ULTRIX

digital

Guide to Network Programming

Order Number: AA–PBKWA–TE

ULTRIX

Guide to Network Programming

Order Number: AA-PBKWA-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

This manual introduces the programmer to the architectures and components of the ULTRIX network programming environment. It discusses how network layering schemes, socket interface, and ULTRIX network components can be used in writing network applications.

digital equipment corporation
maynard, massachusetts

Contents

About This Manual

1 Introduction to Network Architecture

2 Using the Socket Interface

7 ULTRIX Extended SNMP Agent

8 Packet Filter Programming

Examples

Figures

Tables

About This Manual

This manual introduces the programmer to the architectures and components of the ULTRIX network programming environment. It discusses how network layering schemes, socket interface, and ULTRIX network components can be used in writing network applications.

Audience

This guide is intended for experienced programmers who want to write network application programs within the ULTRIX environment. Readers should be familiar with the C programming language and ULTRIX networking concepts.

Organization

This guide consists of eight chapters:

Chapter 1: Introduction to Network Architecture
: This chapter describes the network architecture that is used by Digital and how it relates to the ISO Reference Model for Open System Interconnection (OSI).

Chapter 2: Using the Socket Interface
: This chapter describes how the socket interface can be used to write network applications. The system calls and network library routines that are used for socket-based applications are described.

Chapter 3:Advanced Socket Topics
: This chapter describes some of the facilities described in Chapter 2 in more detail. It also describes facilities that were not described in Chapter 2. The examples used in this chapter are taken from the Berkeley paper *An Advanced 4.3BSD Interprocess Communication Tutorial.*

Chapter 4: Client/Server Model
: This chapter contains code examples for the client/server model. The examples include connection-mode client, connection-mode server, connectionless-mode server, and inetd daemon. These code examples are taken from the Berkeley paper *An Advanced 4.3BSD interprocess Communication Tutorial.*

Chapter 5: X/Open Transport Interface
: This chapter describes how the X/Open transport interface (XTI) fits within the ULTRIX networking environment. It describes what the X/Open transport interface is and points to other documentation for XTI programming details.

Chapter 6: Writing Distributed Applications with DECrpc
: This chapter briefly describes the programming interface to DECrpc, the remote procedure call mechanism supported by the ULTRIX operating system. DECrpc Version 1.0 is based on and is compatible with the RPC component of Apollo's Network Computing System (NCS) Version 1.5, which is a set of tools for heterogeneous distributed computing.

The chapter points to other documentation for programming details.

Chapter 7: ULTRIX Extended SNMP Agent
This chapter describes how to write an Extended SNMP Agent, an extension of the Simple Network Management Protocol (SNMP) Agent. The Extended SNMP Agent allows you to manage a private Management Information Base.

Chapter 8: Packet Filter Programming
This chapter briefly describes the packet filter pseudodevice driver, a kernel-resident network packet demultiplexer supported by the ULTRIX operating system. The packet filter provides a raw interface to Ethernets and similar network data link layers. Packets received that are not used by the kernel (for example, to support the IP and DECnet protocol families) are available through this mechanism.

Related Documents

You should have available the documents in the ULTRIX documentation set, including the *ULTRIX Reference Pages*, appropriate C programming documentation, and the *Guide to X/Open Transport Interface*, and DECrpc documentation set.

Conventions

The following conventions are used in this guide:

macro	In text, bold type is used to introduce new terms.
{ | }	In syntax descriptions and function definitions, braces enclose lists from which one item must be chosen. Vertical bars are used to separate items.
`cat`(1)	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to `cat`(1) indicates that you can find the material on the `cat` command in Section 1 of the reference pages.
`user input`	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in interactive examples to indicate system output and also in code examples and other screen displays. In text, this typeface is used to indicate the exact name of a command, option, partition, pathname, directory, or file.
rlogin	In syntax descriptions and function definitions, this typeface is used to indicate terms that you must type exactly as shown.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.

filename	In examples, syntax descriptions, and function definitions, italics are used to indicate variable values; and in text, to give references to other documents.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.

New and Changed Information

This is a new manual.

