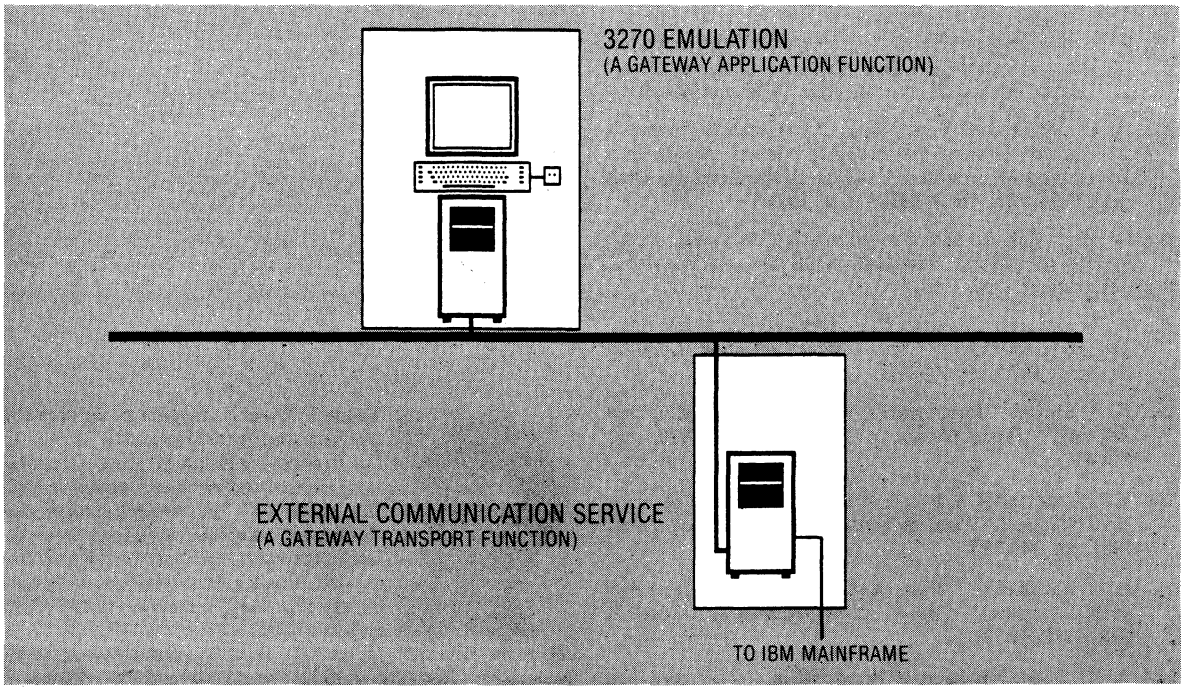

3270 EMULATION
(A GATEWAY APPLICATION FUNCTION)
EXTERNAL COMMUNICATION SERVICE
(A GATEWAY TRANSPORT FUNCTION)
TO IBM MAINFRAME

Figure 5. IBM 3270 emulation.

8. J. B. Postel, "Internetwork Protocol Approaches," *IEEE Trans. Comm.*, Vol. COM-28, No. 4, Apr. 1980, pp. 604-611.

9. *DoD Standard Internet Protocol,* J. Postel, ed., NTIS ADA079730, Jan. 1980, also in *ACM Computer Comm. Review,* Vol. 10, No. 4, Oct. 1980, pp. 2-51, revised; as "Internet Protocol—DARPA Internet Program Protocol Specification," RFC 791, USC/Information Sciences Institute, Sept. 1981.

10. J. B. Postel, C. A. Sunshine, and D. Cohen, "The ARPA Internet Protocol," *Computer Networks*, Vol. 5, No. 4, July 1981, pp. 261-271.

11. *Recommendation X.25/Interface Between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for Terminals Operating in the Packet Mode on Public Data Networks*, CCITT Orange Book, Vol. 7, International Telephone and Telegraph Consultative Committee, Geneva.

12. *Proposal for Provisional Recommendation X.75 on International Interworking Between Packet Switched Data Networks*, CCITT Study Group VII Contribution No. 207, International Telephone and Telegraph Consultative Committee, Geneva, May 1978.

13. G. R. Grossman, A. Hinchley, and C. A. Sunshine, "Issues in International Public Data Networking," *Computer Networks*, Vol. 3, No. 4, Sept. 1979, pp. 259-266.

14. M. H. Unsoy and T. A. Shanahan, "X.75 Internetworking of Datapac and Telenet," *Proc. Seventh Data Comm. Symp.*, Oct. 1981, pp. 232-239.

15. H. Zimmermann, "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Comm.*, Vol. COM-28, No. 4, Apr. 1980, pp. 425-432.

16. *ISO Open Systems Interconnection—Basic Reference Model*, DP 7498, ISO/TC97/SC 16 N 719, Aug. 1981.

17. *Network Layer Principles*, ECMA/TC23/81/169 prepared by TC24 TG NS, Nov. 1981, p. 4.

18. *Internet Transport Protocols*, Xerox System Integration Standard, XSIS-028112, Stamford, Connecticut, Dec. 1981.

19. R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Comm. ACM*, Vol. 21, No. 12, Dec. 1978, pp. 993-999.

20. D. C. Oppen and Y. K. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," Xerox Office Products Division, Palo Alto, OPD-T8103, Oct. 1981.

21. *Courier: The Remote Procedure Call Protocol*, Xerox System Integration Standard, XSIS-038112, Stamford, Connecticut, Dec. 1981.

22. L. G. Roberts, "Telenet: Principles and Practice," *Proc. European Computing Conf. Comm. Networks*, London, England, 1975, pp. 315-329.

23. J. F. Shoch, "Internetwork Naming, Addressing, and Routing," *Proc. Compcon Fall 78*, Sept. 1978, pp. 430-437.

24. Y. K. Dalal and R. S. Printis, "48-bit Internet and Ethernet Host Numbers," *Proc. Seventh Data Comm. Symp.*, Oct. 1981, pp. 240-245.

25. D. D. Redell et al., "Pilot: An Operating System for a Personal Computer," *Comm. ACM*, COM-Vol. 23, No. 2, Feb. 1980, pp. 81-92.

26. Y. K. Dalal, "Broadcast Protocols in Packet Switched Computer Networks," PhD dissertation, Stanford University, DSL Tech. Report 128, Apr. 1977.

27. Y. K. Dalal and R. M. Metcalfe, "Reverse Path Forwarding of Broadcast Packets," *Comm. ACM*, Vol. 21, No. 12, Dec. 1978, pp. 1040-1048.

28. J. M. McQuillan, "Enhanced Message Addressing Capabilities for Computer Networks," *Proc. IEEE*, Vol. 66, No. 11, 1978, pp. 1517-1527.

29. D. R. Boggs, "Internet Broadcasting," PhD dissertation, Stanford University, Jan. 1982 (also available from Xerox Palo Alto Research Center).

30. C. A. Sunshine, "Addressing Problems in Multi-Network Systems," *Proc. IEEE Infocom*, Mar. 1982, pp. 12-18.

31. J. M. McQuillan, G. Falk, and I. Richer, "A Review of the Development and Performance of the ARPANET Routing Algorithm," *IEEE Trans. Comm.*, Vol. COM-26, No. 12, Dec. 1978, pp. 1802-1811.

32. J. F. Shoch, "Packet Fragmentation in Internetwork Protocols," *Computer Networks*, Vol. 3, No. 1, Feb. 1979, pp. 3-8.

33. J. F. Shoch and L. Stewart, "Interconnecting Local Networks via the Packet Radio Network," *Proc. Sixth Data Comm. Symp.*, Nov. 1979, pp. 153-158.

34. J. F. Shoch, D. Cohen, and E. A. Taft, "Mutual Encapsulation of Internetwork Protocols," *Proc. Trends and Applications: 1980—Computer Network Protocols*, May 1980, pp. 1-11; revised version in *Computer Networks*, Vol. 5, No. 4, July 1981, pp. 287-301.

35. S. M. Abraham and Y. K. Dalal, "Techniques for Decentralized Management of Distributed Systems," *Proc. Compcon Winter 80*, Feb. 1980, pp. 430-436.

Yogen K. Dalal is manager of services and architecture for office systems in the Office Products Division of Xerox Corporation. He has been with the company in Palo Alto since 1977. His research interests include local computer networks, internetwork protocols, distributed systems architecture, broadcast protocols, and operating systems. He is a member of the ACM and the IEEE. He received the B. Tech. degree in electrical engineering from the Indian Institute of Technology, Bombay, in 1972, and the MS and PhD degrees in electrical engineering and computer science from Stanford University in 1973 and 1977, respectively.

Reprinted from PROCEEDINGS OF THE SEVENTH DATA COMMUNICATIONS SYMPOSIUM, October 1981

48-bit Absolute Internet and Ethernet Host Numbers

Yogen K. Dalal and Robert S. Printis

Xerox Office Products Division
Palo Alto, California

Abstract

Xerox internets and Ethernet local computer networks use 48-bit absolute host numbers. This is a radical departure from practices currently in use in internetwork systems and local networks. This paper describes how the host numbering scheme was designed in the context of an overall internetwork and distributed systems architecture.

1. Introduction

The Ethernet local computer network is a multi-access, packet-switched communications system for carrying digital data among locally distributed computing systems [Metcalfe76, Crane80, Shoch80, Ethernet80, Shoch81]. The shared communications channel in the Ethernet system is a coaxial cable—a passive broadcast medium with no central control. Access to the channel by stations or *hosts* wishing to transmit is coordinated in a distributed fashion by the hosts themselves, using a statistical arbitration scheme called *carrier sense multiple access with collision detection* (CSMA/CD). Packet address recognition in each host is used to take packets from the channel.

Ethernet packets include both a source and a destination *host number*, that is, the "address" of the transmitter and intended recipient(s), respectively. Ethernet host numbers are 48 bits long [Ethernet80]. 48 bits can uniquely identify 281,474,977 million different hosts! Since the Ethernet specification permits only 1024 hosts per Ethernet system, the question that is often asked is: "why use 48 bits when 10, or 11, or *at most* 16 will suffice?" This paper answers this question, and describes the benefits of using large absolute host numbers.

We view the Ethernet local network as one component in a store-and-forward datagram *internetwork* system that provides communications services to many diverse devices connected to different networks (see, for example, [Boggs80, Cerf78]). Our host numbering scheme was designed in the context of an overall network and distributed system architecture to take into account:

- o the use of host numbers by higher-level software,
- o the identification of a host or a logical group of hosts within the internetwork,
- o the addressing of a host or a logical group of hosts on the Ethernet channel, and
- o the management of distributed systems as they grow, evolve and are reconfigured.

Sections 2, 3, and 4 of this paper describe the pros and cons of various host numbering schemes in inter- and intra-network systems, and describe the properties and advantages of our host numbering scheme. Section 5 discusses our host numbers in the context of "names" and "addresses" in network systems. Sections 6 and 7 describe the reasons for choosing 48 bits, and the mechanisms for managing this space.

2. Addressing Alternatives

The *address* of a host specifies its location. A network design may adopt either of two basic addressing structures: *network-specific host addresses*, or *unique host addresses* [Shoch78]. In the first case, a host is assigned an address which must be unique on its network, but which may be the same as an address held by a host on another network. Such addresses are sometimes called *network-relative addresses*, since they depend upon the particular network to which the host is attached. In the second case, each host is assigned an address which is unique over all space and time. Such addresses are known as *absolute* or *universal addresses*, drawn from a *flat address space*. Both network-specific and absolute host addresses can have any internal structure. For the purposes of this paper, we will treat them as "numbers" and will use host addresses and host numbers interchangeably.

To permit internetwork communication, the network-specific address of a host usually must be combined with a unique network number in order to produce an unambiguous *internet address* at the next level of protocol. Such internet addresses are often called *hierarchical internet addresses*. On the other hand, there is no need to combine an absolute host number with a unique network number to produce an unambiguous internet address. Such internet addresses are often called *flat internet addresses*. However, internetwork systems using flat internet addresses, containing only the absolute host number, will require very large routing tables indexed by the host number. To solve this problem, a unique network number, or other routing information is often included in the internet address as a "very strong hint" to the internetwork routing machinery; the routing information has been separated from host identification.

We anticipate that there will be a large number of hosts and many (local) networks in an internetwork, thus requiring a large internet address space. For example, the Pup-based internetwork system [Boggs80] currently in use within Xerox, as a research network, includes 5 different types of networks, has over 1200 hosts, 35 Experimental Ethernet local networks, and 30 internet routers (often called internetwork gateways). Figure 1 illustrates the topology of the internet in the San Francisco Bay Area.

If network-specific addressing is used, then the host number need only be large enough to accommodate the maximum number of hosts that might be connected to the network. Suitable installation-specific administrative procedures are also needed for assigning numbers to hosts on a network. If a host is moved from one network to another it may be necessary to change its host number if its former number is in use on the new network. This is easier said than done, as each network must have an administrator who must record the continuously changing state of the system (often on a piece of paper tacked to the wall!). It is anticipated that in future office environments, host locations will be changed as often as telephones are changed in present-day offices. In addition, a local network may be shared by uncooperative organizations, often leading to inconsistent management of the network. Duplication of addresses may lead to misdelivered or undeliverable data. Thus, the overall management of network-specific host numbers can represent a severe problem.

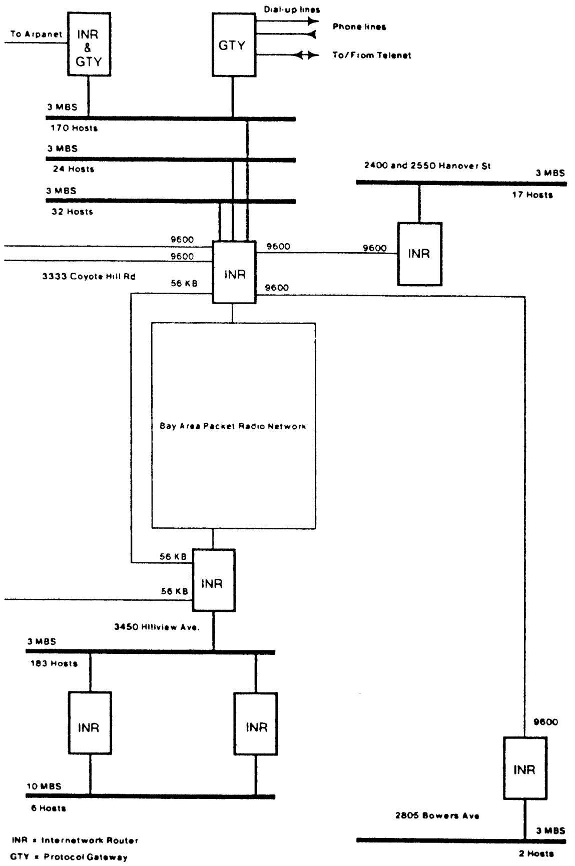


Figure 1. The Xerox Pup-based Experimental Internetwork in the Bay Area

The use of absolute host numbers in an internetwork provides for reliable and manageable operation as the system grows, as machines move, and as the overall topology changes, if the (local) network can directly support these large host numbers. This is true, because the host is given one number or *identity* when it is first built, and this number is never modified when the network configuration changes. A distributed system can be effectively managed if the special purpose parameterizing of the hardware can be reduced to a minimum. The absolute host number space should be large enough to ensure uniqueness and provide adequate room for growth.

Since an absolute host number is a property of a host rather than its location in the network to which it is connected, the number should not be associated with, nor based on, a particular network interface or controller. A host connected to zero or more networks has only one identity, which should not be "hard wired" into a particular interface or controller, but should be setable from the station (see Section 5). The address of this host on all connected networks that directly support absolute host numbers will, in general, be the same as the host's identity (see Sections 3 and 5). A host connected to a network that does not directly support absolute host numbers will, in addition, have an address relative to that network.

Such host numbers can be used by operating systems software to generate unique numbers for use by the file system, resource manager, etc. [Redell80, Abraham80]. By decoupling the host's number from the network to which it is connected, a uniform mechanism can be applied to networked and stand-alone workstations so that they may interact at higher levels. For example, both stand-alone and networked Pilot-based [Redell80] workstations may generate files that are identified by unique numbers and then exchange them by copying them onto removable storage media such as floppy disks.

Xerox internetwork systems will use flat internet addresses containing 48-bit host numbers, and a unique network number as a very strong routing hint. The internet address(es) for an object or resource in the internetwork is obtained from a distributed agent called the *clearinghouse*; it serves a function similar to the telephone system's "white" and "yellow" pages [Oppen81]. The user of the resource does not compute or determine the network number after discovering the host number of the resource—the network number is included in the binding information returned from the clearinghouse. We believe that our host number space is large enough for the foreseeable future (see Section 6). We expect that these internetworks will be built primarily from Ethernet local networks and thus directly support 48-bit absolute host numbers on the Ethernet channel. An internet packet is still encapsulated in an Ethernet packet when it is transmitted on the Ethernet channel.

48-bit host numbers lead to large Ethernet and internet packets. We believe that this will not pose a problem as both local and public data networks continue to offer higher bandwidths at reasonable costs, and the memory and logic costs associated with storing and processing these numbers continue to become lower with the advances in LSI technology.

We further justify our choice of absolute host numbers in the next section by comparing internetwork routing techniques that use hierarchical and flat internet addresses. We show that routing based on flat internet addresses is very general, and especially efficient if the constituent (local) networks directly support the absolute host number.

3. Internetwork Delivery

In this section, we illustrate the pros and cons of using hierarchical and flat internet addresses for internetwork delivery by comparing the techniques prescribed by the Arpa Internet Protocols [IP80] and the Pup Protocols [Boggs80], with those prescribed by the new Xerox internetwork protocols.

A host is identified in an internetwork by its internet address. In general, a host may have many internet addresses, but an internet address can identify only one host.

Hierarchical internet addresses have a routing decision implicitly encoded in them because they specify the network through which a packet must be delivered to the host. This is not necessarily true for flat internet addresses. Flat internet addresses may contain routing information hints in them, and in such cases a sophisticated routing mechanism is free to use or ignore these hints.

The delivery of *internet packets* involves routing the packet from source host to destination host, through zero or more *internet routers* based on the internet address of the destination host. The internet packet usually must be encapsulated for transmission through the various communication networks, on its way from source host to destination host. The encapsulation specifies addressing and delivery mechanisms specific to that communication network. Each communication network may have a different form of internal addressing. When an internetwork packet is to be transported over a communication medium, the *immediate destination* of the packet must be determined and specified in the encapsulation. The immediate destination is determined directly from the internet address if it is the *final destination*, or through a routing table if it is an *intermediate destination*. We do not discuss mechanisms and metrics for managing routing tables in this paper.

The structure of the internet address influences the algorithms used for determining immediate destinations during encapsulation.

Consider flat internet addresses first: the absolute host number in a flat internet address may have no relation to any of the internal addressing schemes used by the communication networks. Hence, during encapsulation, as far as each of the communication networks is concerned, the absolute host number is a *name* that must be translated to an *address* on the network. This involves consulting some form of a *translation table*, possibly in conjunction with the routing table (we assume that the routing table supplies the absolute host number of the next internet router rather than its network-specific address, so that internet routers know one anothers' internet addresses should they wish to directly communicate, for the purpose of exchanging routing information or statistics, etc.). In a very general internetwork, the overhead of performing an absolute host number to internal address translation can be large both in space and time, and also requires the maintenance of translation tables in all hosts. Xerox internetworks will consist primarily of Ethernets. Since absolute host numbers have many other advantages, we chose the internal addressing on an Ethernet system to be identical to the absolute host number to avoid translation. Therefore, as far as Ethernet systems are concerned, the absolute host number is indeed an address and not a name. When Xerox internet packets traverse other communication networks that do not support our absolute host numbers, like the Bell Telephone DDD network, Telenet, or other public or private data networks, translation tables will have to exist in the necessary hosts and internet routers to perform translation from absolute host numbers to internal addresses. We feel that this will not cause many operational problems, other than setting up and maintaining these translation tables at appropriate (and limited) hosts and internet routers. Flat internet addresses are not in widespread use because the designers of internetworks have had little or no control over the design of the constituent communication networks, and thus, have been forced to use hierarchical internet addresses, rather than flat internet address containing routing information or hints.

Flat internet addresses provide a vehicle for solving many of the hard internetwork routing problems in situations like network partitioning, multihoming, mobile hosts, etc. But they create others! These situations are described in greater detail in [Sunshine81].

A host in an internetwork that has hierarchical internet addresses has as many internet addresses as the number of networks to which it is connected. It is the encoding of the network-specific host number itself that distinguishes various schemes in this category. There are two cases, one represented by the Arpa Internet Protocols and the other by the Pup Protocols.

The Arpa Internet Protocols specify that the internet address is an 8-bit network number followed by a 24-bit host number. The host number is encoded such that it is synonymous with the internal addressing scheme of the communication network to which the host is connected. For example, a host connected to the Bay Area Packet Radio Network has a network-relative internal address of 16 bits, and therefore the host number in its internet address will contain these 16 bits in the "least significant" positions. During encapsulation, if the immediate destination is the final destination then it is equal to the host number in the destination internet address, and if the immediate destination is an intermediate destination then it is determined from the routing tables and has the right format. For such a scheme to work, the space reserved for the host number must be as large as the largest internal addressing scheme expected in any communication network. In the case of the Arpa Internet Protocols, this is already too small since it cannot encode new Ethernet host numbers!

The Pup protocols encode the host number in the internet address with only 8 bits, and so cannot be used to encode the various network-specific host numbers. The Pup Protocols were designed to be used in an internetwork environment consisting mainly of interconnected Experimental Ethernet systems which have 8-bit internal addresses, and that is why the host number in the internet address is 8 bits long. Hence, even though the Pup Protocols use network-specific host numbers, when packets are transmitted through non-Experimental Ethernets a translation table is needed just as for absolute host numbers. For example, when Pup internet packets traverse the Bay Area Packet Radio Network, the 8-bit host number of the internet routers must be translated into the 16-bit ID used within the radio network [Shoch79].

Here is another way to look at internet addresses: whether the host number is absolute or network-specific, if it does not encode the communication network's internal addresses, then it may be necessary to translate from the internet host number to the communication network's internal address whenever the packet is to be transmitted over the network.

4. Multicast

In addition to identifying a single host, our absolute host numbering scheme provides several enhanced addressing modes. *Multicast* addressing is a mechanism by which packets may be targeted for more than one destination. This kind of service is particularly valuable in certain kinds of distributed applications, such as the access and update of distributed data bases, teleconferencing, and the distributed algorithms which are used to manage the network (and the internetwork). Multicast is supported by allowing the destination host numbers to specify either a physical or "logical" host number. A logical host number is called a *multicast ID* and identifies a group of hosts. Since the space of multicast IDs is large, hosts must filter out mulitcast IDs that are not of interest. We anticipate wide growth in the use of multicast and all implementations should, therefore, minimize the system load required to filter unwanted multicast IDs.

Broadcast is a special case of multicast; a packet is intended for *all* hosts. The distinguished host number consisting of all ones is defined to be the broadcast address. This specialized form of multicast should be used with discretion, however, since all nodes incur the overhead of processing such packets.

By generalizing the host number to encompass both physical and logical host numbers, and by supporting this absolute host number within the Ethernet system (which is inherently broadcast in nature) we have made it possible to implement multicast efficiently. For example, perfect multicast filtering can be performed in hardware and/or microcode associated with the Ethernet controller. Since logical host numbers are permitted in flat internet addresses we also have the capability for *internetwork multicast*. This is, however, easier said than done as the multicast ID may span many networks. Internetwork multicast and reliable multicast are subjects we are currently researching; an appreciation of the problems can be found in [Dalal78 and Boggs81].

5. Names and Addresses

The words "name" and "address" are used in many different ways when describing components of a computer system. The question that we often get asked is: "is a 48-bit number the name or the address of a host computer?" In the area of computer-communications we have tried to develop a usage that is consistent with that found elsewhere, and an excellent expose of the issues may be found in [Shoch79]. An important result of this paper is that a mode of identification (whether it be a number or a string of characters) is treated as a name or address depending on the context in which it is viewed.

From an internetworking point of view, the 48-bit number asigned to a host is its identity, and never changes. Thus, the identity could be thought of as the "name" (in the very broadest sense) of the host in the internetwork. According to Shoch's taxonomy, this identity could also be thought of as a flat address, as it is recognizable by all elements of the internetwork.

The Ethernet local network is a component of an internet, and was designed to support 48-bit host numbers. One could view this design decision as "supporting host name recognition directly on the Ethernet channel" (since broadcast routing is used to deliver a packet). This would be true if a host was connected to an Ethernet at only one point—a policy decision we made for the Xerox internetwork architecture. However, this is not a requirement of the Ethernet design, and it is possible for a host to be connected to many points on a single Ethernet channel, each one potentially responding to a different 48-bit number. In this situation the 48-bit number does in fact become an address in the classical sense as it indicates "where" the host is located on the channel. One of these 48-bit numbers could also be the host's internet identity; the mapping from internet address to local network address is now more cumbersome.

5. Market Projections

We have described our reasons for choosing absolute host numbers in internet addresses, and for using them as station addresses on the Ethernet channel. The host number space should be large enough to allow the Xerox internet architecture to have a life span well into the twenty-first century. 48 bits allow for 140,737,488 million physical hosts and mulitcast IDs each. We chose this size based on marketing projections for computers and computer-based products, and to permit easy management of the host number space.

An estimate of the number of computer systems that will be built in the 1980s varies, but it is quite clear that this number will be very large and will continue to increase in the decades that follow. The U.S. Department of Commerce, Bureau of Census estimates that in 1979 there were 165 manufacturers of general-purpose computers, producing about 635,000 units valued at $6,439,000,000 [USCensus79]. There were also about 992,000 terminals and about 1,925,000 standard typewriters built! International Data Corporation estimates that during 1980-1984 there will be about 3.5 million general purpose mini, small business, and desktop computers built in the United States [IDC80]. Gnostics Concepts Inc. estimates that during 1980-1988 about 63 million central processing units (cpus) of different sizes with minimum memory will be built in the United States alone [Gnostics80].

We expect that the production of microcomputer chips will increase in the decades that follow, and there will be microprocessors in typewriters, cars, telephones, kitchen appliances, games, etc. While all these processors will not be in constant communication with one another it is likely that every now and then they will communicate in a network of processors. For example, when a car containing a microprocessor chip needs repairs, it might be plugged into a diagnostics system thereby putting the car on a communications system. During the time it is hooked into the communication network it would be very convenient if it behaved like all other computers hooked into the system.

We believe that 32 bits, providing over 2,147,483,648,000 physical host numbers and multicast IDs, is probably enough. However, when this large space is carved up among the many computer manufacturers participating in this network architecture, there are bound to be many thousands of unused numbers. It is for this reason that we increased the size to 48 bits. The next section discusses the problems of managing this space.

7. Management and Assignment Procedures

In order that an absolute host numbering scheme work, management policies are needed for the distribution and assignment of both physical and logical host numbers. The major requirement is to generate host numbers in such a way that the probability of the same number being assigned elsewhere is less than the probability that the hardware used to store the number will fail in an undetected manner. There are two ways to manage the host number space:

1) Partition the host number space into blocks and assign blocks to manufacturers or users on demand. The assignment of numbers within a block to machines is the responsibility of each manufacturer or user.

2) Formulate an appropriate algorithm for generating host numbers in a decentralized manner. For example, use a random number generator that reduces the probability of address collisions to a very small acceptable value.

Both options require the existence of an administrative procedure, and perhaps an agency supported by the user community which will have the overall reponsibility of ensuring the uniqueness of host number assignments.

The second option has a great deal of academic appeal, but nevertheless requires an administrative agency that must control the way the random number generator is used to ensure that users do not initialize it with the same *seed*. One way to accomplish this is to assign unique seeds. This is not very different from assigning unique blocks of numbers! Another way is to provide a thermal noise device on the host to generate a seed or the random host number itself. From a technical standpoint this solution is superior to using software-implemented random number generators, but administrative procedures are still necessary. An agency must certify the "correctness" of the component, i.e., it must guarantee that the component is drawing its numbers from a uniform distribution. In addition to these technical issues, the problem of controlling the assignment of multicast IDs does not lend itself to a random number assignment procedure.

The first option was selected because of its simplicity and ease of administration and control. Xerox Corporataion will manage the assignment of blocks to manufacturers. An in-house database system is being used to assign numbers and produce summaries and reports. This is very similar to the way *uniform product codes* are assigned [UPC78]. The 48-bit host number space is partitioned into 8,388,608 (2^{23}) blocks, each containing 16,777,216 (2^{24}) physical and 16,777,216 (2^{24}) logical host numbers. The partitioning is strictly syntatctic, that is, the "block number" has no semantics, and does not identify a manufacturer.

The owner of a block of host numbers should use all of them before requesting another block. That is, the host numbers within a block should be used "densely", and should not encode the part number, batch number, etc. Mechanisms by which physical host numbers within a block are assigned to machines is manufacturer dependent. Typically, a large-volume manufacturer would make PROMs containing the host number, and then perform quality control tests to ensure that there weren't any duplicates.

Multicast ID assignment is a higher-level, system-wide function, and is a subject we are investigating.

With either assignment option it is possible that two machines inadvertantly received the same host number. Suitable techniques for discovering such anomalies will have to be developed by installations, as part of their network management strategy.

The continued advances in LSI development will make it possible to manufacture an inexpensive "Ethernet chip." Even though host numbers are associated with the host and not a particular network interface, it might be useful to have a unique host number built into each chip and allow the host to read it. The host can then choose whether or not to return this number to the chip as its host number; a host connected to many Ethernet systems can read a unique number from one of the chips and set the physical host number filter to this value in all of them.

The 48-bit host number is represented as a sequence of six 8-bit bytes A, B, C, D, E, F. The bytes are transmitted on the Ethernet channel in the order A, B, C, D, E, F with the least significant bit of each byte transmitted first. The least significant bit of byte A is the *multicast bit*, identifying whether the 48-bit number is a physical or logical host number. Figure 2 illustrates how the bytes of a 48-bit host number are laid out in an Ethernet packet.

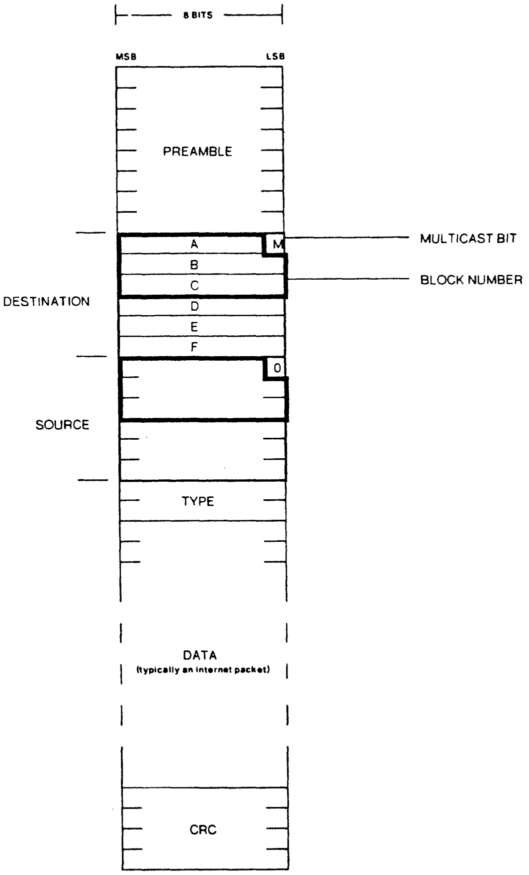


Figure 2. Ethernet Packet and Host Number Format

Although the destination address in an internet or intranet packet may specify either a physical host number or a multicast ID, the source address in a packet is generally the physical host number of the host which sent the packet. Knowing the source address is important for error control, diagnostic tests, and maintenance. A host which receives a multicast packet is also free to use that same multicast ID (the destination) in order to transmit an answer "back" to the multicast group.

8. Summary and Conclusions

We believe that all hosts should have a unique physical host number independent of the type or number of networks to which they are physically connected. With the continuing decline in the cost of computing and communications, we expect that internetworks will be very large. Many of the problems in managing the internetwork can be simplified by directly supporting the large absolute host number in the constituent networks, such as the Ethernet. Thus, addresses in the Ethernet system seem to be very generous, well beyond the number of hosts that might be connected to one local network.

The architecture of the Xerox internetwork communication system has been designed to have a life span well into the twenty-first century. We expect that it will receive wide acceptance as a style of internetworking, and therefore chose the host number to be 48 bits long. As a policy decision our internetwork architecture legislates that a host (mulitiply) connected to one or more Ethernet local networks has the same physical host number on each one.

In summary, absolute host numbers have the following properties:

- o they permit hosts to be added to, or removed from networks in the internetwork with minimum adminstrative overhead.
- o they permit mapping internet addresses to network addresses during encapsulation without translation.
- o they permit the separation of routing from addressing, which is especially useful in internetworks with multihomed or mobile hosts.
- o they provide the basis for unique identification of files, programs and other objects on stand-alone and networked hosts.
- o they support multicast, or the delivery of data to a group of recepients rather than only to a single physical host.

Although a host has the same number for use by operating system software, both within the internetwork and on an Ethernet system, none of the principles of layered protocol design have been violated. Things have simply been conveniently arranged to be optimal in the most common configurations.

We encourage designers of other local computer networks and distributed systems to use absolute host numbers from our 48-bit address space.

Acknowledgements

Our decision to support an absolute host numbering scheme in internetwork and Ethernet systems was based on many years of experience with the Pup internetwork and the Experimental Ethernet system; David Boggs, John Shoch, Ed Taft, Bob Metcalfe and Hal Murray have helped refine our ideas to their current state. Alan Kotok, Bill Strecker and others at Digital Equipment Corporation provided many recommendations on managing the host number space while we were developing the Ethernet specification.

References

[Abraham80]
Abraham, S. M., and Dalal, Y. K., "Techniques for Decentralized Management of Distributed Systems," *20th IEEE Computer Society International Conference (Compcon)*, February 1980, pp. 430-436.

[Boggs80]
Boggs, D. R., Shoch, J. F., Taft, E. A., and Metcalfe, R. M., "PUP: An internetwork architecture," *IEEE Transactions on Communications*, com-28:4, April 1980, pp. 612-624.

[Boggs81]
Boggs, D. R., "Internet Broadcasting," Ph.D. Thesis, Stanford University, 1981, in preparation, (will be available from Xerox Palo Alto Research Center).

[Cerf78]
Cerf, V. G., and Kirstein, P. K., "Issues in Packet-Network Interconnection," *Proceedings of the IEEE*, vol 66, no 11, November 1978. pp. 1386-1408.

[Crane80]
Crane, R. C., and Taft, E. A., "Practical considerations in Ethernet local network design." *Proc. of the 13th Hawaii International Conference on Systems Sciences*, January 1980, pp. 166-174.

[Dalal78]
Dalal, Y. K., and Metcalfe, R. M., "Reverse Path Forwarding of Broadcast Packets." *Communications of the ACM*, 21:12, December 1978. pp. 1040-1048.

[Ethernet80]
Intel, Digital Equipment and Xerox Corporations, *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications*. Version 1.0. September 30, 1980.

[Gnostics80]
Gnostic Concepts, Inc., *Computer Industry Econometric Service, 1980, Volume 1.*

[IDC80]
International Data Corporation. Corporate Planning Service. *Processor Data Book 1980.*

[IP80]
Postel, J., ed., *DoD Standard Internet Protocol*. January 1980. NTIS No. ADA079730, also in *ACM Computer Communication Review*, vol 10, no 4, October 80, pp. 2-51.

[Metcalfe76]
Metcalfe, R. M., and Boggs, D. R., "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM*, 19:7, July 1976. pp. 395-404.

[Oppen81]
Oppen, D. C., and Dalal, Y. K., "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," in preparation.

[Shoch80]
Shoch, J. F., and Hupp, J. A. , "Measured performance of an Ethernet local network," *Communications of the ACM*, 23:12, December 1980, pp. 711-721.

[Shoch81]
Shoch, J. F., *Local Computer Networks*, McGraw-Hill, in preparation.

[Sunshine81]
Sunshine, C., "Addressing Problems in Multi-Network Systems," in preparation.

[UPC78]
UPC Guidelines Manual, January 1978. Available from Uniform Product Code Council, Inc., 7061 Corporate Way, Suite 106, Dayton Ohio.

[USCensus79]
U.S. Department of Commerce, Bureau of Census, "Computers and Office Accounting Machines," Current Industrial Reports, 1979.

[Redell80]
Redell, D. D., Dalal, Y. K., Horsely, T. R., Lauer, H. C., Lynch, W. C. McJones, P. J., Murray, H. G., and Purcell, S. C., "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, 23:2, February 1980. pp. 81-92.

[Shoch78]
Shoch, J. F., "Internetwork Naming, Addressing, and Routing," *17th IEEE Computer Society International Conference (Compcon)*, September 1978, pp. 430-437.

[Shoch79]
Shoch, J. F., and Stewart, L., "Interconnecting Local Networks via the Packet Radio Network," *Sixth Data Communications Symposium*, November 1979, pp. 153-158.

Higher-level protocols enhance Ethernet

Internet Transport Protocols enable system elements on multiple Ethernets to communicate with one another. Courier specifies the manner in which a work station invokes operations provided by a server.

The Ethernet specification announced by Digital Equipment Corp., Intel Corp., and Xerox Corp. in 1980 only covers the lowest level hardware and software building blocks necessary for an expandable distributed computer network that can serve large office environments. Additional levels of protocol are needed to allow communication between networks and communication between processes within different pieces of equipment from different manufacturers.

Xerox' recently announced Network Systems Internet Transport Protocols and Courier: The Remote Procedure Call Protocol, define protocols that address these issues.

To serve large office environments, Ethernet's basic communication capability must be augmented in various ways. Interconnecting multiple Ethernets will circumvent the maximum end to end cable length restriction of 2.5 km, but requires mechanisms for internetwork communication. The Internet Transport Protocols offer a richer addressing scheme and a more sophisticated routing algorithm, and will enable Ethernets to be interconnected by telephone lines, public data networks, or other long-distance transmission media but will allow transmission of data larger than the 1526-byte packet-size restriction imposed by the Ethernet.

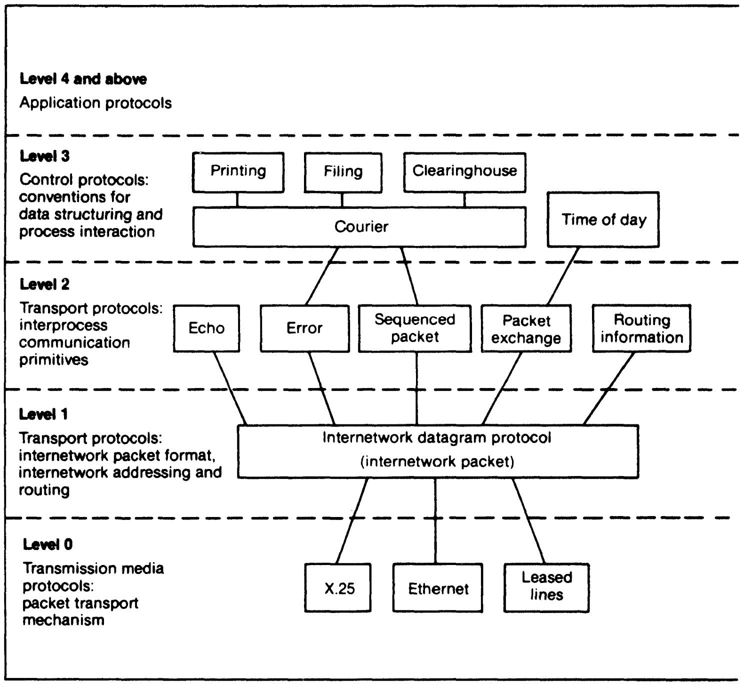
Network system protocols

As illustrated by Xerox' five-level Network Systems protocol architecture (Fig. 1), the new protocols go well beyond the original Ethernet specification, which covers level 0—physically transmitting data from one point to another. This corresponds to the physical, data link, and network (network-specific sublayer) layers in the International Standards Organization's Open Systems Interconnect (OSI) reference model. The Internet Transport protocols cover levels 1 and 2; the first level decides where the data should go, and the second for structured sequences of related packets. Levels 1 and 2 correspond to the network (internet-specific sublayer), transport, and session layers of the OSI model.

At level 3, the protocols have less to do with communication and more to do with the content of data and the control of manipulation of resources. Level 3 corresponds to the OSI model's presentation layer and is covered by Courier. Level 4 defines specific applications and corresponds to the OSI model's application layer; Xerox plans to disclose some of them later this year.

There are several protocols in this family:

- The internet datagram protocol, which defines the fundamental unit of information flow within the internetwork—the internet datagram packet.


Level 4 and above
Application protocols
Level 3
Control protocols: conventions for data structuring and process interaction
Printing
Filing
Clearinghouse
Courier
Time of day
Level 2
Transport protocols: interprocess communication primitives
Echo
Error
Sequenced packet
Packet exchange
Routing information
Level 1
Transport protocols: internetwork packet format, internetwork addressing and routing
Internetwork datagram protocol (internetwork packet)
Level 0
Transmission media protocols: packet transport mechanism
X.25
Ethernet
Leased lines

1. Network system protocols are arranged in five levels. The internet transport protocols are at levels 1 and 2; the Courier remote procedure call protocol is at level 3. Xerox plans to announce the application protocols at level 4 later this year.

James White, Manager, Electronic Mail
Yogen Dalal, Manager, Advanced Network Services
Xerox Corp. Office Products Division
3450 Hillview Ave., Palo Alto, Calif. 94304

- The sequenced packet protocol, which provides for reliable, sequenced, and duplicate-suppressed transmission of a stream of packets.
- The packet exchange protocol, which supports simple transaction-oriented communication involving the exchange of a request and its response.
- The routing information protocol, which provides for the exchange and dissemination of internetwork topological information necessary for the proper routing of datagrams.
- The error protocol, which is intended to standardize the manner in which low-level communication or transport errors are reported.
- The echo protocol, which is used to verify the existence and correct operation of a host, and the path to it.

The internet packet transport protocols embody the fundamental principles of store-and-forward internetwork packet communications. The fundamental unit of information flow is the internet packet, which is media-, processor-, and application-independent (Fig. 2).

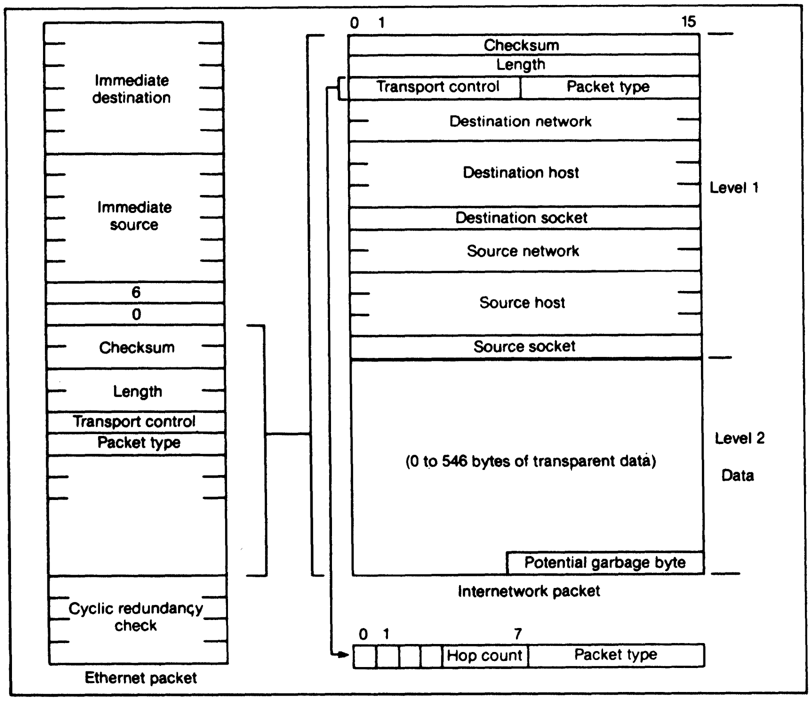
Internetwork packets are routed from one network to another via store-and-forward system elements called internetwork routers that connect transmission systems. Each datagram is treated independently by the routing machinery; it gives its best effort, but will not guarantee that packets will be delivered once and only once, or that they will be delivered in the same order in which they were transmitted.

When an internet packet is received over a transmission medium, it is first decapsulated by stripping away the immediate source and destination addresses. If the packet is destined for this host, it will be delivered to a local socket (a uniquely identified port within the operating system in a host). If the packet is to be routed to another network, it will be reencapsulated and subsequently transmitted according to the conventions of the second transmission medium.

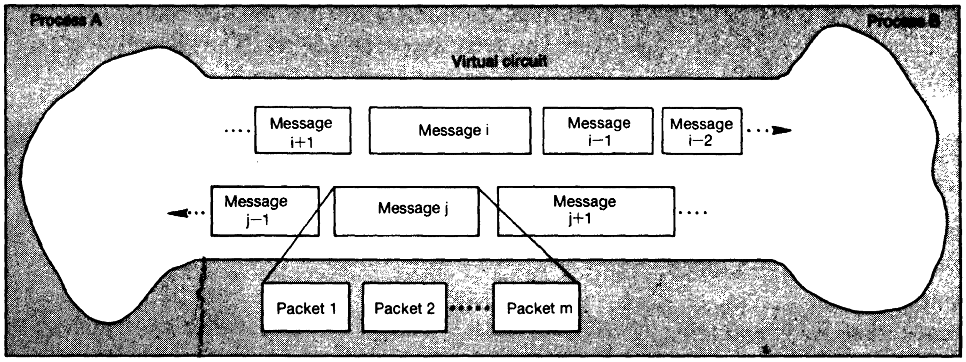
Internet packet fields fall into three categories: addressing fields, which specify the address of the destination and source of the internet packet and consist of source and destination network addresses; control fields, which are related to controlling data transmission and consist of checksum, length, transport control, and packet type fields; and data fields, which carry the data and consist of information that is interpreted only at level 2.

The network address fields provide a more general addressing mechanism than the 48-bit host number used on the Ethernet by a 32-bit network number and a 16-bit socket number. The network number reaches out to encompass multiple interconnected Ethernets or other transmission media. The socket number reaches in to distinguish among multiple post-office-box–like objects within the operating system in a machine.

The checksum is an end-to-end checksum (unlike the Ethernet's cyclic redundancy check) that is computed once by the original source of the packet and checked once by the ultimate recipient to verify the integrity of all the data it encompasses. It is an optional one's-complement add-and-left cycle (rotate) of all the 16-bit words of the internet packet, excluding the checksum word itself. Internet packets are always transmitted as an integral number of 16-bit words. A garbage byte is added at the end if the numbers of bytes is odd; this byte



2. An internet packet (16 bits wide) is encapsulated in an Ethernet packet.



3. A connection is a transient association between two processes that allows messages to flow back and forth. The sequenced packet protocol allows packets to be assembled into messages and removes the limitation on packet size at lower architectural levels.

is included in the checksum, but not in the length.

The length field carries the complete length of the internet packet measured in bytes, beginning with the checksum and continuing to the end of the data field. However, the possible garbage byte at the end is not included.

The transport control field contains a hop-count subfield, which is incremented by 1 each time the packet is handled by an internetwork router. An internetwork packet reaching its sixteenth internetwork router is discarded.

The packet type field describes how the data field is to be interpreted, providing a bridge to level 2.

A client process typically interfaces to the internetwork datagram protocol package in an operating system by acquiring a socket and then transmitting and receiving internet packets on that socket.

Two modes of communication are particularly important in building a distributed system: connections and simple transactions. Connection-oriented communications, which is supported by the sequenced packet protocol, involves an extended conversation by two machines in which much more information is conveyed than can be sent in one packet going in one direction. Thus, the need arises for a series of related packets that could number in the thousands.

Simple transaction-oriented communication, which is supported by the packet exchange protocol, involves one machine (the consumer) simply sending a request to perform an operation; the other machine (the server) performs the operation and provides information about its outcome.

Sequenced packet protocol

The sequenced-packet protocol provides reliable, sequenced, and duplicate suppressed transmission of successive internetwork packets by implementing the virtual-circuit connection abstraction, which is common to many communications systems (Fig. 3). The connection links two processes in different machines and carries a sequence of messages, each consisting of a sequence of packets, in each direction.

Arranging packets into messages and message sequences is one way to circumvent the packet-size limitation at lower levels of the protocol architecture. The sequenced packet protocol provides a mechanism to punctuate the stream of packets with end-of-message boundaries.

Each client packet gets a sequence number when it is transmitted by the source; sequence numbers are used to order the packets, to detect and suppress duplicates and, when returned to the source, to acknowledge reception of the packets. The flow of data from sender to receiver is controlled on a packet basis. The protocol specifies the format of the packets (Fig. 4) and the meaning of packet sequences.

Throughput vs buffering

One of the major design goals when implementing connections is to maximize throughput—controlling the packet flow so that the receiver accepts packets at the speed the source is sending them. But another goal is to minimize the amount of buffer resources allocated to the connection, since a typical machine, particularly a server, might have to maintain many connections (to different work stations) at the same time. Since these two goals could conflict, the system designer will have to make tradeoffs according to individual requirements.

The connection control field contains four bits that control the protocol's actions: system packet, send acknowledgment, attention, and end-of-message. The system packet bit enables the recipient to determine whether the data field contains client data or is empty and the packet has been sent only to communicate control information required for the connection to function properly. If the send acknowledgment bit is set, the source wants the receiver to acknowledge previously received packets.

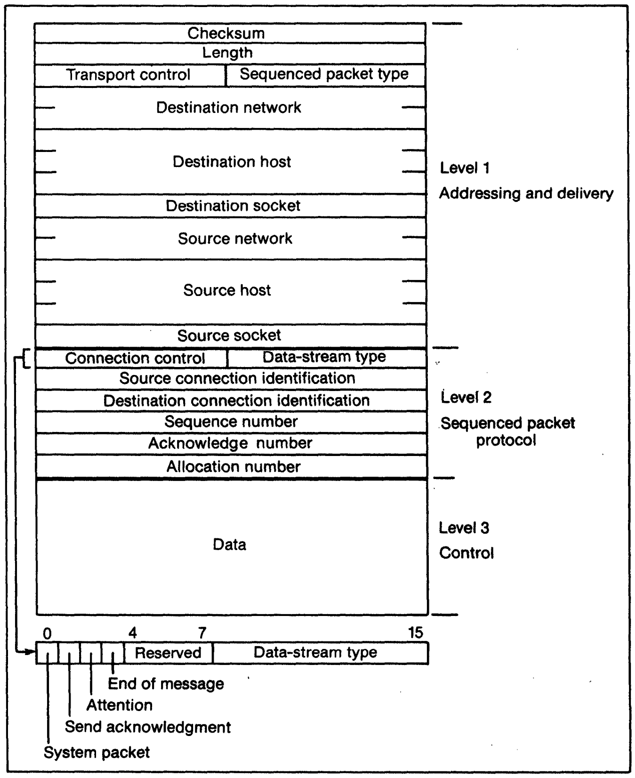
In a distributed environment, special procedures must be provided to bypass the normal flow control and interrupt a process. If the attention bit is set, the source client process wants the destination client process to be notified that this has arrived. If the end-of-message bit is set, then the packet and its contents will terminate a message and the next packet will begin the following message.

The primary bridge between this level 2 prototype and any level 3 protocols is the data stream type field, which provides information that may be useful to higher-level software in interpreting data transmitted over the connection.

Should one of the partners in a connection fail, it must be noticed by the other partner. Accordingly, each packet includes two 16-bit connection identifiers, one specified by each end of the connection. Each end tries to ensure that if it fails and is restarted, it will not reuse the same identifier. Thus, the restarted program will be easy to distinguish from the old instance of the same program.

The sequence number is a unique number assigned to each packet sent on the connection. Each direction of data flow is independently sequenced. One purpose of the sequence number is to provide a means for the receiver to reorder the incoming packets (as necessary) before presenting them to the application software. The sequence number also provides a basis for the acknowledgment and flow-control mechanisms.

The acknowledgment number field specifies the sequence number of the first packet, which has not yet been seen traveling in the reverse direction, thus identifying the next expected packet. The allocation number field specifies the sequence number of the last packet that will be accepted from the other end. However, if the attention bit is set, the allocation mechanism described will be ignored and the packet will be sent, even though the destination may have no room.



4. A sequenced packet protocol packet allows successive transmission of internet packets.

Flow control by windowing

The sequenced-packet protocol has been designed to support both high- and low-bandwidth communication. The receiving end controls the rate at which data may be sent to it; the sending end controls the frequency with which the receiving end must return acknowledgments.

The protocol controls data flow with windowing (Fig. 5). A window is a contiguous set of sequence numbers that form the current focus of the transmission. The window is a range of packets such that all packets to the left of the window—the lower-numbered packets—are understood to have been received by the destination machine. All packets to the right of the window—the higher sequence numbers—are not to be sent at that moment. All packets in the window are packets that the receiver has allowed to be sent, not all that may have been received. As the window is filled from the left, it is advanced to the right.

There are several compatible strategies for implementing this window mechanism. A conservative implementation could have windows one packet wide; an ambitious implementation might have very wide windows. The amount of buffer space allocated to the connection is traded off against performance because a very small window forces a complete two-way interaction between source and destination on every packet. But with wide windows, an entire sequence of packets can be sent in bulk by the source.

In a certain sense, these strategies conflict, two machines employing different strategies can still communicate, but at the lowest common denominator.

Establishing and terminating connections

A connection, of course, must be created before it can be used and discarded when no longer required. One end of a connection is said to be established when it knows the address (host and socket number) and connection identification of both ends of the connection. If both ends are established symmetrically, the connection is said to be open. Data can only flow reliably on an open connection; that is, a packet will be delivered to the client process only if its source-and-destination host number, socket number, and connection identification match those associated with the connection.

The first packet on a new connection will address some particular socket in the machine, and the implementation of the sequenced packet protocol will know whether any application in that machine has expressed interest in that network address. If no process has expressed an interest in the socket, the sequenced packet protocol implementation will inform the sender via the error protocol.

In order to open a connection between a consumer process and a server process that advertises service on a well-known socket, the server first establishes a service-listener process at a well-known or well-advertised socket. This process accesses the Internet Transport Protocol package at the level of the internet datagram protocol and indicates a willingness to accept packets from any source. The consumer process then creates an unestablished end of a connection. Once the consumer's packet is received, the service listener creates a new service process and creates one end of the unestablished connection. An empty packet returned by the new service process causes the consumer's end of the connection to be established.

Termination of a connection is not handled by the sequence packet protocol, but by the communicating clients. There are three separate but interlocking messages they transmit—one signifying that all data has been sent; one signifying that all the data has been received and processed; and one signifying that the sender understands and is turning to other things.

Packet exchange protocol

Transmitting a request in a packet and receiving a response via the packet exchange protocol (Fig. 6) will be more reliable than transmitting internet packets directly as datagrams, but less reliable than the sequenced packet protocol.

There are only three fields in the packet. An identification field, which contains a transaction identifier, is the means by which a request and its response are associated. A client type field indicates how the data field should be interpreted at higher levels. A data field contains whatever the higher-level protocols specify. Such a protocol might be used in locating a file server through a resource-location service, such as the Xerox Clearinghouse.

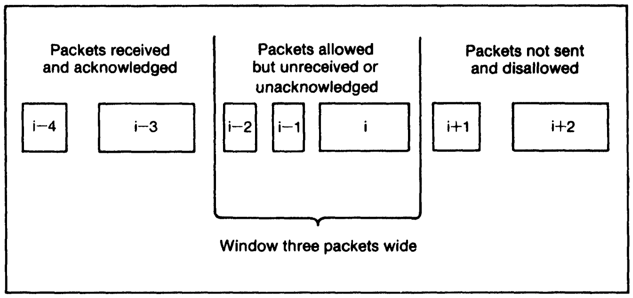
Other protocols

As dominant as the sequenced packet and packet exchange protocols are at level 2, they do not handle everything. The routing-information protocol, for one, provides for the exchange of topological information among internetwork routers and work stations.

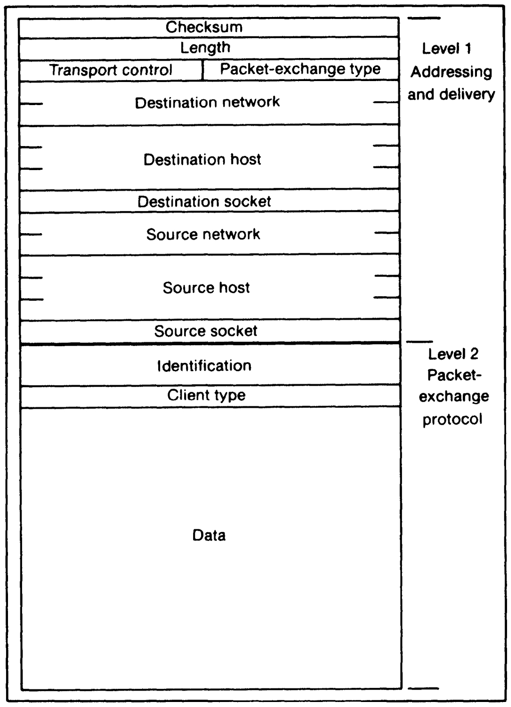
Two packets are defined by the protocol: one of them requests routing information, and the other supplies it. The information supplied is a set of network numbers and an indication of how far away those networks are. This information is either sent on specific request or periodically distributed by all internetwork routers, which use the data to maintain routing tables that describe all or part of the internetwork topology.