Introduction to Network Architecture 1

A **network** consists of two or more computing systems linked together for the purpose of exchanging information and sharing resources. Network activity involves the flow of information between systems. Data originated on one system is routed through the network until it reaches its destination on another system.

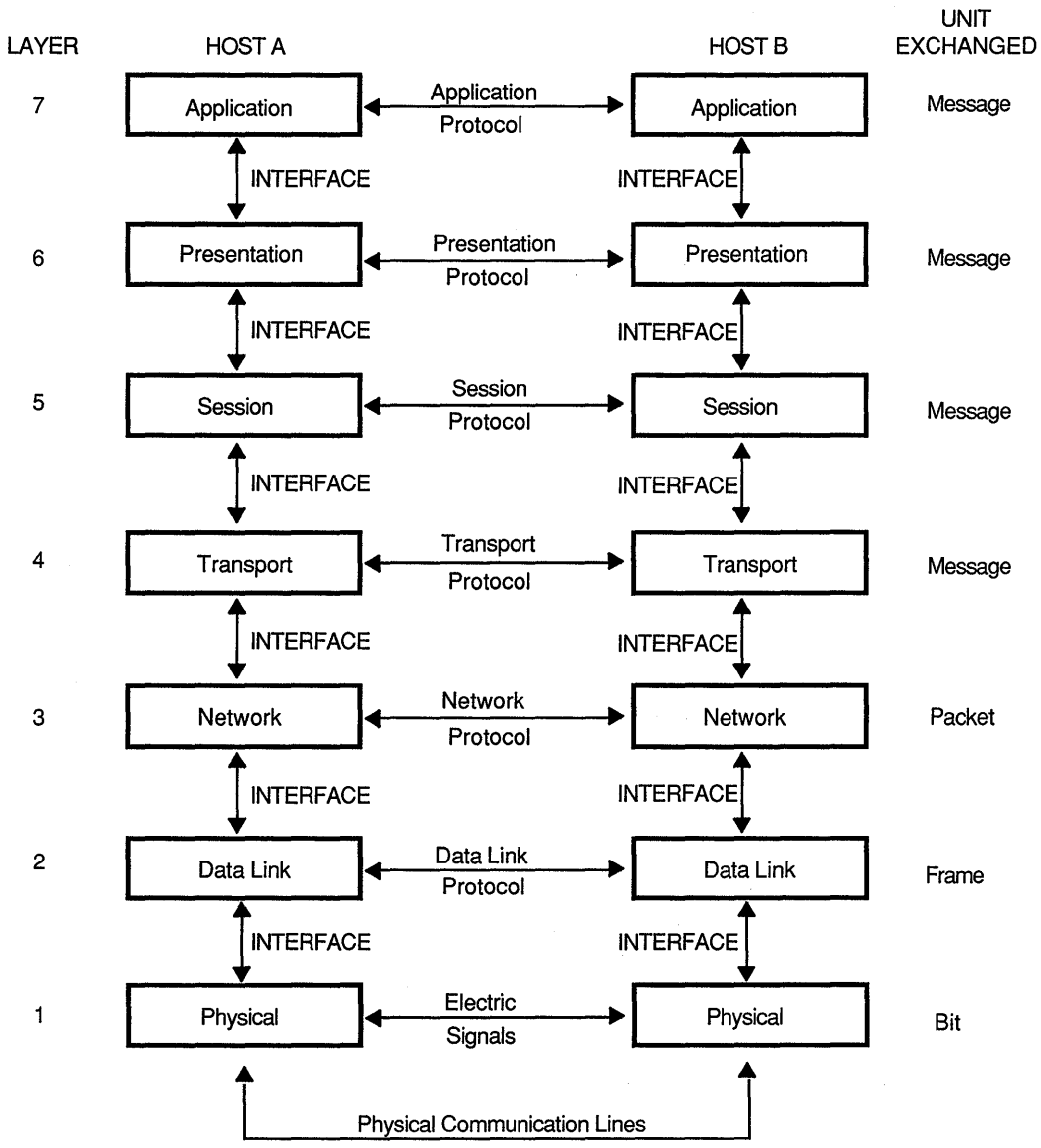
Each system on the network is called a host. Every host has a unique name and address. Hosts in the network are connected by communication **lines** over which communications take place. There are two types of communications, connection-oriented and connectionless-oriented (datagrams). **Connection-oriented** communications use virtual circuits for transmission. Virtual circuits provide a perfect channel by providing automatic sequencing, error control, and flow control.

Connectionless-oriented communications provide a datagram mode of communications within the environment of a computer network or networks. Datagram transmission is transaction-oriented, and delivery and duplication protection are not guaranteed. The datagrams do not use connections but attempt to deliver each datagram as an isolated message. The messages may arrive out of order or not at all.

A computing system can run many different **processes** (running programs). For two processes to communicate with each other, they have to establish contact and exchange data. A common way for processes to communicate with each other is to use a **client/server model**. In this scheme, client processes request services from a server process.

For processes to communicate with each other, there have to be rules of procedures stating just how two or more processes are suppose to interact, for example, how to send messages to each other. A set of agreements for interaction is known as a **protocol**. Individual protocols that work together are known as the **network architecture**. The network architecture, which consists of layers, is the complete definition of all the layers necessary to build the network. Each definition is a set of protocols that act within the same layer or between layers. The interaction between adjacent layers is known as an **interface**. Figure 1-1 shows how protocol communications take place between two hosts.

Figure 1-1: Protocol Communications between Two Hosts



ZK-0147U-R

1.1 ISO Reference Model

The International Organization for Standardization (ISO) has developed a 7-layer model for a standard network architecture, the ISO Reference Model for Open System Interconnection (OSI). The purpose of the ISO Reference Model is to enable systems from different manufacturers to work together. The model provides an architectural basis for the development of standards, which different systems can implement in order to provide the user with true distributed processing.

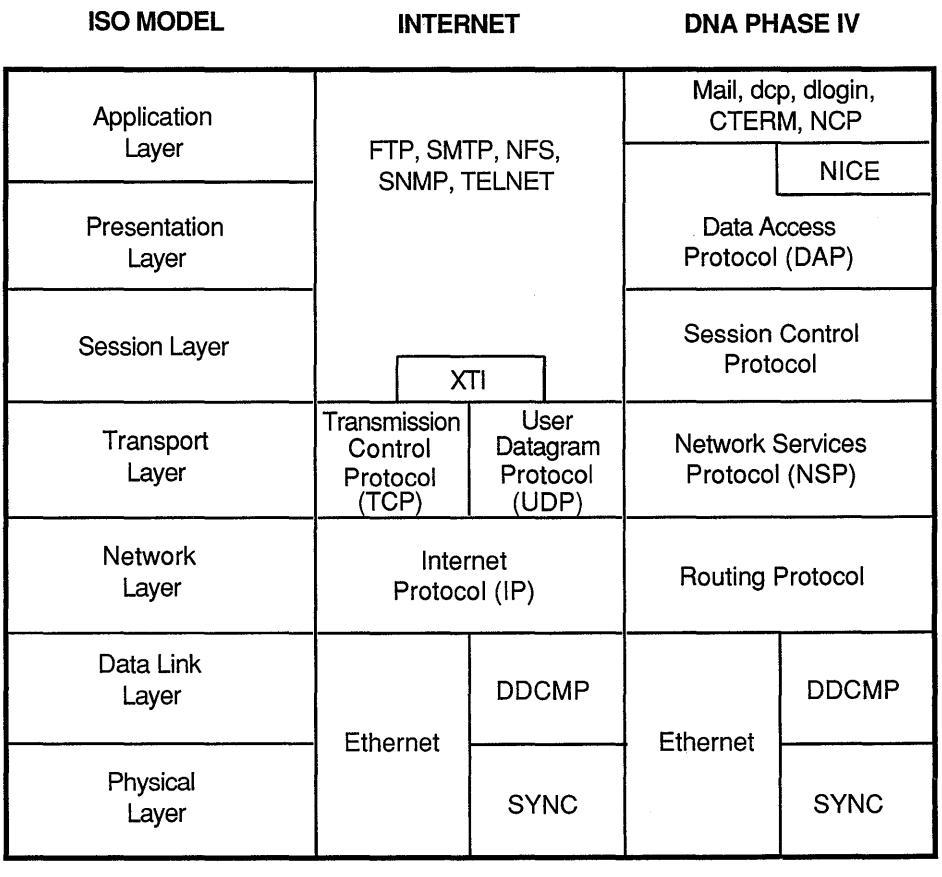
Being a hierarchical structure, the OSI architecture consists of layers. Each **layer** consists of an independent set of related functions, with its own characteristic purpose, protocols, and functions. The functions within a layer are spread across the systems connected in the network.