An error protocol is intended to standardize the manner in which low-level communication or transport errors are reported. Moreover, it can be used as a debugging tool. If, for example, a machine receives a packet that it detects as invalid, it may return a portion of that packet by means of the error protocol, along with an indication of what is wrong. If, say, the packet is too large to be forwarded through some intermediate network, the error protocol can be used to report that fact and to indicate the length of the longest packet that can be accommodated. If too many of these return, the system designer may conclude that something is wrong with his implementation.

Another useful diagnostic and debugging tool is a protocol called the echo protocol, which is used to verify the existence and correct operation of a host and the path to it. It specifies that all echo-protocol packets received shall be returned to the source. The echo protocol also can be used to verify the correct operation of


Packets received and acknowledged
Packets allowed but unreceived or unacknowledged
Packets not sent and disallowed
i−4
i−3
i−2
i−1
i
i+1
i+2
Window three packets wide

5. A flow-control window is set up by the sequenced packet protocol, using its sequence, acknowledgment, and allocation numbers. The wider the window, the fewer the number of interactions between source and destination during message transmission.



6. A packet exchange protocol packet simply transmits a request and receives a response.

an implementation of the internet datagram protocol.

Protocols above the Internet Transport Protocols are required when, for example, a work station requests a particular file from a remotely located file server. Agreements are needed on how a work station will ask for the service and indicate the file name and how the file server will indicate that it can or cannot find the file (among other things).

Courier is a level 3 protocol that facilitates the construction of distributed systems by defining a single request-reply discipline for an open-ended set of higher-level application protocols such as filing. Courier specifies the manner in which a work station or other active system element invokes operations provided by a server or other passive system element (Fig. 7).

Courier uses the subroutine or procedure call as a metaphor for the exchange of a request and its positive reply. An operation code is modeled as the name of a remote procedure, the parameters of the request as the arguments of that procedure, and the parameters of the positive reply as the procedure's results. Courier uses the raising of an exception condition or error as a metaphor for the return of a negative reply. An error code is modeled as the name of a remote error and the parameters of the negative reply as the arguments of that error. Courier uses the module or program as a collection of related operations and their associated exception conditions. A family of remote procedures and the remote errors those procedures can raise are said to constitute a remote program.

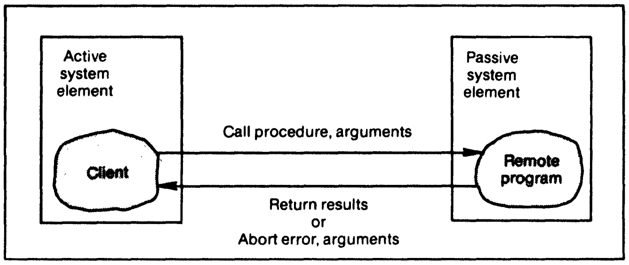
Courier does for distributed-system builders some of what a high-level programming language does for implementers of more conventional systems. Pascal, for example, allows the system builder to think in terms of procedure calls, not in terms of base registers, save areas, and branch-and-link instructions. Courier allows the distributed-system builder to think in terms of remote procedure calls, not in terms of socket numbers, network connections, and message transmission. Pascal allows the system builder to think in terms of integers and strings, rather than in terms of sign bits, length fields, and character codes. Courier allows the distributed-system builder to do the same.

Request, reply parameter types

Courier defines a family of data types from which request and reply parameters can be constructed (see "Courier data types"). Many high-level languages define data types that are semantically equivalent (or similar) to those defined by Courier. In such environments, it is often useful to define mappings between Courier data types and those of the host language. A Courier implementation can then provide software that converts a Courier data object (in its standard representation) to or from a form in which it can be manipulated using normal language or run-time facilities.

Courier also defines four standard message formats for requests and replies: a call message calls a remote procedure, i.e., invokes a remote operation; a reject message rejects such a call, i.e., reports an inability to even attempt a remote operation; a return message reports a procedure's return, i.e., acknowledges the operation's successful completion; and an abort message raises a remote error, i.e., reports the operation's failure. The message formats are defined using the same standard notation described for request and reply parameters.

Every remote program is assigned a program number, which identifies it at run time. Every remote program is further characterized by a version number, which distinguishes successive versions of the program and helps to ensure at run time that caller



7. The Courier remote procedure call protocol covers the manner in which a client invokes operations from a remote program. It simply calls for a procedure and expects the results to be returned or the operation to be aborted.

```
SimpleFile Transfer: PROGRAM 13 VERSION 1 =
BEGIN
-- types
Credentials: TYPE = RECORD [user, password: STRING];
Handle: TYPE = UNSPECIFIED;
-- procedures
OpenDirectory: PROCEDURE [name: STRING, credentials:
  Credentials]
  RETURNS [directory: Handle] REPORTS [NoSuchUser,
  IncorrectPassword, NoSuchDirectory, AccessDenied] = 1;
  Store File: PROCEDURE [name: STRING, directory: Handle]
  REPORTS [NoSuchFile, InvalidHandle] = 2;
  RetrieveFile: PROCEDURE [name: STRING, directory: Handle]
  REPORTS [NoSuchFile, InvalidHandle] = 3;
  CloseDirectory: PROCEDURE [directory: Handle] REPORTS
  [InvalidHandle] = 4;
-- errors
NoSuchUser:          ERROR = 1;
NoSuchDirectory:     ERROR = 2;
NoSuchFile:          ERROR = 3;
IncorrectPassword:   ERROR = 4;
AccessDenied:        ERROR = 5;
InvalidHandle:       ERROR = 6;
END.
```

8. As part of Courier's operation, a simple file-transfer protocol requests access to a directory to store or retrieve a file, gains the access and then closes the directory. Note the use of the high-level-like programming language in Courier's standard notation.

and callee have agreed upon the calling sequences of the program's remote procedures.

Each remote program has its own version-number space. Whenever a program's declaration is changed in any way, its version number is incremented by 1. A remote program consists of zero or more remote procedures and the errors they can raise. The specification of a remote program defines a numeric value of each procedure and error.

A call message invokes the remote procedure whose program number, program version number, and procedure value are specified.

A reject message rejects a call to a remote procedure, specifying the nature of the problem encountered. A return message reports a procedure's return and supplies its results. An abort message raises, with the arguments supplied, the remote error whose error value is specified.

In addition, a standard notation is defined for formally specifying the remote procedures and errors of a remote program, which means higher-level protocol specifications are written in what resembles a high-level programming language.

To see how Courier is used, consider a user named Stevens (password etyyq), who wishes to retrieve a file named Drawings from a directory named Projects on a file server named Development. The work station in Stevens' office and the file server at a branch office in another part of the state are attached to different Ethernet local networks, which are interconnected by means of a leased phone line. The file server is supplied by Xerox; the work station is not.

A simple file-transfer protocol is assumed to provide access to a two-level hierarchical file system maintained by the file server. The file system contains one or more named directories, each of which comprises one or more named files. The hypothetical file-transfer protocol is formally specified using Courier's standard notation (Fig. 8). Remote procedures are provided for gaining and relinquishing access to directories and for storing and retrieving files.

To retrieve the file, Stevens' work station locates and then establishes a connection to the file server. The work station opens the directory, retrieves the file, and closes the directory. The work station then terminates the connection. The work station opens and closes the directory by calling the remote procedures named OpenDirectory and CloseDirectory, respectively, in the file server. It requests retrieval of the file by calling the remote procedure named RetrieveFile, which tells the file server of the intention to retrieve. As soon as that procedure returns, the file server transmits the contents of the file on the connection using a protocol not described here.

Before anything can happen, however, the work station must discover the network address of the file

Courier data types

The data types defined by Courier fall into two classes: predefined and constructed. Predefined data types are fully specified by Courier, whereas constructed data types are defined by an application-protocol designer, in most cases using predefined or other constructed data types. Courier covers seven predefined data types:

- **Boolean:** a logical quantity that can assume either of two values, true and false.
- **Cardinal:** an integer in the interval 0 to 65535 (that is, an unsigned integer representable in 16 bits).
- **Long-cardinal:** an integer in the interval 0 to 4,294,967,295 (32 bits).
- **Integer:** a signed integer in the interval −32768 to 32767 (that is, a signed integer representable in 16 bits).
- **Long-integer:** a signed integer in the interval −2,147,483,648 to 2,147,483,647 (32 bits).
- **String:** an ordered collection of text characters whose number need not be specified until run time.
- **Unspecified:** a 16-bit quantity whose interpretation is unspecified.

Courier also defines seven constructed data types:

- **Enumeration:** a quantity that can assume any of a relatively few named integer values in the interval 0 to 65535.
- **Array:** an ordered, one-dimensional, homogeneous collection of data objects whose type and number are specified at documentation time.
- **Sequence:** an ordered, one-dimensional homogeneous collection of data objects whose type and maximum number are specified at documentation time but whose actual number can be specified at run time.
- **Record:** an ordered, possibly heterogeneous collection of data objects whose types and number are specified at documentation time.
- **Choice:** a data object whose type is chosen at run time from a set of candidates specified at documentation time.
- **Procedure:** the identifier or code for an operation that one system element will perform at the request of another. The operation may require parameters when it is invoked, return parameters if it succeeds, and report exception conditions if it fails. The arguments and results of a procedure are data objects whose types and number are specified at documentation time.
- **Error:** the identifier or code for an exception condition that one system element may report to another in response to a request to perform an operation. Parameters may accompany the report. The arguments of an error are data objects whose types and number are specified at documentation time.

server named Development by contacting a resource location service (the Clearinghouse). It does this by broadcasting an internet packet with a specially structured network address. The network number field contains a code that means "the local network"; the processor field contains a code that means "broadcast"; the socket number field is the Clearinghouse's well-known socket number.

Clearinghouse operations

The Clearinghouse consults its (distributed) data base and returns the file server's network address. The work station then initiates a connection by sending the first packet to the file server.

Once a connection has been established, the work station makes three remote procedure calls on the file server and then terminates the connection. The steps carried out to make these calls are shown in Fig. 9. Each step is hierarchically divided into substeps, which show the Courier messages exchanged by the work station and server (taking the work station's point of view), as well as how those messages appear on the connection as a sequence of 16-bit words (shown in hexadecimal).

But the document transfer may not work out as described; various problems may crop up. The most common mistakes are made by the human user, such as specifying a nonexistent file server, directory, or

1. Open the directory named *Projects*, on behalf of the user named *Stevens* (password *etyyq*):

1a. Call the remote procedure named OpenDirectory, with:
Arguments: name: "Projects," credentials: [user: "Stevens," password: "etyyq"].

Results: directory: 10A4H ("H" signifies hexadecimal).

1a1. Send a call message, with parameters:
transactionID:0, programNumber: 13, versionNumber: 1, procedureValue: 1, procedureArguments: [name: "Projects", credentials: [user: "Stevens", password: "etyyq"]]

1a1a. Send the following 16-bit words (shown in hexadecimal) on the connection:

message type (call):	0000
transactionID:	0000
programNumber:	000D
versionNumber;	0001
procedureValue:	0001
name:	0008 5072 6F6A 6543 7473
user:	0007 5374 6576 656E 7300
password:	0005 6574 7979 7100

1a2. Receive a return message, with parameters:
transactionID: 0, procedureResults: [directory: 10A4H]

1a2a. Receive the following 16-bit words (shown in hexadecimal) on the connection:

message type (return):	0002
transactionID:	0000
directory:	10A4

2. Retrieve the file named *Drawings*:

2a. Call the remote procedure named RetrieveFile, with:
Arguments: name: "Drawings", directory: 10A4H.
Results: *none.*

2a1. Send a call message, with parameters:
transactionID: 0, programNumber: 13, versionNumber: 1, procedureValue: 3, procedureArguments: [name: "Drawings", directory: 10A4H]

2a1a. Send the following 16-bit words (shown in hexadecimal) on the connection:

message type (call):	0000
transactionID:	0000
programNumber:	000D
versionNumber:	0001
procedureValue:	0003
name:	0008 4472 6177 696E 6773
directory:	10A4

2a2. Receive a return message, with parameters:
transactionID: 0, procedureResults: □

2a2a. Receive the following 16-bit words (shown in hexadecimal) on the connection:

message type (return):	0002
transactionID:	0000

2b. Receive the contents of the file transmitted via the connection (details unspecified here).

3. Close the directory:

3a. Call the remote procedure named CloseDirectory, with:
Arguments: directory: 10A4H.
Results: *none.*

3a1. Send a call message, with parameters:
transactionID: 0, programNumber: 13, versionNumber: 1, procedureValue: 4, procedureArguments: [directory: 10A4H]

3a1a. Send the following 16-bit words (shown in hexadecimal) on the connection:

message type (call):	0000
transactionID:	0000
programNumber:	000D
versionNumber:	0001
procedureValue:	0004
directory:	10A4

3a2. Receive a return message, with parameters:
transactionID: 0, procedureResults: □

3a2a. Receive the following 16-bit words (shown in hexadecimal) on the connection:

message type (return):	0002
transactionID:	0000

9. With Courier, a user named Stevens (password etyyq) retrieves a file from a directory. Each step is hierarchically divided into substeps. The messages appear as a sequence of 16-bit words, shown in hexadecimal.

file. Such mistakes are reported to the work station by the file server or the Clearinghouse, using the Courier remote error reporting mechanism.

In addition, a connection may not go through for a number of reasons—the file server has crashed, an internetwork router has crashed, there is an undetected break in the network, or the telephone line has failed in some way not directly detectable by the software.

Testing and debugging may be needed

When no response is returned, the first task is to isolate the failure. A call to the system administrator may help ascertain which part of the communication path is at fault. If there is a print server on the same Ethernet as the file server, and something can be sent to the print server, the file server is probably at fault. The internetwork router can be checked in the same way. If none of these attempts isolates the problem, the system implementer can turn to one of several software tools.

Many of these tools depend on the broadcasting nature of the Ethernet medium, and the resulting ability of one machine to observe packets sent by another. For example, a peek-type tool makes visible on the screen (in a convenient format) the contents of packets. An internet peek-type tool can also do selective filtering of packets based on sequenced-packet protocol connections or Courier calls, and display them symbolically, which proves useful in debugging. Another useful tool tests the network hardware, microcode, and software within a single machine. Yet another program permits the user to examine routing tables and network device driver statistics in any internetwork router and to echo packets from any machine. □

Bibliography

Digital Equipment Corp., Intel Corp., and Xerox Corp., Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications, Sept. 30, 1980.

Internet Transport Protocols: Xerox System Integration Standard. Xerox Corp., Stamford, Conn., December, 1981.

Xerox Corp., Courier: The Remote Procedure Call Protocol. Xerox System Integration Standard. Stamford, Conn.; Dec., 1981; XSIS-038112.

Oppen, D.C., and Dalal, Y.K., The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment, Xerox Office Products Div., Palo Alto, Calif., October, 1981.

Early Experience with Mesa

Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite
Xerox Palo Alto Research Center

The experiences of Mesa's first users—primarily its implementers—are discussed, and some implications for Mesa and similar programming languages are suggested. The specific topics addressed are: module structure and its use in defining abstractions, data-structuring facilities in Mesa, an equivalence algorithm for types and type coercions, the benefits of the type system and why it is breached occasionally, and the difficulty of making the treatment of variant records safe.

Key Words and Phrases: programming languages, types, modules, data structures, systems programming
CR Categories: 4.22

1. Introduction

What happens when professional programmers change over from an old-fashioned systems programming language to a new, modular, type-checked one like Mesa? Considering the large number of groups developing such languages, this is certainly a question of great interest.

This paper focuses on our experiences with strict type checking and modularization within the Mesa programming system. Most of the local structure of Mesa was inspired by, and is similar to, that of Pascal [14] or Algol 68 [12], while the global structure is more like that of Simula 67 [1]. We have chosen features from these and related languages selectively, cast them in a different syntax, and added a few new ideas of our own. All this has been constrained by our need for a language to be used for the production of real system software right now. We believe that most of our observations are relevant to the languages mentioned above, and others like them, when used in a similar environment. We have therefore omitted a comprehensive description of Mesa and concentrated on annotated examples that should be intelligible to anyone familiar with a similar language. We hope that our experiences will help others who are creating or studying such languages.

An interested reader can find more information about the details of Mesa elsewhere. A previous paper [7] addresses issues concerning transfer of control. Another paper [3] discusses some more advanced data-structuring ideas. A paper on *schemes* [8] suggests another possible direction of advance. In this paper we restrain our desires to redesign or extend Mesa and simply describe how we are using the language as currently implemented.

The version of Mesa presented in this paper is one component of a continuing investigation into programming methodology and language design. Most major aspects of the language were frozen when implementation was begun in the autumn of 1974. Although we were dissatisfied with our understanding of certain design issues even then, we proceeded with implementation for the following reasons.

- We perceived a need for a "state of the art" implementation langauge within our laboratory. It seemed possible to combine some of our ideas into a design that was fairly conservative, but that would still dominate the existing and proposed alternatives.
- We wanted feedback from a community of users, both to evaluate those ideas that were ready for implementation and to focus subsequent research on problems actually encountered in building real systems.
- We had accumulated a backlog of ideas about implementation techniques that we were anxious to try.

It is important to understand that we have consciously decided to attempt a complete programming system for demanding and sophisticated users. Their own research projects were known to involve the construction of "state of the art" programs, many of which tax the limits of available computing resources. These users are well aware of the capabilities of the underlying hardware, and they have developed a wide range of programming styles that they have been loath to abandon. Working in this environment has had the following consequences.

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

A version of this paper was presented at the SIGPLAN/SIGOPS/SICSOFT Conference on Language Design for Reliable Software, Raleigh, N.C., March 28-30, 1977.

Authors' address: Computer Science Laboratory, Palo Alto Research Center, Xerox Corporation, 3333 Coyote Hill Road, Palo Alto CA 94304

- We could not afford to be too dogmatic. The language design is conservative and permissive; we have attempted to accommodate old methods of programming as well as new, even at some cost in elegance.
- Efficiency is important. Mesa reflects the general properties of existing machines and contains no features that cannot be implemented efficiently (perhaps with some microcode assistance); for example, there is no automatic garbage collection.

A cross-compiler for Mesa became operational in the spring of 1975. We used it to build a small operating system and a display-oriented symbolic debugger. By early 1976, it was possible to run a system built entirely in Mesa on our target machine, and rewriting the compiler in its own language was completed in the summer of 1976. The basic system, debugger, and compiler consist of approximately 50,000 lines of Mesa code, the bulk of which was written by four people. Since mid-1976, the community of users and scope of application of Mesa have been expanding rapidly, but its most experienced and demanding users are still its implementers. It is in this context that we shall try to describe our experiences and to suggest some tentative conclusions. Naturally, we have discovered some bugs and omissions in the design, and the implemented version of the language is already several years from the frontiers of research. We have tried to restrain our desire to redesign, however, and we report on Mesa as it is, not as we now wish it were.

The paper begins with a brief overview of Mesa's module structure. The uses of types and strict type checking in Mesa are then examined in some detail. The facilities for defining data structures are summarized, and an abstract description of the Mesa type calculus is presented. We discuss the rationale and methods for breaching the type system and illustrate them with a "type-strenuous" example that exploits several of the type system's interesting properties. A final section discusses the difficulties of handling variant records in a type-safe way.

2. Modules

Modules provide a capability for partitioning a large system into manageable units. They can be used to encapsulate *abstractions* and to provide a degree of *protection*. In the design of Mesa, we were particularly influenced by the work of Parnas [10], who proposes *information hiding* as the appropriate criterion for modular decomposition, and by the concerns of Morris [9] regarding protection in programming languages.

Module Structure

Viewed as a piece of source text, a *module* is similar to an Algol procedure declaration or a Simula class definition. It typically declares a collection of variables that provide a localized database and a set of procedures performing operations upon that database. Modules are designed to be compiled independently, but the declarations in one module can be made visible during the compilation of another by arranging to reference the first within the second by a mechanism called *inclusion*. To decouple the internal details of an implementation from its abstract behavior, Mesa provides two kinds of modules: *definitions* and *programs*.

A definitions module defines the interface to an abstraction. It typically declares some shared types and useful constants, and it defines the interface by naming a set of procedures and specifying their input/output types. Definitions modules claim no storage and have no existence at run time. Included modules are usually definitions modules, but they need not be.

Certain program modules, called *implementers*, provide the concrete implementation of an abstraction; they declare variables and specify bodies of procedures. There can be a one-to-many relation between definitions modules and concrete implementations. At run time, one or more instances of a module can be created, and a separate *frame* (activation record) is allocated for each. In this respect, module instances resemble Simula class objects. Unlike procedure instances, the lifetimes of module instances are not constrained to follow any particular discipline. Communication paths among modules are established dynamically as described below and are not constrained by, e.g., compile-time or run-time nesting relationships. Thus lifetimes and access paths are completely decoupled.

The following skeletal Mesa modules suggest the general form of a definitions module and one of its implementers:

```
Abstraction: DEFINITIONS =
  BEGIN
  . . .
  it: TYPE = . . .; rt: TYPE = . . .;
  . . .
  p: PROCEDURE;
  p1: PROCEDURE [INTEGER];
  . . .
  pi: PROCEDURE [it] RETURNS [rt];
  . . .
  END
```

```
Implementer: PROGRAM IMPLEMENTING Abstraction =
BEGIN
OPEN Abstraction;
x: INTEGER;
. . .
p: PUBLIC PROCEDURE = ⟨code for p⟩;
p1: PUBLIC PROCEDURE [i: INTEGER] = ⟨code for p1⟩;
. . .
pi: PUBLIC PROCEDURE [x: it] RETURNS [y: rt] =
      ⟨code for pi⟩;
 . . .
 END
```

Longer but more complete and realistic examples can be found in the discussion of *ArrayStore* below; *ArrayStoreDefs* and *ArrayStore* correspond to *Abstraction* and *Implementer*, respectively.

Mesa allows specification of attributes that can be used to control intermodular access to identifiers. In the definition of an abstraction, some types or record fields are of legitimate concern only to an implementer, but they involve or are components of other types that are parts of the advertised interface to the abstraction. Any identifier with the attribute PRIVATE is visible only in the module in which it is declared and in any module claiming to implement that module. Subject to the ordinary rules of scope, an identifier with the attribute PUBLIC is visible in any module that includes and *opens* the module in which it is declared. The PUBLIC attribute can be restricted by specifying the additional attribute READ-ONLY. By default, identifiers are PUBLIC in definitions modules and PRIVATE otherwise.

In the example above, *Abstraction* contains definitions of shared types and enumerates the elements of a procedural interface. *Implementer* uses those type definitions and provides the bodies of the procedures; the compiler will check that an actual procedure with the same name and type is supplied for each public procedure declared in *Abstraction*.

A module that uses an abstraction is called a *client* of that abstraction. Interface definitions are obtained by including the *Abstraction* module. Any instance of a client must be connected to an instance of an appropriate implementer before the actual operations of the abstraction become available. This connection is called *binding*, and there are several ways to do it.

Binding Mechanisms

When a relatively static and purely procedural interface between modules is acceptable, the connection can be made in a conventional way. Consider the following skeleton:

```
Client1: PROGRAM =
  BEGIN
  OPEN Abstraction;
  . . .
  px: EXTERNAL PROCEDURE;
  . . .
  p[ ]; px[ ];
  . . .
  END.
```

A client module can request a system facility called the *binder* to locate and assign appropriate values to all external procedure names, such as *px*. The binder follows a well-defined *binding path* from module instance to module instance. When the binder encounters an actual procedure with the same name as, and a type compatible with, an external procedure, it makes the linkage. The compiler automatically inserts an EXTERNAL procedure declaration for any procedure identifier, such as *p*, that is mentioned by a client but defined only in an included definitions module. The binder also checks that all identifiers from a single definitions module are bound consistently (that is, to a single implementer).

The observant reader will have noticed that this binding mechanism and the undisciplined lifetimes of module instances leave Mesa programs vulnerable to dangling reference problems. We are not happy about this, but so far we have not observed any serious bugs attributable to such references.

As an alternate binding mechanism, Mesa supports the Simula paradigm as suggested by the following skeleton (which assumes that *x* is a public variable):

```
Client2: PROGRAM =
  BEGIN
  OPEN Abstraction;
  frame: POINTER TO FRAME[Implementer] ←
          NEW Implementer;
  . . .
  frame ↑ .x ← 0;
  frame ↑ .p[ ];
  . . .
  END.
```

Here, the client creates an instance of *Implementer* directly. Through a pointer to the frame of that instance, the client can access any public variable or invoke any public procedure. Note that the relevant declarations are in *Implementer*; the *Abstraction* module is included only for type definitions. Some of the binding has been moved to compile time. In return for a wider, not necessarily procedural interface (and potentially more efficient code), the client has committed itself to using a particular implementation of the abstraction.

Because Mesa has procedure variables, it is possible for a user to create any binding regime he wishes simply by writing a program that distributes procedures. Some users have created their own versions of Simula classes. They have not used the binding mechanism described above for a number of reasons. First, the actual implementation of an abstract object is sometimes unknown when a program is compiled or instantiated; there might be several coexisting implementations, or the actual implementation of a particular object might change dynamically. Their binding scheme deals with such situations by representing objects as record structures with procedure-valued fields. The basic idea was described in connection with the implementation of streams in OS6 [11]: some fields of each record contain the state information necessary to characterize the object, while others contain procedure values that implement the set of operations. If the number of objects is much larger than the number of implementations, it is space-efficient to replace the procedure fields in each object with a link to a separate record containing the set of values appropriate to a particular implementation. When this binding mechanism is used, interface specifications consist primarily of type definitions, as suggested by the following skeleton:

```
ObjectAbstraction: DEFINITIONS =
  BEGIN
  Handle: TYPE = POINTER TO Object;
  Object: TYPE = RECORD [
    ops: POINTER TO Operations,
    state: POINTER TO ObjectRecord,
    ...];
  Operations: TYPE = RECORD [
    p1: PROCEDURE [Handle, INTEGER],
    ...];
  END.
```

A client invokes a typical operation by writing *handle* ↑ *.ops* ↑ *.p1* [*handle*,*x*], where *handle* is an object of type *Handle*.

Observations

We believe that we could not have built the current Mesa system if we had been forced to work with large logically monolithic programs. Assembly language programmers are well aware of the benefits of modularity, but many designers of high-level programming languages pay little attention to the problems of independent compilation and instantiation. Since these capabilities will be grafted on anyway, they should be anticipated in the original design. We have more to say about interface control in our discussion of types, but it is hard to overestimate the value of articulating abstractions, centralizing their definitions, and propagating them through the inclusion mechanism.

3. The Mesa Type System

Strict vs. Nonstrict Type Checking

A widely held view is that the purpose of type declarations is to allow one to write more succinct programs. For example, the Algol 60 declarations

```
real x,y; integer i,j;
```

allow one to attach two different interpretations to the symbol "+" in the expressions $x + y$ and $i + j$. Similarly, the declaration

```
x: RECORD[a: [0..7], b: [0..255]]
```

permits one to write *x.a* and *x.b* in place of descriptions of the shifting and masking that might occur. Descriptive declarations also allow utility programs such as debuggers to display values of variables in a helpful way when the type is not encoded as part of the value.

This view predominated in an earlier version of Mesa. Type declarations were used primarily as devices to improve the expressive power and readability of the language. Types were ignored by the compiler except to discover the number of bits involved in an operation. In contrast, the current version of Mesa checks type agreement as rigorously as languages such as Pascal or Algol 68, potentially rendering compile-time complaints in great volume. This means in effect that the language is more redundant since there are fewer programs acceptable to the compiler.

What benefit do we hope to gain by stricter checking and the attendant obligations on the programmer? We expect that imposing additional structure on the data space of the program and checking it mechanically will make the modification and maintenance of programs easier. The type system allows us to write down certain design decisions. The type checker is a tool that is used to discover violations of the conventions implied by those decisions without a great expenditure of thought.

Type Expressions

Mesa provides a fairly conventional set of expressions for describing types; detailed discussions of the more important constructors are available elsewhere [3]. We shall attempt just enough of an introduction to help in reading the subsequent examples and concentrate upon the relations among types.

There is a set of predefined basic types and a set of *type operators* which construct new types. The arguments of these operators may be other types, integer constants, or identifiers with no *a priori* meanings. Most of the operators are familiar from languages such as Pascal or Algol 68, and the following summary emphasizes only the differences.

Basic Types. The basic types are INTEGER, BOOLEAN, CHARACTER, and UNSPECIFIED, the last of which is a one-word, wild-card type.

Enumerated Types. If $a_1, a_2, \ldots, a_n$ are distinct identifiers, the form $\{a_1, a_2, \ldots, a_n\}$ denotes an ordered type of which the identifiers denote the allowed constant values.

Unique Types. If n is a manifest (compile-time) constant of type INTEGER, the form UNIQUE[n] denotes a type distinct from any other type. The value of n determines the amount of storage allocated for values of that type, which are otherwise uninterpreted. Its use is illustrated by the *ArrayStore* example in Section 4.

Record Types. If $T_1, T_2, \ldots, T_n$ are types and $f_2, \ldots, f_n$ are distinct identifiers, the the form RECORD[f_1: T_1, f_2: T_2, $\ldots$, f_n: T_n] denotes a record type. The f_i are called *field selectors*. As usual, the field selectors are used to access individual components; in addition, linguistic forms called *constructors* and *extractors* are available for synthesizing and decomposing entire records. The latter forms allow either keyword notation, using the field names, or positional notation. Intermodule access to individual fields can be controlled by specifying the attributes PUBLIC, PRIVATE, or READONLY; if no such attributes appear, they are inherited from the enclosing declaration. Some examples:

```
Thing: TYPE = RECORD [n: INTEGER, p: BOOLEAN];
v: Thing; i: INTEGER; b: BOOLEAN;
...
IF v.p THEN v.n ← v.n + 1; --field selection
v ← [100, TRUE];           --a positional constructor
v ← [p:b, n:i];            --a keyword constructor
[n:i, p:b] ← v;            --the inverse extractor.
```

Pointer Types. If T is a type, the form POINTER TO T denotes a pointer type. If x is a variable of that type, then $x\uparrow$ *dereferences* the pointer and designates the object pointed to, as in Pascal. If v is of type T, then $@v$ is its address with type POINTER TO T. The form POINTER TO READ-ONLY T denotes a similar type; however, values of this type cannot be used to change the indirectly referenced object. Such pointer types were introduced so that objects could be passed by reference across module interfaces with assurance that their values would not be modified.

Array Types. If T_i and T_c are types, the form ARRAY T_i OF T_c denotes an array type. T_i must be a finite ordered type. An array a maps an index i from the index type T_i into a value $a[i]$ of the component type T_c. If a is a variable, the mapping can be changed by assignment to $a[i]$.

Array Descriptor Types. If T_i and T_c are types, the form DESCRIPTOR FOR ARRAY T_i OF T_c denotes an array descriptor type. T_i must be an ordered type. An array descriptor value provides indirect access to an array and contains enough auxiliary information to determine the allowable indices as a subrange of T_i.

Set Types. If T is a type, the form SET OF T denotes a type, values of which are the subsets of the set of values of T. T must evaluate to an enumerated type.

Transfer Types. If $T_1, \ldots, T_i, T_j, \ldots, T_n$ are types and $f_1, \ldots, f_i, f_j, \ldots, f_n$ are distinct identifiers, then the form PROCEDURE $[f_1: T_1, \ldots, f_i: T_i]$ RETURNS $[f_j: T_j, \ldots, f_n: T_n]$ denotes a procedure type. Each nonlocal control transfer passes an argument record; the field lists enclosed by the paired brackets, if not empty, implicitly declare the types of the records accepted and returned by the procedure [7]. If x has some transfer type, a control transfer is invoked by the evaluation of $x[e_1, \ldots, e_i]$, where the bracketed expressions are used to construct the input record, and the value is the record constructed in preparation for the transfer that returns control.

The symbol PROCEDURE can be replaced by several alternatives that specify different transfer disciplines with respect to name binding, storage allocation, etc., but the argument transmission mechanism is uniform. Transfer types are full-fledged types; it is possible to declare procedure variables and otherwise to manipulate procedure values, which are represented by procedure descriptors. Indeed, some of the intermodule binding mechanisms described previously depend crucially upon the assignment of values to procedure variables.

Subrange Types. If T is INTEGER or an enumerated type, and m and n are manifest constants of that type, the form $T[m..n]$ denotes a finite, ordered subrange type for which any legal value x satisfies $m \leq x \leq n$. If T is INTEGER, the abbreviated form $[m..n]$ is accepted. These types are especially useful as the index types of arrays. Other notational forms, e.g. $[m..n)$, allow intervals to be open or closed at either endpoint.

Finally, Mesa has adapted Pascal's variant record concept to provide values whose complete type can only be known after a run-time discrimination. Because they are of more than passing interest, variant records are discussed separately in Section 5.

Declarations and Definitions

The form

```
v: Thing ← e
```

declares a variable v of type *Thing* and initializes it to the value of e; the form

```
v: Thing = e
```

is similar except that assignments cannot be made to v subsequently. When e itself is a manifest constant, this form makes v such a constant also.

This syntax is used for the introduction of new type names, using the special type TYPE. Thus

```
Thing: TYPE = TypeExpression
```

defines the type *Thing*. This approach came from ECL [13], in which a type is a value that can be computed by a running program and then used to declare variables. In Mesa, however, *TypeExpression* must be constant.

Recursive type declarations are essential for describing most list structures and are allowed more generally whenever they make sense. To accommodate a mutually recursive list structure, forward references to type identifiers are allowed and do not yield "uninitialized" values. (This is to be contrasted with forward references to ordinary variables.) In effect, all type expressions within a scope are evaluated simultaneously. Meaningful recursion in a type declaration usually involves the type constructor POINTER; in corresponding values, the recursion involves a level of indirection and can be terminated by the empty pointer value NIL. Recursion that is patently meaningless is rejected by the compiler; for example,

```
r: TYPE = RECORD [left, right: r] --not permitted
a: TYPE = ARRAY [0..10) OF s;
s: TYPE = RECORD [i: INTEGER, m: a] --not permitted.
```

Similar pathological types have been noted and prohibited in Algol 68 [6].

Equivalence of Type Expressions

One might expect that two identical type expressions appearing in different places in the program text would always stand for the same type. In Algol 68 they do. In Mesa (and certain implementations of Pascal) they do not. Specifically, the type operators RECORD, UNIQUE, and {...} generate new types whenever they appear in the text.

The original reasons for this choice are not very important, but we have not regretted the following consequences for records:

(a) All modules wishing to communicate using a shared record type must obtain the definition of that type from the same source. In practice, this means that all definitions of an abstraction tend to come from a single module; there is less temptation to declare scattered, partial interface definitions.

(b) Tests for record type equivalence are cheap. In our experience, most record types contain references to other record types, and this linking continues to a considerable depth. A recursive definition of equivalence would, in the worst case, require examining many modules unknown and perhaps unavailable to the casual user of a record type or, alternatively, copying all type definitions supporting a particular type into the symbol table of any module mentioning that type.

(c) The rule for record equivalence provides a mechanism for *sealing* values that are distributed to clients as passkeys for later transactions with an implementer. Suppose that the following declaration occurs in a definitions module:

Handle: PUBLIC TYPE = RECORD [*value*: PRIVATE *Thing*].

The PRIVATE attribute of *value* is overridden in any implementer of *Handle*. A client of that implementer can declare variables of type *Handle* and can store or duplicate values of that type, but there is no way for the client to construct a counterfeit *Handle* without violating the type system. Such sealed types appear to provide a basis for a compile-time capability scheme [2].

(d) Finally, this choice has not caused discomfort because programmers are naturally inclined to introduce names for record types anyway.

The case for distinctness of enumerated types is much weaker; we solved the problem of the exact relationships among such types of {*a*, *b*, *c*}, {*c*, *b*, *a*}, {*a*, *c*}, {*aa*, *b*, *cc*}, etc. by specifying that all these types are distinct. In this case, we are less happy that identical sequences of symbols construct different enumerated types.

Why did we not choose a similar policy for other types? It would mean that a new type identifier would have to be introduced for virtually every type expression, and we found it to be too tedious. In the case of procedures we went even further in liberalizing the notion of equivalence. Even though the formal argument and result lists are considered to be record declarations, we not only permit recursive matching but also ignore the field selectors in doing the match. We were unwilling to abandon the idea that procedures are mappings in which the identifiers of bound variables are irrelevant. We also had a pragmatic motivation. In contrast to records, where the type definitions cross interface boundaries, procedural communication among modules is based upon procedure values, not procedure types. Declaring named types for all interface procedures seemed tiresome. Fortunately all argument records are constructed in a standard way, so this view causes no implementation problems.

To summarize, we state an informal algorithm for testing for type equivalence. Given one or more program texts and two particular type expressions in them:

1. Tag each occurrence of RECORD, UNIQUE, and {. . .} with a distinct number.
2. Erase all the variable names in the formal parameter and the result lists of procedures.
3. Compare the two expressions, replacing type identifiers with their defining expressions whenever they are encountered. If a difference (possibly in a tag attached in step 1) is ever encountered, the two type expressions are not equivalent. Otherwise they are equivalent.

The final step appears to be a semidecision procedure since the existence of recursive types makes it impossible to eliminate all the identifiers. In fact, it is always possible to tell when one has explored enough (cf. [5], Section 2.3.5, Exercise 11).

Coercions

To increase the flexibility of the type system Mesa permits a variety of implicit type conversions beyond those implied by type equivalence. They fall into two categories: *free coercions* and *computed coercions*.

Free Coercions. Free coercions involve no computation whatsoever. For two types T and S, we write $T \subseteq S$ if any value of type T can be stored into a variable of type S without checking, change of representation, or other computation. (By "store" we mean to encompass assignment, parameter passing, result passing, and all other value transmission.) The following recursive rules show how to compute the relation $\subseteq$, assuming that equivalence has already been accounted for:

1. $T \subseteq T$.

In the following assume that $T \subseteq S$.

2. $T[i..j] \subseteq S$ if i is the minimum value of type S.

The restriction is necessary because we chose to represent values of a subrange type relative to its minimum value. Coercions in other cases require computation. Similarly,

3. $T[i..j] \subseteq S[i..k]$ iff $j \leq k$.
4. *var* $T \subseteq S$ if *var* is a variant of T (cf. Section 5).
5. RECORD[*f*: T] $\subseteq$ S for any field name *f* unless *f* has the PRIVATE attribute.
6. POINTER TO T $\subseteq$ POINTER TO READ-ONLY S.

In other words, one can always treat a pointer as a read-only pointer, but not vice versa.

7. POINTER TO READ-ONLY T $\subseteq$ POINTER TO READ-ONLY S.

The relation POINTER TO T $\subseteq$ POINTER TO S is *not* true because it would allow

```
ps: POINTER TO S;
pt: POINTER TO T = @t;
ps ← pt;
ps↑ ← s;
```

which is a sneaky way of accomplishing "$t \leftarrow s$," which is not allowed unless $S \subseteq T$.

8. ARRAY I OF T $\subseteq$ ARRAY I OF S.

Note that the index sets must be the same.

9. PROCEDURE [S′] RETURNS [T] $\subseteq$ PROCEDURE [T′] RETURNS [S] if $T' \subseteq S'$ as well.

Here the relation between the input types is the reverse of what one might expect.

Subrange Coercions. Coercions between subranges require further comment. As others have noted [4], associating range restrictions with types instead of specific variables leads to certain conceptual problems; however, we wanted to be able to fold range restrictions into more complex constructed types. We were somewhat surprised by the subtlety of this problem, and our initial solutions allowed several unintended breaches of the type system.

Values of an ordered type and all its subranges are interassignable even if they do not satisfy cases (2) or (3) above. This is an example of a computed coercion. Code is generated to check that the value is in the proper subrange and to convert its representation if necessary. It is important to realize that computed coercions cannot be extended recursively as was done above. Consider the declarations

```
x: [0..100] ← 15;
y: [10..20];
px: POINTER TO READ-ONLY [0..100] ← @x;
py: POINTER TO READ-ONLY [10..20];
```

The assignment $y \leftarrow x$ is permitted because x is 15; 5 is stored in y since its value is represented relative to 10. However, the assignment $py \leftarrow px$, which rule 7 might suggest, is not permitted because the value of x can change and there is no reasonable way to generate checking code. Even if the value of x cannot change, we could not perform any change in representation because the value 15 is shared. Similar problems arise when one considers rules 6, 8, and 9.

Other Computed Coercions. Research in programming language design has continued in parallel with our implementation work, and some proposals for dealing with uniform references [3] and generalizations of classes [8] suggested adding the following computed coercions to the language:

Dereferencing: POINTER TO $T \rightarrow T$
Deproceduring: PROCEDURE RETURNS $T \rightarrow T$
Referencing: $T \rightarrow$ POINTER TO T.

Initially we had intended to support contextually implied application of these coercions much as does Algol 68. Reactions of Mesa's early users to this proposal ranged from lukewarm to strongly negative. In addition, the data structures and accounting algorithms necessary to deduce the required coercions and detect pathological types substantially complicated the compiler. We therefore decided to reconsider our decision even after the design and some of the implementation had been done. The current language allows subrange coercion as described above. There is no uniform support for other computed coercions, but automatic dereferencing is invoked by the operators for field extraction and array indexing. Thus such forms as $p\uparrow.f$ and $a\uparrow\uparrow[i]$, which are common when indirection is used extensively, may be written as $p.f$ and $a[i]$.

There are hints of a significant problem for language designers here. Competent and experienced programmers seem to believe that coercion rules make their programs less understandable and thus less reliable and efficient. On the other hand, techniques being developed with the goal of decreasing the cost of creating and changing programs seem to build heavily upon coercion. Our experience suggests that such work should proceed with caution.

Why is coercion distrusted? Our discussions with programmers suggest that the reasons include the following:

- Mesa programmers are familiar with the underlying hardware and want to be aware of the exact consequences of what they write.
- Many of them have been burned by forgotten indirect bits and the like in previous programming and are suspicious of any unexpected potential for side effects.
- To some extent, coercion negates the advantages of type checking. One view of coercion is that it corrects common type errors, and some of the detection capability is sacrificed to obtain the correction.

We conjecture that the first two objections will diminish as programmers learn to think in terms of higher-level abstractions and to use the type checking to advantage.

The third objection appears to have some merit. We know of no system of coercions in which strict type checking can be trusted to flag all coercion errors, and such errors are likely to be especially subtle and persistent. The difficulties seem to arise from the interactions of coercion with generic operators. In Algol 68, there are rules about "loosely related" types that are intended to avoid this problem, but the identity operators still suffer. With the coercion rules that had been proposed for Mesa, the following trap occurs. Given the declaration *p, q*: POINTER TO INTEGER, the Mesa expressions $p\uparrow = q\uparrow$ and $2{*}p = 2{*}q$ would compare integers and give identical results; on the other hand, the expression $p = q$ would compare pointers and could give a quite different answer. In the presence of such traps, we believe that most programmers would resolve to supply the "↑" always. If this is their philosophy, coercions can only hide errors. Even if such potentially ambiguous expressions as $p = q$ were disallowed, this example suggests that using coercion to achieve representational independence can easily destroy referential transparency instead.

4. Experiences with Strict Type Checking

It is hard to give objective evidence that increasing compile-time checking has materially helped the programming process. We believe that it will take more effort to get one's program to compile and that some of the effort eliminates errors that would have shown up during testing or later, but the magnitude of these effects is hard to measure. All we can present at the moment are testimonials and anecdotes.

A Testimonial

Programmers whose previous experience was with unchecked languages report that the usual fear and trepidation that accompanied making modifications to programs has substantially diminished. Under previous regimes they would never change the number or types of arguments that a procedure took for fear that they would forget to fix all of the calls on that procedure. Now they know that all references will be checked before they try to run the program.

An Anecdote

The following kind of record is used extensively in the compiler:

RelativePtr: TYPE = [0..37777_8];
TaggedPtr: TYPE = RECORD[*tag*: $\{t_0,t_1,t_2,t_3\}$,
                                        ptr: *RelativePtr*].

This record consists of a 2-bit tag and a 14-bit pointer. As an accident of the compiler's choice of representation, the expressions x and *TaggedPtr*[t_0,x] generated the same internal value. The nonstrict type checker considered these types equivalent, and unwittingly we used *TaggedPtrs* in many places actually requiring *RelativePtrs*. As it happened, the tag in these contexts was always t_0.

The compiler was working well, but one day we made the unfortunate decision to redefine *TaggedPtr* as

RECORD[*ptr*: *RelativePtr*, *tag*: $\{t_0,t_1,t_2,t_3\}$].

This caused a complete breakdown, and we hastily unmade that decision because we were unsure about what parts of the code were unintentionally depending upon the old representation. Later, when we submitted a transliteration of the compiler to the strict type checker, we found all the places where this error had been committed. At present, making such a change is routine. In general, we believe that the benefits of static checking are significant and cost-effective once the programmer learns how to use the type system effectively.

A Shortcoming

The type system is very good at detecting the difference in usage between T and POINTER TO T; however, programmers often use array indices as pointers, especially when they want to perform arithmetic on them. The difference between an integer used as a pointer and an integer used otherwise is invisible to the type checker. For example, the declaration

map: ARRAY [*i*..*j*] OF INTEGER[*m*..*n*];

defines a variable *map* with the property that compile-time type checking cannot distinguish between legitimate uses of k and *map*[k]. Furthermore, if $m \le i$ and $j \le n$, even a run-time bounds check could never detect a use of k when *map*[k] was intended. We have observed several troublesome bugs of this nature and would like to change the language so that indices of different arrays can be made into distinct types.

Violating the Type System

One of the questions often asked about languages with compile-time type checking is whether it is possible to write real programs without violating the type system. It goes without saying that one can bring virtually any program within the confines of a type system by methods analogous to the silly methods for eliminating *gotos*; e.g. simulate things with integers. However, our experience has been that it is not always desirable to remain within the system, given the realities of programming and the restrictiveness of the current language. There are three reasons for which we found it desirable to evade the current type system.

Sometimes the violation is logically necessary. Fairly often one chooses to implement part of a language's run-time system in the language itself. There are certain things of this nature that cannot be done in a type-safe way in Mesa, or any other strictly type-checked language we know. For example, the part of the system that takes the compiler's output and creates values of type PROCEDURE must exercise a rather profound loophole in turning data into program. Another example, discussed in detail below, is a storage allocator. Most languages with compile-time checking submerge these activities into the implementation and thereby avoid the need for type breaches.

Sometimes efficiency is more important than type safety. In many cases the way to avoid a type breach is to redesign a data structure in a way that takes more space, usually by introducing extra levels of pointers. The section on variant records gives an example.

Sometimes a breach is advisable to increase type checking elsewhere. Occasionally a breach could be avoided by declaring two distinct types to be the same, but merging them would reduce a great deal of checking elsewhere. The *ArrayStore* example below illustrates this point.

Given these considerations, we chose to allow occasional breaches of the type system, making them as explicit as possible. The advantages of doing this are twofold. First, making breaches explicit makes them less dangerous since they are clearer to the reader. Second, their occurrences provide valuable hints to a language designer about where the type system needs improvement.

One of the simplest ways to breach the Mesa type system is to declare something to be UNSPECIFIED. The type checking algorithm regards this as a one-word don't-care type that matches any other one-word type. This is similar to PL/I UNSPEC. We have come to the conclusion that using UNSPECIFIED is too drastic in most cases. One usually wants to turn off type checking in only a few places involving a particular variable, not everywhere. In practice there is a tendency to use UNSPECIFIED in the worst possible way: at the interfaces of modules. The effect is to turn off type checking in other people's modules without their knowing it!

As an alternative, Mesa provides a general type transfer function, RECAST, that (without performing any computation) converts between any two types of equal size. It can often be used instead of UNSPECIFIED. In cases where we had declared a particular variable UNSPECIFIED, we now prefer to give it some specific type and to use RECAST whenever it is being treated in a way that violates the assumptions about that type.

The existence of RECAST makes many decisions much less painful. Consider the type CHARACTER. On the one hand we would like it to be disjoint from INTEGER so that simple mistakes would be caught by the type checker. On the other hand, one occasionally needs to do arithmetic on characters. We chose to make CHARACTER a distinct type and use RECAST in those places where character arithmetic is needed. Why reduce the quality of type checking everywhere just to accommodate a rare case?

Pointer arithmetic is a popular pastime for system programmers. Rather than outlawing it, or even requiring a RECAST, Mesa permits it in a restricted form. One can add or subtract an integer from a pointer to produce a pointer of the same type. One can subtract two pointers of the same type to produce an integer. The need for more exotic arithmetic has not been observed.

Here is a typical example: It is common to use a large contiguous area of memory to hold a data structure consisting of many records, e.g. a parse tree. To conserve space one would like to make all pointers relative to the start of the area, thus reducing the size of pointers that are internal to the structure. Furthermore, one might like to move the entire area, possibly via secondary storage. These needs would be met by an unimplemented feature called the *tied pointer*. The idea is that a certain type of pointer would be made relative to a designated base value and this value would be added just before dereferencing the pointer. In other words, if *ptr* were declared to be tied to *base* then *ptr*↑ actually would mean (*base*+*ptr*)↑. Since tied pointers have not yet been implemented, this notation is in fact used extensively within the Mesa compiler. Subsequent versions of Mesa will include tied pointers, and this temporary loophole will be reconsidered.

The Skeleton Type System

Once we provided the opportunity for evading the official type system, we had to ask ourselves just why we thought certain breaches were safe while others were not. Ultimately, we came to the conclusion that the only really dangerous breaches of the type systems were those that require detailed knowledge of the run-time environment. First and foremost, fabricating a procedure value requires a detailed understanding of how various structures in memory are arranged. Second, pointer types also depend on various memory structures' being set up properly and should not be passed through loopholes without some care. In contrast, the distinction between the two types RECORD [*a*,*b*: INTEGER] and RECORD[*c*,*d*: INTEGER] is not vital to the run-time system's integrity. To be sure, the user might wish to keep them distinct, but using a loophole to store one into the other would go entirely unnoticed by the system.

The present scheme that is used to judge the appropriateness of RECAST transformations merely checks to ensure that the source and destination types occupy the same number of bits. Since most of the code invoking RECAST has been written by Mesa implementers, this simplified check has proved to be sufficient. However, as the community of users has grown, we have observed a justifiable anxiety over the use of RECAST. Users fear that unchecked use of this escape will cause a violation of some system convention unknown to them.

We are in the process of investigating a more complete and formal skeletal type system that will reduce the hazards of the present RECAST mechanism. Its aim is to ensure that although a RECAST may do great violence to user-defined type conventions, the system's type integrity will not be violated.

Example – A Compacting Storage Allocator

A module that provides many arrays of various sizes by parceling out pieces of one large array is an interesting benchmark for a systems programming language for a number of reasons:

(a) It taxes the type system severely. We must deal with an array containing variable length heterogeneous objects, something one cannot declare in Mesa.

(b) The clients of the allocator wish to use it for arrays of differing types. This is a familiar polymorphism problem.

(c) As a programming exercise, the module can involve tricky pointer manipulations. We would like help to prevent programming errors such as the ubiquitous address/contents confusion.

(d) A nasty kind of bug associated with the use of such packages is the so-called dangling reference problem: variables or data structures might be used after their space has been relinquished.

(e) Another usage bug, peculiar to compacting allocators, is that a client might retain a pointer to storage that the compacter might move.

The first two problems make it impossible to stay entirely within the type system. One's first impulse is to declare everything unspecified and proceed to program as in days of yore. The remaining problems are real ones, however, and we are reluctant to turn off the entire type system just when we need it most. The following is a compromise solution.

To deal with problem (a), we have two different ways of designating the array to be parceled out, which we call *Storage*. From a client's point of view, the storage is accessible through the definitions shown in the module *ArrayStoreDefs* (cf. Figure 1).

These definitions suggest that the client can get *ArrayPtrs* (i.e. pointers to pointers to array records) by calling *AllocArray* and can relinquish them by calling *FreeArray*. The PRIVATE attribute on *backp* means that the client cannot access that field at all. The READ-ONLY attribute on *length* means that the client cannot change it. Of course these restrictions do not apply to the implementing module. The type *Thing* occupies 16 bits of storage (one word) and matches no other type. Intuitively it is our way of simulating a type variable. The implementing module *ArrayStore* is shown in Figure 2. It declares the array *Storage* to create the raw material for allocation. We chose to declare its element type UNSPECIFIED. This means that every transaction involving *Storage* is an implicit invocation of a loophole. Specifically the initializations of *beginStorage* and *endStorage* store pointers to UNSPECIFIED into variables declared as pointers to *R*.

The general representation scheme is as follows: The storage area [*beginStorage*..*nextR*) consists of zero or more *R*s, each with the form ⟨*backp, length*, $e_0, \ldots, e_{(length-1)}$⟩, where *length* varies from sequence to sequence. The array represented by the record is ⟨$e_0, \ldots, e_{(length-1)}$⟩. If *backp* is not NIL then *backp* is an address in *Table* and *backp* ↑ is the address of *backp* itself. If *Table*[*i*] is not NIL, it is the address of one of these records (cf. Figure 3).

After the initialization, *Storage* is not mentioned again. All the subsequent type breaches in *ArrayStore* are of the pointer arithmetic variety. The expression *endStorage* − *nextR* in *AllocArray* subtracts two *PR's* to produce an integer. The type checker is not entirely asleep here: If we slipped up and wrote

```
IF n + ovh > endStorage − n
```

there would be a complaint because the left-hand side of the comparison is an integer and the right is a *PR*. The assignment

```
nextR ← nextR + (n + ovh)
```

at the end of *AllocArray* also uses the pointer arithmetic breach. The rule *PR* + INTEGER = *PR* makes sense here because *n* + *ovh* is just the right amount to add to *nextR* to produce the next place where an *R* can go.

Despite all these breaches, we are still getting a good deal of checking. The checker would point out (or correct) any address/contents confusions we had, manifested by the omission of ↑'s or their unnecessary

Fig. 1. Definitions module.

```
ArrayStoreDefs: DEFINITIONS =
  BEGIN
  ArrayPtr: TYPE = POINTER TO PR;
  PR: TYPE = POINTER TO R;
  R: TYPE =
     RECORD [ p: Prefix,
              a: ARRAY [0..0] OF Thing ];
  Prefix: TYPE = RECORD [ backp: PRIVATE ArrayPtr,
                          length: READ-ONLY INTEGER ];
  Thing: TYPE = UNIQUE[16];
  AllocArray: PROCEDURE [length: INTEGER]
              RETURNS [new: ArrayPtr];
  FreeArray: PROCEDURE [dying: ArrayPtr];
  END
```

Fig. 2. Implementation of a compacting storage allocator.

```
    DIRECTORY ArrayStoreDefs: FROM "ArrayStoreDefs";
    DEFINITIONS FROM ArrayStoreDefs.
  ArrayStore: PROGRAM IMPLEMENTING ArrayStoreDefs =
    BEGIN
    Storage: ARRAY [0..StorageSize) OF UNSPECIFIED;
    StorageSize: INTEGER = 2000;
    Table: ARRAY TableIndex OF PR;
    Table Index: TYPE = [0..TableSize);
    TableSize: INTEGER = 500;
    beginStorage: PR = @Storage[0];
                --the address of Storage[0]
    endStorage: PR = @Storage[StorageSize];
    nextR: PR ← beginStorage; --next space to put an R
    beginTable: ArrayPtr = @Table[0];
    endTable: ArrayPtr = @Table[TableSize];
    ovh: INTEGER = SIZE[Prefix]; --overhead
  AllocArray: PUBLIC PROCEDURE [n: INTEGER]
                      RETURNS [new: ArrayPtr] =
    BEGIN i:TableIndex;
    IF n < 0 OR n > 77777B − ovh THEN ERROR;
    IF n + ovh > endStorage − nextR THEN
      BEGIN
      Compact[ ];
      IF n + ovh > endStorage − nextR THEN ERROR;
      END;
    --Find a table entry
    FOR i IN TableIndex DO
      IF Table[i] = NIL THEN GOTO found
      REPEAT
        found ⇒ new ← @Table[i];
        FINISHED ⇒ ERROR
      ENDLOOP;
    new ↑ ← nextR;
    --initialize the array storage
    new ↑ ↑ .p.backp ← new;
    new ↑ ↑ .p.length ← n;
    nextR ← nextR + (n + ovh);
    END;
  Compact: PROCEDURE = (omitted)
  FreeArray: PUBLIC PROCEDURE [dead: ArrayPtr] =
    BEGIN IF dead ↑ = NIL THEN ERROR; --array already free
    dead ↑ ↑ .p.backp ← NIL;
    dead ↑ ← NIL;
    END;
  --Initialization
  i: TableIndex;
  FOR i IN TableIndex DO Table[i] ← NIL ENDLOOP;
  END.
```

appearance. We can be sure that integers and *PR*s are not being mixed up. In the (unlikely) event that we wrote something like

```
new↑.p.length ← new↑.a[k]
```

we would be warned because the value on the left is an integer and the value on the right is a *Thing*. Notice that none of this checking would occur if *Thing* were replaced by UNSPECIFIED. Thus, even though the type system is not airtight, we are better off than we would be in a completely unchecked language (unless, perhaps, we get a false sense of security).

Now let us consider how this module is to be used by a client who wants to manipulate two different kinds of arrays: arrays of integers and arrays of strings. At first it looks as if the code is going to have a very high density of RECAST's. For example, to create an array and store an integer in it the client will have to say

```
IA: ArrayPtr = AllocArray[100];
IA↑↑.a[2] ← RECAST[6]
```

because the type of *IA*↑↑*.a*[2] is *Thing,* which does not match anything. Writing a loophole every time is intolerable, so we are tempted to replace *Thing* by UNSPECIFIED, thereby losing a certain amount of type checking elsewhere.

There are much nicer ways out of this problem. Rather than passing every array element through a loophole, one can pass the procedures *AllocArray* and *FreeArray* through loopholes (once, during initialization). The module *ArrayClient* (cf. Figure 4) shows how this is done. Not only does this save our having to make *Thing* UNSPECIFIED, it allows us to use the type checker to ensure that integer arrays contain only integers and that string arrays contain only strings. More precisely, the type checker guarantees that every store into *IA* stores an integer. We must depend upon the correctness of the code in *ArrayStore*, particularly the compactor, to make sure that data structures stay well formed.

This scheme does not have any provisions for coping with problem (d), dangling reference errors. However, somewhat surprisingly, problem (e)—saving a raw pointer—cannot happen as long as the client does not commit any further breaches of the type system. The trick is in the way we declared *IntArray*—all in one mouthful. That makes it impossible to declare a variable to hold a raw pointer. This is because (as mentioned before) every occurrence of the type constructor RECORD generates a new type, distinct from all other types. Therefore, even if we should declare

```
rawPointer: POINTER TO RECORD [
  p: Prefix,
  a: ARRAY[0..0] OF INTEGER ];
```

we could not perform the assignment *rawpointer* ← *IA*↑ because *IA*↑ has a different type, even though it looks the same. If one cannot declare the type of *IA*↑, it is rather difficult to hang onto it for very long. In fact, the compiler has been carefully designed to ensure that no type-checked program can hold such a pointer across a procedure call.

Passing procedure values through loopholes is a rather frightening thing to do. What if, by some mischance, *AllocArray* doesn't have the number of parameters ascribed to it by the client? Since we have waved off the type checker to do the assignment of *AllocArray* to *AllocIntArray* and *AllocStrArray*, no compile-time type violation would be detected and some hard-to-diagnose disaster would occur at run time. To compensate for this, we introduce the curious procedure *Gedanken*, whose only purpose is to fail to compile if the number or size of *AllocArray*'s parameters change. The skeleton type system, discussed earlier in this section, would obviate the need for this foolishness.

We would like to emphasize that, although our examples focus on controlled breaches of the type system, many real Mesa programs do not violate the type system at all. We also expect the density of breaches to decrease as the descriptive powers of the type system increase.

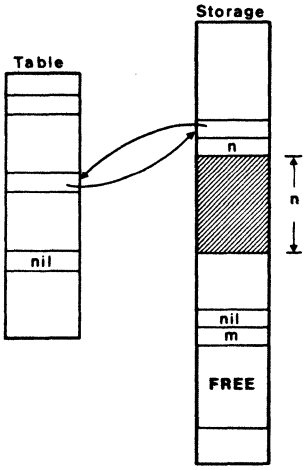
5. Variant Records

Mesa, like Pascal, has variant records. The descriptive aspects of the two languages' notion of variant records are very similar. Mesa, however, also requires strict type checking for accessing the components of variant records. To illustrate the Mesa variant record facility consider the following example of the declaration for an I/O stream:

```
StreamHandle: TYPE = POINTER TO Stream;
StreamType: TYPE = {disk, display, keyboard};
Stream: TYPE = RECORD [
  Get: PROCEDURE[StreamHandle]RETURNS[Item],
  Put: PROCEDURE[StreamHandle, Item],
  body: SELECT type; StreamType FROM
    disk ⇒ [
      file: FilePointer,
      position: Position,
      SetPosition: PROCEDURE [
        POINTER TO disk Stream,
        Position],
      buffer: SELECT size:* FROM
        short ⇒ [b: ShortArray],
        long ⇒ [b: LongArray],
        ENDCASE ],
    display ⇒ [
      first: DisplayControlBlock,
      last: DisplayControlBlock,
      position: ScreenPosition,
      nLines: [0..100]],
    keyboard ⇒ NULL,
    ENDCASE];
```

The record type has three main variants; *disk, display,* and *keyboard.* Furthermore, the disk variant has two variants of its own: *short* and *long.* Note that the field names used in variant subparts need not be unique. The asterisk used in declaring the subvariant of

Fig. 3. *ArrayStore's* data structure.



disk is a shorthand mechanism for generating an enumerated type for tagging variant subparts.

The declaration of a variant record species a type, as usual; it is the type of the whole record. The declaration itself defines some other types: one for each variant in the record. In the above example, the total number of type variations is six, and they are used in the following declarations:

```
r: Stream;
rDisk: disk Stream;
rDisplay: display Stream;
rKeyb: keyboard Stream;
rShort: short disk Stream;
rLong: long disk Stream;
```

The last five types are called *bound variant types*. The rightmost name must be the type identifier for a variant record. The other names are adjectives modifying the type identified to their right. Thus *disk* modifies the type *Stream* and identifies a new type. Further, *short* modifies the type *disk Stream* and identifies still another type. Names must occur in order and may not be skipped. (For instance, *short Stream* would be incorrect since *short* does not identify a *Stream* variant.)

When a record is a bound variant, the components of its variant part may be accessed without a preliminary test. For example, the following assignments are legal:

```
rDisplay.last ← rDisplay.first;
rDisk.position ← rShort.position;
```

If a record is not a bound variant (e.g. *r* in the previous section), the program needs a way to decide which variant it is before accessing variant components. More importantly, the testing of the variant must be done in a formal way so that the type checker can verify that the programmer is not making unwarranted assumptions about which variant is in hand. For this purpose, Mesa uses a *discrimination* statement which resembles the declaration of the variant part. However, the arms in a discriminating SELECT contain statements; and, within a given arm, the discriminated record value is viewed as a bound variant. Therefore, within that arm, its variant components may be accessed using normal qualification. The following example discriminates on *r*:

```
WITH streamRec: r SELECT FROM
  display ⇒
     BEGIN streamRec.first ← streamRec.last;
     streamRec.position ← 73; streamRec.nLines ← 4;
     END;
  disk ⇒
   WITH diskRec: streamRec SELECT FROM
     short ⇒ diskRec.b[0] ← 10;
     long ⇒ diskRec.b[0] ← 100;
     ENDCASE;
ENDCASE ⇒ streamrec.put ← streamrec.newput;
```

The expression in the WITH clause must represent either a variant record (e.g. *r*) or a pointer to a variant record. The identifier preceding the colon in the WITH clause is a synonym for the record. Within each selection, the type of the identifier is the selected bound variant type, and fields specific to the particular variant can be mentioned.

In addition to the descriptive advantages of bound variant types, the Mesa compiler also exploits the more precise declaration of a particular variant to allocate the minimal amount of storage for variables declared to be of a bound variant type. For example, the storage for *r* above must be sufficient to contain any one of the five possible variants. The storage for *rKeyb*, on the other hand, need only be sufficient for storing a *keyboard Stream*.

The Mutable Variant Record Problem

The names *streamRec* and *diskRec* in the example above are really synonyms in the sense that they name the same storage as *r*; no copying is done by the discrimination operation. This decision opens a loophole in the type system. Given the declaration

```
Splodge: TYPE = RECORD [
  refcount: INTEGER;
  vp: SELECT t: * FROM
     blue ⇒
        [x: ARRAY[0..1000) OF CHARACTER],
     red ⇒
        [item: INTEGER, left, right: POINTER TO Splodge],
     green ⇒
        [item: INTEGER, next: POINTER TO green Splodge],
     ENDCASE];
```

one can write the code

```
t: Splodge;
P: PROCEDURE = BEGIN t ← Splodge[0, green[10, NIL]] END;
. . .
WITH s: t SELECT FROM
  red ⇒ BEGIN . . . P[  ] . . . . s.left ← s.right END;
```

The procedure *P* overwrites *t*, and therefore *s*, with a *green Splodge*. The subsequent references to *s.left* and *s.right* are invalid and will cause great mischief.

Closing this breach is simple enough: we could have simply followed Algol 68 and combined the discrimination with a copying operation that places the entire

Fig. 4. Client of a compacting allocator.

```
    DIRECTORY ArrayStoreDefs: FROM "ArrayStoreDefs";
    DEFINITIONS FROM ArrayStoreDefs;
ArrayClient: PROGRAM =
  BEGIN
  --Integer array primitives
  IntArray: TYPE = POINTER TO POINTER TO
        RECORD[p: Prefix, a: ARRAY [0..0] OF INTEGER];
  AllocIntArray: PROCEDURE [INTEGER] RETURNS [IntArray]
          = RECAST[AllocArray];
  FreeIntArray: PROCEDURE [IntArray]
          = RECAST[FreeArray];
  --String array primitives
  StrArray: TYPE = POINTER TO POINTER TO
        RECORD[p: Prefix, a: ARRAY [0..0] OF STRING];
  AllocStrArray: PROCEDURE [INTEGER] RETURNS [StrArray]
          = RECAST[AllocArray];
  FreeStrArray: PROCEDURE [StrArray]
          = RECAST [FreeArray];
  Gedanken: PROCEDURE =
    --This procedure's only role in life is to fail to
    compile if ArrayStore does not have the right sort of
    procedures.
    BEGIN
    uAllocArray:
      PROCEDURE [INTEGER] RETURNS [UNSPECIFIED]
        = AllocArray;
    uFreeArray: PROCEDURE [UNSPECIFIED] = FreeArray;
    END;
  IA: IntArray = AllocIntArray[100];
  SA: StrArray = AllocStrArray[10];
  i: INTEGER;
  FOR i IN [0..IA ↑ ↑ .p.length) DO IA ↑ ↑ .a[i] ← i/3 ENDLOOP;
  SA ↑ ↑ .a[0] ← "zero"; SA ↑ ↑ .a[1] ← "one";
  SA ↑ ↑ .a[2] ← "two"; SA ↑ ↑ .a[3] ← "surprise";
  SA ↑ ↑ .a[4] ← "four";
  FreeIntArray[IA];
  FreeStrArray[SA];
  END.
```

Splodge in a new location (*s*) which is fixed to be *red*. We chose not to do so for three reasons:

(1) Making copies can be expensive.
(2) Making a copy destroys useful sharing relations.
(3) This loophole has yet to cause a problem.

Consider the following procedure, which is representative of those found throughout the Mesa compiler's symbol table processor:

```
Add5: PROCEDURE[ x: POINTER TO Splodge] =
  BEGIN y: POINTER TO green Splodge;
  IF x = NIL THEN RETURN;
  WITH s: x ↑ SELECT FROM
    blue ⇒ RETURN;
    red ⇒
      BEGIN s .item ← s .item + 5;
          Add5[s.left]; Add5[s.right] END;
    green ⇒
      BEGIN y ← @s; -- means y ← x
      UNTIL y = NIL DO
        y ↑ .item ← y ↑ .item + 5; y ← y ↑ .next;
        ENDLOOP;
      END
    ENDCASE
  END
```

As it stands, this procedure runs through a *Splodge*, adding 5 to all the integers in it. Suppose we chose to copy while discriminating: i.e. suppose *x* ↑ were copied into some new storage named *s*. In the *blue* arm a lot of space and time would be wasted copying a 1000-character array into *s*, even though it was never used. In the *red* arm the assignment to *s*'s *item* field is useless since it doesn't affect the original structure.

The *green* arm illustrates the usefulness of declaring bound variant types like *green Splodge* explicitly. If we had to declare *y* and the *next* field of a *green Splodge* to be simply *Splodges*, even though we knew they were always *green*, the loop in that arm would have to be rewritten to contain a useless discrimination.

To achieve the effect we desire under a copy-while-discriminating regime, we would have to redesign our data structure to include another level of pointers:

```
Splodge: TYPE = RECORD [
  refcount: INTEGER;
  vp: SELECT t: * FROM
    blue ⇒ [POINTER TO BlueSplodge],
    red ⇒ [POINTER TO RedSplodge],
    green ⇒ [POINTER TO GreenSplodge],
    ENDCASE];
  BlueSplodge: TYPE = RECORD[
    x: ARRAY[0..1000) OF CHARACTER];
  RedSpolodge: TYPE = RECORD[
    item: INTEGER, left, right: POINTER TO Splodge];
  GreenSplodge: TYPE = RECORD[
    item: INTEGER, next: POINTER TO GreenSplodge];
```

Now we do not mind copying because it doesn't consume much time or space, and it doesn't destroy the sharing relations. Unfortunately, we must pay for the storage occupied by the extra pointers, and this might be intolerable if we have a large collection of *Splodges*.

How have we lived with this loophole so far without getting burnt? It seems that we hardly ever change the variant of a record once it has been initialized. Therefore the possible confusions never occur because the variant never changes after being discriminated. In light of this observation, our suggestion for getting rid of the breach is simply to invent an attribute IMMUTABLE whose attachment to a variant record declaration guarantees that changing the variant is impossible after initialization. This means that special syntax must be invented for the initialization step, but that is all to the good since it provides an opportunity for a storage allocator to allocate precisely the right amount of space.

6. Conclusions

In this paper, we have discussed our experiences with program modularization and strict type checking. It is hard to resist drawing parallels between the disciplines introduced by these features on the one hand and those introduced by programming without *goto*s on the other. In view of the great *goto* debates of recent memory, we would like to summarize our experiences with the following observations and cautions.

(1) The benefits from these linguistic mechanisms, large though they might be, do not come automatically. A programmer must learn to use them effectively. We are just beginning to learn how to do so.

(2) Just as the absence of *goto*s does not always make a program better, the absence of type errors does not make it better if their absence is purchased by sacrificing clarity, efficiency, or type articulation.

(3) Most good programmers use many of the techniques implied by these disciplines, often subconsciously, and can do so in any reasonable language. Language design can help by making the discipline more convenient and systematic, and by catching blunders or other unintended violations of conventions. Acquiring a particular programming style seems to depend on having a language that supports or requires it; once assimilated, however, that style can be applied in many other languages.

Acknowledgments. The principal designers of Mesa, in addition to the authors, have been Butler Lampson and Jim Mitchell. The major portion of the Mesa operating system was programmed by Richard Johnsson and John Wick of the System Development Division of Xerox. In addition to those mentioned above, Douglas Clark, Howard Sturgis, and Niklaus Wirth have made helpful comments on earlier versions of this paper.

References

1. Dahl, O.-J., Myhrhaug, B., and Nygaard, K. The SIMULA 67 common base language. Publ. No. S-2, Norwegian Comptng. Ctr., Oslo, May 1968.

2. Dennis, J.B., and Van Horn, E. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (March 1966), 143–155.

3. Geschke, C., and Mitchell, J. On the problem of uniform references to data structures. *IEEE Trans. Software Eng. SE-1*, 2 (June 1975), 207–219.

4. Habermann, A.N. Critical comments on the programming language PASCAL. *Acta Informatica 3* (1973), 47–57.

5. Knuth, D. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.

6. Koster, C.H.A. On infinite modes. *ALGOL Bull. AB 30.3.3* (Feb. 1969), 109–112.

7. Lampson, B., Mitchell, J., and Satterthwaite, E. On the transfer of control between contexts. In *Lecture Notes in Computer Science, Vol. 19*, G. Goos and J. Hartmanis, Eds., Springer-Verlag, New York. (1974), 181–203.

8. Mitchell, J., and Wegbreit, B. Schemes: a high level data structuring concept. To appear in *Current Trends in Programming Methodologies,* R. Yeh, Ed., Prentice-Hall, Englewood Cliffs, N.J.

9. Morris, J. Protection in programming languages. *Comm. ACM 16*, 1 (Jan 1973), 15–21.

10. Parnas, D. A technique for software module specification. *Comm. ACM 15*, 5 (May 1972), 330–336.

11. Stoy, J.E., and Strachey, C. OS6 – an experimental operating system for a small computer, Part 2; input/output and filing system. *Computer J. 15*, 3 (Aug 1972), 195–203.

12. van Wijngaarden, A., Ed. A report on the algorithmic language ALGOL 68. *Num. Math. 14*, 2 (1969), 79–218.

13. Wegbreit, B. The treatment of data types in EL1. *Comm. ACM 17*, 5 (May 1974), 251–264.

14. Wirth, N. The programming language PASCAL. *Acta Informatica 1* (1971), 35–63.

Operating Systems

R. Stockton Gaines
Editor

Experience with Processes and Monitors in Mesa

Butler W. Lampson
Xerox Palo Alto Research Center

David D. Redell
Xerox Business Systems

The use of monitors for describing concurrency has been much discussed in the literature. When monitors are used in real systems of any size, however, a number of problems arise which have not been adequately dealt with: the semantics of nested monitor calls; the various ways of defining the meaning of WAIT; priority scheduling; handling of timeouts, aborts and other exceptional conditions; interactions with process creation and destruction; monitoring large numbers of small objects. These problems are addressed by the facilities described here for concurrent programming in Mesa. Experience with several substantial applications gives us some confidence in the validity of our solutions.

Key Words and Phrases: concurrency, condition variable, deadlock, module, monitor, operating system, process, synchronization, task

CR Categories: 4.32, 4.35, 5.24

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

A version of this paper was presented at the 7th ACM Symposium on Operating Systems Principles, Pacific Grove, Calif., Dec. 10-12, 1979.

Authors' present address: B. W. Lampson and D. D. Redell, Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, CA 94304.

© 1980 ACM 0001-0782/80/0200-0105 $00.75.

1. Introduction

In early 1977 we began to design the concurrent programming facilities of Pilot, a new operating system for a personal computer [18]. Pilot is a fairly large program itself (24,000 lines of Mesa code). In addition, it must support a variety of quite large application programs, ranging from database management to internetwork message transmission, which are heavy users of concurrency; our experience with some of these applications is discussed later in the paper. We intended the new facilities to be used at least for the following purposes:

Local concurrent programming. An individual application can be implemented as a tightly coupled group of synchronized processes to express the concurrency inherent in the application.

Global resource sharing. Independent applications can run together on the same machine, cooperatively sharing the resources; in particular, their processes can share the processor.

Replacing interrupts. A request for software attention to a device can be handled directly by waking up an appropriate process, without going through a separate interrupt mechanism (e.g., a forced branch).

Pilot is closely coupled to the Mesa language [17], which is used to write both Pilot itself and the applications programs it supports. Hence it was natural to design these facilities as part of Mesa; this makes them easier to use, and also allows the compiler to detect many kinds of errors in their use. The idea of integrating such facilities into a language is certainly not new; it goes back at least as far as PL/I [1]. Furthermore the invention of monitors by Dijkstra, Hoare, and Brinch Hansen [3, 5, 8] provided a very attractive framework for reliable concurrent programming. There followed a number of papers on the integration of concurrency into programming languages, and at least one implementation [4].

We therefore thought that our task would be an easy one: read the literature, compare the alternatives offered there, and pick the one most suitable for our needs. This expectation proved to be naive. Because of the large size and wide variety of our applications, we had to address a number of issues which were not clearly resolved in the published work on monitors. The most notable among these are listed below, with the sections in which they are discussed.

(a) *Program structure.* Mesa has facilities for organizing programs into modules which communicate through well-defined interfaces. Processes must fit into this scheme (see Section 3.1).

(b) *Creating processes.* A set of processes fixed at compile-time is unacceptable in such a general-purpose system (see Section 2). Existing proposals for varying the amount of concurrency were limited to concurrent elaboration of the statements in a block, in the style of Algol 68 (except for the rather complex mechanism in PL/I).

(c) *Creating monitors.* A fixed number of monitors is also unacceptable, since the number of synchronizers should be a function of the amount of data, but many of the details of existing proposals depended on a fixed association of a monitor with a block of the program text (see Section 3.2).

(d) *WAIT in a nested monitor call.* This issue had been (and has continued to be) the source of a considerable amount of confusion, which we had to resolve in an acceptable manner before we could proceed (see Section 3.1).

(e) *Exceptions.* A realistic system must have timeouts, and it must have a way to abort a process (see Section 4.1). Mesa has an UNWIND mechanism for abandoning part of a sequential computation in an orderly way, and this must interact properly with monitors (see Section 3.3).

(f) *Scheduling.* The precise semantics of waiting on a condition variable had been discussed [10] but not agreed upon, and the reasons for making any particular choice had not been articulated (see Section 4). No attention had been paid to the interaction between monitors and priority scheduling of processes (see Section 4.3).

(g) *Input-output.* The details of fitting I/O devices into the framework of monitors and condition variables had not been fully worked out (see Section 4.2).

Some of these points have also been made by Keedy [12], who discusses the usefulness of monitors in a modern general-purpose mainframe operating system. The Modula language [21] addresses (b) and (g), but in a more limited context than ours.

Before settling on the monitor scheme described below, we considered other possibilities. We felt that our first task was to choose either shared memory (i.e., monitors) or message passing as our basic interprocess communication paradigm.

Message passing has been used (without language support) in a number of operating systems; for a recent proposal to embed messages in a language, see [9]. An analysis of the differences between such schemes and those based on monitors was made by Lauer and Needham [14]. They conclude that, given certain mild restrictions on programming style, the two schemes are duals under the transformation

message ↔ process
process ↔ monitor
send/reply ↔ call/return

Since our work is based on a language whose main tool of program structuring is the procedure, it was considerably easier to use a monitor scheme than to devise a message-passing scheme properly integrated with the type system and control structures of the language.

Within the shared memory paradigm, we considered the possibility of adopting a simpler primitive synchronization facility than monitors. Assuming the absence of multiple processors, the simplest form of mutual exclusion appears to be a nonpreemptive scheduler; if processes only yield the processor voluntarily, then mutual exclusion is insured between yield-points. In its simplest form, this approach tends to produce very delicate programs, since the insertion of a yield in a random place can introduce a subtle bug in a previously correct program. This danger can be alleviated by the addition of a modest amount of "syntactic sugar" to delineate critical sections within which the processor must not be yielded (e.g., pseudo monitors). This sugared form of nonpreemptive scheduling can provide extremely efficient solutions to simple problems, but was nonetheless rejected for four reasons:

(1) While we were willing to accept an implementation which would not work on multiple processors, we did not want to embed this restriction in our basic semantics.

(2) A separate preemptive mechanism is needed anyway, since the processor must respond to time-critical events (e.g., I/O interrupts) for which voluntary process switching is clearly too sluggish. With preemptive process scheduling, interrupts can be treated as ordinary process wakeups, which reduces the total amount of machinery needed and eliminates the awkward situations which tend to occur at the boundary between two scheduling regimes.

(3) The use of nonpreemption as mutual exclusion restricts programming generality within critical sections; in particular, a procedure that happens to yield the processor cannot be called. In large systems where modularity is essential, such restrictions are intolerable.

(4) The Mesa concurrency facilities function in a virtual memory environment. The use of nonpreemption as mutual exclusion forbids multiprogramming across page faults, since that would effectively insert preemptions at arbitrary points in the program.

For mutual exclusion with a preemptive scheduler, it is necessary to introduce explicit locks, and machinery which makes requesting processes wait when a lock is unavailable. We considered casting our locks as semaphores, but decided that, compared with monitors, they exert too little structuring discipline on concurrent programs. Semaphores do solve several different problems with a single mechanism (e.g, mutual exclusion, producer/consumer) but we found similar economies in our implementation of monitors and condition variables (see Section 5.1).

We have not associated any protection mechanism with processes in Mesa, except what is implicit in the type system of the language. Since the system supports only one user, we feel that the considerable protection offered by the strong typing of the language is sufficient. This fact contributes substantially to the low cost of process operations.

2. Processes

Mesa casts the creation of a new process as a special procedure activation which executes concurrently with its caller. Mesa allows *any* procedure (except an internal procedure of a monitor; see Section 3.1) to be invoked in this way, at the caller's discretion. It is possible to later retrieve the results returned by the procedure. For example, a keyboard input routine might be invoked as a normal procedure by writing:

```
buffer ← ReadLine[terminal]
```

but since *ReadLine* is likely to wait for input, its caller might wish instead to compute concurrently:

```
p ← FORK Readline[terminal];
... ⟨concurrent computation⟩ ...
buffer ← JOIN p;
```

Here the types are

```
Readline: PROCEDURE [Device] RETURNS [Line];
p: PROCESS RETURNS [Line].
```

The rendezvous between the return from *ReadLine* which terminates the new process and the JOIN in the old process is provided automatically. *ReadLine* is the *root* procedure of the new process.

This scheme has a number of important properties.

(a) It treats a process as a first-class value in the language, which can be assigned to a variable or an array element, passed as a parameter, and in general treated exactly like any other value. A process value is like a pointer value or a procedure value which refers to a nested procedure, in that it can become a dangling reference if the process to which it refers goes away.

(b) The method for passing parameters to a new process and retrieving its results is exactly the same as the corresponding method for procedures, and is subject to the same strict type checking. Just as PROCEDURE is a generator for a family of types (depending on the argument and result types), so PROCESS is a similar generator, slightly simpler since it depends only on result types.

(c) No special declaration is needed for a procedure which is invoked as a process. Because of the implementation of procedure calls and other global control transfers in Mesa [13], there is no extra execution cost for this generality.

(d) The cost of creating and destroying a process is moderate, and the cost in storage is only twice the minimum cost of a procedure instance. It is therefore feasible to program with a large number of processes, and to vary the number quite rapidly. As Lauer and Needham [14] point out, there are many synchronization problems which have straightforward solutions using monitors only when obtaining a new process is cheap.

Many patterns of process creation are possible. A common one is to create a *detached* process, which never returns a result to its creator, but instead functions quite independently. When the root procedure *p* of a detached process returns, the process is destroyed without any fuss. The fact that no one intends to wait for a result from *p* can be expressed by executing:

Detach[*p*]

From the point of view of the caller, this is similar to freeing a dynamic variable—it is generally an error to make any further use of the current value of *p*, since the process, running asynchronously, may complete its work and be destroyed at any time. Of course the design of the program may be such that this cannot happen, and in this case the value of *p* can still be useful as a parameter to the *Abort* operation (see Section 4.1).

This remark illustrates a general point: Processes offer some new opportunities to create dangling references. A process variable itself is a kind of pointer, and must not be used after the process is destroyed. Furthermore, parameters passed by reference to a process are pointers, and if they happen to be local variables of a procedure, that procedure must not return until the process is destroyed. Like most implementation languages, Mesa does not provide any protection against dangling references, whether connnected with processes or not.

The ordinary Mesa facility for exception handling uses the ordering established by procedure calls to control the processing of exceptions. Any block may have an attached exception handler. The block containing the statement which causes the exception is given the first chance to handle it, then its enclosing block, and so forth until a procedure body is reached. Then the caller of the procedure is given a chance in the same way. Since the root procedure of a process has no caller, it must be prepared to handle any exceptions which can be generated in the process, including exceptions generated by the procedure itself. If it fails to do so, the resulting error sends control to the debugger, where the identity of the procedure and the exception can easily be determined by a programmer. This is not much comfort, however, when a system is in operational use. The practical consequence is that while any procedure suitable for forking can also be called sequentially, the converse is not generally true.

3. Monitors

When several processes interact by sharing data, care must be taken to properly synchronize access to the data. The idea behind monitors is that a proper vehicle for this interaction is one which unifies

—the synchronization,
—the shared data,
—the body of code which performs the accesses.

The data is *protected* by a *monitor*, and can only be accessed within the body of a *monitor procedure*. There are two kinds of monitor procedures: *entry procedures*, which can be called from outside the monitor, and *internal procedures*, which can only be called from monitor procedures. Processes can only perform operations on the data by calling entry procedures. The monitor ensures that at most one process is executing a monitor procedure at a time; this process is said to be *in* the monitor. If a process is in the monitor, any other process which calls an entry procedure will be delayed. The monitor procedures are written textually next to each other, and next to the declaration of the protected data, so that a reader can conveniently survey all the references to the data.

As long as any order of calling the entry procedures produces meaningful results, no additional synchronization is needed among the processes sharing the monitor. If a random order is not acceptable, other provisions must be made in the program outside the monitor. For example, an unbounded buffer with *Put* and *Get* procedures imposes no constraints (of course a *Get* may have to wait, but this is taken care of within the monitor, as described in the next section). On the other hand, a tape unit with *Reserve*, *Read, Write*, and *Release* operations requires that each process execute a *Reserve* first and a *Release* last. A second process executing a *Reserve* will be delayed by the monitor, but another process doing a *Read* without a prior *Reserve* will produce chaos. Thus monitors do not solve all the problems of concurrent programming; they are intended, in part, as primitive building blocks for more complex scheduling policies. A discussion of such policies and how to implement them using monitors is beyond the scope of this paper.

3.1 Monitor Modules

In Mesa the simplest monitor is an instance of a *module*, which is the basic unit of global program structuring. A Mesa module consists of a collection of procedures and their global data, and in sequential programming is used to implement a data abstraction. Such a module has PUBLIC procedures which constitute the external interface to the abstraction, and PRIVATE procedures which are internal to the implementation and cannot be called from outside the module; its data is normally entirely private. A MONITOR module differs only slightly. It has three kinds of procedures: *entry*, *internal* (private), and *external* (nonmonitor procedures). The first two are the monitor procedures, and execute with the monitor lock held. For example, consider a simple storage allocator with two entry procedures, *Allocate* and *Free*, and an external procedure *Expand* which increases the size of a block.

```
StorageAllocator: MONITOR = BEGIN
  availableStorage: INTEGER;
  moreAvailable: CONDITION;
  . . .
  Allocate: ENTRY PROCEDURE [size: INTEGER]
  RETURNS [p: POINTER] = BEGIN
    UNTIL availableStorage ≥ size
      DO WAIT moreAvailable ENDLOOP;
    p ← ⟨remove chunk of size words & update availableStorage⟩
    END;
  Free: ENTRY PROCEDURE [p: POINTER, size: INTEGER]
        = BEGIN
    ⟨put back chunk of size words & update availableStorage⟩;
    NOTIFY moreAvailable END;
  Expand: PUBLIC PROCEDURE [pOld: POINTER,
    size: INTEGER]
  RETURNS [pNew: POINTER ] = BEGIN
    pNew ← Allocate[size];
    ⟨copy contents from old block to new block⟩;
    Free[pOld] END;
  END.
```

A Mesa module is normally used to package a collection of related procedures and protect their private data from external access. In order to avoid introducing a new lexical structuring mechanism, we chose to make the scope of a monitor identical to a module. Sometimes, however, procedures which belong in an abstraction do not need access to any shared data, and hence need not be entry procedures of the monitor; these must be distinguished somehow.

For example, two asynchronous processes clearly must not execute in the *Allocate* or *Free* procedures at the same time; hence, these must be entry procedures. On the other hand, it is unnecessary to hold the monitor lock during the copy in *Expand*, even though this procedure logically belongs in the storage allocator module; it is thus written as an external procedure. A more complex monitor might also have internal procedures, which are used to structure its computations, but which are inaccessible from outside the monitor. These do not acquire and release the lock on call and return, since they can only be called when the lock is already held.

If no suitable block is available, *Allocate* makes its caller wait on the *condition* variable *moreAvailable*. *Free* does a NOTIFY to this variable whenever a new block becomes available; this causes some process waiting on the variable to resume execution (see Section 4 for details). The WAIT releases the monitor lock, which is reacquired when the waiting process reenters the monitor. If a WAIT is done in an internal procedure, it still releases the lock. If, however, the monitor calls some other procedure which is outside the monitor module, the lock is not released, even if the other procedure is in (or calls) another monitor and ends up doing a WAIT. The same rule is adopted in Concurrent Pascal [4].

To understand the reasons for this, consider the form of a correctness argument for a program using a monitor. The basic idea is that the monitor maintains an *invariant* which is always true of its data, except when some process is executing in the monitor. Whenever control leaves the monitor, this invariant must be established. In return, whenever control enters the monitor the invariant can be assumed. Thus an entry procedure must establish the invariant before returning, and monitor procedures must establish it before doing a WAIT. The invariant can be assumed at the start of an entry procedure, and after each WAIT. Under these conditions, the monitor lock ensures that no one can enter the monitor when the invariant is false. Now, if the lock were to be released on a WAIT done in another monitor which happens to be called from this one, the invariant would have to be established before making the call which leads to the WAIT. Since in general there is no way to know whether a call outside the monitor will lead to a WAIT, the invariant would have to be established before every such call. The result would be to make calling such procedures hopelessly cumbersome.

An alternative solution is to allow an *outside block* to be written inside a monitor, with the following meaning: on entry to the block the lock is released (and hence the invariant must be established); within the block the protected data is inaccessible; on leaving the block the lock is reacquired. This scheme allows the state represented by the execution environment of the monitor to be maintained during the outside call, and imposes a minimal burden on the programmer: to establish the invariant before making the call. This mechanism would be easy to add to Mesa; we have left it out because we have not seen convincing examples in which it significantly simplifies the program.

If an entry procedure generates an exception in the usual way, the result will be a call on the exception handler from within the monitor, so that the lock will not be released. In particular, this means that the exception handler must carefully avoid invoking that same monitor, or a deadlock will result. To avoid this restriction, the entry procedure can restore the invariant and then execute

```
RETURN WITH ERROR[⟨arguments⟩]
```

which returns from the entry procedure, thus releasing the lock, and then generates the exception.

3.2 Monitors and Deadlock

There are three patterns of pairwise deadlock that can occur using monitors. In practice, of course, deadlocks often involve more than two processes, in which case the actual patterns observed tend to be more complicated; conversely, it is also possible for a single process to deadlock with itself (e.g., if an entry procedure is recursive).

The simplest form of deadlock takes place inside a single monitor when two processes do a WAIT, each expecting to be awakened by the other. This represents a localized bug in the monitor code and is usually easy to locate and correct.

A more subtle form of deadlock can occur if there is a cyclic calling pattern between two monitors. Thus if monitor M calls an entry procedure in N, and N calls one in M, each will wait for the other to release the monitor lock. This kind of deadlock is made neither more nor less serious by the monitor mechanism. It arises whenever such cyclic dependencies are allowed to occur in a program, and can be avoided in a number of ways. The simplest is to impose a partial ordering on resources such that all the resources simultaneously possessed by any process are totally ordered, and insist that if resource r precedes s in the ordering, then r cannot be acquired later than s. When the resources are monitors, this reduces to the simple rule that mutually recursive monitors must be avoided. Concurrent Pascal [4] makes this check at compile time; Mesa cannot do so because it has procedure variables.

A more serious problem arises if M calls N, and N then waits for a condition which can only occur when another process enters N through M and makes the condition true. In this situation, N will be unlocked, since the WAIT occurred there, but M will remain locked during the WAIT in N. This kind of two-level data abstraction must be handled with some care. A straightforward solution using standard monitors is to break M into two parts: a monitor M' and an ordinary module O which implements the abstraction defined by M, and calls M' for access to the shared data. The call on N must be done from O rather than from within M'.

Monitors, like any other interprocess communication mechanism, are a *tool* for implementing synchronization constraints chosen by the programmer. It is unreasonable to blame the tool when poorly chosen constraints lead to deadlock. What is crucial, however, is that the tool make the program structure as understandable as possible, while not restricting the programmer too much in his choice of constraints (e.g., by forcing a monitor lock to be held much longer than necessary). To some extent, these two goals tend to conflict; the Mesa concurrency facilities attempt to strike a reasonable balance and provide an environment in which the conscientious programmer can avoid deadlock reasonably easily. Our experience in this area is reported in Section 6.

3.3 Monitored Objects

Often we wish to have a collection of shared data objects, each one representing an instance of some abstract object such as a file, a storage volume, a virtual circuit, or a database view, and we wish to add objects to the collection and delete them dynamically. In a sequential program this is done with standard techniques for allocating and freeing storage. In a concurrent program, however, provision must also be made for serializing access to each object. The straightforward way is to use a single monitor for accessing all instances of the object, and we recommend this approach whenever possible. If the objects function independently of each other for the most part, however, the single monitor drastically reduces the maximum concurrency which can be obtained. In this case, what we want is to give each object its own monitor; all these monitors will share the same code, since all the instances of the abstract object share the same code, but each object will have its own lock.

One way to achieve this result is to make multiple instances of the monitor module. Mesa makes this quite easy, and it is the next recommended approach. However, the data associated with a module instance includes information which the Mesa system uses to support program linking and code swapping, and there is some cost in duplicating this information. Furthermore, module instances are allocated by the system; hence the program cannot exercise the fine control over allocation strategies which is possible for ordinary Mesa data objects. We have therefore introduced a new type constructor called a *monitored record*, which is exactly like an ordinary record, except that it includes a monitor lock and is intended to be used as the protected data of a monitor.

In writing the code for such a monitor, the programmer must specify how to access the monitored record, which might be embedded in some larger data structure passed as a parameter to the entry procedures. This is done with a LOCKS clause which is written at the beginning of the module:

```
MONITOR LOCKS file↑
  USING file: POINTER TO FileData;
```

if the *FileData* is the protected data. An arbitrary expression can appear in the LOCKS clause; for instance, LOCKS *file.buffers*[*currentPage*] might be appropriate if the protected data is one of the buffers in an array which is part of the *file*. Every entry procedure of this monitor, and every internal procedure that does a WAIT, must have access to a *file*, so that it can acquire and release the lock upon entry or around a WAIT. This can be accomplished in two ways: the *file* may be a global variable of the module, or it may be a parameter to *every* such procedure. In the latter case, we have effectively created a separate monitor for each object, without limiting the program's freedom to arrange access paths and storage allocation as it likes.

Unfortunately, the type system of Mesa is not strong enough to make this construction completely safe. If the value of *file* is changed within an entry procedure, for example, chaos will result, since the return from this procedure will release not the lock which was acquired during the call, but some other lock instead. In this example we can insist that *file* be read-only, but with another level of indirection aliasing can occur and such a restriction cannot be enforced. In practice this lack of safety has not been a problem.

3.4 Abandoning a Computation

Suppose that a procedure P_1 has called another procedure P_2, which in turn has called P_3 and so forth until the current procedure is P_n. If P_n generates an exception which is eventually handled by P_1 (because $P_2 \ldots P_n$ do not provide handlers), Mesa allows the exception handler in P_1 to abandon the portion of the computation being done in $P_2 \dots P_n$ and continue execution in P_1. When this happens, a distinguished exception called UNWIND is first generated, and each of $P_2 \dots P_n$ is given a chance to handle it and do any necessary cleanup before its activation is destroyed.

This feature of Mesa is not part of the concurrency facilities, but it does interact with those facilities in the following way. If one of the procedures being abandoned, say P_i, is an entry procedure, then the invariant must be restored and the monitor lock released before P_i is destroyed. Thus if the logic of the program allows an UNWIND, the programmer must supply a suitable handler in P_i to restore the invariant; Mesa will automatically supply the code to release the lock. If the programmer fails to supply an UNWIND handler for an entry procedure, the lock is *not* automatically released, but remains set; the cause of the resulting deadlock is not hard to find.

4. Condition Variables

In this section we discuss the precise semantics of WAIT, and other details associated with condition variables. Hoare's definition of monitors [8] requires that a process waiting on a condition variable must run immediately when another process *signals* that variable, and that the signaling process in turn runs as soon as the waiter leaves the monitor. This definition allows the waiter to assume the truth of some predicate stronger than the monitor invariant (which the signaler must of course establish), but it requires several additional process switches whenever a process continues after a WAIT. It also requires that the signaling mechanism be perfectly reliable.

Mesa takes a different view: When one process establishes a condition for which some other process may be waiting, it *notifies* the corresponding condition variable. A NOTIFY is regarded as a *hint* to a waiting process; it causes execution of some process waiting on the condition to resume at some convenient future time. When the waiting process resumes, it will reacquire the monitor lock. There is no guarantee that some other process will not enter the monitor before the waiting process. Hence nothing more than the monitor invariant may be assumed after a WAIT, and the waiter must reevaluate the situation each time it resumes. The proper pattern of code for waiting is therefore:

```
WHILE NOT ⟨OK to proceed⟩ DO WAIT c
  ENDLOOP.
```

This arrangement results in an extra evaluation of the ⟨OK to proceed⟩ predicate after a wait, compared to Hoare's monitors, in which the code is:

```
IF NOT ⟨OK to proceed⟩ THEN WAIT c.
```

In return, however, there are no extra process switches, and indeed no constraints at all on when the waiting process must run after a NOTIFY. In fact, it is perfectly all right to run the waiting process even if there is not any NOTIFY, although this is presumably pointless if a NOTIFY is done whenever an interesting change is made to the protected data.

It is possible that such a laissez-faire attitude to scheduling monitor accesses will lead to unfairness and even starvation. We do not think this is a legitimate cause for concern, since in a properly designed system there should typically be no processes waiting for a monitor lock. As Hoare, Brinch Hansen, Keedy, and others have pointed out, the low level scheduling mechanism provided by monitor locks should not be used to implement high level scheduling decisions within a system (e.g., about which process should get a printer next). High level scheduling should be done by taking account of the specific characteristics of the resource being scheduled (e.g., whether the right kind of paper is in the printer). Such a scheduler will delay its client processes on condition variables after recording information about their requirements, make its decisions based on this information, and notify the proper conditions. In such a design the data protected by a monitor is never a bottleneck.

The verification rules for Mesa monitors are thus extremely simple: The monitor invariant must be established just before a return from an entry procedure or a WAIT, and it may be assumed at the start of an entry procedure and just after a WAIT. Since awakened waiters do not run immediately, the predicate established before a NOTIFY cannot be assumed after the corresponding WAIT, but since the waiter tests explicitly for ⟨OK to proceed⟩, verification is actually made simpler and more localized.

Another consequence of Mesa's treatment of NOTIFY as a hint is that many applications do not trouble to determine whether the exact condition needed by a waiter has been established. Instead, they choose a very cheap predicate which implies the exact condition (e.g., some change has occurred), and NOTIFY a *covering* condition variable. Any waiting process is then responsible for determining whether the exact condition holds; if not, it simply waits again. For example, a process may need to wait until a particular object in a set changes state. A single condition covers the entire set, and a process changing any of the objects broadcasts to this condition (see Section 4.1). The information about exactly which objects are currently of interest is implicit in the states of the waiting processes, rather than having to be represented explicitly in a shared data structure. This is an attractive way to decouple the detailed design of two processes; it is feasible because the cost of waking up a process is small.

4.1 Alternatives to NOTIFY

With this rule it is easy to add three additional ways to resume a waiting process:

Timeout. Associated with a condition variable is a timeout interval *t*. A process which has been waiting for time *t* will resume regardless of whether the condition has been notified. Presumably in most cases it will check the time and take some recovery action before waiting again. The original design for timeouts raised an exception if the timeout occurred; it was changed because many users simply wanted to retry on a timeout, and objected to the cost and coding complexity of handling the exception. This decision could certainly go either way.

Abort. A process may be aborted at any time by executing *Abort*[*p*]. The effect is that the next time the process waits, or if it is waiting now, it will resume immediately and the *Aborted* exception will occur. This mechanism allows one process to gently prod another, generally to suggest that it should clean up and terminate. The aborted process is, however, free to do arbitrary computations, or indeed to ignore the abort entirely.

Broadcast. Instead of doing a NOTIFY to a condition, a process may do a BROADCAST, which causes *all* the processes waiting on the condition to resume, instead of simply one of them. Since a NOTIFY is just a hint, it is always correct to use BROADCAST. It is better to use NOTIFY if there will typically be several processes waiting on the condition, and it is known that any waiting process can respond properly. On the other hand, there are times when a BROADCAST is correct and a NOTIFY is not; the alert reader may have noticed a problem with the example program in Section 3.1, which can be solved by replacing the NOTIFY with a BROADCAST.

None of these mechanisms affects the proof rule for monitors at all. Each provides a way to attract the attention of a waiting process at an appropriate time.

Note that there is no way to stop a runaway process. This reflects the fact that Mesa processes are cooperative. Many aspects of the design would not be appropriate in a competitive environment such as a general-purpose time-sharing system.

4.2 Naked NOTIFY

Communication with input/output devices is handled by monitors and condition variables much like communication among processes. There is typically a shared data structure, whose details are determined by the hardware, for passing commands to the device and returning status information. Since it is not possible for the device to wait on a monitor lock, the updating operations on this structure must be designed so that the single-word atomic read and write operations provided by the memory are sufficient to make them atomic. When the device needs attention, it can NOTIFY a condition variable to wake up a waiting process (i.e., the interrupt handler); since the device does not actually acquire the monitor lock, its NOTIFY is called a *naked* NOTIFY. The device finds the address of the condition variable in a fixed memory location.

There is one complication associated with a naked NOTIFY: Since the notification is not protected by a monitor lock, there can be a race. It is possible for a process to be in the monitor, find the ⟨OK to proceed⟩ predicate to be FALSE (i.e., the device does not need attention), and be about to do a WAIT, when the device updates the shared data and does its NOTIFY. The WAIT will then be done and the NOTIFY from the device will be lost. With ordinary processes, this cannot happen, since the monitor lock ensures that one process cannot be testing the predicate and preparing to WAIT, while another is changing the value of ⟨OK to proceed⟩ and doing the NOTIFY. The problem is avoided by providing the familiar wakeup-waiting switch [19] in a condition variable, thus turning it into a binary semaphore [8]. This switch is needed only for condition variables that are notified by devices.

We briefly considered a design in which devices would wait on and acquire the monitor lock, exactly like ordinary Mesa processes; this design is attractive because it avoids both the anomalies just discussed. However, there is a serious problem with any kind of mutual exclusion between two processes which run on processors of substantially different speeds: The faster process may have to wait for the slower one. The worst-case response time of the faster process therefore cannot be less than the time the slower one needs to finish its critical section. Although one can get higher throughput from the faster processor than from the slower one, one cannot get better worst-case real-time performance. We consider this a fundamental deficiency.

It therefore seemed best to avoid any mutual exclusion (except for that provided by the atomic memory read and write operations) between Mesa code and device hardware and microcode. Their relationship is easily cast into a producer-consumer form, and this can be implemented, using linked lists or arrays, with only the memory's mutual exclusion. Only a small amount of Mesa code must handle device data structures without the protection of a monitor. Clearly a change of models must occur at some point between a disk head and an application program; we see no good reason why it should not happen within Mesa code, although it should certainly be tightly encapsulated.

4.3 Priorities

In some applications it is desirable to use a priority scheduling discipline for allocating the processor(s) to processes which are not waiting. Unless care is taken, the ordering implied by the assignment of priorities can be subverted by monitors. Suppose there are three priority levels (3 highest, 1 lowest), and three processes P_1, P_2, and P_3, one running at each level. Let P_1 and P_3 communicate using a monitor M. Now consider the following sequence of events:

P_1 enters M.
P_1 is preempted by P_2.
P_2 is preempted by P_3.
P_3 tries to enter the monitor, and waits for the lock.
P_2 runs again, and can effectively prevent P_3 from running, contrary to the purpose of the priorities.

A simple way to avoid this situation is to associate with each monitor the priority of the highest-priority process which ever enters that monitor. Then whenever a process enters a monitor, its priority is temporarily increased to the monitor's priority. Modula solves the problem in an even simpler way—interrupts are disabled on entry to M, thus effectively giving the process the highest possible priority, as well as supplying the monitor lock for M. This approach fails if a page fault can occur while executing in M.

The mechanism is not free, and whether or not it is needed depends on the application. For instance, if only processes with adjacent priorities share a monitor, the problem described above cannot occur. Even if this is not the case, the problem may occur rarely, and absolute enforcement of the priority scheduling may not be important.

5. Implementation

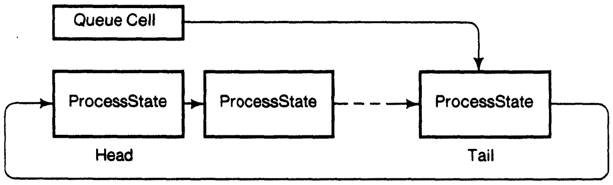
The implementation of processes and monitors is split more or less equally among the Mesa compiler, the runtime package, and the underlying machine. The compiler recognizes the various syntactic constructs and generates appropriate code, including implicit calls on built-in (i.e., known to the compiler) support procedures. The runtime implements the less heavily used operations, such as process creation and destruction. The machine directly implements the more heavily used features, such as process scheduling and monitor entry/exit.

Note that it was primarily frequency of use, rather than cleanliness of abstraction, that motivated our division of labor between processor and software. Nonetheless, the split did turn out to be a fairly clean layering, in which the birth and death of processes are implemented on top of monitors and process scheduling.

5.1 The Processor

The existence of a process is normally represented only by its stack of procedure activation records or *frames*, plus a small (10-byte) description called a *ProcessState*. Frames are allocated from a *frame heap* by a microcoded allocator. They come in a range of sizes which differ by 20 percent to 30 percent; there is a separate free list for each size up to a few hundred bytes (about 15 sizes). Allocating and freeing frames are thus very fast, except when more frames of a given size are needed. Because all frames come from the heap, there is no need to preplan the stack space needed by a process. When a frame of a given size is needed but not available, there is a *frame fault*, and the fault handler allocates more frames in virtual memory. Resident procedures have a private frame heap which is replenished by seizing real memory from the virtual memory manager.

Fig. 1. A process queue.



The *ProcessStates* are kept in a fixed table known to the processor; the size of this table determines the maximum number of processes. At any given time, a *ProcessState* is on exactly one *queue*. There are four kinds of queues:

Ready queue. There is one ready queue, containing all processes which are ready to run.

Monitor lock queue. When a process attempts to enter a locked monitor, it is moved from the ready queue to a queue associated with the monitor lock.

Condition variable queue. When a process executes a WAIT, it is moved from the ready queue to a queue associated with the condition variable.

Fault queue. A fault can make a process temporarily unable to run; such a process is moved from the ready queue to a fault queue, and a fault-handling process is notified.

Queues are kept sorted by process priority. The implementation of queues is a simple one-way circular list, with the queue-cell pointing to the *tail* of the queue (see Figure 1). This compact structure allows rapid access to both the head and the tail of the queue. Insertion at the tail and removal at the head are quick and easy; more general insertion and deletion involve scanning some fraction of the queue. The queues are usually short enough that this is not a problem. Only the ready queue grows to a substantial size during normal operation, and its patterns of insertions and deletions are such that queue scanning overhead is small.

The queue cell of the ready queue is kept in a fixed location known to the processor, whose fundamental task is to always execute the next instruction of the highest priority ready process. To this end, a check is made before each instruction, and a process switch is done if necessary. In particular, this is the mechanism by which interrupts are serviced. The machine thus implements a simple priority scheduler, which is preemptive between priorities and FIFO within a given priority.

Queues other than the ready list are passed to the processor by software as operands of instructions, or through a trap vector in the case of fault queues. The queue cells are passed by reference, since in general they must be updated (i.e., the identity of the tail may change.) Monitor locks and condition variables are implemented as small records containing their associated queue cells plus a small amount of extra information: in a monitor lock, the actual lock; in a condition variable, the timeout interval and the wakeup-waiting switch.

At a fixed interval (~20 times per second) the processor scans the table of *ProcessStates* and notifies any waiting processes whose timeout intervals have expired. This special NOTIFY is tricky because the processor does not know the location of the condition variables on which such processes are waiting, and hence cannot update the queue cells. This problem is solved by leaving the queue cells out of date, but marking the processes in such a way that the next normal usage of the queue cells will notice the situation and update them appropriately.

There is no provision for time-slicing in the current implementation, but it could easily be added, since it has no effect on the semantics of processes.

5.2 The Runtime Support Package

The *Process* module of the Mesa runtime package does creation and deletion of processes. This module is written (in Mesa) as a monitor, thus utilizing the underlying synchronization machinery of the processor to coordinate the implementation of FORK and JOIN as the built-in entry procedures *Process.Fork* and *Process.Join*, respectively. The unused *ProcessStates* are treated as essentially normal processes which are all waiting on a condition variable called *rebirth*. A call of *Process.Fork* performs appropriate "brain surgery" on the first process in the queue and then notifies *rebirth* to bring the process to life; *Process.Join* synchronizes with the dying process and retrieves the results. The (implicitly invoked) procedure *Process.End* synchronizes the dying process with the joining process and then commits suicide by waiting on *rebirth*. An explicit cell on *Process.Detach* marks the process so that when it later calls *Process.End*, it will simply destroy itself immediately.

The operations *Process.Abort* and *Process.Yield* are provided to allow special handling of processes which wait too long and compute too long, respectively. Both adjust the states of the appropriate queues, using the machine's standard queueing mechanisms. Utility routines are also provided by the runtime for such operations as setting a condition variable timeout and setting a process priority.

5.3 The Compiler

The compiler recognizes the syntactic constructs for processes and monitors and emits the appropriate code (e.g., a MONITORENTRY instruction at the start of each entry procedure, an implicit call of *Process.Fork* for each FORK). The compiler also performs special static checks to help avoid certain frequently encountered errors. For example, use of WAIT in an external procedure is flagged as an error, as is a direct call from an external procedure to an internal one. Because of the power of the underlying Mesa control structure primitives, and the care with which concurrency was integrated into the language, the introduction of processes and monitors into Mesa resulted in remarkably little upheaval inside the compiler.

5.4 Performance

Mesa's concurrent programming facilities allow the intrinsic parallelism of application programs to be represented naturally; the hope is that well-structured programs with high global efficiency will result. At the same time, these facilities have nontrivial local costs in storage and/or execution time when compared with similar sequential constructs; it is important to minimize these costs, so that the facilities can be applied to a finer "grain" of concurrency. This section summarizes the costs of processes and monitors relative to other basic Mesa constructs, such as simple statements, procedures, and modules. Of course, the relative efficiency of an arbitrary concurrent program and an equivalent sequential one cannot be determined from these numbers alone; the intent is simply to provide an indication of the relative costs of various local constructs.

Storage costs fall naturally into data and program storage (both of which reside in swappable virtual memory unless otherwise indicated). The minimum cost for the existence of a Mesa module is 8 bytes of data and 2 bytes of code. Changing the module to a monitor adds 2 bytes of data and 2 bytes of code. The prime component of a module is a set of procedures, each of which requires a minimum of an 8-byte activation record and 2 bytes of code. Changing a normal procedure to a monitor entry procedure leaves the size of the activation record unchanged, and adds 8 bytes of code. All of these costs are small compared with the program and data storage actually needed by typical modules and procedures. The other cost specific to monitors is space for condition variables; each condition variable occupies 4 bytes of data storage, while WAIT and NOTIFY require 12 bytes and 3 bytes of code, respectively.

The data storage overhead for a process is 10 bytes of resident storage for its *ProcessState*, plus the swappable storage for its stack of procedure activation records. The process itself contains no extra code, but the code for the FORK and JOIN which create and delete it together occupy 13 bytes, as compared with 3 bytes for a normal procedure call and return. The FORK/JOIN sequence also uses 2 data bytes to store the process value. In summary:

	Space (bytes)	
Construct	*data*	*code*
module	8	2
procedure	8	2
call + return	—	3
monitor	10	4
entry procedure	8	10
FORK+JOIN	2	13
process	10	0
condition variable	4	—
WAIT	—	12
NOTIFY	—	3

For measuring execution times we define a unit called a *tick*: The time required to execute a simple instruction (e.g., on a "one-MIP" machine, one tick would be one microsecond). A tick is arbitrarily set at one-fourth of the time needed to execute the simple statement "$a \leftarrow b + c$" (i.e., two loads, an add, and a store). One interesting number against which to compare the concurrency facilities is the cost of a normal procedure call (and its associated return), which takes 30 ticks if there are no arguments or results.

The cost of calling and returning from a monitor entry procedure is 50 ticks, about 70 percent more than an ordinary call and return. In practice, the percentage increase is somewhat lower, since typical procedures pass arguments and return results, at a cost of 2–4 ticks per item. A process switch takes 60 ticks; this includes the queue manipulations and all the state saving and restoring. The speed of WAIT and NOTIFY depends somewhat on the number and priorities of the processes involved, but representative figures are 15 ticks for a WAIT and 6 ticks for a NOTIFY. Finally, the minimum cost of a FORK/JOIN pair is 1,100 ticks, or about 38 times that of a procedure call. To summarize:

Construct	*Time (ticks)*
simple instruction	1
call+return	30
monitor call+return	50
process switch	60
WAIT	15
NOTIFY, no one waiting	4
NOTIFY, process waiting	9
FORK+JOIN	1,100

On the basis of these performance figures, we feel that our implementation has met our efficiency goals, with the possible exception of FORK and JOIN. The decision to implement these two language constructs in software rather than in the underlying machine is the main reason for their somewhat lackluster performance. Nevertheless, we still regard this decision as a sound one, since these two facilities are considerably more complex than the basic synchronization mechanism, and are used much less frequently (especially JOIN, since the detached processes discussed in Section 2 have turned out to be quite popular).

6. Applications

In this section we describe the way in which processes and monitors are used by three substantial Mesa programs: an operating system, a calendar system using replicated databases, and an internetwork gateway.

6.1 Pilot: A General-Purpose Operating System

Pilot is a Mesa-based operating system [18] which runs on a large personal computer. It was designed jointly with the new language features, and makes heavy use of them. Pilot has several autonomous processes of its own, and can be called by any number of client processes of any priority, in a fully asynchronous manner. Exploiting this potential concurrency requires extensive use of monitors within Pilot; the roughly 75 program modules contain nearly 40 separate monitors.

The Pilot implementation includes about 15 dedicated processes (the exact number depends on the hardware configuration); most of these are event handlers for three classes of events:

I/O interrupts. Naked notifies as discussed in Section 4.2.

Process faults. Page faults and other such events, signaled via fault queues as discussed in Section 5.1. Both client code and the higher levels of Pilot, including some of the dedicated processes, can cause such faults.

Internal exceptions. Missing entries in resident databases, for example, cause an appropriate high level "helper" process to wake up and retrieve the needed data from secondary storage.

There are also a few "daemon" processes, which awaken periodically and perform housekeeping chores (e.g., swap out unreferenced pages). Essentially all of Pilot's internal processes and monitors are created at system initialization time (in particular, a suitable complement of interrupt-handler processes is created to match the actual hardware configuration, which is determined by interrogating the hardware). The running system makes no use of dynamic process and monitor creation, largely because much of Pilot is involved in implementing facilities such as virtual memory which are themselves used by the dynamic creation software.