In this structure, each layer provides a **service** to the layer immediately above, known as its user, and is directly used only by that layer. In other words, each layer uses the service of the layer immediately below and adds functional value to the layer immediately below. The **protocols** communicate with same layer on other hosts. For example, the transport layer protocol on host A communicates with the transport layer on host B.

In OSI, each layer provides two types of standards: service and protocols for use within the layer. A system that supports the OSI standards is known as an **open system**. In an **open systems network**, each system supports the protocol standards.

The ISO Reference Model consists of seven layers. Figure 1-2 shows how the ISO Reference Model relates to the Internet and DNA architectures.

Figure 1-2: ISO Reference Model



ZK-0148U-R

1.2 Application Layer

The application layer is the highest layer in the OSI architecture. The other six layers support this layer, because this layer directly serves the end user by providing the distributed information service appropriate to an application, to its management, and to system management. Management is defined as the functions required to initiate, maintain, terminate, and record data concerning the establishment of connections for data transfer between application processes.

An application consists of cooperating **application processes**, which intercommunicate according to application layer protocols. Application processes are the ultimate sources and destinations for data exchanged. Examples of applications include electronic mail or file transfer program.

1.3 Presentation Layer

The presentation layer provides the set of services that can be selected by the application layer to enable it to interpret the meaning of data exchanged. Typical examples include standard routines that compress text or convert graphic images into bit streams for transmission across a network.

1.4 Session Layer

The session layer supports the interactions between cooperating presentation entities by providing the session administration service and session dialog services. The session administration service binds and unbinds two presentation entities into and out of a relationship. The session dialog service controls the data exchange, any delimiting, and the synchronizing of data operations between two presentation entities.

1.5 Transport Layer

To transfer data between presentation entities, the session layer uses the services provided by the transport layer. The transport layer provides transparent transfer of the data between session entities by relieving the session entities of any concern with how reliable and cost-effective transfer of data is done. In other words, this layer provides transparent data transfer between a source and destination host and can provide optional error recovery and flow control for the data transfer.

1.6 Network Layer

The network layer provides functional and procedural means to exchange network service data units between two transport entities of a network connection. It provides transport entities with independence from routing and switching considerations.

1.7 Data Link Layer

The data link layer provides the functional and procedural means to transfer data along network links between defined points on the network.

1.8 Physical Layer

The physical layer provides mechanical, electrical, functional, and procedural characteristics to establish, maintain, and release the physical connections between data link entities.

Using the Socket Interface 2

The ULTRIX socket interface allows network-based applications to be written independently of the underlying communication facilities. Therefore, the system can support communication networks that use different sets of protocols, different naming conventions, and different hardware. Because of this heterogeneous environment, a communication domain, also known as an address family, must be defined to provide the standard semantics of communication and naming.

For communication to take place between two processes, an endpoint of communication is required. An abstraction known as a socket is the end point of communication, from which messages are sent and received. Sockets are created within an address family like files are created within a filesystem. When an application program requests the ULTRIX operating system to create a socket, a small integer (descriptor) is returned. The application program can use the descriptor to reference the newly created socket. Whenever possible, sockets behave exactly like ULTRIX files or devices, so they can be used with traditional operations like `read` and `write`.

Applications can request the mode of communication, such as datagrams or virtual circuits. The application can provide a destination address each time it uses the socket (for example, when sending datagrams), or it can bind the destination address to the socket and avoid specifying the destination repeatly (for example, when using a TCP connection). The mode of communication is determined by the type of socket.

2.1 Writing Network Applications

The most commonly used paradigm in constructing distributed applications is the client/server model. The requester, known as a **client**, sends a request to a server and waits for a response. The **server** is an application-level program that offers a service that can be reached over the network. Servers accept requests that arrive over the network, perform their service, and return the result to the requester.

An application program interacts with ULTRIX by making system calls. System calls look and behave exactly like other procedure calls. They take arguments and return one or more results. Arguments can contain values (for example, an integer count) or pointers to objects in the application program (for example, a buffer to be filled with characters). The sequence of calling the system calls to establish communication depends upon two major considerations: client or server program and connection or connectionless mode of communication. Tables 2-1 and 2-2 list the calling sequence for each case.

Note

It is highly recommended, as you read this chapter, that you also refer to the reference pages for the respective system calls.

Table 2-1: Calling Sequence for Connection Mode

Operation	Client Call	Server Call
Create socket	Socket()	Socket()
Bind address	Bind()	Bind()
Define listener		Listen()
Request connection	Connect()	
Accept Connection		Accept()
Transfer data	Write()	Write()
	Read()	Read()
	Send()	Send()
	Recv()	Recv()
	Sendmsg()	Sendmsg()
	Rcvmsg()	Rcvmsg()
Discard socket	Close()	Close()
	Shutdown()	Shutdown()

Table 2-2: Calling Sequence for Connectionless Mode

Operation	Client Call	Server Call
Create socket	Socket()	Socket()
Bind address	Bind()	Bind()
Transfer Data	Sendto()	Sendto()
	Recvfrom()	Recvfrom()
	Sendmsg()	Sendmsg()
	Recvmsg()	Recvmsg()
or		
Set destination	Connect()	
Transfer Data	Sendto()	Sendto()
	Recvfrom()	Recvfrom()
Discard socket	Close() Shutdown()	Close() Shutdown()

2.1.1 Creating a Socket

Issuing a `socket` system call creates a socket on demand. The socket call takes three integer arguments: address family (*af*), type (*type*), and protocol (*protocol*). It returns an integer result. It has the form:

```
s = socket(af, type, protocol)
```

Argument *af* specifies the address family (see Table 2-3) to be used with the socket. This argument depends on the environment in which the application is working.

Table 2-3: Socket Address Families

Address Family	Description
AF_DECNET	DECnet
AF_DLI	Direct data link interface
AF_IMPLINK	ARPANET IMP addresses
AF_INET	Internetwork (UDP,TCP)
AF_UNIX	Local to host (pipes)

Argument *type* is selected according to the characteristic properties required by the application. For example, if reliable communication is required, a stream socket might be selected. See Table 2-4. It is assumed that processes communicate only between sockets of the same type, although nothing prevents communication between sockets of different types should the underlying communication protocols support this.

Table 2-4: Socket Types

Socket Type	Description
Datagram	Supports bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated. An important characteristic of a datagram socket is that record boundaries in data are preserved.
Stream	Provides bidirectional, reliable, sequenced, and unduplicated flow of data, without record boundaries.
Sequenced Packet	Provides properties similar to the stream socket, except the sequenced packet preserves the boundaries.
Raw	Provides users access to the underlying communication protocols that support socket abstractions. These sockets are normally datagram-oriented, although their characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they are used mainly for developing new communication protocols or for gaining access to the inner workings of existing protocols.

The *protocol* argument specifies the appropriate protocol for the address family and socket type. Protocols are indicated by well known constants specific to each address family. If the protocol is left unspecified (a value of zero), the system selects an appropriate protocol from those protocols that the address family comprises and that may be used to support the requested socket type. The default protocol (protocol argument to the `socket` call is 0) should be correct for most situations. However, it is possible to specify a protocol other than the default. See Section 3.1, `protocols()`, and `services()` on specifying other protocols.

Normally, a socket blocks, that is, it places the process into suspension until the I/O completes. A socket can be marked as nonblocking to have the socket call return before the process completes, See Section 3.3 for a description on marking sockets as nonblocking.

For additional information, see socket(2) for creating a socket.