The internal structure of Pilot is fairly complicated, but careful placement of monitors and dedicated processes succeeded in limiting the number of bugs which caused deadlock; over the life of the system, somewhere between one and two dozen distinct deadlocks have been discovered, all of which have been fixed relatively easily without any global disruption of the system's structure.

At least two areas have caused annoying problems in the development of Pilot:

(1) *The lack of mutual exclusion in the handling of interrupts.* As in more conventional interrupt systems, subtle bugs have occurred due to timing races between I/O devices and their handlers. To some extent, the illusion of mutual exclusion provided by the casting of interrupt code as a monitor may have contributed to this, although we feel that the resultant economy of mechanism still justifies this choice.

(2) *The interaction of the concurrency and exception facilities.* Aside from the general problems of exception handling in a concurrent environment, we have experienced some difficulties due to the specific interactions of Mesa signals with processes and monitors (see Sections 3.1 and 3.4). In particular, the reasonable and consistent handling of signals (including UNWINDS) in entry procedures represents a considerable increase in the mental

overhead involved in designing a new monitor or understanding an existing one.

6.2 Violet: A Distributed Calendar System

The Violet system [6, 7] is a distributed database manager which supports replicated data files, and provides a display interface to a distributed calendar system. It is constructed according to the hierarchy of abstractions in Figure 2. Each level builds on the next lower one by calling procedures supplied by it. In addition, two of the levels explicitly deal with more than one process. Of course, as any level with multiple processes calls lower levels, it is possible for multiple processes to be executing procedures in those levels as well.

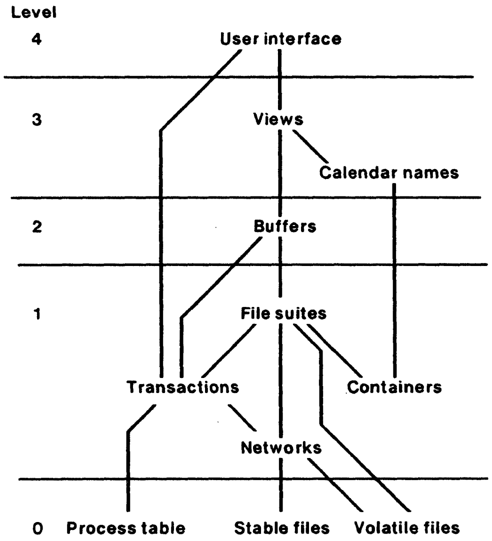
The user interface level has three processes: *Display, Keyboard,* and *DataChanges.* The *Display* process is responsible for keeping the display of the database consistent with the views specified by the user and with changes occurring in the database itself. It is notified by the other processes when changes occur, and calls on lower levels to read information for updating the display. *Display* never calls update operations in any lower level. The other two processes respond to changes initiated either by the user (*Keyboard*) or by the database (*DataChanges*). The latter process is FORKed from the *Transactions* module when data being looked at by Violet changes, and disappears when it has reported the changes to *Display.*

A more complex constellation of processes exists in *FileSuites,* which constructs a single *replicated file* from a set of *representative* files, each containing data from some version of the replicated file. The representatives are stored in a transactional file system [11], so that each one is updated atomically, and each carries a version number. For each *FileSuite* being accessed, there is a monitor which keeps track of the known representatives and their version numbers. The replicated file is considered to be updated when all the representatives in a *write quorum* have been updated; the latest version can be found by examining a *read quorum.* Provided the sum of the read quorum and the write quorum is as large as the total set of representatives, the replicated file behaves like a conventional file.

When the file suite is created, it FORKS and detaches an *inquiry* process for each representative. This process tries to read the representative's version number, and if successful, reports the number to the monitor associated with the file suite and notifies the condition *CrowdLarger.* Any process trying to read from the suite must collect a read quorum. If there are not enough representatives present yet, it waits on *CrowdLarger.* The inquiry processes expire after their work is done.

When the client wants to update the *FileSuite,* he must collect a write quorum of representatives containing the current version, again waiting on *CrowdLarger* if one is not yet present. He then FORKS an *update* process for each representative in the quorum, and each tries to write its file. After FORKing the update processes, the client JOINS each one in turn, and hence does not proceed until all have completed. Because all processes run within the same transaction, the underlying transactional file system guarantees that either all the representatives in the quorum will be written, or none of them.

Fig. 2. The internal structure of Violet.



It is possible that a write quorum is not currently accessible, but a read quorum is. In this case the writing client FORKS a *copy* process for each representative which is accessible but is not up to date. This process copies the current file suite contents (obtained from the read quorum) into the representative, which is now eligible to join the write quorum.

Thus as many as three processes may be created for each representative in each replicated file. In the normal situation when the state of enough representatives is known, however, all these processes have done their work and vanished; only one monitor call is required to collect a quorum. This potentially complex structure is held together by a single monitor containing an array of representative states and a single condition variable.

6.3 Gateway: An Internetwork Forwarder

Another substantial application program which has been implemented in Mesa using the process and monitor facilities is an internetwork gateway for packet networks [2]. The gateway is attached to two or more networks and serves as the connection point between them, passing packets across network boundaries as required. To perform this task efficiently requires rather heavy use of concurrency.

At the lowest level, the gateway contains a set of device drivers, one per device, typically consisting of a high priority interrupt process, and a monitor for synchronizing with the device and with noninterrupt level software. Aside from the drivers for standard devices (disk, keyboard, etc.) a gateway contains two or more drivers for Ethernet local broadcast networks [16] and/

or common-carrier lines. Each Ethernet driver has two processes, an interrupt process, and a background process for autonomous handling of timeouts and other infrequent events. The driver for common-carrier lines is similar, but has a third process which makes a collection of lines resemble a single Ethernet by iteratively simulating a broadcast. The other network drivers have much the same structure; all drivers provide the same standard network interface to higher level software.

The next level of software provides packet routing and dispatching functions. The *dispatcher* consists of a monitor and a dedicated process. The monitor synchronizes interactions between the drivers and the dispatcher process. The dispatcher process is normally waiting for the completion of a packet transfer (input or output); when one occurs, the interrupt process handles the interrupt, notifies the dispatcher, and immediately returns to await the next interrupt. For example, on input the interrupt process notifies the dispatcher, which dispatches the newly arrived packet to the appropriate *socket* for further processing by invoking a procedure associated with the socket.

The *router* contains a monitor which keeps a *routing* table mapping network names to addresses of other gateway machines. This defines the next "hop" in the path to each accessible remote network. The router also contains a dedicated housekeeping process which maintains the table by exchanging special packets with other gateways. A packet is transmitted rather differently than it is received. The process wishing to transmit to a remote socket calls into the router monitor to consult the routing table, and then the same process calls directly into the appropriate network driver monitor to initiate the output operation. Such asymmetry between input and output is particularly characteristic of packet communication, but is also typical of much other I/O software.

The primary operation of the gateway is now easy to describe: When the arrival of a packet has been processed up through the level of the dispatcher, and it is discovered that the packet is addressed to a remote socket, the dispatcher forwards it by doing a normal transmission; i.e., consulting the routing table and calling back down to the driver to initiate output. Thus, although the gateway contains a substantial number of asynchronous processes, the most critical path (forwarding a message) involves only a single switch between a pair of processes.

Conclusion

The integration of processes and monitors into the Mesa language was a somewhat more substantial task than one might have anticipated, given the flexibility of Mesa's control structures and the amount of published work on monitors. This was largely due to the fact that Mesa is designed for the construction of large, serious programs, and that processes and monitors had to be refined sufficiently to fit into this context. The task has been accomplished, however, yielding a set of language features of sufficient power that they serve as the only software concurrency mechanism on our personal computer, handling situations ranging from input/output interrupts to cooperative resource sharing among unrelated application programs.

Received June 1979; accepted September 1979; revised November 1979

References

1. *American National Standard Programming Language PL/I.* X3.53, American Nat. Standards Inst., New York, 1976.
2. Boggs, D.R., et. al. Pup: An internetwork architecture. *IEEE Trans. on Communications 28,* 4 (April 1980).
3. Brinch Hansen, P. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, New Jersey, July 1973.
4. Brinch Hansen, P. The programming language Concurrent Pascal. *IEEE Trans. on Software Eng. 1,* 2 (June 1975), 199–207.
5. Dijkstra, E.W. Hierarchical ordering of sequential processes. In *Operating Systems Techniques,* Academic Press, New York, 1972.
6. Gifford, D.K. Weighted voting for replicated data. *Operating Systs. Rev. 13,* 5 (Dec. 1979), 150–162.
7. Gifford, D.K. Violet, an experimental decentralized system. Integrated Office Syst. Workshop, IRIA, Rocquencourt, France, Nov. 1979 (also available as CSL Rep. 79-12, Xerox Res. Ctr., Palo Alto, Calif.).
8. Hoare, C.A.R. Monitors: An operating system structuring concept. *Comm. ACM 17,* 10 (Oct. 1974), 549–557.
9. Hoare, C.A.R. Communicating sequential processes. *Comm. ACM 21,* 8 (Aug. 1978), 666–677.
10. Howard, J.H. Signaling in monitors. Second Int. Conf. on Software Eng., San Francisco, Calif., Oct. 1976, pp. 47–52.
11. Israel, J.E., Mitchell, J.G., and Sturgis, H.E. Separating data from function in a distributed file system. Second Int. Symp. on Operating Systs., IRIA, Rocquencourt, France, Oct. 1978.
12. Keedy, J.J. On structuring operating systems with monitors. *Australian Comptr. J. 10,* 1 (Feb. 1978), 23–27 (reprinted in *Operating Systs. Rev. 13,* 1 (Jan. 1979), 5–9).
13. Lampson, B.W., Mitchell, J.G., and Satterthwaite, E.H. On the transfer of control between contexts. In *Lecture Notes in Computer Science 19,* Springer-Verlag, New York, 1974, pp. 181–203.
14. Lauer, H.E., and Needham, R.M. On the duality of operating system structures. Second Int. Symp. on Operating Systems, IRIA, Rocquencourt, France, Oct. 1978 (reprinted in *Operating Systs. Rev. 13,* 2 (April 1979), 3–19).
15. Lister, A.M., and Maynard, K.J. An implementation of monitors. *Software—Practice and Experience 6,* 3 (July 1976), 377–386.
16. Metcalfe, R.M., and Boggs, D.G. Ethernet: Packet switching for local computer networks. *Comm. ACM 19,* 7 (July 1976), 395–403.
17. Mitchell, J.G., Maybury, W., and Sweet, R. Mesa Language Manual. Xerox Res. Ctr., Palo Alto, Calif., 1979.
18. Redell, D., et. al. Pilot: An operating system for a personal computer. *Comm. ACM 23,* 2 (Feb. 1980).
19. Saltzer, J.H. Traffic control in a multiplexed computer system. Th., MAC-TR-30, MIT, Cambridge, Mass., July 1966.
20. Saxena, A.R., and Bredt, T.H. A structured specification of a hierarchical operating system. SIGPLAN Notices *10,* 6 (June 1975), 310–318.
21. Wirth, N. Modula: A language for modular multiprogramming. *Software—Practice and Experience 7,* 1 (Jan. 1977), 3–36.

Traits : An Approach to Multiple-Inheritance Subclassing

Gael Curry, Larry Baer, Daniel Lipkie, Bruce Lee
Xerox Corporation, El Segundo, California

Abstract : This paper describes a new technique for organizing software which has been used successfully by the Xerox Star 8010 workstation. The workstation (WS) software is written in an "object-oriented" style : it can be viewed as a system of inter-communicating objects of different *object types*. Most of the WS software considers object types to be constructed by assembling more primitive abstractions called *traits*. A trait is a characteristic of an object, and is expressed as a set of operations which may be applied to objects carrying that trait. The *traits model of subclassing* generalizes the SIMULA-67 model by permitting multiple inheritance paths. This paper describes the relationship of WS software to the traits model and then describes the model itself.

Star Workstation Software and the Traits Model

History : Star WS software has been committed to an object-oriented coding style (discussed shortly) in the Mesa programming language[Mitchell 79] since actual development first started in the spring of 1978 [Harslem 82]. Initial designs did not rely on subclassing. This was partly because the designers had had little experience with it (authors included), and partly because an extensible design based on subclassing seemed to necessitate a violation of Mesa's type system. An early Star text editor was built without the benefit of subclassing. It gradually became clear that significant code-sharing was possible if the design were based on subclassing, since the objects we were dealing with were more similar than different.

By late 1978, we had re-implemented that editor in terms of SIMULA-67-style subclassing, where object types were considered to form a *tree* under the specialization relation. The subclassing was represented as coding conventions in Mesa. That was a great help, particularly the analogue of SIMULA-67 VIRTUAL procedures (which permitted operations to be specified at more abstract levels and interpreted at more concrete ones). Use of this subclassing style extended into other areas of WS software, especially support for property sheets and open icon windows [Smith 82], [Seybold 81]; Star graphics and tables were initially designed in these terms also. As the class hierarchy grew, we began to notice that the constraint of *pure-tree* class hierarchies was causing code to become contorted, and that generalizing the concept of "class hierarchy" to include *directed acyclic graphs* would allow code to be organized more cleanly.

A new subclassing model was defined along those lines. It postulated that object types were constructed from more primitive abstractions, *traits*, corresponding roughly to SIMULA-67 classes. The major difference was that a given trait may be defined in terms of *several* more primitive ones, rather than just a single one. Supporting software - the "Traits Mechanism" - was implemented in late 1979. Star graphics [Lipkie 82] was the first major piece of Star software *designed* in terms of traits and using the full generality of the model. Other areas, especially Star folders and record files, began using the generality permitted by traits heavily.

The Traits Mechanism : A major design goal was to make the new mechanism as efficient as the old coding pattern for the case of static, tree-structured class hierarchies. We found a way to do this with a particular global optimization (outside the scope of this paper), but it required that a central facility, or *trait manager*, collect information about all extant traits. This trait manager collects information from each trait in the system regarding its storage requirements, arranges that trait's storage (in objects) for optimum access, and mediates access to it upon the individual trait's demand. Client code (code calling the trait mechanism) adopts a Mesa coding pattern to use trait-style subclassing.

Another important property of the traits mechanism is that the cost of accessing a trait's data in an object, or the implementation of an operation that the trait introduces, is not a function of the position of the trait in the class hierarchy. There is no run-time searching.

Star today : Star software has been using the Traits Model of subclassing since 1979 with good results. Star-1 was completed in October, 1981. It defined 169 traits. Of those, 129 were *object types*, or *class traits*; i.e., 40 were purely internal abstractions. In general, each trait requires some storage in objects which carry it; 99 were of this sort. Also, each trait introduces some number of operations which can be applied to objects which carry it. While not all of these operations may be "VIRTUAL", 31 traits in Star-1 introduce this kind of operation.

The Traits Model

Object Orientation : Object-orientation is a method for organizing software where, at any time, computation is performed under the aegis of a particular object. Part of the computation may include transferring control and information to another object (message-passing), which then continues the computation; control and other information may be returned to the first subsequently. An object's state is typically represented as some sort of storage; each object has a name. A restricted form of message-passing is typically represented by procedure call, where a distinguished parameter of each procedure is the name of the object which is to continue the computation. Objects' state may be represented as records, pointers to records, names of records, implicit records, or in any number of other ways.

Subclassing : SIMULA-67 noted that often an object is a *specialization* of another, being able to to the job of the first - and more. It provided a means of expressing the common portion once, in order that the specialized object need only specify the way in which it was different from the simpler one. The specialized object *inherited* the properties of the simpler one.

The Traits model notes that an object (type) may be a synthesis of several component abstractions, being able to do the job of its components and more. It provides a means of expressing the common, or shared, parts once.

In both cases advantages come from sharing : clarity of code through factoring or abstraction; uniformity of behavior, including correctness; ease of maintenance; reduced swapping. Another important property for large systems, which both models possess, is extensibility: the addition of a new class or trait does not invalidate existing code.

Instances : There is a wide range of interpretations for the term "object". In order to avoid problems of language, we will use the term *instance* to refer to any of the objects in our universe of discourse. This is left intentionally vague.

Instances have *state*, which allows them to remember information. They also have *names*, or *handles*. Often an instance will remember the names of other useful instances.

Operations : An *operation* is a means of presenting information to and/or extracting information from an instance. Every instance possesses an identifiable set of operations, called its *operation set*. An operation is *applied* to an instance, perhaps presenting some information to the instance (in a well-defined format) and perhaps receiving some information from it (also in a well-defined format) in return. Applying an operation to an instance changes the state of the instance, in general.

Each operation has a *specification* and a *realization*; the realization *meets* the specification. The range of specifications in actual practice extends from strictly functional input/output specifications, to those including some behavioral clauses (operational specifications), to those including contextual clauses (behavior varies with context), to that which is simply "it works when you plug it in". Two operations are *equal* if they have the same specification and realization. They are *equivalent* if they have the same specification; one operation is a *variant* of another if they are equivalent.

Types : Many times instances will have the same operation set, being different only in their internal state and in their identity. The universe of instances can be partitioned into equivalence classes, based on having the same operation set (that is, two instances are in some sense equivalent if they have the same operation set). These equivalence classes are *types*. The *operation set of a type* is also well-defined.

This view says that two instances have different type if their operation sets are different, however minor the difference. While that is correct, it also ignores a lot of information about exactly *how* those operation sets are different.

Type Structure : There are many ways in which the operation sets for types can be related to those of other types.

- UNRELATED - The operation sets for all types can be totally different, so that we see no interesting type structure. This situation is supported well by programming conventions which devote one "module" to each type, and implement operations for the type's operation set within that module.

- VARIATION - It may be that there is a one-to-one correspondence between operations of one type and those of another, and that each operation is equivalent to its corresponding one. Then, each type is a *variant* of the other. This situation is supported well by the programming style which accords each object one procedure variable for each operation; realizations for each operation are recorded in the corresponding procedure variable. Streams are sometimes implemented this way.

- EXTENSION - It may be that all of the operations of one type are equal to operations of another type, but that the latter type has extra operations. Then, the latter type is an *extension* of the former. This situation is supported well by simple inheritance mechanisms.

- SPECIALIZATION - It may be that one type's operation set can be gotten from another 's by variation, perhaps followed by extension. Then, the former type is a *specialization* of the latter. This situation is supported well by SIMULA-67 and Smalltalk-80.

In the cases above, it was possible to see how the operation sets could be derived from the operation sets of other types. In the cases below, type structure is derived from units which are more basic than other types.

- UNIONS - It may be that of three types A, B and C :
  A has operations $O_1 \cup O_2$
  B has operations $O_2 \cup O_3$
  C has operations $O_1 \cup O_3$,

  where O_1, O_2 and O_3 are sets of operations. This is a somewhat contrived case. The same sort of thing happens naturally on a larger scale (indeed, perhaps *only* on a larger scale). Being minimal, the example shows more clearly what is going on. In this case, no type is a specialization of another, yet there is clearly an interesting type structure.

  Notice that the operation sets are not naturally derivable from the operation sets of other *types*, but rather from lower-level *operation subsets*. These operation subsets represent a characterization of some aspect of an instance's behavior in terms of a set of operations. The pattern arises whenever an instance has *several independent aspects*.

  A *trait* is a characterization of an aspect of an instance's behavior. The primary representation of the trait may be an natural language description of that aspect or may be some individual's intent for or understanding of that aspect. The characterization is represented by a set of specifications for operations which, considered together, embody that aspect. A set of operations with those specifications is called an *operation set* for the trait.

- SYNTHESIS - It may be that one type's operation set can be gotten from operation sets of several other *component traits* by variation of the operations in the trait operation sets, followed by union of the results, perhaps followed by extension. Then, the type is a *synthesis* of the component traits.

This is the basic operation adopted by the Traits approach.

- RESTRICTED SYNTHESIS - It may be that one type's operation set can be gotten from those of several other *component traits* by synthesis, followed by discarding some of the resulting operations. This is not well handled by the current Traits mechanism; it can be simulated by re-defining an undesired component trait's operation to have nil realization (Note that the operation may then not meet its specification).

The discussion above has been *analytical*. It assumed instances, operations and types already existed; it tried to dissect the situation. The ensuing discussion is *constructive*. It tries to develop a view of traits as basic design units, in order to show how to incrementally build a system of trait-based instances.

Traits : A *trait* is a characterization of an aspect of an instance's behavior. It is expressed as a set of operations. Some examples of traits are:

- IS-FORWARD-LINKED-LIST-ELEMENT - This represents the notion that an instance carrying this trait will be linked with other instances in some forward-linked list. It specifies operations

  ```
  GetLink :
      PROC [ instance : Instance ]
      RETURNS [ Instance ], and
  SetLink :
      PROC [instance : Instance ,
          instanceLink : Instance ],
  ```

  whose semantics are obvious.

- IS-TREE-ELEMENT - Represents the notion that an instance carrying this trait will be embedded in a tree of instances. It specifies operations

  ```
  GetParent :
      PROC [ instance : Instance ]
      RETURNS [ Instance ],
  SetParent :
      PROC [instance : Instance ,
          instanceParent : Instance ],
  GetNextSibling :
      PROC [ instance : Instance ]
      RETURNS [ Instance ],
  SetNextSibling :
      PROC [instance : Instance ,
          instanceNextSibling : Instance ],
  GetEldestChild :
      PROC [ instance : Instance ]
      RETURNS [ Instance ],
  SetEldestChild :
      PROC [instance : Instance ,
          instanceEldestChild : Instance ],
  ```

  whose semantics are also obvious.

- IS-NAMED - An instance carrying this trait has a textual name. It specifies operations

  ```
  GetName :
      PROC [ instance : Instance ]
      RETURNS [ Name ], and
  SetName :
      PROC [ instance : Instance , name : Name ],
  ```

  whose semantics are obvious.

- IS-PRINTABLE - This represents the notion that the instance can print itself. It specifies the operation

  ```
  Print : PROC [ instance : Instance , printer : Printer ],
  ```

  which causes the instance to emit an image level representation of itself to a printer.

Note that the trait does not include realizations for the various specifications.

Simple Traits : The traits listed above are *simple traits*. A simple trait is completely defined by specifying the operations which characterize it. Figure 1 depicts a simple trait graphically.

T: IS-TREE-ELEMENT

Operation Name	*Specification*
GetParent	$S_{GetParent}$
SetParent	$S_{SetParent}$
GetNextSibling	$S_{GetNextSibling}$
SetNextSibling	$S_{SetNextSibling}$
GetEldestChild	$S_{GetEldestChild}$
SetEldestChild	$S_{SetEldestChild}$

Figure 1. Definition of Simple Trait T

Compound Traits : Sometimes a trait will be best expressed as the "sum" of other traits. For example, the trait

```
IS-IN-NAME-HIERARCHY =
    IS-TREE-ELEMENT ∪ IS-NAMED.
```

specifies operations for an element of a named instance hierarchy. The operations specified by IS-IN-NAME-HIERARCHY might be the union of the operations specified by IS-TREE-ELEMENT and IS-NAMED individually. An instance having that trait would know that it was part of an instance hierarchy, and would know it was named. It may or may not know the same for its subordinates in that hierarchy.

In any case, it might be meaningful to augment that trait's operation set with something like

```
Search:
    PROCEDURE [ instance : Instance, name : Name]
    RETURNS [ Instance ],
```

which would return the name of the subordinate having the indicated name, if there was one, and instanceNil otherwise.

We could define a new trait, IS-SEARCHABLE, which specifies the Search operation as its sole operation - in order to define the compound trait

```
IS-TREE-ELEMENT ∪ IS-NAMED ∪ IS-SEARCHABLE,
```

but it seems more straightforward to associate it directly with the compound trait , as in

```
IS-IN-NAME-HIERARCHY =
    IS-TREE-ELEMENT ∪ IS-NAMED ∪ { Search }.
```

The latter demonstrates the *compounding method* for trait definitions. Figure 2 illustrates the compounding graphically.

The "Carries" Relation : A trait *directly carries* another trait if it is defined in terms of that trait. So, for example, IS-IN-NAME-HIERARCHY carries IS-NAMED directly. "*carries*" is the reflexive transitive closure of "directly carries", and we assume it is acyclic.

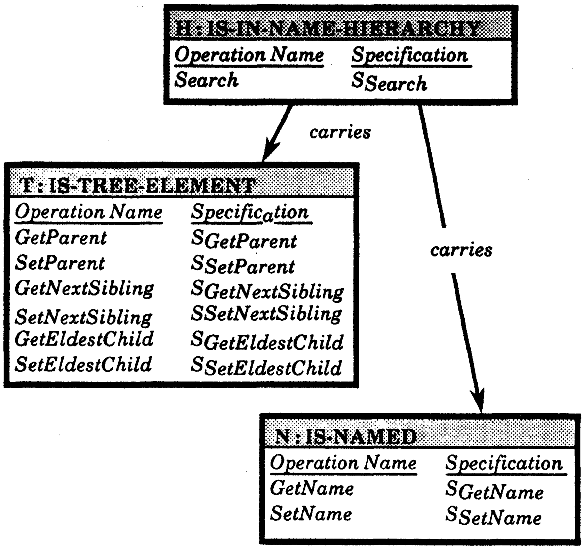


Figure 2. Definition of Compound Trait H

The Traits Graph : The collection of all traits used in a system of instances are inter-related, and form a directed acyclic graph under the "carries" relation. Nodes in the graph represent traits. Arcs represent the "carries" relation. Associated with each node in the traits graph are the specifications for operations *introduced* at that level. For simple traits, that means all of its operations. For compound traits, that means operations over and above those of the component traits. Figure 3 shows a possible trait graph.

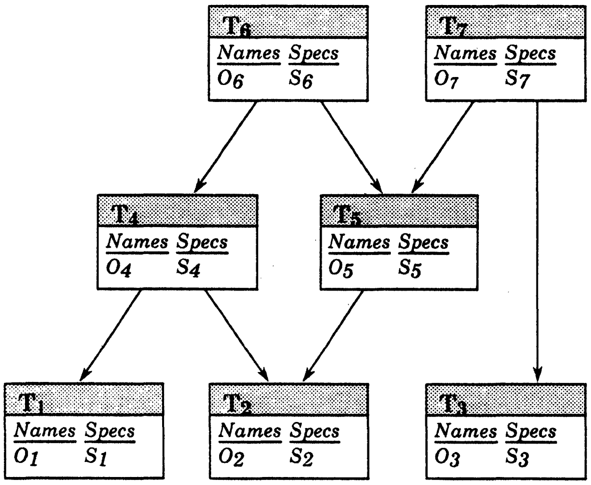


Figure 3. A Possible Traits Graph

Realizations for Trait Operations : Every trait determines a set of "carried" traits (i.e., those that it dominates in the traits graph). A trait may recommend or provide optional realizations for each of its operations - including operations introduced by the traits it carries. Figure 4 shows trait T_6 carrying traits T_1, T_2, T_4, T_5, and T_6 (itself) by displaying them in bold. The boxes adjoining each of those traits represent trait T_6's choices of realizations for the operations of its carried traits. The notation $R_{T_i}[T_j]$ means T_j's choices for the realizations for the operations introduced by trait T_i.

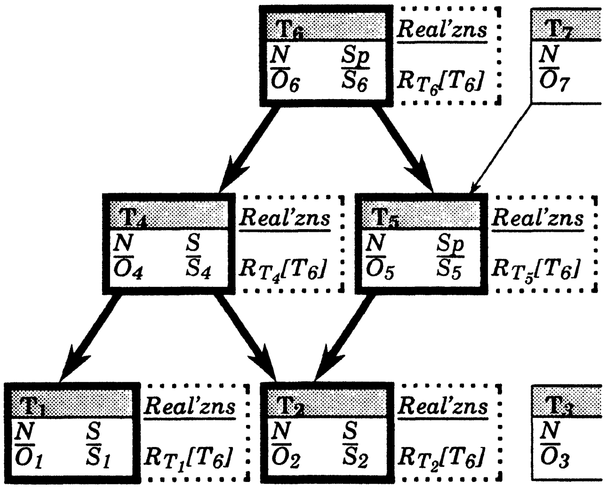


Figure 4. Realizations for Operations of Carried Traits

Default Realizations : A trait always assigns a *default realization* to each of the operations it introduces. The default realization may be the nil realization.

Optional Realizations : A trait sometimes makes *optional realizations* for its operations available. For any of a trait's operations, a trait may designate a pool of realizations from which other traits may choose their default realizations. This helps to maximize sharing. The default realization for an operation should be viewed as a distinguished member of the set of optional realizations for that operation. Figure 5 shows a closeup of the realizations for a particular operation o of a carried trait. The notation $r_o[T]$

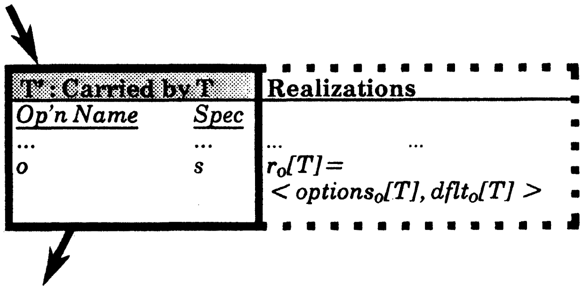


Figure 5. $r_o[T]$ - Realizations for o in T

denotes T's choices for realizations of o. $r_o[T]$ is a 2-tuple. The first element is a set of optional realizations for o from the trait T's point of view; they must all meet the specification s. The second element is a singleton or empty set of realizations from $\mathit{options}_o[T]$ which are T's choices for what it considers to be the default realization for o. All of the operations in $r_o[T']$ must in some sense be *defined below the level of T.*

Inheritance of Realizations for Operations : In principle, each trait in the trait graph for a system is solely responsible for determining the realizations for the operations of all of the traits it carries. In practice we find that most of a trait's choices for operations of a carried trait are exactly the choices of the traits that it directly carries for those operations. For this reason, realizations for a trait's operations are defined initially by *inheritance.*

Pure Inheritance : That is, unless the trait declares otherwise, its assignment of realizations to the operations of carried traits will be the union of assignments made by the traits it immediately carries. If those choices do not suit the trait, it must be able to *override* those assignments. The trait always has opportunity to define optional realizations for the operations that it itself introduces; it has the responsibility for defining default realizations for those operations if it can.

Suppose T is a trait in some trait graph, and that it carries a trait S which introduces an operation o. Suppose that S is carried by immediate sub-traits T_i ... T_k of T. Then we have :

$$r_o[T_j] = \; < \mathit{options}_o[T_j], \mathit{default}_o[T_j] >, \mathit{for}\ j = i, \ldots, k.$$

The trait T initially views its realizations for the operation o as consisting of the union of the realizations as seen by each of the immediate sub-traits, and is potentially confused about the default realization:

$$\mathit{inherited\text{-}r}_o[T] = \; < \mathit{inherited\text{-}options}_o[T], \mathit{inherited\text{-}default}_o[T] >,$$

where

$$\mathit{inherited\text{-}options}_o[T] = \mathit{options}_o[T_i] \cup \ldots \cup \mathit{options}_o[T_k], \mathit{and}$$

$$\mathit{inherited\text{-}default}_o[T] = \mathit{default}_o[T_i] \cup \ldots \cup \mathit{default}_o[T_k].$$

The difficulty is clear - traits T_j and $T_{j'}$ can specify different default realizations for an operation of a shared sub-trait, so that *pure inheritance* does not guarantee well-defined default realizations for operations.

Consistent Inheritance and Conflict Resolution : If o is an operation introduced by trait S carried by trait T, the realizations for o are *consistently inherited at T* iff *inherited-default$_o$[T]* is a singleton or null. If this is not the case, then the trait T *must resolve the inconsistency* by explicitly designating some realization as the default. It is a *design error* for T not to do so.

Qualified Inheritance : Normally, a trait *need not* explicitly designate realizations for any but its own operations (except to resolve occurrences of inconsistent inheritance). However, it is the trait's *prerogative* to modify its realizations for any operation introduced by a trait it carries. This includes changing the default, or modifying the set of optional realizations.

Traits, Class Traits and Instances : There is a set of operation names associated with every trait T in the trait graph for a system. Those include names for operations introduced by the trait itself, as well as names for operations introduced by carried traits. Specifications exist for each of the names. Associated with each of those names is also a default realization (actually, some might be nil, but ignore that problem for now). The operation set for a trait T in a trait graph is the set of operations

{ o : < s, *default$_o$[T]* >,
where o is an operation introduced by a trait carried by T, s is its specification }.

It might be nice to have instances extant in the system with the same operation set as certain traits.

> In some cases, it doesn't seem to make much sense. It doesn't seem like it would be very useful to have instances whose operation set is the same as that of the simple trait IS-FORWARD-LINKED-LIST-ELEMENT; those instances would be pretty uninteresting.

Any trait having such an interesting operation set can be designated a *class trait,* and instances having the same operation set can be generated. The instance is tagged with the name of its class trait; that is the *instance's type.* The instance *carries* the same traits as its class trait.

Specifications for Trait Operations : Any operation for a trait T should have well-defined semantics. The meaning of an operation should be specified as clearly as possible when the trait is defined. That specification is the invariant part of the trait; it does not change (as do realizations for the operation) depending on which other trait is carrying T.

The specification for a trait's operation should be *in terms of the instance carrying the trait.* The client of a trait must have a clear idea of the meaning of an operation's semantics independent of its carrier.

Denoting Applications of Trait Operations : The *application of an operation to an instance* is often denoted as

<results> ← instance.operation[<parameters>].

The denotation is non-committal regarding the trait (subclass) to which the operation belongs. That has the advantage that the denotation need not change if the operation migrates from one trait to another during the course of system development. It has the disadvantage that it presents the instance as having a rather unstructured "pile" of operations, which may make the nature of the instance harder to understand. If structure in the set of operations applicable to an instance can be clearly seen, perhaps it should be expressed, as in

<results> ←
instance.operation$_{\text{trait}}$[<parameters>],

where "operation" is introduced by "trait".

Another possible form is:

<results> ←
trait.operation[instance, <parameters>].

In both of the cases above where the trait is mentioned, it is assumed that the instance carries the trait introducing the operation. The last form is well suited for use in a module-oriented language, where each trait can be represented by a single module.

Expressing Realizations for Trait Operations : It is important to find a way to express realizations of a trait's operations in a way which is independent of context (i.e., who carries that trait), so that realizations for a trait need be implemented when the trait is defined (as opposed to only when it is carried).

A realization is expressed in terms of "code" to be invoked over a particular instance I. That code may express the application of an operation of a carried trait to the same instance I. It may also involve applying operations to another instance I' of which I has knowledge (remembers, or was just told about). It may also involve changing the state of the instance somehow.

The code may also involve computations over other "objects" which happen not to be *instances* in the system in question. For example, it may involve numeric computations. While in principle "number objects" might be instances, performance considerations might recommend against it. All that is required is that the code be able to compute locally, invoke operations over instances, incorporate the results of such invocations, and change appropriate parts of the state of the instance upon which the code is operating.

Instance State vs. Trait State / Trait Data : Suppose T is a simple trait which introduces operation o with specification s, and assigns as its default realization r. Suppose i is an instance carrying T. If the specification s indicates that applying o to i will change the state of i, then it is important to ask how the realization r accesses the state it needs to change.

The problem is addressed in the Traits model by asserting that every trait carried by an instance has its own state, or storage, within the larger state of the instance itself. We go so far as to say that the state space of an instance is the product of the state spaces of the traits that it carries. Figure 6 expresses that idea graphically. Furthermore, only realizations defined by the trait can access or modify that trait storage directly. The internal format for a trait's storage is completely up to the trait itself.

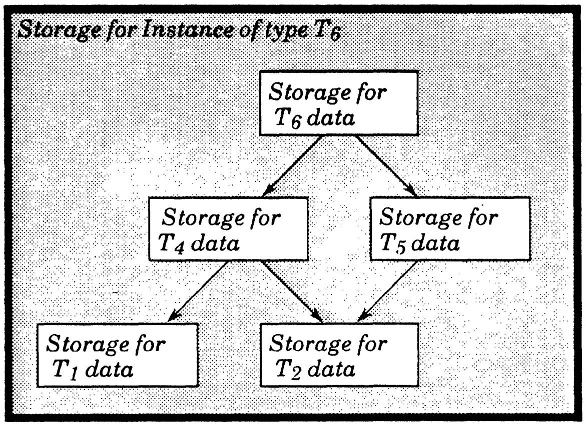


Figure 6. Instance Storage is the Sum of Trait Storage

We will say nothing about the location of storage for a particular trait in instance storage. All that is important is that a realization defined to act directly on the storage for a particular trait must be able to gain access to that storage. For this purpose (and others) there is a *trait manager*, who knows how to access the storage for any particular trait, given the instance's name and the trait's name.

Instance Initialization : When an instance is generated, storage is obtained from somewhere. Embedded in that storage is storage for the individual traits carried by that instance. After the storage is allocated, individual traits are told to initialize their storage. Carried traits initialize their storage before carrying traits. In the example in Figure 6, trait T_1 would be told to initialize its storage before trait T_4 was so instructed, which would be done before trait T_6

was so instructed. The bottom-up order of trait initialization permits carrying traits to invoke carried traits operations during their own initialization.

Classes : Instances may be generated for class traits. If T is a class trait, then it needs to record its choices for realizations for all of the operations it carries. The Traits model postulates a *class* (object) for each class trait. Associated with this class is storage which records the choice of realizations. For brevity, the operation set of the class trait is called the *behavior* of the class.

Every trait which is carried by the class introduces some number of operations whose realizations can be assigned by the class. Associated with each trait T carried by a class trait T_c is enough storage to record the class trait's realizations for the operations of T. Figure 7 depicts that situation.

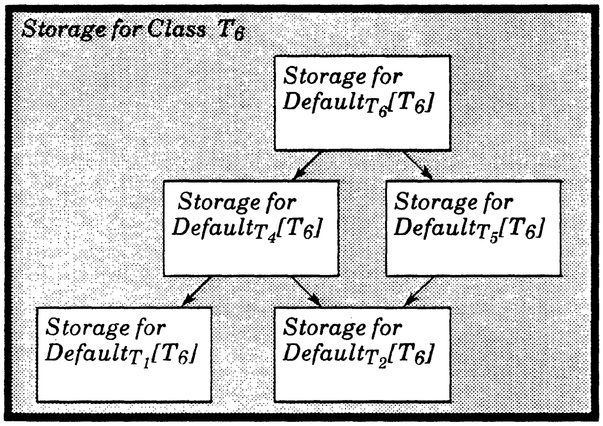

Storage for Class T_6
Storage for $Default_{T_6}[T6]$
Storage for $Default_{T_4}[T6]$
Storage for $Default_{T_5}[T6]$
Storage for $Default_{T_1}[T6]$
Storage for $Default_{T_2}[T6]$

Figure 7. Class Storage Records Realizations for Trait Operations

Again, we will say nothing about the location of storage for a particular trait in class storage. All that is important is that at the time a trait operation is invoked, the realization for that operation can be found. The trait manager knows how to access a particular trait's (realizations) storage, given the name of the instance and the name of the trait.

Class Initialization : Initialization of a class is a bottom-up enumeration of that part of the traits graph dominated by the class' trait. Each trait enumerated should override any default realizations of the traits it carries and should establish its own default realizations. In order to do so, it must be able to obtain access to its component of class storage.

Instantiation : The class (object) is generally viewed as the agent which generates, or *instantiates*, instances. There may be many instances associated with a particular class, but the storage for recording the class trait's choices for default realizations is allocated only once.

Conclusions

Multiple-inheritance subclassing is a valid and useful method for organizing object-oriented software; as demonstrated by the existence of the Star Workstation. The complexity of the Star WS software has been controlled by object-orientation first, subclassing second and multiple-inheritance third.

The Traits Model is a reasonable approach to multiple-inheritance subclassing. It is possible to implement efficient supporting mechanisms, especially for statically specified class structures. The Traits mechanism is optimal for pure-tree class structures, and deep class structures cost nothing extra at run-time.

Acknowledgements : Derry Kabcenell and Tim Rentsch made useful comments during early reviews of the proposed Traits model. Eric Harslem allowed us to apply this unproven software technique to a large and important piece of software - successfully. Dan Ingalls, Alan Borning, and Dave Gifford all later noted the similarities between the traits approach and the flavors approach [Weinreb 81] of the MIT LISP machine and helped to articulate the differences. The Xerox PARC Methodology Discussion Group made plenty of interesting observations.

REFERENCES

[Harslem 82] E. Harslem and L.E. Nelson, "A Retrospective on the Development of Star," to be published in the proceedings of the *6th International Conference on Software Engineering*; Tokyo, Japan; Sept, 1982.

[Lipkie 82] Daniel Lipkie, Steven R. Evans, Robert Weissman, John K. Newlin, "Star Graphics : An Object Oriented Implementation," to be published in the proceedings of SIGGRAPH 1982.

[Mitchell 78] J.G. Mitchell, W. Maybury, and R.E. Sweet, "Mesa Language Manual," Technical report CSL-79-3, Xerox Corporation, Palo Alto Research Center, Palo Alto, California; April 1979.

[Weinreb 81] Daniel Weinreb, David Moon, LISP Machine Manual, Third Edition, March, 1981.

[Seybold 81] Seybold Report, "Xerox's Star," Volume 10, Number 16; April 27, 1981.

[Smith 82] D.C. Smith, E. Harslem, C. Irby, R. Kimball, "The Star User Interface, an Overview," to be published in the proceedings of *NCC '82*.

Pilot: An Operating System for a Personal Computer

David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch,
Paul R. McJones, Hal G. Murray, and Stephen C. Purcell
Xerox Business Systems

The Pilot operating system provides a single-user, single-language environment for higher level software on a powerful personal computer. Its features include virtual memory, a large "flat" file system, streams, network communication facilities, and concurrent programming support. Pilot thus provides rather more powerful facilities than are normally associated with personal computers. The exact facilities provided display interesting similarities to and differences from corresponding facilities provided in large multi-user systems. Pilot is implemented entirely in Mesa, a high-level system programming language. The modularization of the implementation displays some interesting aspects in terms of both the static structure and dynamic interactions of the various components.

Key Words and Phrases: personal computer, operating system, high-level language, virtual memory, file, process, network, modular programming, system structure

CR Categories: 4.32, 4.35, 4.42, 6.20

1. Introduction

As digital hardware becomes less expensive, more resources can be devoted to providing a very high grade of interactive service to computer users. One important expression of this trend is the personal computer. The dedication of a substantial computer to each individual user suggests an operating system design emphasizing close user/system cooperation, allowing full exploitation of a resource-rich environment. Such a system can also function as its user's representative in a larger community of autonomous personal computers and other information resources, but tends to deemphasize the largely ajudicatory role of a monolithic time-sharing system.

The Pilot operating system is designed for the personal computing environment. It provides a basic set of services within which higher level programs can more easily serve the user and/or communicate with other programs on other machines. Pilot omits certain functions that have been integrated into some other operating systems, such as character-string naming and user-command interpretation; such facilities are provided by higher level software, as needed. On the other hand, Pilot provides a more complete set of services than is normally associated with the "kernel" or "nucleus" of an operating system. Pilot is closely coupled to the Mesa programming langauge [16] and runs on a rather powerful personal computer, which would have been thought sufficient to support a substantial time-sharing system of a few years ago. The primary user interface is a high resolution bit-map display, with a keyboard and a pointing device. Secondary storage is provided by a sizable moving-arm disk. A local packet network provides a high bandwidth connection to other personal computers and to server systems offering such remote services as printing and shared file storage.

Much of the design of Pilot stems from an initial set of assumptions and goals rather different from those underlying most time-sharing systems. Pilot is a single-language, single-user system, with only limited features for protection and resource allocation. Pilot's protection mechanisms are *defensive*, rather than *absolute* [9], since in a single-user system, errors are a more serious problem than maliciousness. All protection in Pilot ultimately depends on the type-checking provided by Mesa, which is extremely reliable but by no means impenetrable. We have chosen to ignore such problems as "Trojan Horse" programs [20], not because they are unimportant, but because our environment allows such threats to be coped with adequately from outside the system. Similarly,

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

A version of this paper was presented at the 7th ACM Symposium on Operating Systems Principles, Pacific Grove, Calif., Dec. 10–12, 1979.

Authors' address: Xerox Business Systems, 3333 Coyote Hill Rd., Palo Alto, CA 94304.

© 1980 ACM 0001-0782/80/0200-0081 $00.75.

Pilot's resource allocation features are not oriented toward enforcing fair distribution of scarce resources among contending parties. In traditional multi-user systems, most resources tend to be in short supply, and prevention of inequitable distribution is a serious problem. In a single-user system like Pilot, shortage of some resource must generally be dealt with either through more effective utilization or by adding more of the resource.

The close coupling between Pilot and Mesa is based on mutual interdependence; Pilot is written in Mesa, and Mesa depends on Pilot for much of its runtime support. Since other languages are not supported, many of the language-independence arguments that tend to maintain distance between an operating system and a programming language are not relevant. In a sense, all of Pilot can be thought of as a very powerful runtime support package for the Mesa language. Naturally, none of these considerations eliminates the need for careful structuring of the combined Pilot/Mesa system to avoid accidental circular dependencies.

Since the Mesa programming language formalizes and emphasizes the distinction between an *interface* and its *implementation*, it is particularly appropriate to split the description of Pilot along these lines. As an environment for its client programs, Pilot consists of a set of Mesa interfaces, each defining a group of related types, operations, and error signals. Section 2 enumerates the major interfaces of Pilot and describes their semantics, in terms of both the formal interface and the intended behavior of the system as a whole. As a Mesa program, Pilot consists of a large collection of modules supporting the various interfaces seen by clients. Section 3 describes the interior structure of the Pilot implementation and mentions a few of the lessons learned in implementing an operating system in Mesa.

2. Pilot Interfaces

In Mesa, a large software system is constructed from two kinds of modules: *program* modules specify the algorithms and the actual data structures comprising the *implementation* of the system, while *definitions* modules formally specify the *interfaces* between program modules. Generally, a given interface, defined in a definitions module, is *exported* by one program module (its *implementor*) and *imported* by one or more other program modules (its *clients*). Both program and definitions modules are written in the Mesa source language and are compiled to produce binary object modules. The object form of a program module contains the actual code to be executed; the object form of a definitions module contains detailed specifications controlling the binding together of program modules. Modular programming in Mesa is discussed in more detail by Lauer and Satterthwaite [13].

Pilot contains two kinds of interfaces:

(1) *Public* interfaces defining the services provided by Pilot to its clients (i.e., higher level Mesa programs);
(2) *Private* interfaces, which form the connective tissue binding the implementation together.

This section describes the major features supported by the public interfaces of Pilot, including files, virtual memory, streams, network communication, and concurrent programming support. Each interface defines some number of named items, which are denoted *Interface.Item*. There are four kinds of items in interfaces: types, procedures, constants, and error signals. (For example, the interface *File* defines the type *File.Capability*, the procedure *File.Create*, the constant *file.maxPages PerFile*, and the error signal *File.Unknown.*) The discussion that follows makes no attempt at complete enumeration of the items in each interface, but focuses instead on the overall facility provided, emphasizing the more important and unusual features of Pilot.

2.1 Files

The Pilot interfaces *File* and *Volume* define the basic facilities for permanent storage of data. Files are the standard containers for information storage; volumes represent the media on which files are stored (e.g., magnetic disks). Higher level software is expected to superimpose further structure on files and volumes as necessary (e.g., an executable subsystem on a file, or a detachable directory subtree on a removable volume). The emphasis at the Pilot level is on simple but powerful primitives for accessing large bodies of information. Pilot can handle files containing up to about a million pages of English text, and volumes larger than any currently available storage device ($\sim 10^{13}$ bits). The total number of files and volumes that can exist is essentially unbounded (2^{64}). The space of files provided is "flat," in the sense that files have no recognized relationships among them (e.g., no directory hierarchy). The size of a file is adjustable in units of pages. As discussed below, the contents of a file are accessed by mapping one or more of its pages into a section of virtual memory.

The *File.Create* operation creates a new file and returns a capability for it. Pilot file capabilities are intended for *defensive* protection against errors [9]; they are mechanically similar to capabilities used in other systems for absolute protection, but are not designed to withstand determined attack by a malicious programmer. More significant than the protection aspect of capabilities is the fact that files and volumes are named by 64-bit universal identifiers (uids) which are guaranteed unique in both space and time. This means that distinct files, created anywhere at any time by any incarnation of Pilot, will always have distinct uids. This guarantee is crucial, since removable volumes are expected to be a standard method of transporting information from one

Pilot system to another. If uid ambiguity were allowed (e.g., different files on the same machine with the same uid), Pilot's life would become more difficult, and uids would be much less useful to clients. To guarantee uniqueness, Pilot essentially concatenates the machine serial number with the real time clock to produce each new uid.

Pilot attaches only a small fixed set of attributes to each file, with the expectation that a higher level directory facility will provide an extendible mechanism for associating with a file more general properties unknown to Pilot (e.g., length in bytes, date of creation, etc.). Pilot recognizes only four attributes: size, type, permanence, and immutability.

The *size* of a file is adjustable from 0 pages to 2^{23} pages, each containing 512 bytes. When the size of a file is increased, Pilot attempts to avoid fragmentation of storage on the physical device so that sequential or otherwise clustered accesses can exploit physical contiguity. On the other hand, random probes into a file are handled as efficiently as possible, by minimizing file system mapping overhead.

The *type* of a file is a 16-bit tag which is essentially uninterpreted, but is implemented at the Pilot level to aid in type-dependent recovery of the file system (e.g., after a system failure). Such recovery is discussed further in Section 3.4.

Permanence is an attribute attached to Pilot files that are intended to hold valuable permanent information. The intent is that creation of such a file proceed in four steps:

(1) The file is created using *File.Create* and has temporary status.
(2) A capability for the file is stored in some permanent directory structure.
(3) The file is made permanent using the *File.MakePermanent* operation.
(4) The valuable contents are placed in the file.

If a system failure occurs before step 3, the file will be automatically deleted (by the scavenger; see Section 3.4) when the system restarts; if a system failure occurs after step 2, the file is registered in the directory structure and is thereby accessible. (In particular, a failure between steps 2 and 3 produces a registered but nonexistent file, an eventuality which any robust directory system must be prepared to cope with.) This simple mechanism solves the "lost object problem" [25] in which inaccessible files take up space but cannot be deleted. Temporary files are also useful as scratch storage which will be reclaimed automatically in case of system failure.

A Pilot file may be made *immutable.* This means that it is permanently read-only and may never be modified again under any circumstances. The intent is that multiple physical copies of an immutable file, all sharing the *same* universal identifier, may be replicated at many physical sites to improve accessibility without danger of ambiguity concerning the contents of the file. For example, a higher level "linkage editor" program might wish to link a pair of object-code files by embedding the uid of one in the other. This would be efficient and unambiguous, but would fail if the contents were copied into a new pair of files, since they would have different uids. Making such files immutable and using a special operation (*File.ReplicateImmutable*) allows propagation of physical copies to other volumes without changing the uids, thus preserving any direct uid-level bindings.

As with files, Pilot treats volumes in a straightforward fashion, while at the same time avoiding oversimplifications that would render its facilities inadequate for demanding clients. Several different sizes and types of storage devices are supported as Pilot volumes. (All are varieties of moving-arm disk, removable or nonremovable; other nonvolatile random access storage devices could be supported.) The simplest notion of a volume would correspond one to one with a physical storage medium. This is too restrictive, and hence the abstraction presented at the *Volume* interface is actually a *logical volume*; Pilot is fairly flexible about the correspondence between logical volumes and *physical volumes* (e.g., disk packs, diskettes, etc.). On the one hand, it is possible to have a large logical volume which spans several physical volumes. Conversely, it is possible to put several small logical volumes on the same physical volume. In all cases, Pilot recognizes the comings and goings of physical volumes (e.g., mounting a disk pack) and makes accessible to client programs those logical volumes all of whose pages are on-line.

Two examples which originally motivated the flexibility of the volume machinery were database applications, in which a very large database could be cast as a multi-disk-pack volume, and the CoPilot debugger, which requires its own separate logical volume (see Section 2.5), but must be usable on a single-disk machine.

2.2 Virtual Memory

The machine architecture on which Pilot runs defines a simple linear virtual memory of up to 2^{32} 16-bit words. All computations on the machine (including Pilot itself) run in the same address space, which is unadorned with any noteworthy features, save a set of three flags attached to each page: *referenced*, *written*, and *write-protected.* Pilot structures this homogenous address space into contiguous runs of page called *spaces*, accessed through the interface *Space.* Above the level of Pilot, client software superimposes still further structure upon the contents of spaces, casting them as client-defined data structures within the Mesa language.

While the underlying linear virtual memory is conventional and fairly straightforward, the space machinery superimposed by Pilot is somewhat novel in its design, and rather more powerful than one would expect given the simplicity of the *Space* interface. A space is capable of playing three fundamental roles:

Allocation Entity. To allocate a region of virtual memory, a client creates a space of appropriate size.

Mapping Entity. To associate information content and backing store with a region of virtual memory, a client maps a space to a region of some file.

Swapping Entity. The transfer of pages between primary memory and backing store is performed in units of spaces.

Any given space may play any or all of these roles. Largely because of their multifunctional nature, it is often useful to nest spaces. A new space is always created as a subspace of some previously existing space, so that the set of all spaces forms a tree by containment, the root of which is a predefined space covering all of virtual memory.

Spaces function as allocation entities in two senses: when a space is created, by calling *Space.Create*, it is serving as the unit of allocation; if it is later broken into subspaces, it is serving as an allocation subpool within which smaller units are allocated and freed [19]. Such suballocation may be nested to several levels; at some level (typically fairly quickly) the page granularity of the space mechanism becomes too coarse, at which point finer grained allocation must be performed by higher level software.

Spaces function as mapping entities when the operation *Space.Map* is applied to them. This operation associates the space with a run of pages in a file, thus defining the content of each page of the space as the content of its associated file page, and propagating the write-protection status of the file capability to the space. At any given time, a page in virtual memory may be accessed only if its content is well-defined, i.e., if *exactly one* of the nested spaces containing it is mapped. If none of the containing spaces is mapped, the fatal error *AddressFault* is signaled. (The situation in which more than one containing space is mapped cannot arise, since the *Space.Map* operation checks that none of the ancestors or descendents of a space being mapped are themselves already mapped.) The decision to cast *AddressFault* and *WriteProtectFault* (i.e., storing into a write-protected space) as fatal errors is based on the judgment that any program which has incurred such a fault is misusing the virtual memory facilities and should be debugged; to this end, Pilot unconditionally activates the CoPilot debugger (see Section 2.5).

Spaces function as swapping entities when a page of a mapped space is found to be missing from primary memory. The swapping strategy followed is essentially to swap in the lowest level (i.e., smallest) space containing the page (see Section 3.2). A client program can thus optimize its swapping behavior by subdividing its mapped spaces into subspaces containing items whose access patterns are known to be strongly correlated. In the absence of such subdivision, the entire mapped space is swapped in. Note that while the client can always opt for demand paging (by breaking a space up into one-page subspaces), this is *not* the default, since it tends to promote thrashing. Further optimization is possible using the *Space.Activate* operation. This operation advises Pilot that a space will be used soon and should be swapped in as soon as possible. The inverse operation, *Space.Deactivate*, advises Pilot that a space is no longer needed in primary memory. The *Space.Kill* operation advises Pilot that the current contents of a space are of no further interest (i.e., will be completely overwritten before next being read) so that useless swapping of the data may be suppressed. These forms of optional advice are intended to allow tuning of heavy traffic periods by eliminating unnecessary transfers, by scheduling the disk arm efficiently, and by insuring that during the visit to a given arm position all of the appropriate transfers take place. Such advice-taking is a good example of a feature which has been deemed undesirable by most designers of timesharing systems, but which can be very useful in the context of a dedicated personal computer.

There is an intrinsic close coupling between Pilot's file and virtual memory features: virtual memory is the only access path to the contents of files, and files are the only backing store for virtual memory. An alternative would have been to provide a separate backing store for virtual memory and require that clients transfer data between virtual memory and files using explicit read/write operations. There are several reasons for preferring the mapping approach, including the following.

(1) Separating the operations of mapping and swapping decouples buffer allocation from disk scheduling, as compared with explicit file read/write operations.

(2) When a space is mapped, the read/write privileges of the file capability can propagate automatically to the space by setting a simple read/write lock in the hardware memory map, allowing illegitimate stores to be caught immediately.

(3) In either approach, there are certain cases that generate extra unnecessary disk transfers; extra "advice-taking" operations like *Space.Kill* can eliminate the extra disk transfers in the mapping approach; this does not seem to apply to the read/write approach.

(4) It is relatively easy to simulate a read/write interface given a mapping interface, and with appropriate use of advice, the efficiency can be essentially the same. The converse appears to be false.

The Pilot virtual memory also provides an advice-like operation called *Space.ForceOut*, which is designed as an underpinning for client crash-recovery algorithms. (It is advice-like in that its effect is invisible in normal operation, but becomes visible if the system crashes.) *ForceOut* causes a space's contents to be written to its backing file and does not return until the write is completed. This means that the contents will survive a subsequent system crash. Since Pilot's page replacement algorithm is also free to write the pages to the file at any time (e.g., between *ForceOuts*), this facility by itself does *not* constitute even a minimal crash recovery mechanism; it is intended only as a "toehold" for higher level software to use in providing transactional atomicity in the face of system crashes.

2.3 Streams and I/O Devices

A Pilot client can access an I/O device in three different ways:

(1) *implicitly*, via some feature of Pilot (e.g., a Pilot file accessed via virtual memory);
(2) *directly*, via a low-level device driver interface exported from Pilot;
(3) *indirectly*, via the Pilot stream facility.

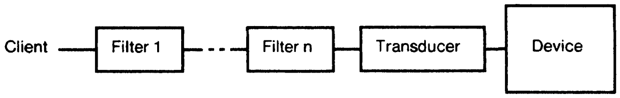
In keeping with the objectives of Pilot as an operating system for a personal computer, most I/O devices are made directly available to clients through low-level procedural interfaces. These interfaces generally do little more than convert device-specific I/O operations into appropriate procedure calls. The emphasis is on providing maximum flexibility to client programs; protection is not required. The only exception to this policy is for devices accessed implicitly by Pilot itself (e.g., disks used for files), since chaos would ensue if clients also tried to access them directly.

For most applications, direct device access via the device driver interface is rather inconvenient, since all the details of the device are exposed to view. Furthermore, many applications tend to reference devices in a basically sequential fashion, with only occasional, and usually very stylized, control or repositioning operations. For these reasons, the Pilot *stream* facility is provided, comprising the following components:

(1) The *stream* interface, which defines device independent operations for full-duplex sequential access to a source/sink of data. This is very similar in spirit to the stream facilities of other operating systems, such as OS6 [23] and UNIX [18].
(2) A standard for *stream components*, which connect streams to various devices and/or implement "on-the-fly" transformations of the data flowing through them.
(3) A means for cascading a number of primitive stream components to provide a compound stream.

There are two kinds of stream components defined by Pilot: the transducer and the filter. A *transducer* is a module which imports a device driver interface and exports an instance of the Pilot *Stream* interface. The transducer is thus the implementation of the basic sequential access facility for that device. Pilot provides standard transducers for a variety of supported devices. A *filter* is a module which imports one instance of the Pilot standard *Stream* interface and exports another. Its purpose is to transform a stream of data "on the fly" (e.g., to do code or format conversion). Naturally, clients can augment the standard set of stream components provided with Pilot by writing filters and transducers of their own. The *Stream* interface provides for dynamic binding of stream components at runtime, so that a transducer and a set of filters can be cascaded to provide a *pipeline*, as shown in Figure 1.

Fig. 1. A pipeline of cascaded stream components.



The transducer occupies the lowest position in the pipeline (i.e., nearest the device) while the client program accesses the highest position. Each filter accesses the next lower filter (or transducer) via the *Stream* interface, just as if it were a client program, so that no component need be aware of its position in the pipeline, or of the nature of the device at the end. This facility resembles the UNIX pipe and filter facility, except that it is implemented at the module level within the Pilot virtual memory, rather than as a separate system task with its own address space.

2.4 Communications

Mesa supports a shared-memory style of interprocess communication for *tightly coupled* processes [11]. Interaction between *loosely coupled* processes (e.g., suitable to reside on different machines) is provided by the Pilot *communications* facility. This facility allows client processes in different machines to communicate with each other via a hierarchically structured family of packet communication protocols. Communication software is an integral part of Pilot, rather than an optional addition, because Pilot is intended to be a suitable foundation for network-based distributed systems.

The protocols are designed to provide communication across multiple interconnected networks. An interconnection of networks is referred to as an *internet*. A Pilot internet typically consists of local, high bandwidth Ethernet broadcast networks [15], and public and private long-distance data networks like SBS, TELENET, TYMNET, DDS, and ACS. Constituent networks are interconnected by *internetwork routers* (often referred to as *gateways* in the literature) which store and forward packets to their destination using distributed routing algorithms [2, 4]. The constituent networks of an internet are used only as a transmission medium. The source, destination, and internetwork router computers are *all* Pilot machines. Pilot provides software drivers for a variety of networks; a given machine may connect directly to one or several networks of the same or different kinds.

Pilot clients identify one another by means of *network addresses* when they wish to communicate and need not know anything about the internet toplogy or each other's locations or even the structure of a network address. In particular, it is not necessary that the two communicators be on different computers. If they are on the same computer, Pilot will optimize the transmission of data between them and will avoid use of the physical network resources. This implies that an isolated computer (i.e., one which is not connected to any network) may still contain the communications facilities of Pilot. Pilot clients on the same computer should communicate with one another using Pilot's communications facilities, as opposed to the tightly coupled mechanisms of Mesa, if the communicators are loosely coupled subsystems that could some day be reconfigured to execute on different machines on the network. For example, printing and file storage server programs written to communicate in the loosely coupled mode could share the same machine if the combined load were light, yet be easily moved to separate machines if increased load justified the extra cost.

A network address is a resource assigned to clients by Pilot and identifies a specific *socket* on a specific machine. A socket is simply a site from which packets are transmitted and at which packets are received; it is rather like a post office box, in the sense that there is no assumed relationship among the packets being sent and received via a given socket. The identity of a socket is unique only at a given point in time; it *may* be reused, since there is no long-term static association between the socket and any other resources. Protection against dangling references (e.g., delivery of packets intended for a previous instance of a given socket) is guaranteed by higher level protocols.

A network address is, in reality, a triple consisting of a 16-bit network number, a 32-bit processor ID, and a 16-bit socket number, represented by a system-wide Mesa data type *System.NetworkAddress.* The internal structure of a network address is not used by clients, but by the communications facilities of Pilot and the internetwork routers to deliver a packet to its destination. The administrative procedures for the assignment of network numbers and processor IDs to networks and computers, respectively, are outside the scope of this paper, as are the mechanisms by which clients find out each others' network addresses.

The family of packet protocols by which Pilot provides communication is based on our experiences with the Pup Protocols [2]. The Arpa Internetwork Protocol family [8] resemble our protocols in spirit. The protocols fall naturally into three levels:

Level 0: Every packet must be encapsulated for transmission over a particular communication medium, according to the network-specific rules for that communication medium. This has been termed level 0 in our protocol hierarchy, since its definition is of no concern to the typical Pilot client.

Level 1: Level 1 defines the format of the *internetwork packet*, which specifies among other things the source and destination network addresses, a checksum field, the length of the entire packet, a transport control field that is used by internetwork routers, and a packet type field that indicates the kind of packet defined at level 2.

Level 2: A number of level 2 packet formats exist, such as error packet, connection-oriented sequenced packet, routing table update packet, and so on. Various level 2 protocols are defined according to the kinds of level 2 packets they use, and the rules governing their interaction.

The *Socket* interface provides level 1 access to the communication facilities, including the ability to create a socket at a (local) network address, and to transmit and receive internetwork packets. In the terms of Section 2.3, sockets can be thought of as *virtual devices*, accessed directly via the *Socket* (virtual driver) interface. The protocol defining the format of the internetwork packet provides end-to-end communication at the packet level. The internet is required only to be able to transport independently addressed packets from source to destination network addresses. As a consequence, packets transmitted over a socket may be expected to arrive at their destination only with *high probability* and not necessarily in the order they were transmitted. It is the responsibility of the communicating end processes to agree upon higher level protocols that provide the appropriate level of reliable communication. The *Socket* interface, therefore, provides service similar to that provided by networks that offer *datagram* services [17] and is most useful for connectionless protocols.

The interface *NetworkStream* defines the principal means by which Pilot clients can communicate reliably between any two network addreses. It provides access to the implementation of the *sequenced packet protocol*—a level 2 protocol. This protocol provides sequenced, duplicate-suppressed, error-free, flow-controlled packet communication over arbitrarily interconnected communication networks and is similar in philosophy to the Pup Byte Stream Protocol [2] or the Arpa Transmission Control Protocol [3, 24]. This protocol is implemented as a transducer, which converts the device-like *Socket* interface into a Pilot stream. Thus all data transmission via a network stream is invoked by means of the operations defined in the standard *Stream* interface.

Network streams provide reliable communication, in the sense that the data is reliably sent from the source transducer's packet buffer to the destination transducer's packet buffer. No guarantees can be made as to whether the data was successfully received by the destination client or that the data was appropriately processed. This final degree of reliability must lie with the clients of network streams, since they alone know the higher level protocol governing the data transfer. Pilot provides communication with varying degrees of reliability, since the communicating clients will, in general, have differing needs for it. This is in keeping with the design goals of Pilot, much like the provision of defensive rather than absolute protection.

A network stream can be set up between two communicators in many ways. The most typical case, in a network-based distributed system, involves a *server* (a supplier of a service) at one end and a *client* of the service at the other. Creation of such a network stream is inherently asymmetric. At one end is the server which advertises a network address to which clients can connect to obtain its services. Clients do this by calling *NetworkStream.Create*, specifying the address of the server as parameter. It is important that concurrent requests from clients not conflict over the server's network address; to avoid this, some additional machinery is provided at the server end of the connection. When a server is operational, one of its processes *listens* for requests on its advertised network address. This is done by calling *NetworkStream.Listen*, which automatically creates a new network stream each time a request arrives at the specified network address. The newly created network stream connects the client to *another* unique network address on the server machine, leaving the server's advertised network address free for the reception of additional requests.

The switchover from one network address to another is transparent to the client, and is part of the definition of the sequenced packet protocol. At the server end, the *Stream.Handle* for the newly created stream is typically passed to an *agent*, a subsidiary process or subsystem which gives its full attention to performing the service for that particular client. These two then communicate by means of the new network stream set up between them for the duration of the service. Of course, the *NetworkStream* interface also provides mechanisms for creating connections between arbitrary network addresses, where the relationship between the processes is more general than that of server and client.

The mechanisms for establishing and deleting a connection between any two communicators and for guarding against old duplicate packets are a departure from the mechanisms used by the Pup Byte Stream Protocol [2] or the Transmission Control Protocol [22], although our protocol embodies similar principles. A network stream is terminated by calling *NetworkStream.Delete.* This call initiates no network traffic and simply deletes all the data structures associated with the network stream. It is the responsibility of the communicating processes to have decided *a priori* that they wish to terminate the stream. This is in keeping with the decision that the reliable processing of the transmitted data ultimately rests with the clients of network streams.

The manner in which server addresses are advertised by servers and discovered by clients is not defined by Pilot; this facility must be provided by the architecture of a particular distributed system built on Pilot. Generally, the binding of names of resources to their addresses is accomplished by means of a network-based database referred to as a *clearinghouse.* The manner in which the binding is structured and the way in which clearinghouses are located and accessed are outside the scope of this paper.

The communication facilities of Pilot provide clients various interfaces, which provide varying degrees of service at the internetworking level. In keeping with the overall design of Pilot, the communication facility attempts to provide a standard set of features which capture the most common needs, while still allowing clients to custom tailor their own solutions to their communications requirements if that proves necessary.

2.5 Mesa Language Support

The Mesa language provides a number of features which require a nontrivial amount of runtime support [16]. These are primarily involved with the control structure of the language [10, 11] which allow not only recursive procedure calls, but also coroutines, concurrent processes, and signals (a specialized form of dynamically bound procedure call used primarily for exception handling). The runtime support facilities are invoked in three ways:

(1) explicitly, via normal Mesa interfaces exported by Pilot (e.g., the *Process* interface);
(2) implicitly, via compiler-generated calls on built-in procedures;
(3) via traps, when machine-level op-codes encounter exceptional conditions.

Pilot's involvement in client procedure calls is limited to trap handling when the supply of activation record storage is exhausted. To support the full generality of the Mesa control structures, activation records are allocated from a heap, even when a strict LIFO usage pattern is in force. This heap is replenished and maintained by Pilot.

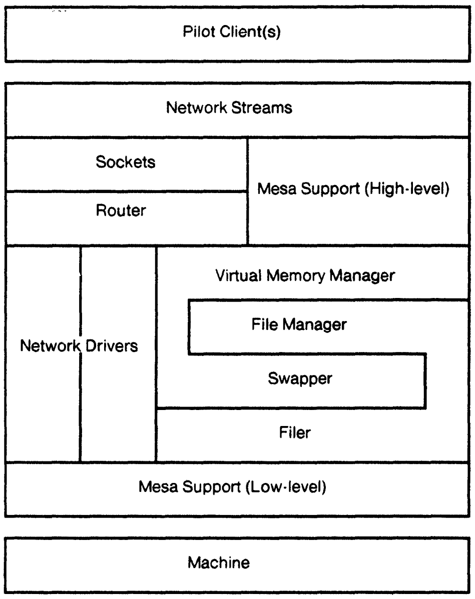
Coroutine calls also proceed without intervention by Pilot, except during initialization when a trap handler is provided to aid in the original setup of the coroutine linkage.

Pilot's involvement with concurrent processes is somewhat more substantial. Mesa casts process creation as a variant of a procedure call, but unlike a normal procedure call, such a FORK statement *always* invokes Pilot to create the new process. Similarly, termination of a process also involves substantial participation by Pilot. Mesa also provides monitors and condition variables for synchronized interprocess communication via shared memory; these facilities are supported directly by the machine and thus require less direct involvement of Pilot.

The Mesa control structure facilities, including concurrent processes, are light weight enough to be used in the fine-scale structuring of normal Mesa programs. A typical Pilot client program consists of some number of processes, any of which may at any time invoke Pilot facilities through the various public interfaces. It is Pilot's responsibility to maintain the semantic integrity of its interfaces in the face of such client-level concurrency (see Section 3.3). Naturally, any higher level consistency constraints invented by the client must be guaranteed by client-level synchronization, using monitors and condition variables as provided in the Mesa language.

Another important Mesa-support facility which is provided as an integral part of Pilot is a "world-swap" facility to allow a graceful exit to CoPilot, the Pilot/Mesa interactive debugger. The world-swap facility saves the contents of memory and the total machine state and then starts CoPilot from a *boot-file*, just as if the machine's bootstrap-load button had been pressed. The original state is saved on a second boot-file so that execution can be resumed by doing a second world-swap. The state is saved with sufficient care that it is virtually always possible to resume execution without any detectable perturbation of the program being debugged. The world-swap approach to debugging yields strong isolation between the debugger and the program under test. Not only the contents of main memory, but the version of Pilot, the accessible volume(s), and even the microcode can be different in the two worlds. This is especially useful when debugging a new version of Pilot, since CoPilot can run on the old, stable version until the new version becomes trustworthy. Needless to say, this approach is not directly applicable to conventional multiuser time-sharing systems.

Fig. 2. Major components of Pilot.



3. Implementation

The implementation of Pilot consists of a large number of Mesa modules which collectively provide the client environment as decribed above. The modules are grouped into larger *components*, each of which is responsible for implementing some coherent subset of the overall Pilot functionality. The relationships among the major components are illustrated in Figure 2.

Of particular interest is the interlocking structure of the four components of the *storage system* which together implement files and virtual memory. This is an example of what we call the *manager/kernel* pattern, in which a given facility is implemented in two stages: a low-level kernel provides a basic core of function, which is extended by the higher level manager. Layers interposed between the kernel and the manager can make use of the kernel and can in turn be used by the manager. The same basic technique has been used before in other systems to good effect, as discussed by Habermann et al. [6], who refer to it as "functional hierarchy." It is also quite similar to the familiar "policy/mechanism" pattern [1, 25]. The main difference is that we place no emphasis on the possibility of using the same kernel with a variety of managers (or without any manager at all). In Pilot, the manager/kernel pattern is intended only as a fruitful decomposition tool for the design of integrated mechanisms.

3.1 Layering of the Storage System Implementation

The kernel/manager pattern can be motivated by noting that since the purpose of Pilot is to provide a more hospitable environment than the bare machine, it would clearly be more pleasant for the code implementing Pilot if it could use the facilities of Pilot in getting its job done. In particular, both components of the *storage system* (the file and virtual memory implementations) maintain internal databases which are too large to fit in primary memory, but only parts of which are needed at any one time. A client-level program would simply place such a database in a file and access it via virtual memory, but if Pilot itself did so, the resulting circular dependencies would tie the system in knots, making it unreliable and difficult to understand. One alternative would be the invention of a special separate mechanism for low-level disk access and main memory buffering, used only by the storage system to access its internal databases. This would eliminate the danger of circular dependency but would introduce more machinery, making the system bulkier and harder to understand in a different sense. A more attractive alternative is the extraction of a streamlined kernel of the storage system functionality with the following properties:

(1) It can be implemented by a small body of code which resides permanently in primary memory.
(2) It provides a powerful enough storage facility to significantly ease the implementation of the remainder of the full-fledged storage system.
(3) It can handle the majority of the "fast cases" of client-level use of the storage system.

Figure 2 shows the implementation of such a kernel storage facility by the swapper and the filer. These two subcomponents are the kernels of the virtual memory and file components, respectively, and provide a reasonably powerful environment for the nonresident subcomponents, the virtual memory manager, and the file manager, whose code and data are both swappable. The kernel environment provides somewhat restricted virtual memory access to a small number of special files and to preexisting normal files of fixed size.

The managers implement the more powerful operations, such as file creation and deletion, and the more complex virtual memory operations, such as those that traverse subtrees of the hierarchy of nested spaces. The most frequent operations, however, are handled by the kernels essentially on their own. For example, a page fault is handled by code in the swapper, which calls the filer to read the appropriate page(s) into memory, adjusts the hardware memory map, and restarts the faulting process.

The resident data structures of the kernels serve as caches on the swappable databases maintained by the managers. Whenever a kernel finds that it cannot perform an operation using only the data in its cache, it conceptually "passes the buck" to its manager, retaining no state information about the failed operation. In this way, a circular dependency is avoided, since such failed operations become the total responsibility of the manager. The typical response of a manager in such a situation is to consult its swappable database, call the resident subcomponent to update its cache, and then retry the failed operation.

The intended dynamics of the storage system implementation described above are based on the expectation that Pilot will experience three quite different kinds of load.

(1) For short periods of time, client programs will have their essentially static working sets in primary memory and the storage system will not be needed.
(2) Most of the time, the client working set will be changing slowly, but the description of it will fit in the swapper/filer caches, so that swapping can take place with little or no extra disk activity to access the storage system databases.
(3) Periodically, the client working set will change drastically, requiring extensive reloading of the caches as well as heavy swapping.

It is intended that the Pilot storage system be able to respond reasonably to all three situations: In case (1), it should assume a low profile by allowing its swappable components (e.g., the managers) to swap out. In case (2), it should be as efficient as possible, using its caches to avoid causing spurious disk activity. In case (3), it should do the best it can, with the understanding that while continuous operation in this mode is probably not viable, short periods of heavy traffic can and must be optimized, largely via the advice-taking operations discussed in Section 2.2.

3.2 Cached Databases of the Virtual Memory Implementation

The virtual memory manager implements the client visible operations on spaces and is thus primarily concerned with checking validity and maintaining the database constituting the fundamental representation behind the *Space* interface. This database, called the *hierarchy*, represents the tree of nested spaces defined in Section 2.2. For each space, it contains a record whose fields hold attributes such as size, base page number, and mapping information.

The swapper, or virtual memory kernel, manages primary memory and supervises the swapping of data between mapped memory and files. For this purpose it needs access to information in the hierarchy. Since the hierarchy is swappable and thus off limits to the swapper, the swapper maintains a resident *space cache* which is loaded from the hierarchy in the manner described in Section 3.1.

There are several other data structures maintained by the swapper. One is a bit-table describing the allocation status of each page of primary memory. Most of the bookkeeping performed by the swapper, however, is on the basis of the *swap unit*, or smallest set of pages transferred between primary memory and file backing storage. A swap unit generally corresponds to a "leaf" space; however, if a space is only partially covered with subspaces, each maximal run of pages not containing any subspaces is also a swap unit. The swapper keeps a *swap unit cache* containing information about swap units such as extent (first page and length), containing mapped space, and state (mapped or not, swapped in or out, replacement algorithm data).

The swap unit cache is addressed by page rather than by space; for example, it is used by the page fault handler to find the swap unit in which a page fault occurred. The content of an entry in this cache is logically derived from a sequence of entries in the hierarchy, but direct implementation of this would require several file accesses to construct a single cache entry. To avoid this, we have chosen to maintain another database: the *projection.* This is a second swappable database maintained by the virtual memory manager, containing descriptions of all existing swap units, and is used to update the swap unit cache. The existence of the projection speeds up page faults which cannot be handled from the swap unit cache; it slows down space creation/deletion since then the projection must be updated. We expect this to be a useful optimization based on our assumptions about the relative frequencies and CPU times of these events; detailed measurements of a fully loaded system will be needed to evaluate the actual effectiveness of the projection.

An important detail regarding the relationship between the manager and kernel components has been ignored up to this point. That detail is avoiding "recursive" cache faults; when a manager is attempting to supply a missing cache entry, it will often incur a page fault of its own; the handling of that page fault must *not* incur a second cache fault or the fault episode will never terminate. Basically the answer is to make certain key records in the cache ineligible for replacement. This pertains to the space and swap unit caches and to the caches maintained by the filer as well.

3.3 Process Implementation

The implementation of processes and monitors in Pilot/Mesa is summarized here; more detail can be found in [11].

The task of implementing the concurrency facilities is split roughly equally among Pilot, the Mesa compiler, and the underlying machine. The basic primitives are defined as language constructs (e.g., entering a MONITOR, WAITing on a CONDITION variable, FORKing a new PROCESS) and are implemented either by machine op-codes (for heavily used constructs, e.g., WAIT) or by calls on Pilot (for less heavily used constructs, e.g., FORK). The constructs supported by the machine and the low-level Mesa support component provide procedure calls and synchronization among existing processes, allowing the remainder of Pilot to be implemented as a collection of monitors, which carefully synchronize the multiple processes executing concurrently inside them. These processes comprise a variable number of client processes (e.g., which have called into Pilot through some public interface) plus a fixed number of dedicated system processes (about a dozen) which are created specially at system initialization time. The machinery for creating and deleting processes is a monitor within the high-level Mesa support component; this places it above the virtual memory implementation; this means that it is swappable, but also means that the rest of Pilot (with the exception of network streams) cannot make use of dynamic process creation. The process implementation is thus another example of the manager/kernel pattern, in which the manager is implemented at a very high level and the kernel is pushed down to a very low level (in this case, largely into the underlying machine). To the Pilot client, the split implementation appears as a unified mechanism comprising the Mesa language features and the operations defined by the Pilot *Process* interface.

3.4 File System Robustness

One of the most important properties of the Pilot file system is robustness. This is achieved primarily through the use of *reconstructable maps.* Many previous systems have demonstrated the value of a *file scavenger*, a utility program which can repair a damaged file system, often on a more or less *ad hoc* basis [5, 12, 14, 21]. In Pilot, the scavenger is given first-class citizenship, in the sense that the file structures were all designed from the beginning with the scavenger in mind. Each file page is self-identifying by virtue of its *label*, written as a separate physical record adjacent to the one holding the actual contents of the page. (Again, this is not a new idea, but is the crucial foundation on which the file system's robustness is based.) Conceptually, one can think of a file page access proceeding by scanning all known volumes, checking the label of each page encountered until the desired one is found. In practice, this scan is performed only once by the scavenger, which leaves behind maps on each volume describing what it found there; Pilot then uses the maps and incrementally updates them as file pages are created and deleted. The logical redundancy of the maps does not, of course, imply lack of importance, since the system would be not be viable without them; the point is that since they contain *only* redundant information, they can be completely reconstructed should they be lost. In particular, this means that damage to any page on the disk can compromise only data on that page.

The primary map structure is the volume file map, a B-tree keyed on ⟨file-uid, page-number⟩ which returns the device address of the page. All file storage devices check the label of the page and abort the I/O operation in case of a mismatch; this does not occur in normal operation and generally indicates the need to scavenge the volume. The volume file map uses extensive compression of uids and run-encoding of page numbers to maximize the out-degree of the internal nodes of the B-tree and thus minimize its depth.

Equally important but much simpler is the volume allocation map, a table which describes the allocation status of each page on the disk. Each free page is a self-identifying member of a hypothetical file of free pages, allowing reconstruction of the volume allocation map.

The robustness provided by the scavenger can only guarantee the integrity of files as defined by Pilot. If a database defined by client software becomes inconsistent due to a system crash, a software bug, or some other unfortunate event, it is little comfort to know that the underlying file has been declared healthy by the scavenger. An "escape-hatch" is therefore provided to allow client software to be invoked when a file is scavenged. This is the main use of the file-type attribute mentioned in Section 2.1. After the Pilot scavenger has restored the low-level integrity of the file system, Pilot is restarted; before resuming normal processing, Pilot first invokes all client-level scavenging routines (if any) to reestablish any higher level consistency constraints that may have been violated. File types are used to determine which files should be processed by which client-level scavengers.

An interesting example of the first-class status of the scavenger is its routine use in transporting volumes between versions of Pilot. The freedom to redesign the complex map structures stored on volumes represents a crucial opportunity for continuing file system performance improvement, but this means that one version of Pilot may find the maps left by another version totally inscrutable. Since such incompatibility is just a particular form of "damage," however, the scavenger can be invoked to reconstruct the maps in the proper format, after which the corresponding version of Pilot will recognize the volume as its own.

3.5 Communication Implementation

The software that implements the packet communication protocols consists of a set of network-specific drivers, modules that implement sockets, network stream transducers, and at the heart of it all, a *router.* The router is a software switch. It routes packets among sockets, sockets and networks, and networks themselves. A router is present on *every* Pilot machine. On personal machines, the router handles only incoming, outgoing, and intramachine packet traffic. On internetwork router machines, the router acts as a service to other machines by transporting internetwork packets across network boundaries. The router's data structures include a list of all active sockets and networks on the local computer. The router is designed so that network drivers may easily be added to or removed from new configurations of Pilot; this can even be done dynamically during execution. Sockets come and go as clients create and delete them. Each router maintains a routing table indicating, for a given remote network, the best internetwork router to use as the next "hop" toward the final destination. Thus, the two kinds of machines are essentially special cases of the same program. An internetwork router is simply a router that spends most of its time forwarding packets between networks and exchanging routing tables with other internetwork routers. On personal machines the router updates its routing table by querying internetwork routers or by overhearing their exchanges over broadcast networks.

Pilot has taken the approach of connecting a network much like any other input/output device, so that the packet communication protocol software becomes part of the operating system and operates in the same personal computer. In particular, Pilot does *not* employ a dedicated front-end communications processor connected to the Pilot machine via a secondary interface.

Network-oriented communication differs from conventional input/output in that packets arrive at a computer *unsolicited*, implying that the intended recipient is unknown until the packet is examined. As a consequence, each incoming packet must be buffered initially in router-supplied storage for examination. The router, therefore, maintains a buffer pool shared by all the network drivers. If a packet is undamaged and its destination socket exists, then the packet is copied into a buffer associated with the socket and provided by the socket's client.

The architecture of the communication software permits the computer supporting Pilot to behave as a user's personal computer, a supplier of information, or as a dedicated internetwork router.

3.6 The Implementation Experience

The initial construction of Pilot was accomplished by a fairly small group of people (averaging about 6 to 8) in a fairly short period of time (about 18 months). We feel that this is largely due to the use of Mesa. Pilot consists of approximately 24,000 lines of Mesa, broken into about 160 modules (programs and interfaces), yielding an average module size of roughly 150 lines. The use of small modules and minimal intermodule connectivity, combined with the strongly typed interface facilities of Mesa, aided in the creation of an implementation which avoided many common kinds of errors and which is relatively rugged in the face of modification. These issues are discussed in more detail in [7] and [13].

4. Conclusion

The context of a large personal computer has motivated us to reevaluate many design decisions which characterize systems designed for more familiar situations (e.g., large shared machines or small personal computers). This has resulted in a somewhat novel system which, for example, provides sophisticated features but only minimal protection, accepts advice from client programs, and even boot-loads the machine periodically in the normal course of execution.

Aside from its novel aspects, however, Pilot's real significance is its careful integration, in a single relatively compact system, of a number of good ideas which have previously tended to appear individually, often in systems which were demonstration vehicles not intended to support serious client programs. The combination of streams, packet communications, a hierarchical virtual memory mapped to a large file space, concurrent programming support, and a modular high-level language, provides an environment with relatively few artificial limitations on the size and complexity of the client programs which can be supported.

Acknowledgments. The primary design and implementation of Pilot were done by the authors. Some of the earliest ideas were contributed by D. Gifford, R. Metcalfe, W. Shultz, and D. Stottlemyre. More recent contributions have been made by C. Fay, R. Gobbel, F. Howard, C. Jose, and D. Knutsen. Since the inception of the project, we have had continuous fruitful interaction with all the members of the Mesa language group; in particular, R. Johnsson, J. Sandman, and J. Wick have provided much of the software that stands on the border between Pilot and Mesa. We are also indebted to P. Jarvis and V. Schwartz, who designed and implemented some of the low-level input/output drivers. The success of the close integration of Mesa and Pilot with the machine architecture is largely due to the talent and energy of the people who designed and built the hardware and microcode for our personal computer.

Received June 1979; accepted September 1979; revised November 1979

References

1. Brinch-Hansen, P. The nucleus of a multiprogramming system. *Comm. ACM 13*, 4 (April 1970), 238–241.
2. Boggs, D.R., Shoch, J.F., Taft, E., and Metcalfe, R.M. Pup: An internetwork architecture. To appear in *IEEE Trans. Commun.* (Special Issue on Computer Network Architecture and Protocols).
3. Cerf, V.G., and Kahn, R.E. A protocol for packet network interconnection. *IEEE Trans. Commun. COM-22*, 5 (May 1974), 637–641.
4. Cerf, V.G., and Kirstein, P.T. Issues in packet-network interconnection. *Proc. IEEE 66*, 11 (Nov. 1978), 1386–1408.
5. Farber, D.J., and Heinrich, F.R. The structure of a distributed computer system: The distributed file system. In Proc. 1st Int. Conf. Computer Communication, 1972, pp. 364–370.
6. Habermann, A.N., Flon, L., and Cooprider, L. Modularization and hierarchy in a family of operating systems. *Comm. ACM 19*, 5 (May 1976), 266–272.
7. Horsley, T.R., and Lynch, W.C. Pilot: A software engineering

case history. In Proc. 4th Int. Conf. Software Engineering, Munich, Germany, Sept. 1979, pp. 94–99.

8. Internet Datagram Protocol, Version 4. Prepared by USC/Information Sciences Institute, for the Defense Advanced Research Projects Agency, Information Processing Techniques Office, Feb. 1979.

9. Lampson, B.W. Redundancy and robustness in memory protection. *Proc. IFIP 1974*, North Holland, Amsterdam, pp. 128–132.

10. Lampson, B.W., Mitchell, J.G., and Satterthwaite, E.H. On the transfer of control between contexts. In *Lecture Notes in Computer Science 19*, Springer-Verlag, New York, 1974, pp. 181–203.

11. Lampson, B.W., and Redell, D.D. Experience with processes and monitors in Mesa. *Comm. ACM 23*, 2 (Feb. 1980), 105–117.

12. Lampson, B.W., and Sproull, R.F. An open operating system for a single user machine. Presented at the ACM 7th Symp. Operating System Principles (*Operating Syst. Rev. 13*, 5), Dec. 1979, pp. 98–105.

13. Lauer, H.C., and Satterthwaite, E.H. The impact of Mesa on system design. In Proc. 4th Int. Conf. Software Engineering, Munich, Germany, Sept. 1979, pp. 174–182.

14. Lockemann, P.C., and Knutsen, W.D. Recovery of disk contents after system failure. *Comm. ACM 11*, 8 (Aug. 1968), 542.

15. Metcalfe, R.M., and Boggs, D.R. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM 19*, 7 (July 1976), pp. 395–404.

16. Mitchell, J.G., Maybury, W., and Sweet, R. Mesa Language Manual. Tech. Rep., Xerox Palo Alto Res. Ctr., 1979.

17. Pouzin, L. Virtual circuits vs. datagrams—technical and political problems. Proc. 1976 NCC, AFIPS Press, Arlington, Va., pp. 483–494.

18. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.

19. Ross, D.T. The AED free storage package. *Comm. ACM 10*, 8 (Aug. 1967), 481–492.

20. Rotenberg, Leo J. Making computers keep secrets. Tech. Rep. MAC-TR-115, MIT Lab. for Computer Science.

21. Stern, J.A. Backup and recovery of on-line information in a computer utility. Tech. Rep. MAC-TR-116 (thesis), MIT Lab. for Computer Science, 1974.

22. Sunshine, C.A., and Dalal, Y.K. Connection management in transport protocol. *Comput. Networks 2*, 6 (Dec. 1978), 454–473.

23. Stoy, J.E., and Strachey, C. OS6—An experimental operating system for a small computer. *Comput. J. 15*, 2 and 3 (May, Aug. 1972).

24. Transmission Control Protocol, TCP, Version 4. Prepared by USC/Information Sciences Institute, for the Defense Advanced Research Projects Agency, Information Processing Techniques Office, Feb. 1979.

25. Wulf, W., et. al. HYDRA: The kernel of a multiprocessor operating system. *Comm. ACM 17*, 6 (June 1974), 337–345.

An Overview of the Mesa Processor Architecture

Richard K. Johnsson
John D. Wick
Xerox Office Products Division
3333 Coyote Hill Road
Palo Alto, California 94304

Introduction

This paper provides an overview of the architecture of the Mesa processor, an architecture which was designed to support the Mesa programming system [4]. Mesa is a high level systems programming language and associated tools designed to support the development of large information processing applications (on the order of one million source lines). Since the start of development in 1971, the processor architecture, the programming language, and the operating system have been designed as a unit, so that proper tradeoffs among these components could be made. The three main goals of the architecture were:

- To enable the efficient implementation of a modular, high level programming language such as Mesa. The emphasis here is not on simplicity of the compiler, but on efficiency of the generated object code and on a good match between the semantics of the language and the capabilities of the processor.
- To provide a very compact representation of programs and data so that large, complex systems can run efficiently in machines with relatively small amounts of primary memory.
- To separate the architecture from any particular implementation of the processor, and thus accommodate new implementations whenever it is technically or economically advantageous, without materially affecting either system or application software.

We will present a general introduction to the processor and its memory and control structure; we then consider an example of how the Mesa instruction set enables significant reductions in code size over more traditional architectures. We will also discuss in considerable detail the control transfer mechanism used to implement procedure calls and context switches among concurrent processes. A brief description of the process facilities is also included.

General Overview

All Mesa processors have the following characteristics which distinguish them from other computers:

High Level Language

The Mesa architecture is designed to efficiently execute high level languages in the style of Algol, Mesa, and Pascal. Constructs in the programming languages such as modules, procedures and processes all have concrete representations in the processor and main memory, and the instruction set includes opcodes that efficiently implement those language constructs (e.g. procedure call and return) using these structures. The processor does not "directly execute" any particular high level programming language.

Compact Program Representation

The Mesa instruction set is designed primarily for a compact, dense representation of programs. Instructions are variable length with the most frequently used operations and operands encoded in a single byte opcode; less frequently used combinations are encoded in two bytes, and so on. The instructions themselves are chosen based on their frequency of use. This design leads to an asymmetrical instruction set. For example, there are twenty-four different instructions that can be used to load local variables from memory, but only twenty-one that store into such variables; this occurs because typical programs perform many more loads than stores. The average instruction length (static) is 1.45 bytes.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Compact Data Representation

The instruction set includes a wide variety of instructions for accessing partial and multiword fields of the memory's basic unit, the sixteen bit word. Except for system data structures defined by the architecture, there are no alignment restrictions on the allocation of variables, and data structures are generally assumed to be tightly packed in memory.

Evaluation Stack

The Mesa processor is a stack machine; it has no general purpose registers. The evaluation stack is used as the destination for load instructions, the source for store instructions, and as both the source and destination for arithmetic instructions; it is also used for passing parameters to procedures. The primary motivation for the stack architecture is not to simplify code generation, but to achieve compact program representation. Since the stack is assumed as the source and/or destination of one or more operands, specifying operand location requires no bits in the instruction. Another motivation for the stack is to minimize the register saving and restoring required in the procedure calling mechanism.

Control Transfers

The architecture is designed to support modular programming, and therefore suitably optimizes transfers of control between modules. The Mesa processor implements all control transfers with a single primitive called **XFER**, which is a generalization of the notion of a procedure or subroutine call. All of the standard procedure calling conventions (call by value, call by reference (result), etc.) and all transfers of control between contexts (procedure call and return, nested procedure calls, coroutine transfers, traps, and process switches) are implemented using the **XFER** primitive. To support arbitrary control transfer disciplines, activation records (called *frames*) are allocated by **XFER** from a heap rather than a stack; this allows the heap to be shared by multiple processes.

Process Mechanism

The architecture is designed for applications that expect a large amount of concurrent activity. The Mesa processor provides for the simultaneous execution of up to one thousand asynchronous preemptable processes on a single processor. The process mechanism implements monitors and condition variables to control the synchronization and mutual exclusion of processes and the sharing of resources among them. Scheduling is event driven, rather than time sliced. Interrupts, timeouts, and communication with I/O devices also utilize the process mechanism.

Virtual Memory

The Mesa processor provides a single large, uniformly addressed virtual memory, shared by all processes. The memory is addressed linearly as an array of 2^{32} sixteen-bit words, and, for mapping purposes, is further organized as an array of 2^{24} pages of 256 words each; it has no other programmer visible substructure. Each page can be individually write-protected, and the processor records the fact that a page has been written into or referenced.

Protection

The architecture is designed for the execution of cooperating, not competing, processes. There is no protection mechanism (other than the write-protected page) to limit the sharing of resources among processes. There is no "supervisor mode," nor are there any "privileged" instructions.

Virtual Memory Organization

Virtual addresses are mapped into real addresses by the processor. The mapping mechanism can be modeled as an array of real page numbers indexed by virtual page numbers. The array can have holes so that an associative or hashed implementation of the map is allowed; the actual implementation is not specified by the architecture and differs among the various implementations of the Mesa processor.

Instructions are provided to enable a program (usually the operating system) to examine and modify the virtual-to-real mapping. The processor maintains "write-protected," "dirty," and "referenced" flags for each mapped virtual page which can also be examined and modified by the program.

The address translation process is identical for *all* memory accesses, whether they originate from the processor or from I/O devices. There is no way to bypass the mapping and directly reference a main memory location using a real address. Any reference to a virtual page which has no associated real page (*page fault*), or an attempt to store into a write-protected page (*writeprotect fault*) will cause the processor to initiate a process switch (as described below). The abstraction of faults is that they occur between instructions so that the processor state at the time of the fault is well defined. In order to honor this abstraction, each instruction must avoid all changes to processor state registers (including the evaluation stack) and main memory until the possibility of faults has passed, or such changes must be undone in the event of a fault.

Virtual memory is addressed by either long (two word) pointers containing a full virtual address or by short (one

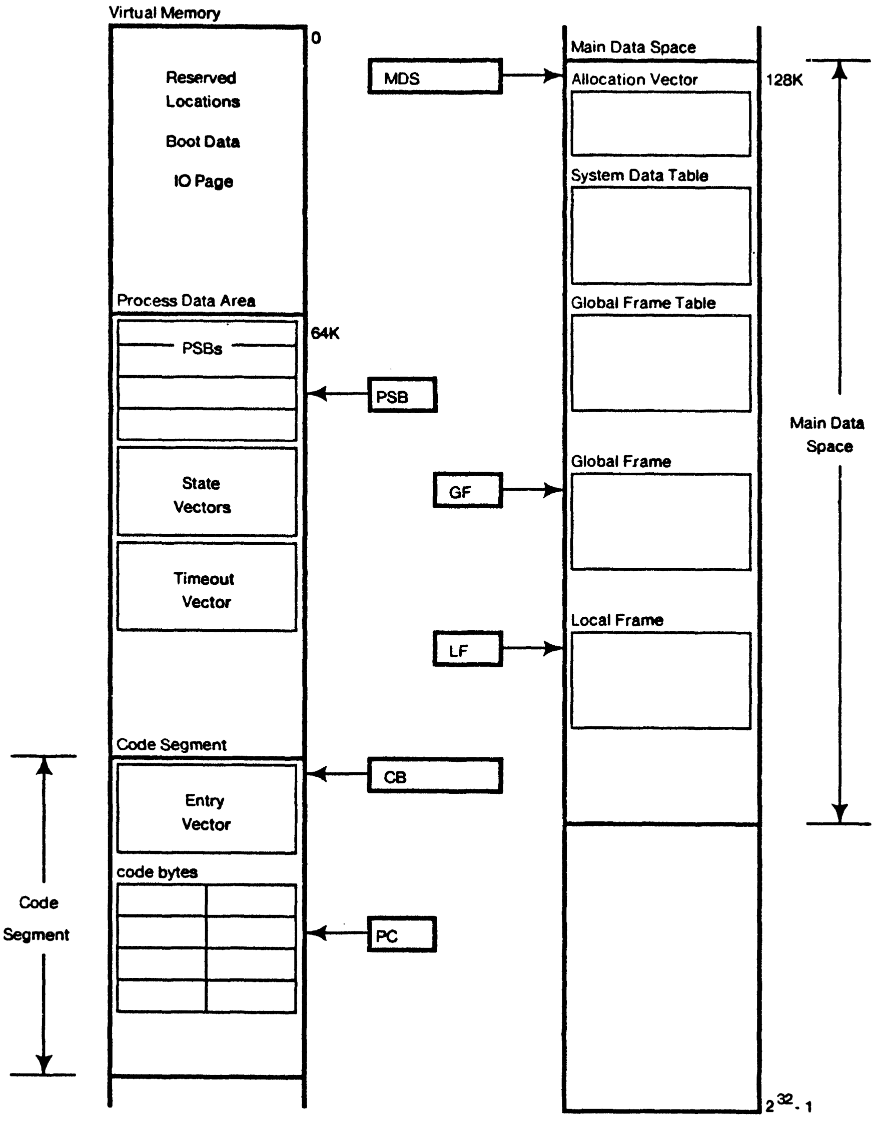


Figure 1. Virtual Memory Structure

word) pointers containing an offset from an implicit 64K word aligned base address. There are several uses of short pointers defined by the architecture:

- The first 64K words of virtual memory are reserved for booting data and communication with I/O devices. Virtual addresses known to be in this range are passed to I/O devices as short pointers with an implicit base of zero.
- The second 64K of virtual memory contains data structures relating to processes. Pointers to data structures in this area are stored as short pointers with an implicit base of 64K.
- Any other 64K region of virtual memory can be a main data space (**MDS**). Each process executes within some **MDS** in which its module and procedure variables are stored; these variables can be referenced by short pointers using as an implicit base the value stored in the processor's **MDS** register.

Code may be placed anywhere in virtual memory, although in general it is not located within any of the three regions mentioned above. A code segment contains read only instructions and constants for the procedures that comprise a Mesa module; it is never modified during normal execution and is usually write-protected. A code

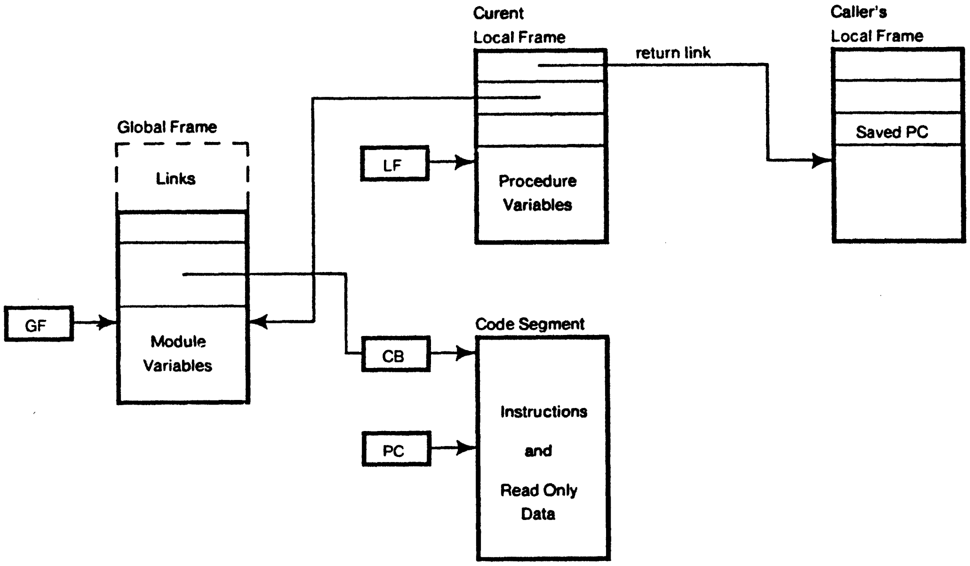


Figure 2. Local and Global Frames and Code Segments

segment is relocatable without modification; no information in a code segment depends on its location in virtual memory.

The data associated with a Mesa program is allocated in a main data space in the form of local and global frames. A global frame contains the data common to all procedures in the module, i.e. declared outside the scope of any procedure. The global frame is allocated when a module is loaded, and freed when the module is destroyed. A local frame contains data declared within a procedure; it is allocated when the procedure is called and freed when it returns.

Any region of the virtual memory, including any main data space, can contain additional dynamically allocated user data; it is managed by the programmer and referenced indirectly using long or short pointers. An MDS also contains a few system data structures used in the implementation of control transfers (discussed below). The overall structure of virtual memory is shown in Figure 1.

Besides enabling standard high level language features such as recursive procedures, multiple module instances, coroutines, and multiple processes, the representation of a program as local data, global data, and code segment tends to increase locality of reference; this is important in a paged virtual memory environment.

Contexts

In addition to a program's variables, there is a small amount of linkage and control information in each frame. A local frame contains a short pointer to the associated global frame and a short pointer to the local frame of its caller (the *return link*). A local frame also holds the code segment relative program counter for a procedure whose execution has been suspended (by preemption or by a call to another procedure). Each global frame contains a long pointer to the code segment of the module. A global frame optionally is preceded by an area called the *link space*, where links to procedures and variables in other modules are stored. This structure is shown in Figure 2.

To speed access to code and data, the processor contains registers which hold the local and global frame addresses (LF and GF), and the code base and program counter (CB and PC) for the currently executing procedure; these are collectively called a context. When a procedure is suspended, the single sixteen bit value which is the MDS relative pointer to its local frame is sufficient to reestablish this complete context by fetching GF and PC from the local frame and CB from the global frame. The management of these registers during context switches is discussed in the section on control transfers below.

The Mesa Instruction Set

As mentioned above, a primary goal of the Mesa architecture is compact representation of programs. The general idea is to introduce special mechanisms into the instruction set so that the most frequent operations can be represented in a minimum number of bytes. See [5] for a description of how the instruction set is tuned to accomplish this goal. Below we enumerate a representative sample of the instruction set.

Many functions are implemented with a family of instructions with the most common forms being a single byte. In the descriptions of instructions below, operand bytes in the code stream are represented by α and β; $\alpha\beta$ represents two bytes that are taken together as a sixteen bit quantity. The suffix **n** on an opcode mnemonic represents a group of instructions with **n** standing for small integers, e.g. **LIn** represents **LI0**, **LI1**, **LI2**, etc. A trailing **B** in an opcode indicates a following operand byte (α); **W** indicates a word ($\alpha\beta$); **P** indicates that the operand byte is a pair of four bit quantities, α.left and α.right.

Operations on the stack. These instructions obtain arguments from and return results to the evaluation stack. Although elements in the stack are sixteen bits, some instructions treat two elements as single thirty-two bit quantities. Numbers are represented in two's complement.

DIS Discard the top element of the stack (decrement the stack pointer).

REC Recover the previous top of stack (increment the stack pointer).

EXCH Exchange the top two elements of the stack.

DEXCH Exchange the top two doubleword elements of the stack.

DUP Duplicate the top element of the stack.

DDUP Duplicate the top doubleword element of the stack.

DBL Double the top of stack (multiply by 2).

unary operations: **NEG**, **INC**, **DEC**, etc.

logical operations: **IOR**, **AND**, **XOR**.

arithmetic: **ADD**, **SUB**, **MUL**.

doubleword arithmetic: **DADD**, **DSUB**.

Divide and other infrequent operations are relegated to a multibyte escape opcode that extends the instruction set beyond 256 instructions.

Simple Load and Store instructions. These instructions move data between the evaluation stack and local or global variables.

LIn Load Immediate n.

LIB α Load Immediate Byte.

LIW $\alpha\beta$ Load Immediate Word.

LLn Load Local n; load the word at offset n from **LF**.

LLB α Load Local Byte; load the word at offset α from **LF**.

SLn Store Local n.

SLB α Store Local Byte.

PLn Put Local n; equivalent to **SLn REC**, i.e. store and leave the value on the stack.

LGn Load Global n; load the word at offset n from **GF**.

LGB α Load Global Byte; load the word at offset α from **GF**.

SGB α Store Global Byte.

LLKB α Load Link; load a word at offset α in the link space.

There are also versions of these instructions that load doubleword quantities. Note that there are no three-byte versions of these loads and stores and no one-byte Store Global instructions. These do not occur frequently enough to warrant inclusion in the instruction set.

Jumps. All jump distances are measured in bytes relative to the beginning of the jump instruction; they are specified as signed eight or sixteen bit numbers.

Jn short positive jumps.

JB α jump -128 to +127 bytes.

JW $\alpha\beta$ long positive or negative jumps.

JLB α compare (unsigned) top two elements of stack and jump if less; also **JLEB**, **JEB**, **JGB**, **JGEB** and unsigned versions.

JEBB α β if top of stack is equal to to α, jump distance in β; also **JNBB**.

JZB α jump if top of stack is zero; also **JNZB**.

JEP α if top of stack is equal to α.left, jump distance in α.right; also **JNEP**.

JIB $\alpha\beta$ at offset $\alpha\beta$ in the code segment find a table of eight bit distances to be indexed by the top of stack; also **JIW** with a table of sixteen bit distances.

Read and Write through pointers. These instructions read and write data through pointers on the stack or stored in local variables.

Rn Read through pointer on stack plus small offset.

RB α Read through pointer on stack plus offset α.

WB α Write through pointer on stack plus offset α.

RLIP α Read Local Indirect; use pointer in local variable α.left; add offset α.right.

WLIP α Write Local Indirect.

RnF α Read Field using pointer on the stack plus n; α contains starting bit and bit count as four bit quantities.

RF α β Read Field using pointer on the stack plus α; β contains starting bit and bit count as four bit quantities.

WF α β Write Field.

RKIB α Read Link Indirect; use the word at offset α in the link space as a pointer.

There are also versions of these instructions that take long pointers and versions that read or write doubleword quantities.

Control Transfers. These instructions handle procedure call and return. Local calls (in the same module) specify the *entry point number* of the destination procedure; external calls (to another module) specify an index of a *control link* in the module's link space (see the section on Control Transfers).

LFCn Local Function Call using entry point n.

LFCB α Local Function Call using entry point α.

EFCn External Function Call using control link n.

EFCB α External Function Call Byte using control link α.

SFC Stack Function Call; use control link from the stack.

RET Return. XFER using the return link in the local frame as the destination; free the frame.

BRK Breakpoint; a distinguished one-byte instruction that causes a trap.

Miscellaneous. These instructions are used to generate and manipulate pointer values.

LAn Local Address n; put the address of local variable n on the stack.

LAB α Local Address Byte; put the address of local variable α on the stack.

LAW αβ Local Address Word; put the address of local variable αβ on the stack.

GAn Global Address n; put the address of global variable n on the stack.

GAB α Global Address Byte; put the address of global variable α on the stack.

GAW αβ Global Address Word; put the address of global variable αβ on the stack.

LP Lengthen Pointer; convert the short pointer on the stack to a long pointer by adding MDS; includes a check for invalid pointers.

An example. Consider the program fragment below. The statement c ← Q[p.f + i] means "call procedure Q, passing the sum of i and field f of the record pointed to by local variable p; store the result in global variable c." The statement RETURN [a[i].c] means "return as the value of the procedure Q field c of the ith record of global array a."

```
Prog: PROGRAM =
   BEGIN

   c: CHARACTER;
   a: ARRAY INTEGER OF RECORD [
      b: BOOLEAN,
      c: CHARACTER,
      s: INTEGER[0..128),
      w: CARDINAL];

   P: PROCEDURE =
      BEGIN
      i: INTEGER = 2;
      p: POINTER TO RECORD [. . . , f: INTEGER];
      . . . ;
      c ← Q[p.f + i];
      . . . ;
      END;

   Q: PROCEDURE [i: INTEGER]
         RETURNS [CHARACTER] =
      BEGIN
      RETURN [a[i].c];
      END;

   END.
```

Below we have shown the code generated for this program fragment in a generalized Mesa instruction set, and then in the current optimized version of the instruction set.

Source	Mesa/Gen	Mesa/Opt
p.f	LL p R f	RLIP (p, f)
i	LI i	LIi
p.f + i	ADD	ADD
Q[..]	LFC q	LFCq
n ←	SG n	SG n
	11 Code Bytes 6 Instructions	7 Code Bytes 5 Instructions
Q:	SL i	PLi
i	REC	
	DBL	DBL
	GAB a	GAa
a[i]	ADD	ADD
a[i].c	RF 0,(1,8)	ROF (1,8)
RETURN	RET	RET
	11 Code Bytes 7 Instructions	7 Code Bytes 6 Instructions

Although this is admittedly a contrived example, it cannot be called pathological, and it does illustrate quite well several of the ways the Mesa instruction set achieves code size reduction. In particular:

- *Use of the evaluation stack.* The stack is the implicit destination or source for load and store operations; instructions can be smaller because they need not specify all operand locations. Since the stack is also used to pass parameters, no extra instructions are needed to set up for the procedure call. Most statements and expressions are quite simple so that the added generality of a general register architecture is a liability rather than an asset.
- *Control transfer primitive.* By using a single, standard calling convention with built-in storage allocation, almost all of the overhead associated with a call is eliminated. There is minimal register saving and restoring.
- *Common operations are single instructions.* Operations that occur frequently are encoded in single instructions. Reading a word from a record given a pointer to the record in a local variable is a good example (**RLIP**). There are similar instructions for storing values through pointers. There are instructions that deal with partial word quantities or that include runtime as well as compile time offsets. Procedure calls are also given single instructions.
- *Frequently referenced variables are stored together.* Most operands are addressed with small offsets from local or global frame pointers or from variable pointers stored in the local or global frame. Using small offsets means that instructions can be smaller because fewer bits are needed to record the offset. The compiler assists by assigning variable locations based on static frequency so that the smallest offsets occur most often.

These last two points are the guiding principles of the Mesa instruction set. If an operation, even a complex one involving indirection and indexing, occurs frequently in "real" programs, then it should be a single instruction or family of instructions. For instruction families with compile time constant operands such as offsets, assigning operand values by frequency increases the payoff of merging small operand values into the opcode or packing multiple values into a single operand byte. There are a small number of cases in which an infrequently used function is provided as an instruction because it is required for technical reasons or for efficiency (e.g. disable interrupts or block transfer).

Control Transfers

The Mesa architecture supports several types of transfers of control, including procedure call and return, nested procedure calls, coroutine transfers, traps and process switches, using a single primitive called XFER [1]. In its simplest form, XFER is supplied with a destination *control link* in the form of a pointer to a local frame; XFER then establishes the context associated with that frame by loading the processor state registers: the PC and global frame pointer GF are obtained from the local frame, and the code base CB is obtained from the global frame. Most control transfer instructions perform some initial setup before invoking the XFER primitive; some specify action to be taken after the XFER. If after the XFER we add code to free the source frame, we have the mechanism for performing a procedure return. On the other hand, if we add code before the XFER to save the current context (only the PC), we have the basic mechanism to implement a coroutine transfer between any two existing contexts.

A process switch is little more than a coroutine transfer, except that it may be preemptive, in which case the evaluation stack must be saved and restored on each side of the XFER. In the Mesa architecture, we have also added the ability to change the main data space on a process switch (see the next section).

The procedure call is the most interesting form of control transfer in any architecture; it is complicated by the fact that the destination context does not yet exist, and must be created out of whole cloth. We represent the context of a not-yet-executing procedure by a control link called a *procedure descriptor.* It must contain enough information to derive all of the following:

The global frame pointer of the module containing the procedure,

The address of the code segment of the module,

The starting PC of the procedure within the code segment, and

The size of the frame to allocate for the procedure's local variables.

Note that in the case of a local call within the current module, only the last two items are needed; the first two remain unchanged.

It is desirable to pack all of this information into a single word, and at the same time make room for a tag bit to distinguish between local frames and procedure descriptors, so the two can be used interchangeably. Then, at the Mesa source level, a program need not concern itself with whether it is calling a procedure or a coroutine.