2.1.2 Binding an Address

A socket is created without a name (address). Until an address is bound to a socket, processes have no way to reference a socket and, consequently, no messages can be received on it. Applications can explicitly specify a socket's address or can permit the system to assign one. Socket addresses can be reused if the address family permits, although address families normally ensure that a socket address is unique on each host, so that the association between two sockets is unique within the address family. The address to be bound to a socket must be formulated in a socket address structure. Applications find addresses of well-known services by looking up their names in a database. The format of addresses can vary among address families; to permit a wide variety of different formats, the system treats addresses as variable-length byte arrays.

Communicating processes are bound by an **association**. An association is a logical binding between two communication endpoints that must be established before communication can take place. Associations can be long-lived, such as in virtual circuit-based communication, or short-lived, such as in a datagram-based communication.

The `bind` system call is used to associate (bind) an address to a socket. It has the form:

```
bind(s, name, namelen)
```

The *s* argument is the integer descriptor of the socket to be bound. The *name* argument is a structure that specifies the local address to which the socket should be bound, and the *namelen* argument specifies the length of the address in bytes. Interpretation of the address may vary from address family to address family. For example, *name* would contain Internet address and port number in the Internet address family (AF_INET), whereas, in the UNIX address family (AF_UNIX), it would contain a pathname.

See Section 3.2 and `bind(2)` for additional information on binding addresses to sockets.

2.1.3 Establishing a Connection

In connection-oriented communication, the process that initiates a connection request is called a client process, and the process that receives a connection request is called a server process. The server offers its services, advertises, by binding a socket to a

well-known address associated with the service and then passively **listens** on its socket. An unrelated process can make a connection with the server. The client requests services from the server by initiating a **connection** to the server's socket. On the client side, the `connect` call is used to initiate a connection. It has the format:

```
connect(s, name, namelen)
```

The *s* argument is the integer descriptor of the socket to connect. The *name* argument is a structure that specifies the destination address to which the socket should be bound. The *namelen* argument specifies the length of the address in bytes.

If the socket of the client is unbound at the time of the connect call, the system automatically selects and binds a name to the socket, if necessary. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer can begin. If a connection attempt fails, an error value is returned in *errno*. See `socket(2)` for the possible errors.

For the server to receive a client's connection, it must perform two steps after binding its socket. The first step is to indicate a willingness to listen for incoming connection requests. This is done with a `listen` call. It has the form:

```
listen(s, backlog)
```

The *backlog* parameter of the `listen` call specifies the maximum number of pending connections that should be queued for acceptance. Should a connection be requested while the queue is full, the connection is not refused, but, rather, the individual messages which make up the request are ignored. This gives a server time to make room in its pending connection queue while the client retries the connection request.

With a socket marked as listening, a server may `accept` a connection. It has the format:

```
accept(s, addr, addrlen)
```

The *s* argument specifies the descriptor of the socket on which to wait for a connection. The *addr* argument points to a structure of type `sockaddr`, and *addrlen* points to an integer. When a connection request arrives, the system places the address of the requesting client into *addr* and sets *addrlen* to the length of the address. The system creates a new socket that has its destination connected to the requesting client and returns the new socket descriptor to the caller.

The `accept` call normally blocks. That is, the `accept` call does not return until a connection is available or the system call is interrupted by a signal to a process. Further, there is no way for a process to indicate if it accepts connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not want to communicate with the process.

For additional information, see the `listen(2)`, `connect(2)`, and `accept(2)` reference pages.

2.1.4 Transferring Data

After establishing a connection, a variety of calls can be used to send and receive data. Because the peer entity at each end of a connection is established, a user can send or receive a message without specifying the peer. The usual `read`, `readv`, `write`, and `writev` system calls can be used. The `write` call has the form:

```
write(d, buf, nbytes)
```

The *d* argument contains either an integer socket descriptor or a file descriptor. The *buf* argument contains a sequence of bytes to be sent, and the *nbytes* argument specifies the number of bytes. If the internal socket buffers are full when a `write` call is issued, the call blocks until data can be transferred.