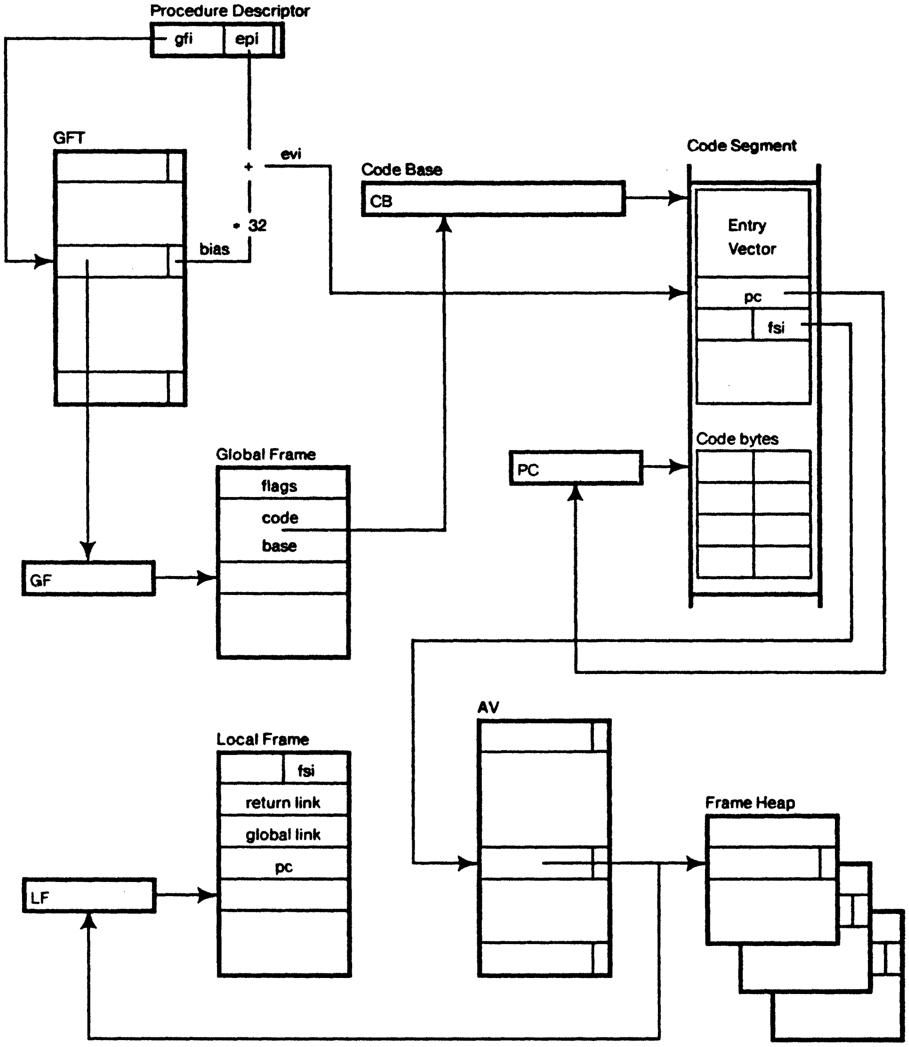


Figure 3. Procedure Calls

The obvious representation of a procedure descriptor would include the global frame address (sixteen bits), the code segment address (thirty-two bits), the starting PC (sixteen bits), and the local frame size (sixteen bits), for a total of eighty bits. We use a combination of indirection, auxiliary tables, and imposed restrictions to reduce this to the required fifteen bits, leaving one bit for the frame/procedure tag (refer to Figure 3).

We eliminate the code segment address by noticing that it is available in the global frame of the destination module, at the cost of a double word fetch.

We replace the PC and frame size by a small (five bit) *entry point index* into a table at the beginning of each code segment containing these values for each procedure. This costs another double word fetch, and limits the number of procedures per module to a maximum of thirty-two. (By an encoding trick, we will increase this to 128 later.)

We replace the global frame pointer by a ten bit index into an MDS-unique structure called the global frame table (GFT); it contains a global frame pointer for each module in the main data space. This costs one additional memory reference per XFER and limits the number of modules in an MDS to 1024 and the number of procedures in an MDS to 32,768.

We obtain our tag bit by aligning local frames to at least even addresses; the low order bit of all procedure descriptors is one.

To increase the maximum number of procedures per module, we first free up two bits in each entry of the global frame table by aligning all global frames on quad word boundaries. We use these two bits to indicate that the entry point index should be increased by 0, 32, 64, or 96 before it is used to index the code segment entry vector. Of course, this requires multiple entries in the global frame table for modules with more than thirty-two procedures.

So, XFER's job in the case of a procedure call is conceptually the same as a simple frame transfer, except that it must pick apart the procedure descriptor and reference all the auxiliary data structures created above. It also needs a mechanism for allocating a new local frame, given its size.

As mentioned above, local frames are allocated from a heap rather than a stack, so that a pool of available frames can be shared among several processes executing in the same MDS. We organize this pool as an array of lists of frames of the most frequently used sizes; each list contains frames of only one size. Rather than actual frame sizes, the code segment entry vector contains frame size indexes into this array, called the allocation vector, or AV (see Figure 3).

Assuming that a frame is present on the appropriate list, it costs three memory references to remove the frame from the list and update the list head. This scheme requires that the frame's frame size index be kept in its overhead words, so that it can be returned to the proper list; it therefore requires four memory references to free a frame. Again we take advantage of the fact that frames are aligned to make use of the low order bits of the list pointers as a tag to indicate an empty list. There is also a facility for chaining a list to a larger frame size list.

In the (rare) event that no frame of the required size (or larger) is available, a trap to software is generated; it may resume the operation after supplying more frame storage. Of course, the frequency of traps depends on the initial allocation of frames of each size, as well as the calling patterns of the application; this is determined by the obvious static and dynamic analysis of frame usage.

Calling a nested procedure involves additional complexity because the new context must be able to access the local variables of the lexically enclosing procedure. The semantics of procedure variables in the Mesa language dictate that the caller of a nested procedure cannot be aware of its context or depth of nesting; all of the complexity must be handled by the called procedure. The implementation of this is beyond the scope of this paper.

Concurrent Processes

The Mesa architecture implements concurrent processes as defined by the Mesa programming language for controlling the execution of multiple processes and guaranteeing mutual exclusion [2].

The process implementation is based on queues of small objects called Process State Blocks (PSBs), each representing a single process. When a process is not running, its PSB records the state associated with the process, including the process's MDS and the local frame it was last executing. If the process was preempted, its evaluation stack is also saved in an auxiliary data structure; the evaluation stack is known to be empty when a process stops running voluntarily (by waiting on a condition or blocking on a monitor). The PSB also records the process's priority and a few flag bits.

When a process is running, its state is contained in the evaluation stack and in the processor registers that hold pointers to the current local and global frames, code segment and MDS. An MDS may be shared by more than one process or may be restricted to a single process. All of these processor registers are modified when a process switch takes place.

Each PSB is a member of exactly one process queue. There is one queue for each monitor lock, condition variable, and fault handler in the system. A process that is not blocked on a monitor, waiting on a condition variable, or faulted (e.g. suspended by a page fault) is on the ready queue and is available for execution by the processor. The process at the head of the ready queue is the one currently being executed.

The primary effect of the process instructions is to move PSBs back and forth between the ready queue and a monitor or condition queue. A process moves from the ready to a monitor queue when it attempts to enter a locked monitor; it moves from the monitor queue to the ready queue when the monitor is unlocked (by some other process). Similarly, a process moves from the ready queue to a condition queue when it waits on a condition variable, and it moves back to the ready queue when the condition variable is notified, or when the process has timed out. The instruction set includes both notify and broadcast instructions, the latter having the effect of moving all processes waiting on a condition variable to the ready queue.

Each time a process is requeued, the scheduler is invoked; it saves the state of the current process in the process's PSB, loads the state of the highest priority ready process, and continues execution. To simplify the task of choosing the highest priority task from a queue, all queues are kept sorted by priority.

In addition to normal interaction with monitors and condition variables, certain other conditions result in process switches. Faults (e.g. page faults or write-protect faults) cause the current process to be moved to a fault queue (specific to the type of fault); a condition variable associated with the fault is then notified. An interrupt (from an I/O device) causes one of a set of preassigned condition variables to be notified. Finally, a timeout causes a waiting process to be moved to the ready queue, even though the condition variable on which it was waiting has not been notified by another process.

Conclusions

The Mesa architecture accomplishes its goals of supporting the Mesa programming system and allowing significant code size reduction. Key to this success is that the architecture has evolved in conjunction with the language and the operating system, and that the hardware architecture has been driven by the software architecture, rather than the other way around.

The Mesa architecture has been implemented on several machines ranging from the Alto [6] to the Dorado [3], and is the basis of the Xerox 8000 series products and the Xerox 5700 electronic printing system. The ability to transport almost all Mesa software (i.e. all except unusual I/O device drivers) among these machines while retaining the advantages of the semantic match between the language and the architecture has been invaluable. The code size reduction over conventional architectures (which averages about a factor of two) has allowed considerable shoehorning of software function into relatively small machines.

Acknowledgments

The first version of the Mesa architecture was designed and implemented by the Computer Science Laboratory of the Xerox Palo Alto Research Center. Butler Lampson was responsible for much of the overall design and many of the encoding tricks. Subsequent development and maintenance have been done by the Systems Development Department of the Office Products Division. Chuck Geschke, Richard Johnsson, Butler Lampson, Roy Levin, Jim Mitchell, Dave Redell, Jim Sandman, Ed Satterthwaite, Dick Sweet, Chuck Thacker, and John Wick have all made major technical contributions.

References

[1] Lampson, B., Mitchell, J., and Satterthwaite, E. On the transfer of control between contexts. *Lecture Notes in Computer Science 19,* (1974).

[2] Lampson, B. W. and Redell, D. D. Experience with processes and monitors in Mesa. *Comm. ACM 23,* 2 (Feb. 1980), 105-117.

[3] Lampson, B. W. *et. al.* The Dorado: A high-performance personal computer—three papers. Tech. Rep. CSL 81-1, Xerox Palo Alto Res. Ctr., 1981.

[4] Mitchell, J. G., Maybury, W., and Sweet, R. Mesa Language Manual. Tech. Rep. CSL 79-3, Xerox Palo Alto Res. Ctr., 1979.

[5] Sweet, R. E. and Sandman, J. G. Empirical Analysis of the Mesa Instruction Set, ACM Symposium on Architectural Support for Programming Languages & Operating Systems, March 1982.

[6] Thacker, C.P. *et. al.* Alto: a personal computer, in *Computer Structures: Readings and Examples,* Second edition, Sieworek, Bell, and Newell, Eds., McGraw-Hill, 1981. Also available as Tech. Rep. CSL 81-1, Xerox Palo Alto Res. Ctr., 1981.

Empirical Analysis of the Mesa Instruction Set

Richard E. Sweet
James G. Sandman, Jr.
Xerox Office Products Division
Palo Alto, California

1. Introduction

This paper describes recent work to refine the instruction set of the Mesa processor. Mesa [8] is a high level systems implementation language developed at Xerox PARC during the middle 1970's. Typical systems written in Mesa are large collections of programs running on single-user machines. For this reason, a major design goal of the project has been to generate compact object programs.

The computers that execute Mesa programs are implementations of a stack architecture [5]. The instructions of an object program are organized into a stream of eight bit bytes. The exact complement of instructions in the architecture has changed as the language and machine micro architecture have evolved.

In Sections 3 and 4, we give a short history of the Mesa instruction set and discuss the motivation for our most recent analysis of it. In Section 5, we discuss the tools and techniques used in this analysis. Section 6 shows the results of this analysis as applied to a large sample of approximately 2.5 million instruction bytes. Sections 7 and 8 give advice to others who might be contemplating similar analyses.

2. Language Oriented Instruction Sets

There has been a recent trend toward tailoring computer architecture to a given programming language. Availability of machines with writeable control stores has accelerated this trend. A recent *Computer* issue [2] contains several general discussions of the subject.

There are at least two reasons for choosing a language oriented architecture: space and time. We can get improved speed by assuring that operations done frequently have efficient implementations. We can get more compact object programs by using variable length opcodes, assigning short opcodes to common operations. The use of variable length encodings based on probabilities is, of course, not new; see the classical papers by Shannon[10]and Huffman [4].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Both space and time optimizations rely on knowledge of the statistical properties of programs. Static statistics are sufficient for code compaction, while dynamic statistics help in the area of execution speed. As most of today's computers have some sort of virtual memory, anything that makes programs smaller tends to speed them up by reducing the amount of swapping.

One of the first published empirical stuaies of programming language usage was by Knuth [6], where he studied FORTRAN programs. Several other studies have also been published, including [1], [12], and [14]. Similar studies have been made of Mesa programs before each change in the instruction set.

Basing an instruction set on statistical properties of programs leads to an asymmetric instruction set. For example, variables are read more often than they are assigned, so it makes sense to have more short load instructions than short store ones; certain short jump distances are more common than others, so variable length jump instructions can make address assignment a rather complicated operation. There is a misconception held by some that a language oriented architecture is one in which the compiler's code generators have a very easy task. Quite the contrary, in a production environment, we are willing to put considerable complexity into code generation in order to generate compact object programs.

There are trade-offs between code compaction and processor complexity. Encoding techniques such as variable bit length opcodes and conditional encoding add to the amount of microcode or hardware needed, and slow down decoding. The Mesa machines use a fixed size opcode (eight bits), and have instructions with zero, one, or two data bytes. A similar architecture was independently proposed by Tanenbaum [12].

The paper by Johnsson and Wick [5] describes the current Mesa architecture.

3. History of the Mesa Instruction Set

Each machine that runs Mesa provides a microcoded implementation of the Mesa architecture. Machines have spanned more than an order of magnitude in processing power, from the Alto [13] to the Dorado [7], with several machines in between. All have a 16 bit word size.

The overall concepts of the Mesa architecture have not changed since 1974, but the exact complement of instructions has changed several times. New language features, such as a larger address space, have required new instructions. New insights into the usage of these language features have allowed more compact encoding of common operations.

The first implementation of Mesa was done in 1974 for the Alto. Peter Deutsch's experience with Byte LISP [3] had shown the feasibility of a byte code interpreter to run on the Alto. A stack architecture was chosen to allow "addressless" instructions. Decisions on stack size and procedure parameter passing, etc. were partially based on statistics gathered on programs written in MPL, a precursor to Mesa that ran on Tenex (and partially forced by the limitations of the Alto hardware). The MPL study is described briefly in Sweet's thesis [11].

In 1976, a reasonable body of Mesa code existed and was analyzed. A study of source programs is described in [11]. There was also a study of the object code. These analyses lead to small changes in the instruction set; in particular to some two byte instructions where the second (operand) byte was divided into two four-bit fields.

It soon became clear that the small 16 bit address space of the original Alto implementation was too restrictive. There were several proposals for adding virtual memory to the Alto, but they were rejected in favor of designing a new machine whose microarchitecture was better suited for Mesa emulation. In 1978, we had a machine with virtual memory, and the type LONG POINTER (32 bits) was added to the language. This, of course, required instructions for dealing with the new pointers: loading, storing, dereferencing, etc. At the same time, 32 bit arithmetic was also added to the language (and Mesa architecture).

4. Experimental Sample

Today, Mesa has reached a significant level of maturity. Our programmers are working in a development environment written completely in Mesa; there are products in the field, such as the Xerox 8000 series, including the Star workstation, that are programmed entirely in Mesa. These are large programs that make extensive use of the virtual memory. Since the LONG POINTER instructions were added to the architecture before we had any body of code using long pointers to analyze, we were sure that there was room for improvement in their encoding. We did not have the resources at this time to completely redesign the instruction set, but we decided that it was worth our while to see if small changes to the instruction set could lead to more compact object programs.

We started with a sample of programs that was representative of all software running under Pilot [9], the Mesa operating system. We had to decide whether to analyze the source code or the object code generated by the then current compiler. We chose to do both, but this paper deals primarily with the object code analysis.

Some changes, such as increasing the stack depth, or adding new instructions for record construction, have significant effects on the code generating strategy in the compiler. These were studied by instrumenting the compiler or producing a new compiler that generated the expanded instruction set.

Most anticipated instruction set changes were sufficiently similar to the existing set that observing patterns in object code was a workable plan. This certainly included decisions about the proper mix of one, two, and three byte instructions for a given function. In fact, the compiler waits until the very last phase of code generation, the peephole optimizer, to choose the exact opcodes. This concentrates knowledge of the exact instruction set in a single place in the compiler.

5. Experimental Plan

The general plan of attack was as follows:

1. Normalize the object code.

   We converted the existing object code into a canonical form. This included breaking the code into straight line sequences, and undoing most peephole optimizations. The sample resulted in 2.5 million bytes of normalized instructions.

2. Collect statistics by pattern matching.

   Patterns took two general forms: compiled in patterns that looked at things like operator pair frequencies, and interactive patterns, where the user could type in a pattern and have the data base searched for that pattern.

3. Propose new instructions.

   Based upon the statistics gathered in step 2, we proposed new instructions.

4. Convert to new opcodes by peephole optimization.

   We wrote a general framework for peephole optimization that read and wrote files in a format compatible with the pattern matching utilities. This allowed us to write procedures that would convert sequences of simple instructions into new fancier instructions.

5. Repeat steps 2 through 4.

   While the statistics from step 2 tell us how many of each new instruction we will get in step 4, the ability to partially convert the data file was helpful for questions of the form "What local variables are we loading when the load is not folded into another instruction?"

Normalization

The version of the Mesa instruction set under analysis used 240 of the possible 256 byte values. Moreover, many of the instructions are single byte encodings of what is logically an operation and an operand value, e.g. "Load Local 6" or "Jump 8." Other instructions replace two or three instruction sequences that are sufficiently common to warrant a more compact encoding. To simplify analysis, all code sequences were transformed into semantically equivalent sequences of a subset of the instructions, comprising slightly over 100 opcode values.

1. Expand out imbedded operand values.

   All instructions with embedded operand values were replaced by a corresponding two or three byte instructions where the operand is given explicitly. For example "Jump 8", a single byte opcode was replaced by the three byte sequence: the "Jump word" opcode, and a two byte operand with a value of 8.

2. Break apart multi-operation opcodes.

   Most complicated instructions were replaced by sequences of equivalent simpler instructions. For example, "Jump Not Zero" was replaced by the sequence "Load 0," "Jump Not Equal." Notable exceptions were the "Doubleword" instructions. These could often have been replaced by two single word instructions, but a major thrust of this analysis was finding out how doublewords were used in the language.

The procedure that did the normalization first made a pass over the code to find the targets of all jumps. These were then sorted so that the normalizing procedure could put a marker byte in the output file between each sequence of straight line code.

The analysis software was written so that the normalization routine could run as a coroutine with any of the pattern matchers, converting object files to a stream of normalized bytes. While not a complete waste of effort, this option was not used when the mass of data became large. The normal mode of operation was to convert a related set of object programs to a single output file, and then use that data file, or a collection of such files, as the input to pattern matching and peephole optimization.

When working with large amounts of data, you should plan for expansion. Consider the format of the code sequence data file. The normalization step reduces the opcodes to a set with approximately a hundred members. On the other hand, the peephole optimization (step 3 above) adds new opcodes. In fact, before we were done we had more than 256 logical opcodes (some of them became two or three byte sequences in the resulting instruction set using an escape sequence). As we desired to have the output of peephole acceptable to the pattern matchers, we used two bytes for each operation "byte" of the stream.

Pattern Matching

The collected files of normalized instructions may now be used to answer questions about language usage. One obvious question is "How many of each opcode do I have?" It is easy to write a routine that reads the data file and counts the opcodes. This was one of a class of generic patterns that we ran on our data file. The set of generic patterns waxed and waned throughout the several months of analysis, but at the end, we found the following patterns most interesting:

1. Static opcode frequency.

   Count the number of occurrences of each opcode.

2. Operands values.

   For each opcode, get a histogram of operand values.

3. Opcode successors.

   For each opcode, get a histogram of the set of next opcodes in the code sequences.

4. Opcode predecessors.

   For each opcode, get a histogram of the set of previous opcodes in the code sequences.

5. Popular opcode pairs.

   Consider the set of all pairs of adjacent opcodes; sort them by frequency.

The reader will doubtless observe that patterns 3, 4, and 5 all report the same information. Patterns 3 and 4 are valuable because, even when the frequency of an opcode pair is not especially high, the conditional probability of one based on the other might be high. Additionally, all three patterns provide information that can suggest additional areas of study, as described below.

We also wrote patterns for finding popular triples, and in fact popular n-tuples, where the search space is seeded with allowed (n-1)-tuple initial strings. These weren't as interesting as we had suspected; we got mountains of n-tuples that occurred only a few times, and we tended to run out of storage. Looking at pairs, along with a knowledge of the language and the compiler's code generation strategies, allowed us to generate patterns that gave us statistics on most interesting multibyte constructs.

User Specified Patterns

For matching of longer patterns, or answering specific questions about instruction use, we preferred not to have to recompile the matching program for every new pattern. We therefore wrote an interactive program where the user typed in a pattern which was parsed, and then matched against the data base. A pattern was a sequence of instructions; each instruction consisted of an operator and its operands. The operator/operands could be given explicitly in the pattern, or a certain amount of "wild carding" was allowed. For wild card slots, we provided the option of collecting statistics on the actual values.

Consider the pattern: `LLB * IN [0..16), RB $`. The instruction `LLB` is a two byte "load local variable" instruction where the second byte gives the offset of the variable in the frame (procedure activation record). Similarly, `RB` says "dereference the pointer on the stack, adding the offset specified by the operand byte." This pattern finds all occurrences of `LLB` followed by `RB` where one of the first sixteen local variables is a pointer being loaded. The $ is a wild card match like the *, except it tells the pattern matcher to gather statistics on the actual operand values for the `RB` instructions. The output of the pattern matcher looked something like this:

```
Total data: 1289310 inst, 2653970 bytes
--------------------------
LLB * IN [0..16), RB $  total: 22813

   value    count     %     cum.%

       0     7575  33.20   33.20
       1     3638  15.94   49.15
       2     2838  12.44   61.59
       3     1700   7.45   69.04
       4     1291   5.65   74.70
       5      823   3.60   78.31
       6      746   3.27   81.58
       7      577   2.52   84.10
      13      344   1.50   85.61
      15      328   1.43   87.05
      10      315   1.38   88.43
      11      283   1.24   89.67
      14      277   1.21   90.89
      12      252   1.10   91.99
```

Figure 1. Sample Pattern Matcher Output

These data tell us that the vast majority of offsets are small. If the first "*" had been a "$", statistics would have been collected on which local variable was loaded as well. The statistics for this field are even more skewed—over 90% of the matches are for locals at offset 0, 1, or 2.

Peephole Optimizer

Based on the statistics gathered by pattern matching, we proposed some new instructions. Some of these new instructions were single byte opcodes that encoded a common operand value of what was logically a two or three byte operation; other new instructions were combinations of operations that occurred frequently in code sequences.

Decisions about the two types of instructions were interrelated. The question "How many single byte 'load local' instructions should we have" is best answered by looking at the load local statistics after any loads have been combined into fancier instructions. We solved this problem by writing a peephole optimizer to convert normalized code sequences into sequences of new instructions. This simplified the patterns needed for decisions and also allowed us to look for patterns involving the new instructions. The actual peephole conversion was done by straightforward case analysis, but the framework that it was built upon is worthy of some discussion.

There are several problems with operating directly on the data files. Variable length instructions cannot be read backward, and some instructions have two operand bytes that are logically a single sixteen bit operand. For this reason, the file reading procedure produced fixed sized Mesa records containing the opcode and an array of parameters, correctly decoding multibyte operands. These were maintained in an array as shown in the figure below.

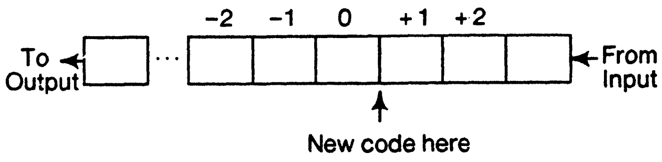


Figure 2. Peephole Optimization Framework

The optimizing procedures typically dealt with the element at index 0, based upon previous instructions ($-i$) and following instructions ($+i$). The range of index values depends on how much history is required in the peephole procedure. For all of our routines, a range from -5 to $+3$ was more than adequate. The framework provided the following operations:

1. Delete *i*.

   Any instruction not already written to the output may be deleted.

2. Output new code.

   New instructions may be generated; they are buffered until the next shift, but will appear just to the right of index 0.

3. Shift left.

   The first new output, or the element at +1, is moved to index 0. Deleted cells are compacted. The buffered new code is moved into the array, possibly pushing some of the previous +*i* elements into a buffer at the right. Any instruction forced out the left is written to the output file. In the case of no change, this reduces to a write, a block transfer in memory, and a read; in the general case, the operation can be rather complicated.

One useful feature of the framework was a display facility that showed the entire array on the screen, with the instruction given as a mnemonic and the parameter array shown only to the extent that the given instruction had parameters. We had several stepping modes, allowing us to see the instructions streaming by, or allowing us to stop and display only when an optimization was to take place.

6. Results

There is certainly not room in this paper to show the complete results of our analysis. Instead, we will show some of the generally interesting results, and go into considerable detail for one class of jump instructions.

Statistics of the Normalized Instruction Data

Table 1 shows the most frequently occurring elements of the original normalized instruction set, together with their statistics.

Op	*count*	*%*	*cum.%*	
LI	208924	16.90	16.90	Load immediate
LL	156848	12.68	29.59	Load local variable
SL	81270	6.57	36.16	Store local variable
REC	64145	5.18	41.35	Recover previous top of stack
LLD	62950	5.09	46.44	Load local doubleword
EFC	55982	4.52	50.97	External function call
J	50726	4.10	55.08	Unconditional jump
R	42328	3.42	58.50	Dereference pointer on stack
SLD	37747	3.05	61.56	Store local doubleword
LA	29205	2.36	63.92	Address of local variable
ADD	28987	2.34	66.26	Add top two words of stack
JNE	25499	2.06	68.33	Jump not equal
RET	24176	1.95	70.28	Return
JE	23335	1.88	72.17	Jump equal
LG	21594	1.74	73.92	Load global variable
LFC	21450	1.73	75.65	Local function call
DADD	20652	1.67	77.32	Doubleword add
LGD	17895	1.44	78.77	Load global doubleword
LLK	16193	1.31	80.08	Load link

Table 1. Frequency of normalized instructions

Table 1 contains some interesting data about language usage. Note that the local variables of procedures are loaded twice as often as they are stored. Doubleword (32 bit) variables are loaded and stored almost half as often as single word ones. Over 6% of the instructions were procedure calls (EFC+LFC), and there were statically three times as many procedure calls as returns. Knowing that the compiler generates a single return from a procedure to facilitate setting breakpoints, we can conclude that procedures are called from an average of three places. Almost 17% of the instructions load constants (LI). Table 2 shows the most popular constants. Bear in mind that some of the loads of constants go away when then are combined into fancier instructions, as we will see in the section on conditional jumps.

Value	*count*	*%*	*cum.%*
0	96652	45.83	45.83
1	29546	14.01	59.84
2	8901	4.22	64.06
3	7094	3.36	67.42
4	5895	2.79	70.22
-1	5553	2.63	72.85
5	3411	1.61	74.47
6	3198	1.51	75.99
8	2220	1.05	77.04
13	2037	0.96	78.01
9	1853	0.87	78.88
7	1841	0.87	79.76

Table 2. Distribution of values for load immediate instructions

The distribution of local variables loaded is shown in Table 3. The reader should be aware that the compiler sorts the local variables by static usage before assigning addresses in the local frame.

Offset	*count*	*%*	*cum.%*
0	63152	40.29	40.29
1	23151	14.77	55.07
2	15125	9.65	64.72
3	10116	6.45	71.17
4	7886	5.03	76.21
5	5837	3.72	79.93
6	4323	2.75	82.69
7	3754	2.39	85.08
8	2718	1.73	86.82
9	2096	1.33	88.16

Table 3. Distribution of offsets of local variables loaded

Analysis of Conditional Jumps

We observe from Table 1 that approximately 4% of the instructions are testing the top two elements of the stack for equality (JE or JNE). It is instructive to describe in some detail the steps that we took in deciding upon what specific instructions to generate for the "Jump Not Equal" class of instructions (JNE).

In Tanenbaum's proposed architecture [11], he allocates 20 one byte instructions and one two byte instruction to each of "Jump Not Equal" and "Jump Not Zero." We would rather not use this much of our opcode space. We looked to see if some of the conditional jumps could be combined with other operations.

From the predecessor data, we observed that 84.7% of the JNE instructions are preceded by a load immediate. We next wrote a pattern that gave a distribution of the values being tested against. Table 4 shows the most frequent values.

Value	*count*	*%*	*cum.%*
0	11792	54.07	54.07
1	2181	10.00	64.07
3	1441	6.60	70.68
2	1032	4.73	75.41
4	390	1.78	77.20
5	314	1.43	78.64
6	238	1.09	79.73
7	232	1.06	80.80
-1	220	1.00	81.81
15	198	0.90	82.72

Table 4. Constants loaded before Jump Not Equal instructions

It comes as no surprise that 0 is the most common value, since 1% of the pre-normalization instructions were "Jump Not Zero," and they were normalized to the sequence LI 0, JNE. We clearly needed to put back in at least the two byte version of this instruction, "Jump Not Zero Byte" (JNZB), where the operand byte specifies the jump distance. The frequency of other small constants lead us to propose a new instrucion: "Jump Not Equal Pair," a two byte instruction where the operand byte is treated as two four bit fields, one a constant, and the other a jump distance. Since jump distances are measured from the first byte of a multibyte instruction, the first reasonable value to jump is 3 bytes—jump over a single byte. When we looked at the jump distances for JNE, however, we saw that 3 byte jumps occur very seldom, and that 5 bytes is the winner, followed by 4 bytes. For this reason, we biased our distances by 4.

By using the data byte to hold a constant between 0 and 15, and a jump distance between 4 and 19, we found 4464 opportunities for the new JNEP instruction. This did not count the situations where the constant value was 0, since they could be encoded by the equally short JNZB instruction.

After the JNZB and JNEP instructions are removed from JNE statistics, there are still over 5000 cases of LI *, JNE left. In these, either the constant value or the jump distance was out of range. We decided to include a "Jump Not Equal Byte Byte" instruction—one with two operand bytes: a value for comparison, and a signed jump distance. This took care of most of the remaining cases.

Now it was time to look at the operands of the remaining JNEB instructions to see if we should have any one byte JNE instructions. The distribution was fairly flat, with the most frequent occurring around 450 times. For this reason, we declined to include single byte JNE instructions.

We also looked at the operands of the JNZB instructions. There were two values, 4 and 5, that were frequent enough to warrant single byte instructions. We added the instructions JNZ3 and JNZ4 (remembering that the jump distance counts from the first byte of the instruction).

In summary, our Not Equal testing is now supported by the following instructions:

Opcode	*bytes*	*count*	*% of* JNE
JNEB	2	4501	18
Jump Not Equal Byte (all byte jumps are signed bytes)			
JNZB	2	8878	35
Jump Non-Zero Byte			
JNEP	2	4464	17
Jump Not Equal Pair (value in [0..15], dist in [4..19])			
JNEBB	3	4742	19
Jump Not Equal Byte Byte (value in [0..255], dist in [-128..127])			
JNZ3	1	1029	4
Jump Non-Zero 3			
JNZ4	1	1885	7
Jump Non-Zero 4.			

Table 5. Jump Not Equal in the new instruction set

The then current opcode set under analysis had a two byte JNZB instruction, a two byte JNEB instruction and eight single byte JNE instructions. The new instruction set has no single byte JNE instructions; most of them occurred in situations where we could combine the jump with the preceding instruction into a new two byte jump. The overall net change was a 13% decrease in code bytes used for not-equal testing compared to the previous instruction set, even though there are four fewer JNE instructions.

Statistics of the Final Instruction Set

From information theory, we know that the best encoding would have all single byte opcodes equally probable. While we do not meet this ideal, the distribution of opcode frequencies is a lot flatter than that of the normalized set. Table 6 shows the most frequently occurring instructions in the new instruction set. Note that of the twenty-two instructions shown in Table 6, fourteen are straightforward single operation opcodes with any operand values given explicitly as additional bytes, six are single byte instructions where operand values are encoded in the opcode, and two are compound operations combined into a single opcode.

Opcode	*count*	*%*	
LI0	46956	4.57	Load immediate 0
LL0	35242	3.43	Load local 0
JB	25587	2.49	Jump byte—a relative, signed byte distance
RET	24256	2.36	Return
LIB	19944	1.94	Load immediate byte—operand is literal value
LL1	18951	1.84	Load local 1
EFCB	17074	1.66	External function call byte—operand specifies a link number
LAB	16706	1.62	Local address byte—load address of a local variable
LI1	16244	1.58	Load immediate 1
REC	15929	1.55	Recover value just popped from stack
SLB	13977	1.36	Store local byte—operand is offset in frame
JZB	13618	1.32	Jump zero byte—pop stack, jump if value = 0
LLD0	13553	1.32	Load local doubleword 0
LLB	13269	1.29	Load local byte—operand is offset in frame
LL2	13132	1.27	Load local 2
ADD	12435	1.21	Add—adds the top two elements of the stack
SLDB	12400	1.20	Store local doubleword byte—operand is offset in frame of first word
LLDB	11222	1.09	Load local doubleword byte—operand is offset in frame of first word
LIW	11205	1.09	Load immediate word—next two bytes are a 16 bit literal
JW	10322	1.00	Jump word—next two bytes are a 16 bit relative jump distance
LLKB	10306	1.00	Load link byte—operand specifies link number
RLIP	9691	0.94	Read local indirect pair—operand has four bits to specify local variable pointer, four bits to specify offset of word relative to that pointer.

Table 6. Most frequent instruction of the new set.

It is interesting to compare the contents of Tables 1, 2, and 3 with that of Table 6. We see that over half of the LI0 instructions have been folded into new instructions. Eighty percent of the LL instructions are either encoded as single byte instructions such as LL0, or folded into more complicated instructions such as RLIP. Several of the most common instructions are load immediate ones (LI*). In fact, the complete frequency data show that almost 13% of all new instructions are some form of load immediate. The most frequent instruction, weighted by instruction size, is JB, a two byte unconditional jump. The most frequent conditional jump is a test against zero, JZB; many of these arise from tests of Boolean variables. Table 7 shows the set of one and two byte load and store local instructions of the new instruction set.

Load instructions—push local variable onto stack.	*bytes*	*total*	*%*
LLn, for n=0,1,2,3,4, 5,6,7,8,9,10,11	1	103402	10.1
LLB	2	13269	1.3
LLDn, for n=0,1,2,3,4, 5,6,7,8,9	1	39989	3.9
LLDB	2	11222	1.1
Store instructions—pop from stack into local variable.			
SLn, for n=0,1,2,3,4, 5,6,7,8,9,10	1	44598	4.3
SLB	2	13977	1.4
SLDn, for n=0,1,2,3,4, 5,6,8	1	21829	2.1
SLDB	2	12400	1.2
Put instructions—store from stack into local variable, don't pop.			
PLn, for n=0,1,2	1	10540	1.0
PLB	2	4195	0.4
PLDn, for n=0	1	2350	0.2
PLDB	2	5238	0.5

Table 7. Distibution of load and store local instructions

Variables outside the first 256 words of the frame are loaded and stored so infrequently that the compiler first generates their address on the stack and then uses the pointer dereferencing instructions. We considered a three byte "Load Local Word" instruction with a sixteen bit offset, but found that "Local Address Word," which loaded the address of a local variable, was more useful. The compiler needs to generate the address of large variables (larger than two words) in order to use the "Block Transfer" instruction; if a variable is at a large offset in the frame, it is probably a large variable as well.

We implemented fewer short instructions for storing local variables than for loading them. Note in Table 6 that four of the single byte load local instructions appear in the top fifteen instructions. Table 7 says that the most frequently referenced (and hence the first in the frame) locals are loaded over twice as often as stored. The variables that are loaded with the two byte LLB are loaded and stored at about the same frequency. The "put" instructions arise primarily at statement boundaries where a variable is stored in one statement and then immediately used in the next; such situations are found by the peephole optimizer of the compiler.

7. Analysis

The most useful patterns for finding sequences of instructions to combine are successors, predecessors, and popular pairs. A simple minded scheme for generating instructions is to start down the list of popular pairs and make a new instruction for each pair until the number of occurrences of that pair reaches some threshold. Of course, each new instruction potentially changes the frequencies of all other pairs containing one of the instructions.

Popular pairs will find many sequences but the data from the successors and predecessors patterns should not be overlooked. For example, the WS (Write Swapped) instruction writes a word in memory using a pointer and value popped from the stack. The REC (Recover) instruction recovers the value that was previously on the stack; after a WS, it recovers the pointer. The successor data showed that 91.4% of the WS instructions were followed by a REC. These two instructions were combined into the PS (Put Swapped) instruction which left the pointer on the stack. We could then eliminate the WS instruction entirely and use the sequence PS, DIS (Discard) the remaining 8.6% of the time.

It helps to know what the compiler does when analyzing patterns. We were suprised to find no occurrences of the pattern LI 0, LI 0. We found them when we looked at popular pairs—the compiler had changed that sequence into LI 0, DUP (Duplicate). This sequence was one of the more popular pairs, which lead us to include the new instruction LIDO (Load Immediate Double Zero).

The pattern showing histograms of operand values is useful for deciding when to fold an operand value into a single byte opcode. Remember that combining instructions may change the operand distribution. For example, the initial operand data for JNEB showed very popular jump distances of 3 through 9 bytes. The original instruction set had single oyte instructions for these jumps. After the analysis, most of these short jumps had been combined into the JNEP or JNEBB instructions. The operand data obtained after peephole optimization did not warrant putting the short JNE instructions back into the instruction set.

8. Implementation Issues

One cannot blindly apply the statistical results of the analysis to decide what instructions to have in the new instruction set. It is necessary to temper these data with knowledge of the compiler, history and expected future trends of language use, and details of the implementations of the instruction set.

There are some operations that are needed in the machine, even though they occur infrequently—the divide operation is an example. Many such operations can be encoded as a single opcode, ESC (Escape), followed by an operand byte specifying the infrequently used operation. This makes available more single byte opcodes for more frequently occurring operations. Mathematically, it makes sense to move any operation to ESC if the available opcode can hold a new operation that gives a net savings in code size. On the other hand, each new opcode adds complexity to the implementation.

Suppose there are two potential new instructions with the same code size savings, one that combines two operations, and the other that combines an operand value with an operation. The latter often results in less complexity in the implementation of the instruction set. In particular, if you already have a LL6 instruction, it typically takes only a single microinstruction to add LL7.

There are many encoding tricks that can be used to save space. Some of these can be decoded at virtually no cost, others are more costly. In the analysis of JNE above, we ended up with an instruction, JNEP, where the operand byte was interpreted as two four bit fields, a literal value and a jump distance. The jump distance was biased, i.e. the microcode added 4 to the value before interpreting the jump. The literal value, on the other hand was unbiased, even though the compiler would not generate the instruction for one of the values. For one of the microprocessors implementing the instruction set, biasing the compared value would have significantly slowed down the execution of the instruction.

In an integrated system such as Mesa, global issues must be considered when making instruction set decisions. For example, many procedures return a value of zero. The statistics showed that an opcode that loads zero and returns would be cost effective. However, the source level debugger takes advantage of the fact that a procedure has a single RET instruction when setting exit breakpoints (all of the procedure's returns jump to this RET). We were unwilling at this time to add the complexity to the debugger of finding all possible return instructions (RET and the new RETZ) in order to set exit breakpoints. Therefore we declined to add this new instruction.

Finally, be careful when analyzing data obtained about an evolving system. Be aware that some common code sequences reflect attempts by older programs to cope with restrictions that are no longer in the architecture. For example, programs written to live in a small address space use different algorithms than those written to live in a large address space.

9. Conclusions

We began our analysis with limited goals: we had a short time in which to make recommendations about changes to the instruction set, we were generally happy with the old instruction set, and we didn't have the resources to handle the necessary rewriting of microcode and compiler that a massive change in the instruction set would require.

Our experience showed that our chosen method, analysis of existing object code, was a workable approach to the problem. Normalization of the code to a canonical form proved valuable for simplifying the subsequent pattern matching used.

We found that simple minded analysis of n-tuples becomes unworkable for n>2, but that informed study of opcode pairs allowed us to postulate longer patterns for study. An interactive pattern matching program was valuable for answering questions about longer patterns.

Our analysis predicted an overall reduction in code size of 12%. We converted the compiler to generate the new instructions and realized the expected savings on a large sample of programs.

10. Acknowledgments

The first opcode analysis of Mesa was done by Chuck Geschke, Richard Johnsson, Butler Lampson, and Dick Sweet. Loretta Guarino Reid helped to develop the current analysis tools, and LeRoy Nelson helped to produce the program sample. The analyses were run on a Dorado, whose processing power was invaluable for handling the large amount of data that we had.

Bibliography

[1] Alexander, W. G., and Wortman, D. B., "Static and Dynamic Characteristics of XPL Programs," *Computer,* vol 8. pp. 41-46, 1975.

[2] Chu, Yaohan, ed., Special issue on Higher-Level Architecture, *Computer,* vol. 14, no. 7, July 1981.

[3] Deutsch, L. Peter, "A LISP machine with very Compact Programs," *Third International Joint Conference on Artificial Intelligence,* Stanford University, 1973.

[4] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE,* vol 40, pp. 1098-1101, September, 1952

[5] Johnsson, Richard K., and Wick, John D., "An Overview of the Mesa Processor Architecture," *Symposium on Architectural Support for Prog. Lang. and Operating Sys.,* Palo Alto, Mar. 1982.

[6] Knuth, Donald E., "An Empirical Study of FORTRAN Programs," *Software—Practice and Experience,* vol. 1, pp. 105-133, 1971

[7] Lampson, Butler W. *et. al., The Dorado: A High-Performance Personal Computer—Three papers.* CSL-81-1, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

[8] Mitchell, James G., Maybury, William, and Sweet, Richard E., *Mesa Language Manual.* Version 5.0. CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, California, 1979.

[9] Redell, David D. *et. al.,* "Pilot: An Operating System for a Personal Computer," *Communications of the ACM,* vol. 23, pp. 81-92, 1980.

[10] Shannon, C. E., "A Mathematical Theory of Communication," *Bell System Technical Journal,* vol 27, pp. 379-423, 623-656, 1948.

[11] Sweet, Richard E., *Empirical Estimates of Program Entropy.* CSL-78-3, Xerox Palo Alto Research Center, Palo Alto, California, 1978.

[12] Tanenbaum, Andrew S., "Implications of Structured Programming for Machine Architecture," *Communications of the ACM,* vol. 31, pp. 237-246, 1978.

[13] Thacker. C. P. *et. al.,* "Alto: A personal computer," in *Computer Structures: Readings and Examples,* Second edition, Sieworek, Bell and Newell, Eds., McGraw-Hill, 1981. Also in Technical Report CSL-79-11, Xerox Palo Alto Research Center, 1979.

[14] Wade, James F., and Stigall, Paul D., "Instruction Design to Minimize Program Size," *Proceedings of the Second Annual Symposium on Computer Architecture,* pp. 41-44, 1975.

Pilot: A Software Engineering Case Study

Thomas R. Horsley and William C. Lynch

Xerox Corporation, Palo Alto, California

Abstract

Pilot is an operating system implemented in the strongly typed language Mesa and produced in an environment containing a number of sophisticated software engineering and development tools. We report here on the strengths and deficiencies of these tools and techniques as observed in the Pilot project. We report on the ways that these tools have allowed a division of labor among several programming teams, and we examine the problems introduced within each different kind of development programming activity (ie. source editing, compiling, binding, integration, and testing).

Introduction

The purpose of this paper is to describe our experiences in implementing an operating system called Pilot using a software engineering support system based on the strongly typed language Mesa [Geschke *et al*, 1977, Mitchell *et al*, 1978], a distributed network of personal computers [Metcalfe *et al*, 1976], and a filing and indexing system on that network designed to coordinate the activities of a score or more of programmers. In this paper we will present a broad overview of our experience with this project, briefly describing our successes and the next layer of problems and issues engendered by this approach. Most of these new problems will not be given a comprehensive discussion in this paper, as they are interesting and challenging enough to deserve separate treatment.

That the Mesa system, coupled with our mode of usage, enabled us to solve the organizational and communication problems usually associated with a development team of a score of people. These facilities allowed us to give stable and non-interactive direction to the several sub-teams.

We developed and used a technique of incremental integration which avoids the difficulties and schedule risk usually associated with system integration and testing.

The use of a Program Secretary, not unlike Harlan Mills' program librarian, proved to be quite valuable, particularly in dealing with situations where our tools had weaknesses. We showed the worth of the program librarian tool, which helped coordinate the substantial parallel activity we sustained; and we identified the need for some additional tools, particularly tools for scheduling consistent compilations and for controlling incremental integrations.

We determined that these additional tools require an integrated data base wherein consistent and correct information about the system as a whole can be found.

Background

Pilot is a medium-sized operating system designed and implemented as a usable tool rather than as an object lesson in operating system design. Its construction was subjected to the fiscal, schedule, and performance pressures normally associated with an industrial enterprise.

Pilot is implemented in Mesa, a modular programming system. As reported in [Mitchell *et al*, 1978], Mesa supports both definitions and implementing modules (see below). Pilot is comprised of some 92 definitions modules and 79 implementation modules, with an average module size of approximately 300 lines.

Pilot consists of tens of thousands of Mesa source lines; it was implemented and released in a few months. The team responsible for the development of Pilot necessarily consisted of a score of people, of which at least a dozen contributed Mesa code to the final result. The coordination of four separately managed sub-teams was required.

There are a number of innovative features in Pilot, and it employs some interesting operating system technology. However, the structure of Pilot is not particularly relevant here and will be reported in a series of papers to come [Redell *et al*, 1979], [Lampson *et al*, 1979].

Development Environment and Tools

The hardware system supporting the development environment is based on the Alto, a personal interactive computer [Lampson 1979], [Boggs, *et al*, 1979]. Each developer has his own personal machine, leading to a potentially large amount of concurrent development activity and the potential for a great degree of concurrent development difficulty. These personal computers are linked together by means of an Ethernet multi-access communication system [Metcalfe *et al*, 1976]. As the Altos have limited disk storage, a file server machine with hundreds of megabytes of storage is also connected to the communications facility. Likewise, high-speed printers are locally available via the same mechanism. The accessing, indexing, and bookkeeping of the large number of files in the project is a serious problem (see below). To deal with this, a file indexing facility (librarian) is also available through the communications system..

The Alto supports a number of significant wideranging software tools (of which the Mesa system is just one) developed over a period of years by various contributors. As one might imagine, the level of integration of these tools is less than perfect, which led to a number of difficulties and deficiencies in the Pilot project. Many of these tools were constructed as separate, almost stand-alone systems.

The major software tools which we employed are described below.

Mesa is a modular programming language [Geschke *et al*, 1977]. The Mesa system consists of a compiler for the language, a Mesa binder for connecting the separately compiled modules, and an interactive debugger for debugging the Mesa programs. Optionally, a set of procedures called the Mesa run-time may be used as a base upon which to build experimental systems.

The language defines two types of modules: *definitions* modules and *implementation* modules. Both of these are compiled into binary (object) form. A definitions module describes an interface to a function by providing a bundle of procedure and data declarations which can be referenced by *client programs (clients)*. Declarations are fully type specified so that the compiler can carry out *strong type checking* between clients and implementation modules. The relevant type information is supplied to the clients (and checked against the implementations) by reading the object modules which resulted from previous compilation(s) of the relevent definitions module(s). The implementing modules contain the procedural description of one or more of the functions defined in some definitions module. Since an implementing module can be seen only through some definitions module, a wide variety of implementations and/or versions is possible without their being functionally detectable by the clients. Thus Mesa enforces a form of information hiding [Parnas, 1972].

The Mesa binder [Mitchell *et al.* 1978] defines another language, called C/Mesa, which is capable of defining *configurations.* These assemble a set of modules and/or sub-configurations into a new conglomerate entity which has the characteristics of a single module. Configurations may be nested and used to describe a tree of modules. Configurations were used in the Pilot project as a management tool to precisely define the resultant output of a contributing development sub-team.

Another software tool is the Librarian. It is designed specifically to index and track the history of the thousands of files created during the project. In addition to its indexing, tracking, and status reporting functions, the Librarian is constructed to adjudicate the frequent conflicts arising between programmers attempting to access and update the same module.

Organization, Division, and Control of the Development Effort

The size of the Pilot development team (itself mandated by schedule considerations) posed the usual organizational and management challenges. With 20 developers, a multi-level management structure was necessary despite the concomitant human communication and coordination problems.

As described below, we chose to use the modularization power of the Mesa system to address these problems, rather than primarily providing the capability for rapid interface change as reported in [Mitchell, 1978]. The resultant methodology worked well for the larger Pilot team. We believe that this methodology will extrapolate to organizations at least another factor of five larger and one management level deeper. A description and evaluation of this methodology are the topics of this section.

Another aspect of our approach was the use of a single person called the Program Secretary, a person not unlike the program librarian described by Harlan Mills [Mills, 1970] in his chief programmer team approach. As we shall describe, the Secretary performed a number of functions which would have been very difficult to distribute *in our environment.* This person allowed us to control and make tolerable a number of problems, described below, which for lack of time or insight we were not able to solve directly.

The Pilot Configuration Tree

We organized Pilot into a tree of configurations isomorphic to the corresponding people tree of teams and sub-teams. The nodes of the Pilot tree are C/Mesa configuration descriptions and the leaves (at the bottom of the tree) are Mesa implementation modules. By strictly controlling the scope (see below) of interfaces (through use of the facilities of the configuration language C/Mesa), different branches of the tree were developed independently. The configuration tree was three to four layers deep everywhere. The top level configuration implements Pilot itself. Each node of the next level down maps to each of the major Pilot development teams, and the next lower level to sub-teams. At the lowest level, the modules themselves were usually the responsibility of one person. This technique of dividing the labor in correspondence with the configuration tree proved to be a viable management technique and was supported effectively by Mesa.

Management Of Interfaces

It quickly became apparent that the *scope* of an interface was an important concept. It is important because it measures the number of development teams that might be impacted by a change to that interface. The *scope* of an interface is defined as the least configuration within which all clients of that interface are confined. This configuration corresponds to the lowest C/Mesa source module which does *not* export the interface to a containing configuration. Thus the scope of a module may be inferred from the C/Mesa sources. The impact of a change to an interface is confined to the development organization or team that corresponds to the node which is the scope of the interface. Thus the scope directly identifies the impacted organization and its sub-organizations.

The higher the scope of an interface, the more rigorously it must be (and was) controlled and the less frequently it was altered since changes to high scope interfaces impact broader organizations. Changing a high level interface was a management decision requiring careful project planning and longer lead times, while a lowest-level interface could be modified at the whim of the (usually) individual developer responsible for it. In general, changing an interface required project planning at the organizational level corresponding to its scope. In particular, misunderstandings between development sub-teams about interface specifications were identified early at design time rather than being discovered late at system integration time. Also obviated were dependencies of one team on another team's volatile implementation details. The result of all of this was 1) the elimination of schedule slips during system integration by the elimination of nasty interface incompatibility surprises and, even stronger, 2) the reduction of system integration to a pro-forma exercise by the (thus enabled) introduction of *incremental integration* (see below).

[Mitchell 1978] reported good success with changing Mesa interface specifications, followed by corresponding revisions in the implementing modules and a virtually bug-free re-integration. While we also found this to be a valid and valuable technique for low-level interfaces (the scope of which corresponded to a three-to-five-person development sub-team), the project planning required to change high-level interfaces affecting the entire body of developers was obviously much greater as was the requirement for stability of such interfaces. It should be noted that the experience reported by [Mitchell 1978] refers to a team of less than a half dozen developers.

Thus, we chose to use the precise interface definition capabilities and strong type checking of the Mesa system differently for the high-level interfaces than for the low-level ones. High-level interfaces were changed only very reluctantly, and were frozen several weeks prior to system integration. This methodology served to decouple one development team from another since each team was assured that they would not be affected by the on going implementation changes made by another developer. Each could be dependent only on the shared definitions modules, and these were controlled quite carefully and kept very stable [Lauer *et al.* 1976].

The Master List

As the system grew, it became painfully obvious that we had no single master description of what constituted the system. Instead we had a number of overlapping descriptions, each of which had to be maintained independently.

One such description was the *working directory* on the file server. Its subdirectory structure was a representation of the Pilot tree. Another description of this same tree was embodied in the librarian data base which indexed the file server. Yet another description was implicit in the C/Mesa configuration files. Early in the project we found it necessary to create a set of command files for compiling and binding the system from source: these files contained still another description of tne Pilot tree.

The addition of a module implied manually updating each of these related files and data bases; it was a tedious and error prone process. In fact, not until the end of the project were all of these descriptions made consistent.

We never did effect a good solution to this problem. We dealt with it in an *ad hoc* fashion by establishing a rudimentary data base called the Master List. This data base was fundamental in the sense that all other descriptions and enumerations were required to conform to it. A program was written to generate from the Master List some of the above files and some of the required data base changes.

A proper solution to this problem requires merging the various lists into a single, coherent data base. This implies that each tool take direction from such a data base and properly update the data base. Since many of the tools were constructed apart from such a system, they would all require modification. Thus the implementation of a coherent and effective data base is a large task in our environment.

Incidentally, this problem was one of those controlled by our Program Secretary. It is quite clear what chaos would have resulted if the updating of the numerous lists described above had not been concentrated in the hands of a single developer.

Pilot Update Cycle

In this section we will examine some of the interesting software engineering aspects of the inner loop of Pilot development. This inner loop occurs after design is complete and after a skeletal system is in place. The typical event consists of making a coordinated set of changes or additions to a small number of modules.

In our environment, a set of modules is fetched from the working directory on the file server to the disk on the developers personal machine. Measures must be taken to ensure that no one changes these modules without coordinating these modifications with the other developers. Usually edits are made to the source modules: the changed modules (and perhaps some others) are recompiled; and a trial Pilot system is built by binding the new object modules to older object modules and configurations. The resulting system is then debugged and tested using the symbolic Mesa debugger and test programs which have been fetched from the working directory. When the system is operating again (usually a few days later), the result is integrated with the current contents of the working directory on the file server, and the changed modules are stored back onto the working directory.

A number of interesting problems arise during this cyclic process:

Consistent Update Of Files

Pilot has been implemented in the context of a distributed computing network. The master copies of the Mesa source modules and object modules for Pilot are kept in directories on a file server on the network. In order to make a coordinated batch of changes to a set of Pilot source files, the developer transfers the current copies of the files from the file server to his local disk, edits, compiles, integrates, and tests them, and then copies them back to the file server.

This simple process has a number or risks. Two developers could try to change the same file simultaneously. A developer could forget to fetch the source, and he would then be editing an old copy on his local disk. He could fetch the correct source but forget to write the updated version back to the file server.

All of these risks were addressed (after the project had begun) by the introduction of the program librarian server. This server indexes the files in the file server and adjudicates access to them via a checkin/checkout mechanism. To guarantee consistency between local and remote copies of files, it provides atomic operations for "checkout and fetch the file" and "checkin and store the file". In the latter case, it also deletes the file from the local disk, thus removing the possibility of changing it without having it checked out (n.b. check-in is prevented unless the developer has the module currently checked out).

Consistent Compilation

Each Mesa object file is identified by its name and the time at which it was created; it contains a list of the identifications of all the other object modules used in its creation (e.g., the definitions module it is implementing). The Mesa compiler will not compile a module in the presence of definitions modules which are not *consistent*, nor will the the binder bind a set of inconsistent object modules. *Consistent* is loosely defined to mean that, in the set of all object modules referenced directly or indirectly, there is no case of more than one version of a particular object module. Each recompilation of a source module generates a new version.

For example, module A may use definitions modules B and C, and definitions module B may also refer to C. It can easily happen that we compile B using the original compilation of C, then we edit the source for Ç "slightly" and recompile, and then we attempt to compile A using C (the new version) and using B (which utilized the original version of C). The compiler has no way of knowing whether the "slight" edit has created compatibility problems, so it "plays safe" and announces a consistency error.

Thus, editing a source module implies that it recompile not only itself, but also all of those modules which include either a direct or an indirect reference to it. Correctly determining the list of modules to be recompiled and an order in which they are to be recompiled is the *consistent compilation* problem.

This "problem" is, in fact, not a problem at all but rather an aid enabled by the strong type checking of Mesa. In previous systems the developer made the decision as to whether an incompatibility had been introduced by a "slight" change. Subtle errors due to the indirect implications of the change often manifested themselves only during system integration or system testing. With Mesa, recompilation is forced via the Mesa systems auditing and judging the compatability of all such changes, thus eliminating this source of subtle problems.

A consistent compilation order for a system (such as Pilot) having a configuration tree can be determined largely by the following analysis:

1) As a direct consequence of the consistency requirement, two modules cannot reference each other, nor can any other cyclical dependencies exist: otherwise the set cannot be compiled. This implies the existence of a well-defined order of compilation.

2) Pilot implementation modules may not refer to each other but must refer only to definitions modules. Therefore only those implementation modules which import recompiled definitions modules need themselves be recompiled. Such implementation modules are recompiled in any order after the recompilation of the definitions modules.

3) An individual definitions module can have compilation dependencies only on modules having the same or a higher scope (from the definition of *scope*). The proper compilation order for definitions modules with different scopes is thus determined by the C/Mesa configuration sources (compile the one with the higher scope first). The Pilot tree of configurations thus imposes a global and fairly restrictive partial ordering on the compilation order of definitions modules. The set of "difficult" compilation dependencies are hence limited and localized to definitions modules of the same scope and described in the same C/Mesa source module.

4) By point 1) there exists a well-defined order of compilation among interfaces possessing the same scope. The compilation order of such sets of interfaces was determined at design time, and, as a matter of policy, the interfaces were not often modified so as to change this ordering.

As an aside, it is clear that it is possible to build a tool which, given that a specified module has been changed, will examine the source modules of the system, determine which modules must be recompiled, and give the order of their recompilation. This is a *Consistent Compilation Tool.* A practical consistent compilation tool need not be omniscient, and it could occasionally cause a module to be compiled when this was not really necessary. Our attempts to build such a tool have been less than completely successful.

Consistent compilation and the design of associated tools is one of those topics which requires a separate paper for a complete treatment.

System Building

As already mentioned, the nodes of the Pilot tree are C/Mesa configuration descriptions. Associated with each is an object module built by binding all associated modules and configurations below the node in the Pilot tree. If a module changes, the system is rebound bottom up through the tree. First, the changed module is bound with its siblings in its parent configuration. Next, the parent is bound with its siblings in its parent's configuration, and so on.

Since the binding must be done on the developers personal computer and the object modules are stored in the file server, it is necessary to fetch from the file server the object modules involved in the binding and to store (after testing [see below]) the newly bound replacements back onto the file server.

The process of fetching (from the file server) the correct siblings for each level of binding is somewhat tedious and error prone. It was not automated except by individual developers using command files. Clearly this information should have been derived automatically from the Master List or from the hypothesized data base.

Each rebinding yields a new version of the object module. The Mesa Binder enforces *consistent binding* by ensuring that only one version of a module or sub-configuration is used either directly or indirectly in a bind. This situation has a number of similarities to the consistent compilation issue. The subtleties of consistent binding also merit treatment in a separate paper.

Integration and Testing

A key software engineering technique which we implemented for the Pilot project was that of *incremental integration.* This kept Pilot integrated and tested in a state which was no more than a few days behind the lead developers.

Each developer integrated and tested changes as he made them. Bugs arose incrementally and were usually restricted to the last set of changes; there was always a current working version of the system. This technique was particularly useful in the early stages of development, when the various teams were quite dependent on what the other teams were doing (i.e., they needed new functions as soon as they were implemented).

Substantial payoff was realized at the time of release. Final *systems integration* and *systems test* proved to be almost trivial: essentially no bugs showed up at this stage. (In many projects it is during this phase that project failure occurs [often with no prior warning]). We were also required to designate several system integrations as internal releases. This provided a continuing sequence of milestones by which progress could be measured.

Key to meeting this objective of incremental integration is the requirement to maintain consistency among the sources and objects in the working directory on the file server. In this case consistent means that the stored modules are consistently compiled and consistently bound and that the resultant *Pilot* object module has been system tested using regression-test programs also stored consistently in this same working directory.

When the Pilot object module had been constructed as described above, the test modules were fetched from the working directory and executed. Nothing was to be stored in the working directory until these tests had been passed. We referred to this whole process as *incremental integration.* (It is intended that the update performed in an incremental integration require only a small amount of work, [i.e., a few man-days]).

The steps in storing a change to Pilot onto the working directory were as follows: 1) test the change on a private version of Pilot in one's local environment. 2) fetch the latest object modules from the working directory, rebuild the system, and test again. 3) via the librarian, acquire sole right to update the master copy. 4) again fetch the latest object modules, rebuild the system and test. 5) write the source and new object modules back onto the working directory. 6) relinquish sole right to update the master copy of the object modules via the librarian.

Steps 3-6 are, of course, necessary to resolve the "store race" which sometimes results from two developers performing incremental integrations in parallel. This procedure permits such parallel incremental integrations provided that they are independent updates and that the order in which they are performed matters not. Step 2) minimizes the time that the universal directory lock is held. Note that if independent and parallel incremental integrations are, in fact, taking place, the modules fetched at step 4) may very well be different than those fetched at step 2). Unless there is a subtle interaction error between the changes of the two concurrent incremental integrations, the test at step 4) will not fail.

While this procedure was effective in managing parallel incremental integrations, its implementation was not very satisfactory. The procedure was executed manually, introducing the potential for error. The fetching and storing were accomplished by command files derived from the Master List rather than from an integrated data base. This situation could be considerably improved by a tool flexing off the appropriate data base. While the overhead of our incremental integration procedure was considerable, the payoff more than justified it.

It should be pointed out that certain classes of changes could not be made as small increments to the current version of Pilot. For example, the changing of high-level interfaces usually had system wide repercussions. These changes were coordinated via internal releases (described below).

Releases

Internal Releases

Internal releases of Pilot were generated when major interface changes were required and also periodically to serve as milestones for the measurement of progress. Internal releases are also useful to assure the consistency of the source and object modules in the directory. In our environment it is possible (through human error) for the source and object modules to be inconsistent with each other due to the lack of unique version identification (e.g., a timestamp) in each source module. (Source modules may be updated and checked back in without being recompiled and rebound.) Ultimately, the only way to guarantee that the sources and objects are consistent is to recompile the source.

To make an internal release, the working directory was write-locked and the system was brought to a guaranteed consistent state by completely recompiling and rebuilding it from source files. The working directory was then tested and finally backed-up to an archive directory. This was all done by the Program Secretary using command files generated from the Master List. Any outstanding changes to high level interfaces were made and frozen several weeks prior to the internal release.

External Releases

An external release is accomplished simply by moving a completed internal release from the working directory to a public test directory. Substantial testing must take place and documentation must be created. At the completion of the testing period, the release should be moved from the public test directory to the proper public release directory.

The execution of this activity was another of the Program Secretary's duties.

Forking

Forking is defined to be the creation of a copy of a system followed by the development of that copy in a fashion inconsistent with the continuing development of the original. This usually means that there is at least one module in which changes must be made which are incompatible between the two branch systems of the fork. We forked at one point early in the development, and found it sufficiently unmanagable that we did not try it again. The extra complexity of maintaining two development paths and the problems of making parallel bug fixes were the major shortcomings of forking. The software engineering procedures described in this paper o not address the problems of forking.

File Management

All of these machinations create file and directory logistics problems. In addition to the main working directory, we also have a public test and a public release directory for the previous external release. Additionally, each external release and each internal release (four or five per external release) are captured on a structured archive directory.

By the end of the project, there were 600 current versions of files stored on just the working directory. This included almost 200 source files, their corresponding object files and symbols files (for the symbolic Mesa Debugger), and a number of other files, including about 150 associated with the test programs. With snapshots of past releases of the system on the archive directory, the actual number of online files approached 5000. The time spent keeping this data base up to date and backed up was very significant. The Master List and command files generated therefrom helped alleviate some of the logistics problems.

Conclusion

What is the upshot of all of this? In short, most of the development environment and control concepts which we used worked well. Of even more interest is the catalog of newly discovered issues which are the ones now constraining our performance. Our systems are never fast enough, particularly in switching from one major task to another. Many tasks which we perform manually cry out to be automated, to have their speed of execution improved but, more important, to have their accuracy increased. The automation of these tasks generally requires a much more integrated data base than is easily constructed in concert with our unintegrated tools.

Successes

What worked really well? The configuration and interface definition capabilities of the Mesa language, the C/Mesa configuration language, and the Mesa Binder worked spectacularly well in allowing us to divide, organize and control our development effort. Such facilities are clearly a must in any modern systems language and implementation.

The important notion of the scope of an interface and the concept of grading and controlling the volatility of each interface according to its scope gave the project the appropriate amount of stability at each organizational level. This stability in turn was one of the enabling factors for incremental integration.

The Program Secretary was clearly a vital post in this scheme. He was instrumental in maintaining the structure and consistency of the Master List, the directories, and the many command files. He was also the prime mover in the execution of both internal and external releases. We do have some vague suspicions, however, that the Program Secretary's main value was in carrying the integrated data base in his head, as we had no automated mechanism for doing so. Certainly the implementation of an effective and integrated data base (of which the Master List would be a part) would reduce his duties considerably.

The program librarian proved its worth in dealing with the problem of updating the working directory consistently. Since this tool was introduced slightly after the beginning of the Pilot project, its impact was clearly observable. It was an important facility in the implementation of the incremental integration technique.

Last, the incremental integration technique itself, despite its largely manual implementation, was quite successful, particularly from the point of view of avoiding a monolithic system integration and test just before a scheduled release.

Deficiencies

With respect to our development environment, the relative autonomy of each of our tools reflected itself in our inability to achieve an integrated data base which would control the tools in a consistent way. It also manifested itself in the relative slowness of the system in switching from one tool to another. Something as elementary as switching from the compiler to the editor requires a fraction of a minute. This slowness raises the cost of the update cycle and effectively imposes a minimum size on a change. The resulting increased batching of changes tends to make the process more error prone.

Maintaining and updating the librarian and Master List data bases was a tedious error-prone operation. In these cases the tools are in a relatively early stage, and not all of the improvements possible to the user interaction have yet been made.

A strong requirement for some additional tools has been established. The requirement for a Consistent Compilation Tool (for determining the modules to recompile and the order of recompilation) was proposed quite some time ago by members of our staff (not participants in the Pilot project), but the necessity for such a tool was not generally accepted at that time; the requirement for a Consistent Compilation Tool is now quite clear. As a result of the Pilot experience. The requirement for a Consistent Binding Tool has been also now established, whereas before the Pilot project this was not a particularly visible requirement. A third addition which would have a large positive impact is a tool for controlling and automating the incremental integration process.

The design and implementation of such tools constitutes a major effort in itself. Central to any solution is an integrated data base.

Acknowledgements

We are particularly indebted to our colleagues Hugh Lauer, Paul McJones, Steve Purcell, Dave Redell, Ed Satterthwaite, and John Wick, who both lived through, and furnished the raw data for, the experiences related in this paper and provided encouragement and constructive criticism for the text itself. We are also indebted to Jude Costello for a her many suggestions for improvement in this paper.

References

Geschke, C. M., Morris, J. H., and Satterthwaite, E. H., "Early Experience with Mesa," *Communications of the ACM* 20 8 (August 1977), pp. 540-553.

Lampson, B. W., "An Open Operating System for a Single User Machine." *to be published, Proceedings - Seventh Symposium on Operating System Principles*, (Dec., 1979)

Lampson, B. W. and Redell, D. D., "Experience with Processes and Monitors in Mesa," *to be published, Proceedings - Seventh Symposium on Operating System Principles*, (Dec., 1979)

Lauer, H.C. and Satterthwaite, E.H., "The Impact of Mesa on System Design," *Proceedings of the 4th International Conference on Software Engineering*, 1979

Metcalfe, R. M., and Boggs, D.R., "Ethernet: Distributed Packet Switching For Local Computer Networks," *Communications of the ACM* 19 7 (July 1976), pp. 395-404

Mills, H. D., *Chief Programmer Teams: Techniques and Procedures*, IBM Internal Report, January 1970

Mitchell, J. G., "Mesa: A Designer's User Perspective", *Spring CompCon 78* (1978), pp. 36-39

Mitchell, J. G., Maybury, W., and Sweet, R. E., "Mesa Language Manual," Technical report CSL-78-1, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, February 1978.

Parnas, D. L., "A Technique For Software Module Specification With Examples," *Communications of the ACM* 15 5 (May 1972), pp. 330-336

Redell D. D., Dalal, Y. K., Horsley, T. H., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C., "Pilot: An operating system for a personal computer," *to be published, Proceedings - Seventh Symposium on Operating System Principles*, (Dec., 1979)

Boggs, D., Lampson, B. W., McCreight, E., Sproull, R., and Thacker, C. P., "Alto: A Personal Computer", Technical report *to be published*, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, 1979.

The Impact of Mesa on System Design

Hugh C. Lauer and Edwin H. Satterthwaite

Xerox Corporation, Palo Alto, California

Abstract

The Mesa programming language supports program modularity in ways that permit subsystems to be developed separately but to be bound together with complete type safety. Separate and explicit interface definitions provide an effective means of communication, both between programs and between programmers. A configuration language describes the organization of a system and controls the scopes of interfaces. These facilities have had a profound impact on the way we design systems and organize development projects. This paper reports our recent experience with Mesa, particularly its use in the development of an operating system. It illustrates techniques for designing interfaces, for using the interface language as a specification language, and for organizing a system to achieve the practical benefits of program modularity without sacrificing strict type-checking.

Mesa is a programming language designed for system implementation. It is used within the Xerox Corporation both by research laboratories as a vehicle for experiments and by development organizations for 'production' programming. Some of our initial experience with Mesa was reported previously [Geschke *et al*, 1977]. Since that time, the language has evolved in several directions and has acquired a larger and more diverse community of users. That community has accumulated a substantial amount of experience in using Mesa to design and implement large systems, a number of which are now operational. It has become increasingly clear that the value of Mesa extends far beyond its enforcement of type-safety within individual programs. It has profoundly affected the ways we think about system design, organize development projects, and communicate our ideas about the systems we build.

This paper reports some of our recent experience with Mesa. It is based primarily upon the development of one particular system—what we refer to as the Pilot operating system—for a small, personal computer. We also draw upon the lessons learned from other systems. These represent a non-trivial amount of programming; a survey of just the authors' immediate colleagues at the end of 1978 uncovered several hundred thousand lines of stable, operational Mesa code. Pilot itself is a 'second generation' client of Mesa. It is the first major system to take advantage of explicit interface and configuration descriptions (discussed below) in its original design. In addition, its designers were able to make careful assessments of earlier systems to discover both the benefits and pitfalls of using Mesa. As a result, we were able to profit from, as well as add to, the accumulated 'institutional learning' about the practical problems of developing large systems in Mesa.

The purpose of this paper is to communicate those lessons, which deserve more emphasis and discussion than they have received to date in the literature. We concentrate upon the impact and adequacy of the Mesa programming language and its influence upon system design; a companion paper [Horsley and Lynch, 1979] focuses upon organizational and management issues. This paper contains three main sections. First, the facilities provided by Mesa for supporting the development and organization of modular programs are discussed. In the second section, we describe the role played by the Mesa interface and configuration languages in system design, particularly from the perspective of Pilot. The final section is a qualitative assessment of the adequacy of Mesa as a system implementation language.

Context

Mesa is both a language and a system. The Mesa language [Mitchell *et al*, 1979] features strict type-checking much like that of PASCAL [Wirth, 1971] or EUCLID [Lampson *et al*, 1977], with similar advantages and disadvantages. In particular, the type-checking moves a substantial amount of debugging from run-time to compile-time. Much has been written on this subject; our views and design decisions have changed little since our earlier report [Geschke *et al*, 1977]. The type system of Mesa pervades all other aspects of the language and system. The latter consists of a compiler, a binder, a source language debugger, and a number of other tools and utilities. The system has been implemented on machines that can be microprogrammed at the register transfer level; thus we have also been able to design and implement a machine architecture specifically tailored to Mesa.

The Pilot operating system upon which this report is based is programmed entirely in Mesa, as are all of its clients. In addition to providing the usual set of operating-system facilities, Pilot implements all of the run-time machinery needed to support the execution of Mesa programs, including itself. The clients are assumed to be friendly and cooperating, not hostile or malicious. Since no debugging takes place on machines that are simultaneously supporting other users, no attempt has been made to provide a strong protection mechanism; instead the goal has been to minimize the likelihood of uncontrolled damage due to residual errors. Pilot was designed and implemented by a core group of six people, with important contributions by members of other groups in specialized areas. By late 1978, the total system consisted of approximately twenty-five thousand lines of Mesa code.

Modularity in Mesa

Systems built in Mesa are collections of modules. The general structure of a Mesa *module* is described in [Geschke *et al*, 1977]. A module declaration defines a data structure consisting of a collection of variables and a set of procedures with access to those variables. In form, a module resembles an ALGOL procedure or SIMULA class. Although the Mesa language enforces no particular style of module usage, a *de facto* standard has evolved. An instance of a module typically manages a collection of *objects*. Each object contains information characterizing its own state. The module instance provides a set of procedures to create, operate upon, and destroy the objects; it contains any data shared by the entire collection (e.g., a table of allocated resources) and perhaps some initialization code also.

Modules communicate with each other via *interfaces.* A module may *import* an interface, in which case it may use facilities defined in that interface and implemented in other modules. We call the importer a *client* of the interface. A module may also *export* an interface, in which case it makes its own facilities available to other modules as defined by that interface. Modules and interfaces provide a basis for separate compilation. In the current version of Mesa, they must in fact be compiled separately; there is no provision for nesting modules within a single compilation unit. Instead a collection of modules is bound together into a *configuration* by the Mesa *binder*; this causes all imported interfaces to be connected to corresponding exported interfaces.

This section contains a brief, simplified description of Mesa interface definitions and of the configuration description language. At the end of the section is a note on the consistent compilation requirement, a constraint that has an important impact on the style and organization of any large system programmed in Mesa.

Interfaces

An interface consists of a sequence of declarations and is defined by a separate compilation unit called a **DEFINITIONS** module. An interface definition can be partitioned into two parts, either of which may be empty. A *static part* declares types and constants that are to be shared between client and implementor. Such interface components have values that are completely specified in the interface definition and can be used by any module with access to that definition. The *operations part* defines the operations available to clients importing the interface. In general, the operations are defined in terms of procedures and signals (dynamically bound unique names, used primarily for exception handling). Only the names and types of operations (including the types of their arguments) are specified in the interface, not their implementations. The operations part of an interface implicitly declares a record type with procedure- and signal-valued fields. We call this an *interface record.*

Figures 1a and 1b are excerpts from the definition of a hypothetical **Channel** interface. They illustrate the declarations typically found in the static and operations parts respectively. Note that each operation is defined to accept or return a **Handle**. Only this type (and its distinguished value **nullHandle**) are of interest to clients. The type **Object** is defined within **Channel** because it is required for the declaration of **Handle**; the attribute **PRIVATE** hides the definition of **Object** from clients of the interface.

A module that uses an interface is said to *import* an instance of the corresponding interface record. Every module lists the interfaces that it imports. In essence, the importer is parametrized with respect to these interfaces. The compiler reads (the compiled version of) each of the imported modules and obtains all of the information necessary to compile the importing module. No knowledge about any implementors of the interfaces is required, but the types and parameters of all references to an interface are fully checked at compile time. The compiler also allocates space in the object program for (the required components of) the imported interface records but does not initialize that space.

Similarly, a module that implements an interface is said to *export* it. Such a module contains procedure and/or signal declarations, each with the **PUBLIC** attribute, for the procedures and/or signals defined in the interface. The compiler ensures that the types in the exporter are assignment compatible with the corresponding fields of the interface record and thus with the types expected by importers of the interface. In essence, instantiation of an exporter yields an instance of the exported interface record in which procedure and signal descriptors have been assigned to the fields. Figure 1c suggests the form of a module that exports **Channel**. In this example, **ChannelImplementation** imports another interface, **Device**, so that it can use operations defined there.

The Mesa binder collects exported interface records and assigns their values to the corresponding interface records of the importers. The rules for collection and assignment are expressed in a configuration description language, which is discussed below.

The Mesa approach to interfaces has several important advantages:

> Once an interface has been agreed upon, construction of the importer and exporter can proceed independently. In particular, interfaces and implementations are decoupled. Not only is information better hidden, but minor programming bugs can be fixed in exporting modules without invalidating a previously established interface and without sacrificing full type-checking across module boundaries.
>
> In large projects, interface specifications are units of communication among design and programming groups (see below under *Interfaces and Specifications*).
>
> Interfaces partition the name space and effectively reduce the number of global names that must be kept distinct within a project.
>
> Interfaces enforce consistency in the connections among modules. The operations upon a class of objects are collected into a single interface, not defined individually and in potentially incompatible ways. An earlier binding scheme, using component-by-component connection, could for example obtain *Allocate* from one module and *Free* from an entirely unrelated one.
>
> Nearly all of the work required for the type-checking of interfaces is done by the compiler.

```
Object: PRIVATE TYPE = RECORD [ . . . ];

Handle: TYPE = POINTER TO Channel.Object;

nullHandle: Channel.Handle = NIL;
```

Figure 1a

```
Create: PROCEDURE [a: arguments] RETURNS [h:
        Channel.Handle];

Operation: PROCEDURE [h: Channel.Handle, a:
        arguments];
```

Figure 1b

```
ChannelImplementation: PROGRAM IMPORTS Device
        EXPORTS Channel =

BEGIN
   . . .
   Create: PUBLIC PROCEDURE [a: arguments]
          RETURNS [h: Channel.Handle] =
   BEGIN
      . . .
   END;
   . . .
   Operation: PUBLIC PROCEDURE [h: Channel.Handle,
          a: arguments] =
   BEGIN
      . . .
   END;
   . . .

END.
```

Figure 1c

This approach should be contrasted with the alternatives. Interfaces in typical assembly-language programming are defined implicitly by attributes attached to symbols scattered through the text of the implementors. The associated binders (linkage editors) and loaders do no type checking and impose little structure on the use of names. Implementations of higher-level languages that are constrained to use the same binders seldom do any better, even when they offer strict intra-module type-checking. *We believe that the type-checking of interfaces is the most important application of the type machinery of Mesa.* In a few PASCAL derivatives (see, for example, [Kieburtz *et al*, 1978]), inter-module type-checking is provided by a special binder, but interfaces are still defined implicitly.

If, importers and exporters refer to inconsistent versions of an interface, the type-checking scheme used by Mesa will fail. The following rather conservative approach has therefore been adopted to guarantee consistency. Whenever a **DEFINITIONS** module is compiled, the compiler generates a unique internal name for the interface (essentially a time stamp). Interfaces are 'the same' for the purposes of binding only if they have the same internal name. This rule is an extension of Mesa's equivalence rule for record types (see [Geschke *et al*, 1977] for further discussion). The compiler places the unique name of the interface in the object code generated for any importer or exporter compiled using that interface. *It is this internal name that is used by the binder to match interfaces.* Thus the binder checks that each interface is used in the *same version* by every importer and exporter.

This strategy has profound effects on the organization and management of large systems. It guarantees complete type-safety and consistency among all modules in a system communicating via a particular interface. On the other hand, it introduces both direct and indirect dependencies among modules to the level of exact versions; establishing consistency can require a great deal of recompilation. Subsequent sections discuss these issues.

Configurations and Binding

Mesa provides a separate *configuration description language*, C/Mesa, for specifying how separately compiled modules are to be bound together to form *configurations*. In the simple cases considered here, configuration descriptions are just lists of modules and (sub)configurations. These descriptions can be nested, however; and the nesting implicitly determines the scope of an interface according to the following rules:

> A component of a configuration (i.e., a module or 'sub-configuration' named within the configuration description) may import an interface if and only if that interface is either imported by the configuration itself or is exported by some component of that configuration.
>
> A configuration may export an interface only if it is exported by one of its components.

The Mesa configuration language is, in fact, more general than this; it has many of the attributes of a 'module interconnection language' as defined by [DeRemer and Kron, 1976]. C/Mesa provides such features as multiple, named instances of interfaces, the assignment of specific instances to specific importers, and the joint or shared implementation of an interface by more than one module. This generality is little used by Pilot and is not discussed here.

A complete system is represented by a hierarchy of configuration descriptions. The scope rules for interfaces permit an interface to be confined to, or excluded from, any given branch of the hierarchy. This can best be illustrated by an example. Let **A**, **B**, **C**, . . . be interfaces, and let **U**, **V**, **W**, **X**, . . . be modules that import and export them as indicated in the comments. Consider the following three C/Mesa configuration descriptions:

```
Config1: CONFIGURATION
        IMPORTS A
        EXPORTS B =
BEGIN
    U;          --imports A, C
    V;          --exports B, C
END.

Config2: CONFIGURATION
        IMPORTS B =
BEGIN
    W;          --imports B, exports C
    X;          --imports B, C
END.

Config3: CONFIGURATION
        IMPORTS A =
BEGIN
    Config1;
    Config2;
END.
```

These configuration descriptions guarantee the following properties of the interfaces (among others):

> The scope of interface **C** in **Config1** is just that configuration; that is, this instance of **C** is known to all components of **Config1** but to no component outside it. Every component of **Config1** which imports **C** will be bound to the same implementation, the one provided by **V**.
>
> The interface **C** in **Config2** is entirely independent of the interface **C** in **Config1**. Whether these two interfaces are different instances of the same interface definition does not matter; *they do not represent the same implementation.* All components of **Config2** that import **C** are bound to the implementation in **W**, not **V**.
>
> Interface **A** is imported into **Config3** (from some yet-to-be-specified, larger configuration), but it is imported only into the branch of the hierarchy represented by **Config1**. Thus no component of **Config2** may import **A**, even though it is known at a higher level in the hierarchy.

The scope rules for configurations provide a powerful tool for controlling the interactions among different parts of the system. Individual subgroups of the development team can define their own interfaces for their own purposes without involving larger units, without having to cope with unexpected calls from unrelated parts of the system, and without having any naming conflicts. Similarly, the organization of the whole system is subject to scrutiny, and *all* interfaces between different parts of the system are fully exposed. No private, undocumented interfaces between low-level components in unrelated branches of the configuration hierarchy can exist.

Pilot makes extensive use of nested configurations to limit the scopes of interfaces. The configuration descriptions are organized as a four-level hierarchy. The highest level exports just the 'public' interfaces defined in the *Functional Specification* (see below). At the next level are the major internal interfaces, used for communication among the major subsystems of Pilot—e.g., input/output, memory management, etc. At lower levels are the interfaces that provide communication within a subsystem. At each level, the interfaces are defined and managed by the group or individual responsible for that configuration. This has been an important factor in keeping the logistics of the project manageable and its schedule reasonable.

Consistent Compilation

When one module is referenced during the course of compiling another, a *compilation dependency* is established. This dependency imposes a partial ordering on a collection of modules. If one module is changed and recompiled, all those that follow it in the ordering must also be recompiled before the collection is again consistent. It is seldom possible to bind a system together so long as any inconsistencies remain. An example illustrates the problem. Let **A** be an interface between modules **U** and **V**. If some change is required in **A**, it is a relatively simple matter to recompile first **A** and then **U** and **V**. These three are then consistent with each other and may be correctly bound together. If only **U** or only **V** were recompiled, the binder would report an error. Suppose, however, that interface **B** uses a type defined in **A**, say as the type of an object pointed to by a field of a record. Suppose further that modules **X** and **Y** communicate using **B**. If **X** also references **A**, any attempt to recompile **X** will fail until **B** is recompiled; then consistent binding requires recompilation of **Y** also. Thus **Y** has an indirect compilation dependency on **A**. Whenever **A** is recompiled, **B**, **X**, and **Y** must be also.

If the number of modules and interfaces in a system is large and if interfaces are evolving, ensuring this strictly-checked consistency becomes a major logistic problem for the project manager. The practical effect of this *consistent compilation* requirement is to force system designers to pay very close attention to when and how modules are updated. Without careful planning and system design, small changes to one or a few interfaces can trigger a recompilation of an entire system. For small systems this is not significant, but for larger projects it is a headache; and it sacrifices many of the operational benefits of modularity. All members of the project must bring their work into phase and 'check in' their outstanding modules. These must then be recompiled in a sequence consistent with the partial order.

In our experience, such a universal recompilation effort nearly always reveals newly introduced inconsistencies and interactions among modules. These must be resolved immediately to allow the recompilation and rebinding to proceed. In the development of Pilot, the recompilation effort took more than a week the first time it was tried; this eventually converged to one-and-one-half or two days once the logistics were debugged. Note that this period is one of enforced inactivity among the members of the project—i.e., they are not able to continue the coding and development of the system being integrated. (Because of the hierarchical structure of Pilot, universal recompilations were rare. In most cases, only the components of one of the nested configurations needed to be recompiled, requiring much less time and effort and affecting fewer people.)

The enforcement of consistent compilation is a result of Mesa's strict type- and version-checking at the module level. We have found that a utility program capable of computing the partial ordering and scheduling the required compilations is of great help in dealing with consistent compilation. Three more drastic alternatives can be imagined:

> First, compatibility of interfaces might be defined recursively in terms of component-by-component compatibility of types and values. This not only involves the binder in much more elaborate type checking but also requires access to large symbol tables during binding. Previous use of this scheme in Mesa demonstrated that it had unacceptable performance and introduced a different set of operational problems.
>
> Second, the compiler and binder could be more discriminating and enforce recompilation of **B**, **X**, and **Y** only when they are actually affected by the changes made to **A**. So far, attempts to do this in ways that do not reduce to the first alternative have not been very successful.
>
> Finally, the onus could be placed on the programmer to recompile **B**, **X**, and **Y** when required. This, however, sacrifices the type-safeness of the Mesa language in one of the places where it is most required: at the interface between two modules. Failure to recompile at the appropriate times will result in a discrepancy between those modules *that is not apparent in any source text.* (In fact, one early version of Mesa used 'unique' names that were incorrectly computed and were not always unique. We found that debugging in the presence of undetected version mismatches was extremely tedious and frustrating.)

The universal recompilation effort is, in effect, the root of a software release policy. Observe that the clients of Pilot itself must be recompiled whenever the external interfaces (those exported by Pilot) are recompiled. This, of course, can be very time-consuming and costly. Therefore, new releases of system software—i.e., new versions with updated interfaces—must be carefully planned and must not be undertaken lightly. 'Maintenance' releases, on the other hand, involve updates only to program modules or strictly internal interfaces. These releases can be absorbed very easily by clients at will and at the cost of a few seconds' or minutes' binding.

While consistent compilation is a logistic problem for the project manager, it can also be a programming benefit. Sometimes it becomes necessary to change an interface, e.g., to change the representation of a shared type or to repartition functions within a system. When this occurs, the type- and version-checking done by the compiler and binder will detect all references to that interface and will expose all parts of the system that must be modified to accommodate the change. *The experience of many projects in Mesa is that once a previously running system has been successfully recompiled and rebound following changes to its internal or external interfaces, it will immediately run with the same reliability as before.* The correct use of strict interface checking is not always obvious, but it must be mastered if the potential benefits are to be obtained. (This parallels our experience with intra-module type checking.)

Programming in the Interface Language of Mesa

Designing interfaces and reducing them to Mesa DEFINITIONS modules are as much acts of programming as designing algorithms and reducing them to executable code. In Mesa, *interfaces are not derived* ex post facto *from the compiled modules constituting a system.* Most of the early 'programming' of Pilot was, in fact, interface programming, and one member of the design team was recognized as the 'interface programmer.' This was a senior member of the group who had the responsibility of ensuring that all interfaces were complete, were consistent with each other, and conformed to project standards.

The notion of an interface programmer did not exist *a priori* but arose from the methods used in the specification and design of Pilot. The original assignment of the interface programmer was to act as editor of the *Functional Specification*, a document describing the external characteristics of the Pilot operating system. However, it soon became apparent that Mesa text was an inherent part of this specification. In addition, while each of the designers contributed an interface and draft specification that was satisfactory for the area of his responsibility, the collection of these had to be integrated into a coherent whole. Thus, the editing task evolved into one resembling programming. The first part of this section illustrates the specification method and the use of the Mesa interface language for defining the external characteristics of Pilot.

One of the most important responsibilities of the interface programmer was to ensure that there were no compilation dependencies between client programs and internal details of Pilot. This is not as easy as it sounds, and we had suffered some bitter experience in previous systems that failed to do this. In one case, a field of a record representing a low-level data structure was accidentally omitted in some code shared between Pilot and the Mesa system itself. The omission did not affect the operation of the Mesa system and was discovered only after most of the testing of a new release of that system had been completed. Unfortunately, the DEFINITIONS module in which the record was located was near the root of the tree of compilation dependencies and, because of schedule commitments, could not be corrected prior to release. As a consequence, all versions of Pilot built on that release of Mesa had to avoid using a fundamental feature of the system architecture. Considerable pains were taken in the subsequent design of the Pilot interfaces to avoid this kind of problem. The second part of this section describes a language feature that reduces the number of such undesirable interactions.

The third part of this section describes how the explicit and strictly checked interfaces of Mesa permitted the functional simulation of Pilot using an older operating system. Contrary to our expectations and previous experience in operating-system design, the conversion of the client programs from the simulated system to the real one was painless.

Interfaces and Specifications

The interface language of Mesa served as the nucleus of the functional specification of the Pilot operating system. This provided a means for defining the scope and character of the system, for documenting it for clients and potential clients, and for focusing the programming effort.

In this particular project, two versions of a *Functional Specification* document were prepared before coding began. The first of these was the culmination of a long study in which the general nature of the system, its goals, and its requirements were identified. This first version of the *Functional Specification* was circulated, and detailed design of the system was begun. Approximately six months later, the second version of the *Functional Specification* was prepared. It incorporated changes and refinements resulting from the design effort and from comments by the client organizations. Following this, Pilot was coded and tested for a period of approximately six months. Finally, the *Functional Specification* was edited to make minor changes and distributed as a programmer's reference manual.

The external specification of Pilot at the functional level is essentially a specification of its public interfaces—i.e., of the types and constants defined by the system, of the procedures that clients can call, and of the signals representing error conditions detected by the system. These interfaces consist of approximately a dozen DEFINITIONS modules representing the major functional areas of the system. They are named according to function, e.g., **File** and **Volume** to describe the file storage system. **Space** to describe memory management, etc.

Figure 2 illustrates two fragments of the *Functional Specification* for the **File** interface. The two parts of the figure illustrate, respectively, the definition of the notion of file capabilities in this system and the operation for creating files. File capabilities are simply and conveniently described in terms of the type **File.ID** (described earlier in the *Functional Specification*). The null value of a file capability is also defined at this point in terms of **File.nullID**, a previously defined null value of **File.ID**, and the empty set. Figure 2a contains all of the information about file capabilities needed by a Mesa programmer designing a client of Pilot, and it illustrates the self-documenting nature of high-level languages such as Mesa.

In Figure 2b, the file creation operation is defined. First, definitions of the procedure and associated error signals are presented as they appear in the interface (note that the **Create** operation defined in the **File** interface can cause signals defined in the **Volume** interface to be raised). Following this is a narrative describing the function of the **Create** operation and the error responses that can occur. The initial state of the file is fully defined (including values of attributes defined elsewhere in the *Functional Specification*). The **type** attribute of the file is defined in conjunction with **Create** and consists of a CARDINAL (i.e., non-negative integer) encapsulated in a record (to create a unique type).

When the *Functional Specification* was completed, the Mesa text was extracted using a text editor, embedded in a prototype DEFINITIONS module and compiled. This revealed a host of minor errors and several circularities. Several omissions were also detected, indicating that the document was incomplete in these respects. These, of course, were corrected both in the interfaces and in the document. The result was twofold: First, the interfaces compiled from the document became the 'official' versions and were used in the implementation. Second, we had confidence that we had adequately documented the whole system as an integral part of its development, in advance and not as a last-minute chore.

A **File.Capability** is an encapsulation of a **File.ID**, along with a set of **permissions**, and is used to represent the right to perform a specific set of operations on a specific file or volume.

```
File.Capability: TYPE = PRIVATE RECORD [
      fID: File.ID, permissions: File.Permissions];
File.Permissions: TYPE = SET OF {read, write, grow,
      shrink, delete};
File.nullCapability: File.Capability = [fID: File.nullID,
      permissions: {}];
```

. . .

Note: Capabilities are *redundant specifications of intent*, not "ironclad" vehicles for protection. If a client program conscientiously limits the permissions in its capabilities to those it expects to use, it will reduce its chances of accidentally destroying its own data in case of minor hardware or software malfunctions.

Figure 2a

```
File.Create: PROCEDURE [volume: Volume.ID, initialSize:
      File.PageCount,
      type: File.Type] RETURNS [file: File.Capability];
File.Error: ERROR [type: File.ErrorType];
File.ErrorType: TYPE = {reservedType, . . . };
Volume.InsufficientSpace: ERROR;
Volume.Unknown: ERROR [volume: Volume.ID];
```

The **Create** operation creates a new file on the specified volume. The operation returns a **File.Capability** (with all permissions) for the new file. If **volume** does not name a volume known to Pilot, **Volume.Unknown** is signaled. The signal **Volume.InsufficientSpace** is generated if there is not enough space on the volume to contain the file. The file initially contains the number of pages specified by **initialSize** (filled with zeros) and has the following other attributes (see §*5.2.5*):

type = **type** parameter to **Create**

immutable = FALSE

temporary = TRUE

The **type** attribute of the file is a tag provided by Pilot for the use of higher level software. . . .

```
File.Type: TYPE = RECORD [CARDINAL];
```

The type of a file is set at the time it is created and may not be changed. . . .

Create may signal **File.Error[reservedType]** if its **type** argument is one of a set of values reserved by the Pilot file implementation.

Figure 2b

From the Pilot experience, we conclude that the combination of Mesa and English in the style we have described is an effective specification tool. There is no formal or mechanical verification method to ensure or 'prove' that the resulting system satisfies the specifications. Nevertheless, our experience has been that human 'verification' is tractable; i.e., the redundancy in this description plus ordinary debugging and testing techniques are sufficient to convince us that the operating system meets its specifications with a reasonable degree of reliability. There were very few cases in which the specifications were misinterpreted or interpreted differently by different people.

A Note on Exporting Types

At the time Pilot was developed, Mesa did not permit modules to export types, only procedures and signals. Constants and types could, of course, be declared in interfaces, but these were known at compile-time to both the importers and the exporters of the interfaces. Unlike procedures and signals, types to be used by one module could not be bound at some later time to types defined by implementation modules elsewhere. Thus every module using instances of a type had to be compiled in an environment in which that type was completely defined, even if the compilation actually required no knowledge of the internal structure of the type.

This restriction introduced unreasonable compilation dependencies between implementation details and the external interfaces of Pilot. This is partly a result of the 'object' style of programming. Consider, for example, the specification of the **Channel** interface introduced previously. The desired interface must provide the type **Channel.Handle** (to be used by Pilot to identify objects describing channels) and a number of operations, such as **Channel.Create**, requiring handles as arguments or returning them as results. Figures 1a and 1b suggest the obvious mapping of these requirements into a Mesa DEFINITIONS module.

While Figure 1 shows the most type-safe way to define a **Channel.Handle**, that interface has a serious operational shortcoming. A client program is not concerned with the actual values of **Channel.Handle**; it only stores them and passes them as parameters. The implementation might use a pointer, an array index, or some other kind of token to represent a **Channel.Handle**. In particular, the implementor should be free to change its representation without impacting **Channel** clients (i.e., without forcing them to be recompiled). Unfortunately, the definition in Figure 1 requires a commitment to the representation of not only **Channel.Handle** but also **Channel.Object** at the time the interface is defined. The only flexibility retained by the implementor is in the algorithms and data structures hidden within **ChannelImplementation**. Thus, fixing bugs and improving the system behavior must be confined to major releases of Pilot, at which time it is expected that all clients will, at least, be recompiled.

Clients also suffer in this approach. Because the representation of the **Channel.Object** is clearly exposed in the interface (even though it is marked PRIVATE), the client programmer is tempted to make unwarranted assumptions about the properties of the objects. Indeed, he can even reference objects directly (using a very simple breach of the type system subject only to administrative control), rather than via the exported procedures of the interface. If the implementation of channels is changed in a subsequent release of Pilot, the client program must be revised, not just recompiled.

In Pilot, introducing implementation details into public interfaces was avoided by carefully placed breaches of the type system. The Mesa version of the **Channel** specification was defined as shown in Figure 3. The declaration in Figure 3a defines **Channel.Handle** to be a unique record type occupying one word of storage. This change has no effect on clients of the interface (see Figure 3b) and does not sacrifice type-checking of channel handles within clients. The actual representation of the **Channel.Handle** is defined in the implementation module as suggested by Figure 3c, where the ***LOOPHOLE*** construct changes the type of its first argument to its second argument, with no change in representation.

Note that the implementation module can be recompiled whenever necessary and rebound to the rest of the system without affecting any interfaces. In particular, the implementation details of the embedded types **Object** and **InternalHandle** (except the latter's size) can be changed at will. The type **InternalHandle** is bound at compile time to the current version of **Object**, but the type **Channel.Handle** is constant for the life of the interface.

This need to breach the type system to minimize compilation dependencies has suggested an improvement to the Mesa language, namely, the exporting of types. To do this, we replace the declaration of **Channel.Handle** in Figure 3a by:

```
Handle: TYPE WITH SIZE [POINTER];
```

This defines **Channel.Handle** to be a type that will be bound at a later time. The size, if specified, grants to an importer the right to use the declaration and assignment operations for that type. An implementation module then exports the type in exactly the same way it exports procedures—by declaring a PUBLIC type with the required name. The compiler checks that the representation of the

```
Handle: TYPE = PRIVATE RECORD[UNSPECIFIED];
```

Figure 3a

```
Create: PROCEDURE [a: arguments] RETURNS [h:
        Channel.Handle];

Operation: PROCEDURE [h: Channel.Handle, a:
        arguments];
```

Figure 3b

```
ChannelImplementation: PROGRAM IMPORTS Device
        EXPORTS Channel =

BEGIN
    Object: TYPE = RECORD [...];
    InternalHandle: TYPE = POINTER TO Object;
    ...
    Create: PUBLIC PROCEDURE [a: arguments]
            RETURNS [h: Channel.Handle] =
    BEGIN
        h1: InternalHandle;
        ...
        h1 ← ...;
        ...
        h ← LOOPHOLE[h1, Channel.Handle];
    END;
    ...
    Operation: PUBLIC PROCEDURE [h: Channel.Handle,
            a: arguments] =
    BEGIN
        h1: InternalHandle = LOOPHOLE[h,
                InternalHandle];
        ...
    END;
    ...

END.
```

Figure 3c

exported type is consistent with the specified size. In Figure 3c, the declaration of **InternalHandle** is replaced by:

```
Handle: PUBLIC TYPE = POINTER TO Object;
```

references to **h1** are replaced by references to **h**, and the assignments using ***LOOPHOLE***s are removed. Breaches of the type system are no longer required in the source code. Clients of **Channel** are unaffected. The binder checks that each exported type is exported by precisely one implementing module and that therefore all modules of a configuration refer to the same type. The only information that needs to be known about the type when the interface is designed or a client is compiled is the size of the representation of that type. Note that an exported type does not have a run-time representation that is available to clients; only the exporter can have any knowledge of the internal structure of that type.

Functional Simulation of the Pilot Operating System

A side effect of the explicit definition of interfaces in separate compilation units is that the same set of interfaces can be implemented by two different systems, and a client can be bound to either one. Provided that corresponding procedures of the two systems implement the same 'semantics,' the client perceives no functional difference between them. This proved to be a valuable feature for the early clients of Pilot. To allow them to begin their own testing before Pilot was complete, a simulated version of Pilot was provided using an older operating system.

This simulated version used exactly the same interfaces (i.e., source and object **DEFINITIONS** modules) as the real one. It consisted of only a small amount of code that converted calls upon Pilot procedures into calls upon old operating-system procedures. In the configuration description of the simulated system, all interfaces of the old system were carefully concealed from clients. For all of the basic operating system facilities, the simulated system and the real one provided virtually identical functional behavior.

The conversion from the simulated environment to the real environment took very little time and effort. In one typical case, an operational version of an application system was demonstrated using the simulated Pilot system. Within two weeks, it was operational on the real system and had successfully executed the same tests as it had in the simulated environment. We attribute this success primarily to the strict interpretation of interface equivalence in Mesa, which, along with the English narrative in the *Functional Specification*, provided sufficient redundancy to permit the implementation of exactly the same functions on two different systems.

This simulated system was not our first attempt. An earlier effort demonstrated that compatibility requires more than a collection of appropriately named operations. In that effort, the old operating system interfaces were not concealed, and the interface modules of the simulated system were only 'approximately' the same as those of the real system. As a result, conversion from the simulated system was a very painful process. Programs that worked well on the simulated system needed extensive revision prior to conversion because (much to the surprise of their implementors) they were found to contain extensive dependencies upon the facilities of the old system, which were still available and visible.

Adequacy of Mesa as a System Programming Language

Previous sections have discussed some potential benefits of high-level languages, particularly in the areas of consistency checking, information hiding, and control of interfaces. These languages offer other well-known advantages, such as greater descriptive power and the suppression of many coding details. A question often raised, however, is whether a language such as Mesa is adequate for implementing components of 'real' systems, especially very low-level programs such as the kernel of an operating system or a device driver. In the case of the Pilot project, the answer is an unqualified 'yes.' *All* system software, including all run-time support for the language, trap handlers, interrupt routines, etc., is coded in Mesa. Even a bootstrap loader that fits into a single 256-word disk block has been written in Mesa.

We must, however, expand upon our answer. In our opinion, several easily overlooked characteristics of our environment contributed substantially to our success. The more important of these are discussed in this section.

Access to the Hardware

Mesa was designed to provide complete but controlled access to the underlying machine. There are several aspects of this. Note that the features described below appear quite infrequently in our code, and the use of most of them is subject to strict administrative control. Each one, however, seems crucial in certain situations.

The programmer has the option of specifying the representation to be used for a particular type. If, for example, the attribute **MACHINE DEPENDENT** is attached to a record declaration, the mapping from the fields of that record to the bit positions in its representation is precisely defined and guaranteed by the compiler. An important use of this attribute is to create structures that exactly match hardware-defined formats; thereafter, interaction with the hardware can be described symbolically. Another use is to specify the formats of records placed on secondary storage media. The Mesa system is still evolving; each release defines a 'virtual machine' that may differ from its predecessors in certain details. Any data structure likely to outlive a particular release is, in effect, dependent upon the virtual machine that created it. Clients are encouraged to recognize this dependency explicitly, either by specifying some fixed format in the original declaration or by inventing their own unique naming scheme for version control. (We have found that using the **MACHINE DEPENDENT** attribute for this purpose is overly tedious; an adequate and more satisfactory alternative would be an attribute enforcing some standard, release-independent format.)

The Mesa language allows explicit breaches of the type system. For essentially the same reasons reported previously [Geschke *et al*, 1977], we have made modest use of such breaches, often to decode representations. Trap handling, for example, sometimes requires inspection of a procedure descriptor as a string of bits. We use another breach, the assignment of an integer to a pointer, to access hardware-defined memory locations. This is one of the rare cases in which a non-pointer value must be assigned to a pointer, and it is almost always done by a constant declaration in an internal **DEFINITIONS** module rather than by an executable program.

The language also includes a low-level 'transfer' primitive, as defined in [Lampson *et al*, 1974], for the transfer of control between contexts. Use of this primitive sacrifices a certain amount of readability and type checking; in conjunction with the heap (non-stack) allocation of frames, however, it has allowed us to experiment with unconventional control structures and to implement the lowest levels of trap handlers, interrupt routines, process schedulers and the like in Mesa.

Finally, Mesa permits bodies of procedures to be specified as sequences of machine instructions. When one of these procedures is called, that sequence is compiled 'inline' in the body of the caller. This facility permits direct access to any special operations of the machine not reflected in the Mesa language, such as I/O control, interrupt masking, etc.

Efficiency

Implementing Mesa on a microprogrammed machine has given us the opportunity to design an instruction set that is well matched to the requirements of the language. In our experience, space has proved more critical than time in most systems for small, personal computers; overall performance depends more upon the amount of primary memory available than on raw execution speed. We have therefore emphasized compactness in our design.

Mesa object code is very compact. This is due primarily to the design of the instruction set itself. We used techniques for program analysis similar to those described in [Sweet, 1978] to discover common operations and to choose efficient encodings of them. The current compiler does little global analysis and optimization, but extensive 'peephole' optimization does contribute further to the compactness of the object code. That code is considerably more compact than the code produced by most other compilers known to us, even those that perform extensive optimization [Wichmann, 1977]. In fact, Mesa object code is often more compact than good assembly code for machines with a conventional instruction set.

We have been careful to define operations that have reasonable implementations in microcode. Execution speed is therefore adequate also; critical timing-dependent code, such as a disk interrupt handler that operates on each sector, can be satisfactorily programmed in Mesa without making undue demands on processor time. We seldom find it necessary to resort to obscure coding styles to achieve fast programs; when bottlenecks are discovered, it is often more profitable to improve the microcode.

Tools

Another essential requirement for programming in a language such as Mesa is a set of tools that maintains the illusion of a Mesa 'virtual machine'. The most notable of these is a powerful source-language debugger, which is routinely used by all Mesa programmers. To allow the debugging of programs such as Pilot itself, our debugger operates on the 'world-swap' principle. Embedded in the program to be debugged is a small *nub* which fields traps, faults, breakpoints, and other conditions. Using a few carefully chosen primitive operations, this nub causes the entire state of the memory to be saved on a file and then loads a debugging system to examine that file. Because of the swap, an errant program cannot damage the debugger, and the debugger is not dependent upon the system being debugged for any of its operations.

The debugger provides the usual facilities: for example, it is possible to display variables, to set conditional breakpoints and to display the state or call stack of any process. All interactions with the programmer are symbolic and are expressed in terms of his original program. Thus each displayed value is formatted according to its type, the original source code is used to specify the location of a breakpoint, etc. In addition, the debugger contains an interpreter of a subset of Mesa; it is valuable for following paths through data structures, setting variables, and calling procedures in the user's memory image.

System Integration

The entire Mesa system is integrated and can evolve to meet new requirements as they are recognized. We can influence all levels of the implementation; to add new facilities or remove a bottleneck, changes can be made where they are most appropriate.

The evolution of processes in Mesa demonstrates this. Earlier versions of the language had no special support for processes in any form. Because of the accessibility of the underlying machine, particularly the transfer primitives, users were able to write their own packages supporting process creation and scheduling. In fact, several such packages were written, each designed to perform well for certain classes of applications. Most of the packages were mutually incompatible, however, and since the language had no notion of a 'process' or 'critical section,' the compiler could offer no help in checking for process-related inconsistencies.

After much discussion of the alternatives, we decided to adopt a 'procedure-oriented model' of processes [Lauer and Needham, 1978] as our standard. The concepts of processes, monitors, and condition variables were added to the language. While it is possible (and, at the lowest levels of the system, sometimes necessary) to ignore these additions, they provide a standard way of programming that is adequate for most applications. The compiler was extended not only to accept these constructs but also to check for obvious inconsistencies in their use. In our initial implementation, process scheduling was done largely in software; this was relatively easy and gave us some flexibility for experimentation. Subsequently, certain parts of the scheduler were moved into microcode to obtain a substantial performance improvement.

Conclusions

The correct uses of the type system, interface language, and configuration language of Mesa are not always obvious. They must be mastered both by individuals and by organizations if the benefits are to be obtained. The benefits, however, can be very substantial. Mesa provides a measure of control over the design and development of systems that greatly exceeds anything else available to us within the resources of a modest-sized development project. As a result, sophisticated systems can be implemented robustly and reliably by small groups within reasonable times. One of the most important practical benefits of Mesa is that the 'easy' bugs are eliminated almost at once and the 'hard' bugs are encountered much sooner in the life of a system.

Acknowledgments

Many of our colleagues have shared experiences and insights that contributed to the ideas expressed in this paper. We are particularly indebted to the other implementors of Mesa and Pilot. Butler Lampson, Charles Simonyi and John Wick made major contributions to the design of Mesa's interfaces and configuration descriptions.

References

DeRemer, F., and Kron, H. H., Programming-in-the-large versus programming-in-the-small, *IEEE Transactions on Software Engineering* SE-2 2 (June 1976), 80-86.

Geschke, C. M., Morris, J. H., and Satterthwaite, E. H., Early experience with Mesa, *Communciations of the ACM* **20** 8 (August 1977), 540-553.

Horsley, T. R., and Lynch, W. C., Pilot: a software engineering case study, submitted to this conference, 1979.

Kieburtz, R. B., Barabash, W., and Hill, C. R., A type-checking program linkage system for Pascal, in *Proceedings 3rd International Conference on Software Engineering*, (Atlanta, May 1978), 23-28.

Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. L., Report on the programming language Euclid, *SIGPLAN Notices* **12** 2 (February 1977), 1-79.

Lampson, B. W., Mitchell, J. G., and Satterthwaite, E. H., On the transfer of control between contexts, in *Lecture Notes in Computer Science, Vol. 19*, G. Goos and J. Hartmannis, Eds., Springer-Verlag, New York (1974), 181-203.

Lauer, H. C., and Needham, R. M, On the duality of operating system structures, in *Proceedings of the Second International Symposium on Operating Systems*, IRIA, Rocquencourt, France, October 1978.

Mitchell, J. G., Maybury, W., and Sweet, R. E., *Mesa Language Manual*, Technical report CSL-79-3, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, April 1979.

Sweet, R. E., *Empirical Estimates of Program Entropy*, Technical report CSL-78-3, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, September 1978.

Wichmann, B., How to call procedures, or second thoughts on Ackermann's function, *Software—Practice and Experience* 7 3 (June-July 1977), 317-329.

Wirth, N., The programming language Pascal, *Acta Informatica* 1 1 (1971), 35-63.

A Retrospective on the Development of Star

Eric Harslem and LeRoy E. Nelson

Xerox Corporation, El Segundo, California

Abstract

Star, officially known as the *Xerox 8010 Information System*, is a workstation for professionals, providing a comprehensive set of capabilities for the office environment. The Star software consists of just over 250,000 lines of code. Its development required 93 work years over a 3.5 year period.

The development of Star depended heavily on the use of powerful personal computers connected to a local-area network and on the use of the *Mesa* language and development environment. An *Integration Service* was introduced to speed up the building of Star and to relieve the programmers of many complex, but repetitive, tasks.

Background

In 1975, the *Systems Development Department* (SDD) was formed inside Xerox to effect the technology transfer of research from the *Xerox Palo Alto Research Center* (PARC) into mainline Xerox office products. Central to this strategy was the development of a superior professional workstation, subsequently named Star, that was to provide a major step forward in several different domains of office automation.

PARC had developed a number of experimental software development tools and office tools based on the Alto personal computer [Thacker 82]. The most important of these tools was a combined modular implementation language and interactive development environment called Mesa [Mitchell 78]. Mesa played a key role in the construction of an integrated, distributed development environment of personal computers connected by a local-area network.

The experimental office tools were the result of several research projects that had produced extensive user-interface design knowledge. But these tools were not consistent in terms of software design, implementation language or user interface. The goal of Star was to use the base of experience accumulated at PARC to build an integrated system with a uniform user interface.

The first release of Star entered the marketplace in 1981. Star provides a relatively powerful personal computer in an elegant professional workstation (electronic *Desktop*) connected to a 10 mega-bits-per-second (Mbps), local-area network (Ethernet [Dalal 81]). Star provides a unique user interface and comprehensive office functions [Smith 82] including multi-font text editing integrated with graphics, sophisticated interactive layout[1], electronic mail, printing and filing, as well as a "personalized" data management system.

In this paper, we generally refer to Star development in the past tense, as if it had ended with the first release. Actually, that first release has already been replaced by later releases. Star is expected to evolve as a product for several more years--adding new functions and encompassing new domains.

Summary of Star Development

From the start of 1977 through the first quarter of 1978, a *functional specification* was written for Star. Product specifications are often overly ambitious due to pressures from the marketing organization. In the case of Star, the problem was compounded because the very charter of the development organization was to be innovative. The staff of designers and implementors aspired to build the ultimate professional workstation. Concurrent with the writing of the functional specification, two experimental prototypes were developed at a cost of approximately 15 work years.

The design of the Star software began in the spring of 1978. The first release was completed early in October of 1981. Over that period of 3.5 years, 93 work years were expended by the Star development group, including project leaders, first- and second-level managers, and software integrators. The staff grew to 20 people within the first six months and then gradually increased to 45 people over the next three years.

The progress of Star development was interrupted several times when we discovered that fundamental

Xerox®, 8010 and Star are trademarks of Xerox Corporation.

[1] This paper was prepared on Star. The layout, fonts and graphics, as you see them, were viewed and edited directly on Star. The original document was printed on a Xerox 8044 laser printer.

components of the system required redesign to meet objectives. For example, three successive designs for interactive text, graphics and page layout were implemented.

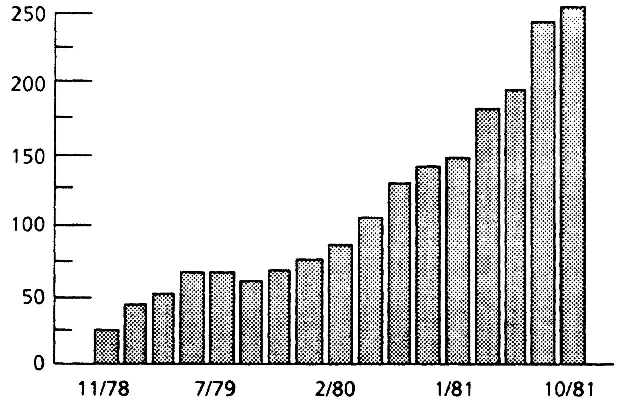
About one-fifth of Star was working by the time it was determined that a subclassing mechanism would be needed to complete the system. Such a mechanism was subsequently designed and implemented [Curry 82]. The existing Star code was converted to use this new mechanism in several phases.

The initial text display and editing implementation supported an 8-bit character space. A little over a year before the end of development, this was replaced with a new design to support a 16-bit character space, including Japanese, Chinese, Russian and European characters.

Star workstation software was built on a specialized operating system known as Pilot [Horsley 79]. For historical reasons, we include a tree-structured file system as part of Star, even though it was split off as a separate project in 1980. There were six major releases of the Pilot operating system and five major releases of the file system during the course of Star development.

Each new release of underlying software required significant changes to Star code–sometimes to take advantage of new functionality, and sometimes to adapt to radically restructured interfaces. The last major restructuring of the file system occurred only three months before the first release of Star.

The barchart below shows the size of the operational corpus of Star code over the period of its development. A plot of functionality would have roughly the same shape. Star actually shrank during the conversion to the subclassing mechanism.



Size of Star over time
(in 1000 lines of code)

The first release of Star (including the file system but not Pilot) was composed of 401 interface (definitions) modules, 440 implementation (code) modules and 88 configuration description files. The modules contained 255,000 lines of code and the compiled system consisted of 908,000 bytes of Mesa opcodes.

Altogether, we probably wrote, integrated and had working, for some period of time, in excess of 400,000 lines of code. Moreover, we expect major components of the existing system to be rewritten in the future, based on our most recently evolved insights.

The *8000* Workstation

The *8000* workstation hardware was developed concurrently with the Star software. The *8000* workstation had two distinct uses for Star development: as an electronic *Office Desktop,* which is the Star product, and as a *Programmer's Desktop*–a vehicle for software development which is currently only available inside Xerox.

The *8000* workstation consists of a processing unit and a user terminal. The *8000* processor was designed to be the processor in all SDD products (the *8000* series). The processing unit is installed in a wheeled cabinet that is small enough (12"x25"x28") to fit conveniently under a table or beside a desk. It consists of a central processor (implemented with bit-slice microprocessor technology), a rigid disk (8 mega bytes (MB) or, with a second cabinet, 24MB), a floppy disk, connections for Ethernet and user terminal, and optional controllers for other devices. The memory system implements a 22-bit virtual and 20-bit real address space. The typical memory configuration includes 512KB of RAM.