The `read` call has the form:

```
read(d, buf, nbytes)
```

The argument *d* gives the socket descriptor or file descriptor from which to read data. The *buf* argument specifies where in memory to store the data, and the *nbytes* argument specifies the number of bytes to be read.

Note

The `read` operation can be used only when the socket is connected.

The `writev` call is same as the `write` call, except the `writev` call uses what is known as a `gatherwrite`. The `gatherwrite` allows an application program to write a message without copying the message into contiguous bytes. It has the form:

```
writev(d, iov, ioveclen)
```

The *d* argument is used the same as in the `write` call. The *iov* argument points to a **iovec** structure type that contains a sequence of pointers to blocks of bytes that form the message. See `write(2)` for the details of the **iovec** structure. The argument *ioveclen* specifies the number of entries in **iovec**.

The `readv` system call uses a `scatterread` operation. The `scatterread` operation allows the incoming data to be placed in noncontiguous locations. It has the form:

```
readv(d, iov, iovcnt)
```

The *d* argument is the same as in the `read` call. The *iov* argument points to a `iovec` structure type that contains a sequence of pointers to blocks of memory into which incoming data should be stored. See `read(2)` for details of the `iovec` structure. The *iovcnt* argument specifies the number of entries in *iov*.

In addition to `read`, `readv`, `write`, and `writev` calls, the `send` and `recv` can be used. The `send` call has the form:

```
send(s, msg, len, flags)
```

The *s* argument specifies the socket to use. The *msg* argument gives the address of bytes to be sent and the *len* argument specifies the number of bytes to be sent. The *flags* argument controls the transmission.

The `recv` call has the form:

```
recv(s, buf, len, flags)
```

The *s* argument specifies a socket descriptor from which data should be received. The *buf* argument specifies the address in memory into which the message should be placed and the *len* argument specifies the length of the buffer area. The *flags* argument allows the caller to control the reception.

The `send` and `recv` calls differ from the `read` and `write` calls in that both support an additional *flags* argument. The *flags* argument, defined in `<sys/socket.h>`, can be used to peek at incoming data on reception (MSG_PEEK), to send or receive out-of-band data (MSG_OOB), and to send data without network routing (MSG_DONTROUTE). See Section 3.6 for additional information on sending out-of-band data.

The `sendmsg` and `rcvmsg` system calls can also be used. These two calls support all that `read` and `write` system calls do and also provide `scatter-gather` operations, specifying and receiving addresses, and transmitting and receiving specially interpreted data, called access rights.

The `sendmsg` call has the format:

```
sndmsg(s, msg, flags)
```

The *s* and *flags* arguments are used the same as in the `send` call. The difference is in the *msg* argument which contains the address of a `msghdr` structure that contains the address for the outgoing message as well as locations for the sender's address. See `recv(2)` for a description of the `msghdr` structure.

The `recvmsg` call has the format:

```
recvmsg(s, msg, flags)
```

The *s* and *flags* arguments are used the same as in the `recv` call. The *msg* call contains the address of a `msghdr` structure that contains the address for the incoming message as well as locations for the sender's address. This structure is the same as the one produced by the `sndmsg` call, allowing them to function as a pair.

For additional information, see `send(2)` for the `send` and `sendmsg` calls; `write(2)` for `write` and `writev` calls; `read(2)` for the `read` and `readv` calls; and `recv(2)` for `recv` and `recvmsg` calls.

2.1.5 Discarding Sockets

After a socket is no longer of interest, it can be discarded with the `close` call to the descriptor (*d*).

```
close(d)
```

In normal operation, closing a socket causes any queued but unaccepted connections to be discarded. If the socket is in a connected state, a disconnect is initiated. The socket is marked to indicate that a file descriptor is no longer referencing it, and the close operation returns successfully. When the disconnect request completes, the socket resources are reclaimed.

Alternatively, a socket can be marked explicitly to force the application process to **linger** when closing until there is no more data and the connection has shut down. This option is marked in the socket data structure, using the `setsockopt` system call with the *SO_LINGER* option. When an application indicates that a socket is to linger, it also specifies a duration for the lingering period. If the lingering period expires before the disconnect is completed, the socket shuts down and discards any data still pending.

See Section 2.1.8 for additional information on `setsockopt` call.