The central processor is microprogrammed, so it allows an efficient implementation of a modular, high-level language such as Mesa through the specialization of the Mesa opcodes [Johnsson 82]. The *8000* processor runs about 500,000 Mesa opcodes per second. For typical (non-floating-point) processing, the *8000* processor has about one-half the speed of a DEC VAX 11/780 processor.

The user terminal consists of an 808 raster by 1024 pixel bitmapped display, a keyboard, and a pointing device called a "mouse" [Thacker 82]. The bitmapped display and mouse are of particular significance to the functionality of Star as well as for the use of the *8000* as a programming environment.

The Evolution of Workstation Hardware

From the beginning of Star software development, *every* programmer had his own Alto. Later, Altos were supplemented by a more powerful personal computer, the Dolphin. Finally, *8000* workstations replaced both Altos and Dolphins as the personal computers for programmers. About two-thirds of the Star programmers have used all three of these computers in the course of Star's development.

The following table summarizes the capabilities of these three computers.

	Avail-ability	Virtual memory	Typical memory	Disk capacity	Mesa compile
Alto	1977-81	NO	256KB	5MB	2 min.
Dolphin	1979-81	YES	576KB	24MB	1.5 min.
8000	1981-	YES	512KB	24MB	1 min.

Availability is for personal use of Star developers.
Compile times are for a typical 600 line program.

The Alto was very good hardware for research purposes, but it lacked virtual memory and was therefore not suitable for a sophisticated product such as Star. The Dolphin (also known as the D0) was still research hardware, but its virtual memory capability, disk capacity and user terminal put it much closer to the *8000* than the Alto was. The Dolphin's principal use for Star development was to emulate the *8000* workstation, while the latter was under development. During 1980, Dolphins were a critical resource for Star development. One Dolphin was provided for every 3-4 programmers. While each programmer had an Alto for his programming tasks, all execution and debugging had to be done on shared Dolphins.

An important feature was that all three computers were micro-programmed to run Mesa. All three systems could be used to compile and build systems as well as to execute them. In the terminology of Brooks [Brooks 75], our two early *vehicle* machines were able to imitate the *target* machine, albeit with a certain degradation in performance. This feature accelerated Star development and provided partial independence from the delivery schedule for *8000* hardware.

The Network Development Environment

All three of the hardware systems used in the development of Star were interconnected via the Ethernet. As noted above, these systems came with only modest disk capacity and with no printing capability. Our requirements for large disk storage and printing were fulfilled by *server elements* on the Ethernet.

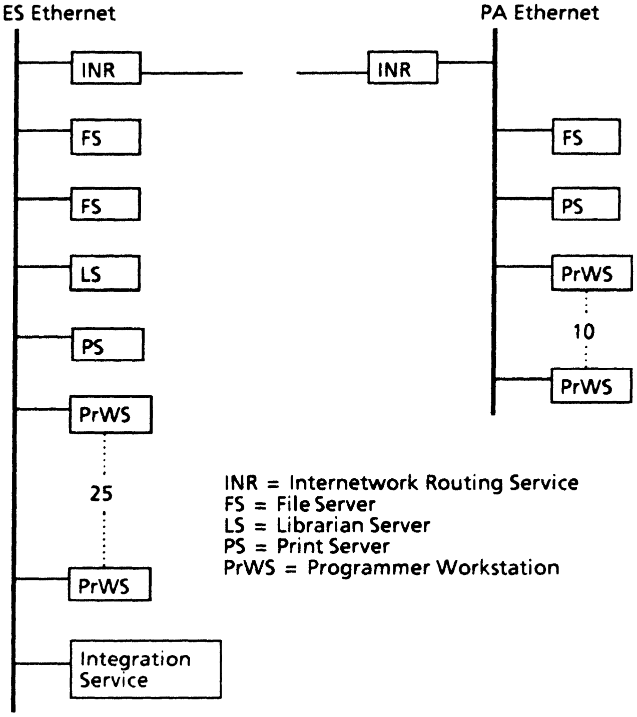
For most of the development of Star, the servers were Alto-based. File servers used in Star development had between 600 and 1,200 MB of online file storage. Print servers provided between 10 and 60 pages-per-minute raster printing to personal systems on the network. Librarian servers mediated access to libraries of source files stored on file servers.

To further complicate the process of developing Star, SDD was geographically split between El Segundo and Palo Alto, California. During 1981, roughly twenty-five programmers were located in El Segundo, and ten were located in Palo Alto. Communication servers connected the El Segundo Ethernet with the Palo Alto Ethernet via a 56 Kbps leased line.

The following diagram shows the portion of the Xerox internetwork (internal, connected Ethernets) that was relevant to the development of Star.

The internetwork made large-scale resources available to geographically-separated, personal computers on a cost-effective basis. It was also essential to the *Integration Service*, which we discuss later.

The network link between El Segundo and Palo Alto provided high-bandwidth, inter-personal electronic communication. Through electronic mail, minor developments, which might otherwise have gone unnoticed, were broadly publicized. This constant flow of information was a primary vehicle for the implementors to keep up to date on such a large project.



The Fragment of the XEROX Internetwork Used for Star Development

The Implementation Language

Much has already been written about the advantages of using Mesa for the development of system software [Horsley 79] [Lauer 79]. Our experience with Mesa in the development of Star supported the following key points made in the earlier papers:

- High-level language. This was the most obvious benefit of Mesa. It provided the ability to "say" a lot in a single statement and improved the readability of the code.

- Interface definitions modules (Defs). Defs allowed use of logical software objects by client modules without any knowledge of, or dependency on, the implementation modules. Such information hiding greatly facilitated the rapid growth of Star, as well as the redesigns and re-implementations of key functions in the system. Star contained 401 Defs.

- Independent compilation of modules. Star was composed of 440 implementation modules, each of which could be recompiled without recompiling any of the others. The compiled modules were bound together into a tree-structured hierarchy of 88 configurations. Only those configurations on a direct path between a recompiled implementation module and the root configuration needed to be rebound. This feature made it possible for programmers to routinely achieve six or more compile-build-test cycles per day.

- Strict type enforcement. The compiler and binder found many errors that traditionally would require tedious debugging. Also, when a working part of the system was converted to new interfaces, type enforcement found most errors, allowing Star to become operational again quickly.

- Coding conventions. Star developmers chose to observe a set of formally specified coding style conventions, particularly in regard to naming types and variables. While not specifically part of Mesa, these conventions were made possible by Mesa's provision for user-defined types. The naming conventions significantly enhanced the readability of Star code.

The Software Development Environment

Many papers on software engineering have noted that software tools are as critical to the effectiveness of an environment as the hardware. The key tools, ranging from the obvious to the sublime, that were part of our development environment, are listed below.[2]

- Compiler. Read interface and implementation source modules and produced object files called Binary Configuration Descriptions (BCDs).
- Binder. Merged implementation module BCDs and/or configuration BCDs into configuration BCDs.
- IncludeChecker. Examined the partial ordering of dependencies in a set of modules and/or configurations to determine which ones must be recompiled and/or rebound, and in what order, to insure type compatibility.
- Packager. Rearranged code within bound configurations BCDs to change swapping performance.
- MakeBoot. Produced a "boot file" from a BCD which contains Pilot. A boot file was the basic executable object on *8000* systems.
- CoPilot. Provided interactive, *source-level* debugging for Mesa programs, including a variety of performance monitoring and debugging tools.
- Adobe. Submited problem reports to a data base on a file server, maintained the data base, and generated data for status reports. Was used extensively to keep track of problems with hardware as well as problems with software.
- Bravo/Editor/Formatter/Find/Waterlily. Created and manipulated Mesa source programs and documentation, including string searches and line-by-line file comparisons. In the future, Star itself will be used for many of these same purposes.
- Lister. Produced human-readable listings of BCD contents in many different formats.
- Access/Librarian. Checked modules in and out of a source library maintained on a file server via a librarian server and updated the librarian server's data base.
- FTP/FileTool/Chat/Brownie. Transfered data files between workstations and file servers.
- Empress/Print. Printed source listings, documentation, etc. on print servers.
- Laurel/Hardy. Used to compose, send, receive and organize electronic mail.

As mentioned previously [Horsley 79], the original Alto-based environment was highly fragmented. Each tool was a separate context to and from which the programmer had to switch. This was burdensome, due as much to the attention and keystrokes required as to the time consumed. Furthermore, these tools had been developed by many different groups. They had radical differences in user interfaces, which taxed even the adaptability of systems programmers. These programmers frequently cited the context switching and differing user interfaces as major aggravations, if not measurable deterrents to progress.

Over the years, an integrated environment–a *Programmer's Desktop*–was developed. This environment provided editing, compiling, binding, debugging, etc. in multiple co-existing *windows* on the *Desktop*. This integrated environment was only available for the last few months of Star development.

Personal Computers for Programmers

The increasing cost of software as a portion of overall computer system development is a well known fact of life. SDD was committed from the start to maximize the effectiveness of a relatively scarce and expensive resource–programmers.

Gutz, Wasserman and Spier present a thorough case for providing professional programmers with personal workstations, in the context of a network of larger-scale services [Gutz 81]. The Star experience supported their case. The personal environment was a key to the high productivity of Star development. The advantages included the following:

- Vehicle machine ≡ Target machine. This condition held at two points during Star's development: at the beginning and at the end. Almost any degree to which a personal vehicle machine can approximate the target machine is a major benefit. The one year when it was necessary to "sign up" for Dolphins was significantly more frustrating and less productive.

- Reliability. The operational independence of personal computers meant that a single workstation failure affected only one person.

- Consistency/Performance. Second in importance to reliability is the consistent feedback/response that a personal computer provides. "A dependable 2-h[our] turnaround is better than an average 1-h[our] turnaround with high variability." [Mills 76] Furthermore, the *8000* processor provided computing power comparable to modest timesharing systems.

- Physical environment. Personal computing in an environment rich in services provided remotely via Ethernet freed programmers from many traditional restrictions–extreme noise, cold air, etc. The power of the *8000* and of the *Programmer's Desktop* reduced programmers' reliance on printed output. Portability was certainly an option, but the capital expense of an *8000* was such that they were not made available for home use.

- Staffing. Hiring talented programmers has been especially difficult in recent years. We found that being able to offer a programmer his own personal

[2] The diversity of tool names reflects the variety of groups which contributed to the development of these tools. Most of the tools with self-explanatory names were created for the integrated Mesa development environment.

computer with a powerful set of development tools was of great benefit in attracting and holding a talented staff.

- Job satisfaction. Our programmers appreciate and enjoy having sufficient computing resources readily available to make full use of their time and talents.

We also noticed a few pitfalls with personal computing. There was a tendency for individuals to isolate themselves because of the autonomy of personal computing. Without careful coordination and direction, it was easy for for considerable time and effort to be wasted.

Another interesting effect was that the overhead of doing a given task could be reduced to the point where it was tempting, if not automatic, to do it without appropriate forethought. For example, when it was easy to build a new version of the system, programmers would tend to stop debugging as soon as they found the first bug rather than making the most efficient use of each debugging opportunity. Under those conditions, it was also tempting to try a quick fix for a problem rather than taking time to really understand it.

Going to Extremes with Personal Computers

At the end of the first year of development, Star had grown to 300 modules and 67,000 lines of code. This was far more than an individual programmer could fit on his Alto disks along with the necessary tools to build the system.

Of course, not all changes required compiling all of modules in Star. In fact, most changes required only a small number of modules to be recompiled. However, binding Star and building a boot file each required most of the available disk space as well as significant amounts of time.

File servers could solve the disk space problem. However, swapping files required a large number of keystrokes and introduced opportunities for error. A lot of creative time was spent on the rather mundane task of system building.

In part, this problem could have been alleviated by a *system modeling/building* tool. A system modeling tool would provide a formal description of the system and a means of automatically building a new version given an arbitrary revision. Such a tool would have allowed programmers more of an option to rebuild the system themselves, particularly overnight. System modeling was–and still is–an active research topic in SDD. However, even if system bulding were completely automatic, the time involved was frequently not appopriate for a personal workstation.

The Integration Service

After the first few months of Star development we "rediscovered" the need for *integrations*–versions of the system that were complete, consistent, operational, and stored on a file server for public use as a basis for development. The requirements for an *Integration Service* enumerated below may seem quite logical and even obvious. While we recognized a few of them from the beginning, we discovered most of them along the way–by identifying bottlenecks and by experimenting with proposed solutions that we found in the literature or (re)invented ourselves.

Initially we tried to have individual programmers perform software system integrations on a rotating basis. It became increasingly difficult for each new integrator to learn the techniques that had been developed by his predecessors.

- The *Integration Service* permanently accepted the responsibility for producing and storing integrations and for documenting the structural and procedural aspects of integrations.

Producing an integration included the following functions, among others:

- compiling the right modules in the correct order;
- binding the configurations in the correct order;
- correcting syntax errors; and
- building an executable system.

- The *Integration Service* was staffed with para-programmers under the direction of a programmer.

We found performing integrations to be a sub-optimal use of programming talent. Performing integrations was a specialized, repetitive task, in many ways having little to do with programming and requiring good skills for interpersonal communication. The bulk of the work of performing Star integrations was done by a single para-programmer. A second integrator was added during 1981 when the *Integration Service* was expanded to support development of Services products and the file system in addition to Star.

- The *Integration Service* tested what it built to insure operability of the system.

For an integration to be useful as a base for further development, it had to operate with a sufficient, known degree of correctness. We generally required that all modules submitted to the mainline integration be "unit tested," but as Mills points out " . . . the difficulties show up at system integration time. . . . there is seldom difficulty in programming the pieces, the modules; the main difficulty is that the modules seldom all run together as designed." [Mills 76]

The Star project had a separate product testing group, but it was oriented toward testing fully documented "final" products. The *Integration Service* was in the best position to quickly determine the level of operability of the system and then either to initiate corrective action or to inform the programmers of any pitfalls that had been discovered. Even a malfunctioning system could often be used successfully by programmers as long as each programmer could be spared the burden of having to discover each malfunction himself.

- The *Integration Service* acquired and organized computing resources to handle the burdens of large computing tasks.

The computing resources of the *Integration Service* expanded gradually to include nine Altos for compiling and binding, two Dolphins for system building, and an *8000* system for checkout. In addition, integrations used a library of over 200 2.5MB Alto disk packs, a dedicated file server with 600 MB of storage, and 300MB of storage on three public file servers.

By taking advantage of opportunities to automate the integration procedures, we enabled a single para-programmer to keep up to eight workstations busy building different parts of Star for several different integrations simultaneously.

The integrator's principal interactions with the programmers were through electronic mail and a well-publicized set of uniquely named directories on file servers for each integration. The Ethernet environment allowed these forms of interaction to work equally well with programmers in El Segundo or Palo Alto, except for the degradation caused by the 56 Kbps line between sites.

Parallel Integrations

● The most interesting contribution of the *Integration Service* was its management of parallel integrations.

Most Star development was planned to follow a single thread–the mainline series of integrations. But there were several situations where we were compelled to do some development work in parallel with the mainstream until it reached stability, and then to merge it into the mainstream. We had as many as four parallel integrations in progress at the peak of Star development.

Reworks of critical parts of the system were one such situation. For example, the input handler was largely reworked 3 years into the development of Star to provide a "virtual" or "soft" keyboard capability. Untested changes could not be introduced into the mainline integration sequence without running the risk of stopping all development until the new feature was operational.

In many cases, this problem could be handled adequately by an individual on his personal workstation. However, the changes frequently affected interface definitions which had such wide compilation dependencies that the computing effort would have swamped the programmer and his workstation.

One alternative to parallel integrations was to develop stand-alone test vehicles for critical parts of the system. We only chose this alternative in a few cases. In general, we found that testing changes in parallel integrations required much less programming resources than developing and maintaining test programs.

● The *Integration Service* supported reworks and conversions.

Parallel integrations were also used when we received a new version of Mesa, Pilot or the file system–especially when interfaces changed or massive recompilations were required. We call such parallel integrations *conversions*. In converting to a new version of Mesa, Pilot, the file system or to some reworked component within Star, much of the work was systematic and straightforward editing that could be performed by para-programmers. We found that this was also a good training activity for new, junior programmers. The work was fairly easy, and it provided an opportunity to expose them to Mesa, toour coding standards, and to various domains of Star in a non-threatening way.

● The *Integration Service* merged parallel integrations into the mainstream.

This was another example of work that could be done by para-programmers. Most of the work was routine, given a good source comparison program and complete lists of the sources that were changed in each integration being merged.

The key to our success with this technique was our simple method of accounting for all of the source changes that went into each integration. Every source file that was modified for a particular integration was stored by the programmers on a uniquely-named directory on a file server. As the files were processed by the integrator, the original source files were moved to a different uniquely-named directory. The names of both directories included the version number of the integration for easy traceability. At the end of the integration, the second directory contained all of the files which had been changed for that integration.

Conclusions

The net productivity of Star development was just over 2700 lines of code per work year. This compares favorably with Brooks, who cites 1500 lines per year for a typical project with "many interactions" [Brooks 75]. Boehm's formulas for the nominal productivity of a project delivering 250K source lines yield 2066 lines per year for "semidetached software" and 1420 lines per year for "embedded software." Star was probably more "semidetached" than "embedded". The main factors contributing to our productivity were as follows:

● High-level language. The number of lines of assembly language code required for Star would have been larger than the Mesa code by at least a factor of three. The manpower required would have increased by an even larger factor.

● Language explicitness. Manually debugging the errors caught at compile time by Mesa's type checking could easily have doubled the duration and cost of the project.

● Source-level interactive debugging. This roughly halved the debugging time, thereby increasing overall progress by about 10%.

● Sophisticated development environment. We estimate that the power of our tools and the flexibility of the *Programmer's Desktop* have increased our programming output by about 30%. Overall gain for the project was about 10%. The attractiveness and rarity of our personal development environment made hiring easier and reduced turnover.

- Personal computers. Personal computers with adequate capacity to support the Mesa development environment were essential to the development of this particular product. We feel certain that personal computers were cost effective relative to timesharing, but we do not have an adequate basis for quantative comparison.

- *Integration Service* and parallel integrations. Our semi-automated integration procedures made parallel integrations practical. Parallel integrations enabled us to more than double the amount of development work that could be integrated and system tested in a given period of time, with a modest increase in less-skilled manpower. We estimate that parallel integrations shortened our total development schedule by 40%.

- Electronic mail. Electronic mail was extremely important to Star development because the group was geographically split. Moreover, the ease of information dissemination to a large group in a non-preemptive fashion (vs. the telephone, for example) eliminated a lot of disruptive administrative overhead at a low cost.

Star was an ambitious undertaking. The design and implementation were inherently complex and entailed many uncertainties. Many future software projects will have these same characteristics. Like Star, such projects will benefit from a Mesa-like language, a source-level debugger, a comprehensive set of development tools, personal workstations connected to a local-area network, and an *Integration Service*.

References

[Boehm 81] Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[Brooks 75] Frederick P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley Publishing Company, 1975.

[Curry 82] Gael Curry, Larry Baer, Daniel Lipkie and Bruce Lee, "Traits - An Approach to Multiple-Inheritance Subclassing," *Proceedings of the Proceedings Conference on Office Automation Systems*, Philadelphia, Pennsylvania, June 1982.

[Dalal 81] Y.K. Dalal, "The Information Outlet: A new tool for office organization," *Proceedings of the Online Conference on Local Networks & Distributed Office Systems*, London, England, May 1981. Also Xerox Office Products Division, Palo Alto, California, OPD-T8104, October 1981.

[Gutz 81] Steve Gutz, Anthony I. Wasserman, and Michael J. Spier, "Personal Development Systems for the Professional Programmer," *Computer*, The IEEE Computer Society, April 1981.

[Horsley 79] Thomas R. Horsley and William C. Lynch, "Pilot: A Software Engineering Case Study," *Proceedings of the 4th International Conference on Software Engineering*, The IEEE Computer Society, September 1979.

[Johnsson 82] Richard K. Johnsson and John D. Wick, "An Overview of the Mesa Processor Architecture," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, March 1982.

[Lauer 79] Hugh C. Lauer and Edwin H. Satterthwaite, "The Impact of Mesa on System Design," *Proceedings of the 4th International Conference on Software Engineering*, The IEEE Computer Society, September 1979.

[Mills 76] Harlan D. Mills, "Software Development," *IEEE Transactions on Software Engineering*, v. SE-2, no. 4, pp. 265-273, 1976.

[Mitchell 78] J.G. Mitchell, W. Maybury, and R.E. Sweet, "Mesa Language Manual," Technical report CSL-78-1, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, February 1978.

[Smith 82] David Canfield Smith, Eric Harslem, Charles Irby, and Ralph Kimball, "The Star User Interface: An Overview," to be published in the proceedings of *NCC '82*.

[Thacker 82] C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull and D.R. Boggs, "Alto: A Personal Computer," *Computer Structures: Principles and Examples*, D. Siewiorek, D.G. Bell and A. Newell, editors, McGraw-Hill, 1982.

Observations on the Development of an Operating System

Hugh C. Lauer
Xerox Corporation
Palo Alto, California

The development of Pilot, an operating system for a personal computer, is reviewed, including a brief history and some of the problems and lessons encountered during this development. As part of understanding how Pilot and other operating systems come about, an hypothesis is presented that systems can be classified into five kinds according to the style and direction of their development, independent of their structure. A further hypothesis is presented that systems such as Pilot, and many others in widespread use, take about five to seven years to reach maturity, independent of the quality and quantity of the talent applied to their development. The pressures, constraints, and problems of producing Pilot are discussed in the context of these hypotheses.

Key words and phrases: Operating system, system development, software engineering, Pilot, personal computer, system classification.
CR Categories: 4.35, 4.30.

This paper contains my personal observations about the development of Pilot, an operating system for a personal computer [Redell *et al*], compared and contrasted with some other operating systems with which I have had contact. In these observations, I concentrate not on the anatomy of these systems but rather on their life cycles, particularly their formative years from conception to birth to maturity. This is a somewhat unorthodox point of view in the technical literature which abounds with papers on operating system techniques and structures, software engineering tools and methods, and general exhortations about the right and wrong ways to develop systems. But it is a useful one, not only for the student of operating systems and system development, but also for the managers or sponsors of development projects who need some understanding about why systems are so dramatically different from each other, why some succeed and others fail, and what might expected from development organizations.

In comparing Pilot with other operating systems, I have found it useful to classify operating systems into five categories according to how they came about, how successful they were, and their impact on the computing community. This classification is one of the main themes of this paper. It is interesting to observe that systems classified as the second kind, including Pilot and many of the major operating systems in widespread use, seem to take from five to seven years to grow from birth to maturity. Furthermore, it seems that this five to seven years is necessary, independent of the amount or quality of the talent applied to an operating system development project.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

The paper has four main parts. In the first two parts, I chronicle the development of Pilot and some of the data and problems pertaining to it; this chronicle is presented not because it is new, different, or novel, but because we rarely talk about these things in the literature and there are lessons to be learned. In the third part, I offer the classification and some comments on systems in each category. In the final part, I make some observations about the five-to-seven year rule, why it seems to be true, and what its consequences are.

History of Pilot

We use the term 'Pilot' in three different ways, to mean an operating system kernel, a major project and system, and a way of life.

- As an operating system kernel, Pilot is the system described in [Redell, *et al*]. It consists of approximately 25,000-50,000 lines of code (depending upon how you count) in the Mesa programming language [Mitchell *et al*] and was developed over four years by a group of four to eight people, many of whom had other responsibilities at the same time.
- As a system development project, Pilot consists of approximately 250,000 lines of Mesa code in about two dozen major subsystems, including the operating system kernel, CoPilot (the Mesa debugger), a common user interface package and framework for building development tools, various utilities and communications packages, microcode for defining the Mesa architecture in several processors, and other facilities. For historical and organizational reasons, another 200,000 lines of code in compilers, binders, Mesa utilites, librarian tools, change request tools, etc.—much of which was Alto-based until early 1981—are not included in Pilot. Together, these two bodies of software represent both the operating system to be embedded in client application systems and the necessary support to develop and test those applications. Approximately 40 people have contributed to these systems over five years.
- As a way of life, Pilot defines a framework for thinking about, designing, and implementing systems and for communicating among subsystems within machines and across networks. It is the operating system for the Mesa machine architecture and hence part of a number of Xerox products, the foundation for the Mesa development environment, and a tool for supporting software research through the Cedar system [Deutsch and Taft].

Most other operating systems suffer the same multiple uses of their names. In this paper, I will use the term Pilot and any other operating sytem in the second sense, to include all the supporting software that makes the system usable. Note that Pilot is not ordinarily visible to Xerox customers; it was primarily intended to be embedded in products and to be used internally in the research and development environments.

Although it has roots in earlier work on the Alto system [Sproull and Lampson] and Mesa at the Xerox Palo Alto Research Center (PARC), serious work on Pilot began in the System Development Department (SDD) in early 1976. Most of the people who worked on Pilot over the past five years had previous experience primarily in the academic and research communities, and very few had ever built any 'production' system, let alone an operating system, to commercial pressures of schedule and budgets. We knew about the differences between the big, ponderous, ungainly systems sold by some computer manufacturers and the simple, elegant, lightweight systems imagined in the research community, and we were determined to do it right. This meant using a good, high-level language (Mesa), assigning a fairly small group of knowledgeable people to the project, carefully studying the lessons of others in the computer science community and from the Alto environment at PARC, and designing the hardware and software together. This ought to take two or perhaps three years, after which the designers would be available to work on advanced, Pilot-based applications for products, research, and development. The following is a short chronicle of our actual experience:

Jan. 1976: Architectural principles established; work began on design of a machine architecture optimized for Mesa. The Mesa machine extended the Alto architecture in a number of ways, including expansion of the basic machine address from 16 to 32 bits, stack-oriented operation, addition of virtual memory, improved handling of I/O devices, etc.

mid-1976: Mesa language and system becomes operational on the Alto [Geschke *et al*].

Dec. 1976: Pilot Functional Specification (version 1) released to clients. Unfortunately, the system specified in this document looked a lot like a traditional operating system and did not take account of the characteristics of a personal computer, the role of Mesa, or the (dimly perceived) needs of distributed applications.

early 1977: Mesa process facilities [Lampson and Redell] and Pilot file system redefined.

Sept. 1977: Pilot Functional Specification (version 2) released to clients. This set the style of Pilot as it is today and used the Mesa interface language as the primary specification tool [Lauer and Satterthwaite].

Sept. 77–Apr. 78: Design of Pilot kernel implementation.

Apr. 78–Oct. 78: Implementation of Pilot kernel and basic communication facilities.

July 1978: First Dolphin processor delivered to Pilot group (the Dolphin is one of three machines supported by Pilot). This was microcoded to be compatible with the Alto but with virtual memory; some Mesa emulator functions were still coded in BCPL and compiled into Alto machine language.

Oct. 1978: First release of Pilot to clients. This version retained compatiblility with the Alto system in many areas such as disk layout, boot file formats, and Mesa instruction set, even though they were not adequate for our long term needs. The debugger was a modification of the Alto/Mesa debugger and operated in Alto mode. Since at this time there was no established client base, there was very little testing except from our own test programs; consequently, this release was of limited use.

Nov. 78–Dec. 79: Bootstrapped away from the Alto environment. During this time, we eliminated remaining Alto compatibility from Pilot, implemented the Mesa architecture entirely in microcode, supported Pilot disk and boot file formats, etc. CoPilot, the Pilot debugger, became operational as the first major Pilot client. This was a particularly painful period for the implementors.

Dec. 1979: Second release of Pilot to clients. Note that at this time, the Mesa compiler, binder, and other facilities, as well as the program editors, were still based on the Alto. The performance, reliability, and usability of Pilot was grim; as each new client tried it, a new set of disabling bugs was uncovered. Among other things, we discovered that Pilot could read or write the disk at the rate of only one 256-word sector per revolution.

Mar. 1980: Third release of Pilot to clients. This was a cleaned up version of the second release and the first to appear in a product (the Xerox 5700 electronic printing system). We redesigned the file system to solve the disk performance problem and concentrated on the reliability issues with respect to that specific client. Even so, the application programmers had to resort to some unnatural contortions to avoid problem areas in the operating system.

May. 1980: First Dandelion processor available, but with no disk or Ethernet (the Dandelion is the basis of all Xerox 8000 series products).

July. 1980: The Pilot disk utility runs on the Dandelion (with a simulated Ethernet).

Oct. 1980: Fourth release of Pilot to clients. The primary emphasis of this release was to provide the function necessary to support Xerox 8000 series products, the Mesa development environment, and the Cedar project at PARC. Unfortunately, Pilot by this time had become too large and too slow to do any of these very well.

Dec. 1980: Pilot runs on the Dorado processor at PARC (the Dorado is a very high speed, single-user system for the research environment [Dorado]).

Feb. 1981: First Xerox 8000 network system, including Pilot, delivered to a customer.

Apr. 1981: Fifth release of Pilot to clients. The emphasis of this release was to improve the performance and reliability and to reduce the working set size of the system to acceptable levels within the memory available on the various processors. The first version of the Pilot-based Mesa development environment (i.e., compiler, binder, editor, utilities, etc.) was made available to friendly clients.

In April 1981, we received a letter from our most demanding clients in PARC indicating that with the fifth release, Pilot had become the system of choice (on processors that were capable of running either Pilot or Alto software). After five years, Pilot had come of age.

During this whole time, we always had the support of the corporation, even during the hard years when we 'should have' had a nicely running system. Of course, it was necessary to regularly show progress, which we did partly by demonstrating some early but limited applications (to show that Pilot actual worked) and partly by demonstrating Alto-based prototypes of some advanced applications (to show the feasibility of the kinds of things we were aiming for). We also had to fend off the usual kinds of pressures from other parts of the corporation—for example, that we use a commercially available language (e.g., Pascal) rather than Mesa or that we purchase OEM computers rather than design our own architecture.

Selected problems and lessons

In this section, I will recount a few of the problems we encountered and lessons we learned during the development. Most of these we should have avoided, and afterwards there were plenty of people to say they told us so (none of whom had any responsibility for actually releasing the system, of course). Nevertheless, these problems and lessons happened anyway.

Sizes of the system. Table 1 shows some statistics for the five releases of Pilot. Included are the sizes of the Pilot kernel and of the entire system in terms of lines of Mesa code, bytes of object code, and numbers of modules. From the table it can be seen that the Pilot kernel dominated the first release, but by the fifth release it represented barely more than twenty percent of the system. The growth of the total system is accounted for partly by new development and partly by absorbing and converting code which was orginally developed for other systems. Other development, in which major subsystems were completely rewritten or replaced, is not apparent in this kind of summary table, but represents a non-trivial portion of the work that went into all but the first release.

When we began work on Pilot, none of us imagined that we would be developing and managing a system so large. Yet in retrospect this was probably inevitable, given that it was intended to support several major products plus all of our software development, some research, and a number of specialized applications. As we look to the future, it is not clear whether the sizes of either the kernel or the system will stabilize soon or whether they will continue to grow as we respond to new or different needs from our clients.

Working set sizes. Table 1 also shows the size of the working set of the Pilot kernel when supporting 'typical' applications—i.e., the amount of real memory required to hold the virtual memory actively needed by Pilot without thrashing. A working set size is inferred by first artificially restricting the amount of real memory available on the machine and then timing a selected benchmark with a stopwatch. This is repeated for various memory sizes and the results plotted. The total system working set for that benchmark is defined to be at the knee of the curve and can be determined accurately within 1%. Then another experiment is run by setting the real memory size to the working set size, executing the benchmark again (most benchmarks take a few seconds), and taking a memory dump. With some detective work, it is possible to attribute specific pages swapped into real memory to the Pilot kernel, common software and other packages, and the application. This whole exercise is repeated for various benchmarks and for each release of Pilot (and also for each release of critical applications on a given version of Pilot).

An unexpected result was that the content of the working set of the Pilot kernel (i.e., the actual pages swapped in) is nearly constant across all benchmarks. We were also surprised that by April 1980, Pilot exceeded its share of the real memory of our product configurations by nearly a factor of two. In retrospect, of course, we should not have been surprised. Although the requirement for a small working set was in the front of our minds, we had no feedback or reinforcement to achieve it, even at the expense of some features or function. In this sense, the developers suffered from the availability of a good virtual memory system. It is too easy to add more memory to their machines in order to meet critical schedules, even if business reasons precluded such memory in the products. The result was a year of hard work to bring the Pilot memory requirement down to a reasonable level, at some cost in the overall schedule. (An Alto programmer, by contrast, is *forced* to make his applications fit into a non-expandable real memory and address space because he cannot proceed with his own work until they do.) Unfortunately, similar pressures affect many of our clients, and some are caught by the convenience of virtual memory the way we were.

Programmer Productivity. None of the ways that we know of for measuring the productivity of our developers is very statisfying. We do, of course, measure whether or not a release is on time, how many trouble reports are submitted against it, how big it is, etc. But none of these tell us how good the system is. We also have, on occasion, tried to make one traditional measurement of programmer productivity, namely the number of lines of code produced per work-year. There are two types of problems with this measurement, an obvious one and a subtle one. The obvious one is deciding which people and what time to count, so that an effective comparison might be made among organizations and/or programming environments (companies such as Xerox are always interested in such comparisons, even if individual development groups are not). There is also the question of how to count the lines of code, especially when someone is reorganizing or making major modifications to existing modules.

The subtle problem is illustrated by the following observation: in my organization, a group of four or five developers, including a project leader, can specify, design, implement, test, and release a complete system or subsystem of approximately 25,000 lines of Mesa code in twelve months. This includes vacations and holidays and time to attend conferences and seminars, to write professional papers or continue education, to get and train some users or clients to test the system, and to be generally effective members of the organization. In most cases, *if the same people had twice as much time, they could produce as good a system in, perhaps, half that amount of code*—i.e., the extra year yields negative productivity. Thus, one of the conflicts that we have to manage constantly is that between the need to get a system done and working satisfactorily and the desire to make it smaller, faster, easier to use, etc.

Holy wars. In the early days of Pilot development, we got bogged down in a number of basic questions of operating system design and spent a lot of time, energy, and emotions before resolving them. One of these concerned the model of processes and synchronization and whether we should have a facility based on procedures and monitors or a facility based on messages. Each side was firmly entrenched and unable to accept the position of the other; not until we developed the duality hypothesis presented in [Lauer and Needham] were we able to resolve the issue and implement the scheme described in [Lampson and Redell]. Even afterward, our organization bore serious scars from this debate.

Another basic question concerned the kind of access to the file system that Pilot would provide. One alternative was a simple read-write facility with which client programs transfer pages directly between virtual memory and files. The other alternative was a 'mapping' facility, whereby a portion of the file is made the backing store for a portion of virtual memory (for example, as in MULTICS [Bensoussan *et al*]). While this question did not inflame emotions the way the process question did, the proponents of each view felt that the two models were incompatible with each other. Pilot chose the mapping approach, providing a convenience for some but causing headaches for others, particularly those who were trying to convert

TABLE I — SIZES OF PILOT RELEASES

Release	I	II	III	IV	V
Date	Oct 78	Dec 79	Mar 80	Oct 80	Apr 81
Contributors*	~20	~27	~27	~35	~35
Pilot Kernel:					
lines	24K	30K	33K	44K	53K
codebytes	89K	111K	110K	152K	162K
modules	88	102	114	123	135
interfaces	93	132	146	183	204
Pilot System:					
lines	48K	129K	125K	171K	249K
codebytes	211K	508K	508K	754K	1025K
modules	390	420	unknown	530	710
Working Sets (256-word pages)	unknown	unknown	~320	~260	~190

* Most contributors had responsibility for other, non-Pilot software at the same time.

programs from other systems based on the read-write model. Subsequently, the perception grew that perhaps the two approaches are duals of each other, and that a client program structured for one approach might have a natural counterpart of similar performance and complexity for the other approach. However, we never found a duality transformation to support this view. Finally, we realized that neither model excludes the other. In particular, code files and certain data files of limited size are better supported by the mapping model—the swapping characteristics are understood and address space management in virtual memory is more convenient than explicit reading and writing. Large data files with known, high performance access requirements, on the other hand, are better served by the read-write approach—these files are often larger than the virtual address space, and the complexity and overhead of buffer management is worthwhile to achieve the desired performance. Pilot now supports both approaches with consistent interfaces.

Files and transactions. When the Pilot file system was being designed, the question naturally arose about whether or not it should include a transaction facility for crash recovery and atomic updating of files. We did not include one because our experience was limited and we were only just beginning to see results from research in this area at PARC. However, we did provide enough facilities so that a client could build a specialized transaction mechanism on top of the Pilot kernel. Someone built such a facility and also undertook an evangelical mission to persuade people that it offered the solution to all file reliability and recovery problems. As a consequence, several of our major clients came to depend upon the transaction concept, even in cases where it is not appropriate.

Naturally, a transaction facility built on top of Pilot could not have as high performance as one integral with it, and in this case, its interfaces were of a substantially different style than those of Pilot itself. It was also extremely unreliable. Thus we were forced to do a quick implementation of a new transaction facility as part of, and consistent with, the Pilot kernel. This has significantly better performance and is reliable in spite of known bugs; client programs that depended on the previous facility became simpler with the new one. However, the performance is still not good enough for high intensity activities such as data base accesses and updates. At the same time, some of the clients began to realize that even the best transaction facility would offer inappropriate performance for their applications and that their failure modes did not require this generality. For example, in the user level directory facility for the Xerox 8000 series products, it is much better to accept that crashes can occasionally occur in the middle of updates and to rely on a scavenger to restore things from the natural redundancy in the file system.

It is not clear what the future of transactions in Pilot will be, but since we currently satisfy no one in this area, it is likely that the facility will change substantially again.

Virtual memory implementation. In designing the Pilot virtual memory facilities, we recognized the client program would want to manage the address space, map files to pieces of virtual memory, and control (or influence) the swapping between real memory and the backing file. We began with three different interfaces and concepts, but quickly unified them into the single concept of the *space*. The Pilot space is the unit of allocation, mapping, and swapping; spaces can be declared within other spaces, so that the set of all spaces forms a hierarchy according to the containment relation. This was a remarkably simple generalization, but it was hard to implement and is not used by clients. Clients have evolved a style in which almost all mapped spaces are subspaces of the primordial space (all of virtual memory) and only a few of these are further partitioned into subspaces for swapping control. The implementation requires such large data structures for each space that they have to be swappable, and only the very active items are cached in real memory. In the end, several caches were needed and a lot of resident code was written to manage them.

An assumption of the virtual memory implementation is that disk accesses are expensive. Thus, we set up a lot of queues and expected a lot of multiprogramming to overlap computing with disk operation. In fact, disk accesses are cheap on both the Dandelion and Dolphin configurations. If no arm movement is required (this appears to be true most of the time), the computation required to field a page fault, locate the disk address, set up a disk command, receive the disk interrupt, and dispatch the faulted process takes about the same time as the average latency to read a sector. We found that the system could not accept back-to-back requests for adjacent sectors and read them on the same revolution, and thus we had to rewrite the file system to submit single disk requests for runs of pages whenever it could. Even so, it would almost be cheaper to treat the disk as a synchronous device and simply wait until each operation completes without trying to do anything else.

In view of this experience, we are currently reexamining the basic design of the Pilot kernel virtual memory system and will probably make major revisions in both the strategy and the implementation.

Pipes, filters, and streams. One of the strong features of the UNIX system [Ritchie and Thompson] is the uniform facility for input and output which allows separate programs to be connected together by 'pipes.' The UNIX programmer's toolkit includes a large number of simple programs (called 'filters') which perform simple transformations on streams of data, and it is common practice to concatenate a number of these together for a desired result. We thought that Pilot should have a similar facility but consistent with and implementable in Mesa, and so we designed Pilot *streams* (see [Redell *et al*] for an overview).

Unfortunately, although the Pilot stream facility works satisfactorily, it was not very well received and is not widely used by us or by our clients. One reason probably lies in the Mesa model of program modularity. The type-safety and interface language of Mesa make it convenient to design programs with clearly specified procedural interfaces and bind them together with a little bit of control code for a desired result. Thus the Mesa programmer's toolkit consists of a large number of modules of varying complexity and different kinds of control structures. For example, a module which produces a sequence of objects of some abstract type will export a procedure for its clients to access these one at a time. This can be easily bound to another module that expects to get objects of that type, and it is often more flexible than parsing a stream of characters. Thus, Pilot streams are used almost exclusively at the interface with terminals and other systems over industry standard communication lines and protocols. Procedural interfaces are preferred, both within Pilot-based programs and between system elements over the Ethernet.

Comparing Pilot with other operating systems

From the success of the April 1981 release, it is evident that Pilot will take its place among the ranks of mature, evolving operating systems and have a useful life long after its original designers have moved on to other pursuits. However, it did not happen as planned and its development was very different from that of the Alto system. In reflecting upon this, I found it useful to enumerate some of the other operating systems I have known, either from direct contact or from study of the literature or from contact with others. These systems seem to fall into five categories, which I shall first enumerate and then describe.

1. The Alto system, UNIX.
2. IBM's OS/360, MULTICS, Pilot, etc.
3. MTS (the Michigan Terminal System), TENEX, CP-67
4. CAL-TSS, Project SUE, HYDRA, etc.
5. DOS/360, RS-11, etc.

These categories are the result of my personal observations, not of a systematic study, and hence many systems are not listed because I don't know enough about them to classify them. The ordering of the categories is not significant. An important part of the classification is the maturity or success of a system—i.e., acceptance by its clients as a useful, economic tool for helping to get work done, for supporting applications, or for fulfilling other goals. A characteristic of a successful system is that it is accepted by a non-trivial community of users outside its developing organization as a matter of choice and that this community contributes, directly or indirectly, to its further development and growth.

Systems of the first kind. These are everyone's favourite systems. They are successful by our measure and by most other measures (sometimes too successful for their developers). They begin life as small, simple, unambitious systems meant to serve only their authors and perhaps their immediate colleagues. They have limited requirements, usually in the research area. But their excellence and simplicity attract others who are willing to contribute to the further development, additional features, or maintenance responsiblities in exchange for being able to use such systems in their own work. They become successful partly because potential users find it simple and easy to adapt them when needed facilities are missing or illconceived. Systems of the first kind rarely evolve according to any coherent plan agreed to between the implementors and clients, but rather by the willingness to contribute facilities and features as needed. Thus it is not surprising that these systems sometimes appear a little haphazard.

Systems of the second kind. These the planned systems. They are cut 'from whole cloth,' designed and implemented as major projects, directed toward objectives defined by negotiation, often (but not always) aimed at new architectures, meant to satisfy major clients, and built according to schedules and budget constraints imposed for business, contract, or other external reasons.

Some, but not all, of the major operating systems sold by computer manufacturers fall into this category. For example, IBM's OS/360 was conceived as a whole system to satisfy a new, broad marketplace and to incorporate many of the technological achievements of the previous five years, and thus it is a system of the second kind. So is MULTICS, which was built primarily at MIT as a 'real' system based on the previous experimental system CTSS. Pilot is a system of the second kind: it was conceived as a successor to the Alto system and intended to support a range of product, development, and research applications within Xerox over a specified number of years on a new machine architecture.

Not all systems of the second kind are successes. Some notable failures include IBM's TSS/360, the Berkeley Computer Corporation system [Lampson], and the Elliot 503 Mark II system [Hoare]. Each of these was conceived as a major system and a fairly ambitious project, but none survived the patience of its sponsors or clients to reach maturity.

Systems of the third kind. These systems borrow much of their supporting software from an existing system but represent a fundamental change in the way of life. The Michigan Terminal System, for example, provides a paging, terminal-oriented, time-sharing system especially suited for university use on the IBM 360/67 and IBM 370 systems. Most of its compilers, run-time support, subroutine libraries, program development tools, etc., were taken and converted directly from OS/360, but its operating system kernel is dramatically different from OS/360 and it supports new applications that OS/360 never could. (Of course, there are also many OS/360 applications that MTS cannot support.) The obvious motiviation for building a system of this kind is to avoid the time and expense of designing, implementing, and maintaining all new supporting software for the operating system when it is desired only to implement an operating system kernel and some basic functions.

Systems of the fourth kind. While the population of systems of the first three kinds is relatively small, there are many systems of the fourth kind. They make major contributions to the art and science of operating systems but either never reach maturity or never gain acceptance outside the developing organization. For many of them, there is never any serious intent to promote them for widespread use, to support a variety of applications, or to be 'complete.' They are primarily laboratory exercises to support research and to teach about operating system structures, or to support other laboratory work. Note that systems of the first kind begin life as systems of the fourth kind but suffer the calamity of success.

Systems of the fifth kind. Finally, the world is full of small, uninteresting systems which do little to enhance the machines they support and which contribute little to the technology or have little impact on the computing community. These systems come in all sizes, shapes, colors, and prices, and I have nothing interesting to say about them.

This taxonomy is helpful in comparing like with like when we talk about operating systems in a context of which work and which do not. For example, it does not make sense to berate the excessive generality of OS/360, which did succeed, in comparison with the more limited objectives and elegant structures of, say, CAL-TSS or Project SUE, both of which failed to become generally usable. It was observed in [Lampson and Sturgis] that there is much more work involved in making an operating system usable by general programmers than just providing a nice kernel. Similarly, when we ask why the Pilot development was so different from that of the Alto system, it is important to bear in mind the fundamental difference in objectives and ground rules for the two systems. The following were explicitly *not* objectives of the Alto system:

> "This system must satisfy the corporate needs for the next 10-15 years in specific areas."
>
> "A (nearly) complete list and specification of the functions and facilities required of the system over the next five years must be provided before design starts."
>
> "This system must incorporate all of the wonderful lessons of operating system technology from the past five years."
>
> "This system is expected to have more than one hundred users or to be installed in more than one hundred locations."

These or similar objectives did apply, however, to Pilot and to most other systems of the second kind. The developers of the Alto system are chagrined to find that they now have to spend considerable time and energy supporting a system which has several thousand users and supports a wide variety of corporate needs. By contrast, in the Pilot development, we were chagrined to discover that in striving to meet these objectives or our schedule, we often found ourselves unable to apply what we felt was the best technical solution to a problem.

The five-to-seven-year rule

The above classification identifies operating systems in terms of how they were developed and their impact. It separates systems of the second kind, which are willed into existence and operation, from those of the first, third, and fourth kinds, which evolve in a less deliberate manner or as part of some other research or project. While I have no recipe for producing successful systems of the latter kinds, I can offer an hypothesis which, if true, will be useful to anyone setting out to build a system of the second kind: *it takes from five to seven years for a system of the second kind to grow from birth to maturity.* For example, in the systems I enumerated above:

> OS/360 was begun in 1963-1964. Despite early availability and vigourous promotion by its manufacturer, it was not until 1968-1969 that it really gained wide acceptance by its users as a valuable, economic tool.
>
> MULTICS development began in about 1965. After an initial flurry of publications about the system, its design, and its goals, the outside world heard very little from MULTICS-land until the early 1970's, when it started to acquire a following outside MIT and was subsequently adopted by Honeywell, the manufacturer of MULTICS hardware.
>
> The first five years of the life of Pilot were chronicled above. Although the previous skepticism by its users is now changing to enthusiasm, there is still much to be done before Pilot fulfills their expectations.

It also appears that it takes from five to seven years for other operating systems produced by major manufacturers or as major system projects to mature. Let us consider what happens during those years.

First, there is the period of planning and design. This is a time of exceptional optimism, of desire to incorporate the past successes and avoid past mistakes, of a determination to 'do it right.' In the Pilot project, for example, an attitude that prevailed was "the Alto system demonstrated a lot about personal computing: now let's build one for real, to support the company's business. And incidentally, we should build it in Mesa, and build it with virtual memory, and re-engineer the Ethernet, and unify the protocols, improve the file system, etc., etc., etc."

Next comes the initial implementation and first release. There are no operational client programs against which to validate it, so it gets little testing. Anyway, some of the promised function was deferred in the interests of meeting the delivery schedule.

Then comes a period of trying to make the system work at all. Those first hardy users have had to pick their way through minefields of bugs and problems, and some may have become discouraged and turned to other alternatives. However, through perseverance, the problems are solved one by one, and eventually the system seems to work passably, at least for its few active users.

But now it is important to make it work well. It is too slow or too big; it supports too few users/clients; or it fails to match the performance of its predecessors. Promised enhancements and/or deferred functions are abandoned and development is concentrated on very simple matters. During these two phases, client or user participation is essential, althouah painful. Without the appropriate feedback from others who are trying to use the system for non-trivial reasons (other than its own development), the implementors do not have enough information to guide their work and identify problems in performance, style, or function.

Finally, if the sponsor has not lost patience, there is a period of evolving expectations about the new system. Clients realize that it is not the same as a previous one and that their old models of performance, behaviour, and usage need to be modified to take advantage of the new facilities and the new constraints. Some functions or facilities of the new system may never work as well or as fast as the corresponding ones of the old, and programs converted from the old may appear sluggish in the new environment. This is the beginning of the period of 'community involvement' with the operating system, during which clients learn how to live within the framework defined by that system and how to contribute to its further success and growth.

The five-to-seven-year rule for systems of the second kind is a strong generalization from weak evidence. I know of no analysis which might lead to it as a conclusion. I would even like to see it disproved (and to learn how to disprove it at will). Nevertheless, from my own experience and from observations of the experience of others, it seems to be true, at least most of the time. It seems to apply both to the professional system designers and programmers who populate the industry and to the elite corps of computer scientists from the academic/research community. Even people with impressive credentials fail in their attempts to build systems in less time, and very few succeed. We believed from the beginning that we could do better in the development of Pilot.

Both in casual conversations and detailed analyses of the successes and failures of systems of the second kind, the same terms keep recurring: that the systems are "too ambitious" and/or "too general." Hoare used these terms in his Turing Lecture [Hoare]. I can remember as a graduate student that my colleagues and I would sneer at the manufacturer-supplied operating systems (IBM's OS/360, Univac's Exec VIII, Burroughs' MCP, and all the rest) in exactly the same terms. In almost any cocktail conversation about a system in trouble or one which the users find unsatisfactory, criticism is focused on the generality or amibitousness of the project goals. In my discussions with the original implementors of the kernel of the Alto system, the same terms were used again: "if only we had set more limited goals for Pilot by concentrating on, say, real memory requirements rather than on features, we would have produced a nice, well-performing system in two or three years, just the way they did." However, that begs the question: we *did* concentrate on real memory usage, execution speed, simple structure, and all of the other things that are important in making a successful design. We made task lists of things to do, problems to solve, features to support: we assigned priorities and worked on first things first; we parried requests for yet more features or complexity; we ignored unreasonable constraints imposed externally and let our computer science wisdom prevail. It still took us five years, and our critics at the time still worried about the grandiosity of our system.

I suspect that there is something about the ground rules of projects like Pilot, the Berkeley Computer Corporation system, MULTICS, and other systems of the second kind that makes it difficult or impossible to plan or carry out projects the way we do for systems of the other kinds. (Note that by definition, systems of the other kinds cannot be too ambitious or general: either they succeed on their merits, meaning that they have exactly the right blend of generality and simplicity, or they were never meant to fulfill the kinds of goals that a system of the second kind is.) Part of it is, no doubt, in the way that projects are sponsored. Systems of the first, third, and fourth kinds are usually financed as part of some other project or research.

But systems of the second kind are investments. As such they are subject to the kinds of review of planning, budgeting, scheduling, and scrutiny that the sponsor needs to confirm continuing support (neither the Alto system, MTS, or HYDRA, for example, were ever subject to this kind of review). Furthermore, investments in system development are still so risky these days that most sponsors would rather purchase a satisfactory system, if available, than build one. Thus, by definition, the new system has to be more ambitious and/or more general than its predecessors. If it is necessary to finance a new system, then the sponsors and/or their clients feel entitled to some voice in the facilities, features, style, character, or other attributes of the system. I.e., the system designers do not get to go off alone to build the system of their dreams, and the five-to-seven-year rule prevails.

Summary

I have been a member of the Pilot project since early 1977 and have managed it in its later years. Coming from an academic and research background, I have been somewhat surprised at what it has been possible for us to do and also at what we have not been able to do. I think it is important for people to write about these things occasionally because we too often concentrate on the objects we are creating and not enough on the process of creating them. The world of programming methodology and structured programming is devoted to helping us achieve perfection in the systems we design—an important goal but somewhat at odds with the need to get something done. Pilot had to be delivered and work well enough, despite the fact that we never had time to make it perfect or even as small and as simple as we would have liked. It is helpful to treat the major system as an organism itself, with a life cycle and a personality and characteristics derived from the organizations that build, use, and sponsor it. The successful ones usually outlive the interests and participation of their implementors, and they captivate or even dominate the professional lives and interests of many other people.

I have believed in the five-to-seven-year rule for at least a decade and have not found much evidence against it. Yet this is not proper science, so I challenge graduate students and researchers in the operating system field to conduct systematic studies about how systems are conceived and born and which ones grow, mature, and lead productive lives. This would be a study partly of technology but partly of the sociology and dynamics of system development, and it would teach us how to build better, simpler, less ambitious systems more predictably.

The classification of operating systems into five kinds came about as I tried to compare Pilot with other systems and see where the five-to-seven-year rule applied and where it did not. There is definitely a qualitative difference between the kind of development we carried out and the kind that I have watched or been associated with in universities and research laboratories. Thus it is not surprising that there is a difference in character between the kinds of systems that emerge from these activities. I do not know whether this classification is 'right,' so again I challenge research students to explore the field of operating systems from this point of view, making systematic studies to help us understand how we do better at building them.

Finally, a word of advice to designers, implementors, sponsors, and users: if you are involved with a new, challenging system planned and cut out of whole cloth and meant as a service, not as an experiment, but intended to stretch our horizons and expectations—i.e., a system of the second kind—then have patience. I have not yet seen anyone who has been able to build one as quickly and as well as he thought he could.

Acknowledgements

Approximately forty people in Xerox System Development Department have made contributions to Pilot, all of them valuable. In addition, we gained great benefit from close location to and many conversations with our colleagues at PARC. The number of people deserving acknowledgement is far too great to list here. However, I wish to specially acknowledge Dr. David E. Liddle, who created and sustained SDD and the Pilot project as part of it.

References

[Belady and Lehman]
Belady, L. A., and Lehman, M. M., 'A model of large program development,' *IBM System Journal*, no. 3, 1976.

[Bensoussan *et al*]
Bensoussan, A., Clingen, C. T., and Daley, R. C., 'The MULTICS Virtual Memory: Concepts and Design,' *Communications of the ACM*, vol 15, no 5, May 1972, pp 308-318.

[Deutsch and Taft]
Deutsch, L. P. and Taft, E. A., 'Requirements for an Experimental Programming Environment,' report # CSL-80-10, Xerox Corporation, Palo Alto Research Center, Palo Alto, 1980.

[Dorado]
The Dorado: A High-performance Personal Computer, Three Papers, Technical Report CSL-81-1, Xerox Palo Alto Research Center, Palo Alto, California, January 1981.

[Geschke *et al*]
Geschke, C. M., Morris, J. H., and Satterthwaite, E. H., 'Early Experience with Mesa,' *Communications of the ACM*, vol. 20, no. 8, August 1977

[Hoare]
Hoare, C. A. R., 'The Emperor's Old Clothes,' (1980 ACM Turing Award Lecture), *Communications of the ACM*, vol. 24, no. 2, February 1981.

[Lampson]
Lampson, B. W., 'Dynamic protection structures,' *Proceedings of the AFIPS Fall Joint Computer Conference*, 1969, pp 27-38. (Note: The Berkeley Computer Corporation was a widely publicized venture by a number of respected computer scientists to build a major time-sharing system and utility in 1968-1970. I can find no references to it in the literature except this one, which is mostly about the operating system structure.)

[Lampson and Redell]
Lampson, B. W. and Redell, D. D., 'Experience with Processes and Monitors in Mesa,' *Communications of the ACM*, vol. 23, no. 2, February 1980.

[Lampson and Sturgis]
Lampson, B. W. and Sturgis, H. E., 'Reflections on an Operating System Design,' *Communications of the ACM*, vol. 19, no. 5, May 1976.

[Lauer and Needham]
Lauer, H. C. and Needham, R. M., 'On the Duality of Operating System Structures,' *Proc. Second International Symposium on Operating Systems*, IRIA, Oct. 1978, reprinted in *Operating Systems Review*, vol. 13, no 2, April 1979, pp 3-19.

[Lauer and Satterthwaite]
Lauer, H. C. and Satterthwaite, E. H., 'Impact of Mesa on System Design,' *Proceedings of Fourth International Conference on Software Engineering*, Munich, September 1979, pp 174-182.

[Mitchell *et al*]
Mitchell, J. G., Maybury, W. and Sweet, R., *Mesa Language Manual*, report # CSL-79-3, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, 1979.

[Redell *et al*]
Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C. McJones, P. R., Murray, H. G., Purcell, S. C., 'Pilot: An Operating System for a Personal Computer,' *Communications of the ACM*, vol. 23, no. 2, February 1980.

[Ritchie and Thompson]
Ritchie, D. M. and Thompson, K., 'The UNIX Time-Sharing System,' *Communications of the ACM*, vol. 17, no. 7, July 1974.

[Sproull and Lampson]
Sproull, R. F. and Lampson, B. W., 'An open operating system for a single-user machine,' *Proceeding of the Seventh Symposium on Operating System Principles*, Asilomar, December 1979.

XEROX

Xerox Corporation
Office Systems Division
Palo Alto, California 94304

XEROX® is a trademark of
XEROX CORPORATION.

Printed in U.S.A.