Should a user have no use for any pending data, it can perform a `shutdown` call on the socket prior to closing it. It has the form:

shutdown(s, how)

The *s* socket identifies the socket of the connection to be shut down. The *how* parameter is 0 if the user is no longer interested in reading data, 1 if no more data is to be sent, or 2 if no data is to be sent or received.

For additional information, see `close(2)` and `shutdown(2)`.

2.1.6 Using Connectionless Sockets

Sections 2.1.3 and 2.1.4 have dealt with sockets that use a connection-oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet-switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead of using the `send` and `recv` system calls, the `sendto` and `recvfrom` calls are used because they permit callers to specify or receive the address of the peer with whom they are communicating. These calls are most useful for connectionless sockets, where the peer can vary on each message transmitted or received.

Datagram sockets are created in the same way as described in Section 2.1.1. If a particular local address is needed, the `bind` operation must precede the first data transmission. Otherwise, the system sets the local address and port when data is first sent. To send data, the `sendto` call is used. It has the form:

```
sendto(s, msg, len, flags, to, tolen)
```

The *s*, *msg*, *len*, and *flags* arguments are used the same as the `send` call as described in Section 2.1.4. The *to* and *tolen* arguments point to a socket address structure and an integer. The *to* argument specifies destination address and *tolen* specifies its length. When using an unreliable datagram interface, it is unlikely that any errors are reported to the sender. When information is present locally to recognize a message that cannot be delivered (for instance when a network is unreachable), the call returns -1 and the global value *errno* contains an error number.

To receive messages on an unconnected datagram socket, the `recvfrom` call can be used. The `recvfrom` call is like the `recv` call, except `recvfrom` has two additional arguments that allow the caller to specify the destination from which data should be received. It has the form:

```
recvfrom(s, buf, len, flags, from, fromlen)
```

The first four arguments are the same as the `recv` call as described in Section 2.1.4. The two additional arguments, *from* and *fromlen*, point to a socket address structure and an integer respectively. ULTRIX uses *from* to record the address of the message sender and uses *fromlen* to record the length of the sender's address.

In addition to the two calls, `sendto` and `recvfrom`, datagram sockets can also use the `connect` call to associate a socket with a specific destination address. In this case, any data sent on the socket is automatically addressed to the connected peer, and only data received from that peer is delivered to the user. Only one connected address is permitted for each socket at one time; a second connect changes the destination address, and a connect to a null address (family AF_UNSPEC) disconnects. Connect requests on datagram sockets return immediately, as this simply

results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end-to-end connection). The `accept` and `listen` calls are not used with datagram sockets.

While a datagram socket is connected, errors from recent `send` calls may be returned asynchronously. These errors can be reported on subsequent operations on the socket, or a special socket option used with `getsockopt`, SO_ERROR, can be used to interrogate the error status.

Two other system calls can be used with datagrams: `sendmsg` and `recvmsg`. The `sendmsg` call is used instead of the `sendto` call, when a long list of arguments are to be sent. It has the form:

```
sendmsg(s, msg, flags)
```

The *s* and *flags* arguments are used the same as in the `sendto` call. The *msg* argument is a `msghdr` structure that contains the message to be sent. The `msghdr` structure is used to minimize the the number of directly supplied parameters. See `recv(2)` reference pages for a description of the `msghdr` structure. This call is especially useful when used with the `recvmsg` call, because they both can produce a message structure in the same format.

The `recvmsg` call functions like the `recvfrom` call but requires fewer arguments. It has the format:

```
recvmsg(s, msg, flags)
```

The *s* and *flags* arguments are used the same as in the `recvfrom` call. The *msg* argument points to a `msghdr` structure that holds the address for the incoming message. See `recv(2)` for a description of the `msghdr` structure.

For additional information, see `send(2)` for the `sendto` and `sendmsg` calls; `recv(2)` for the `recvfrom` and `recvmsg` calls.

2.1.7 Obtaining Local and Remote Socket Addresses

Newly created processes inherit the set of open sockets from the process that created them. If the newly created process needs to determine the address of the destination to which a socket connects, `getpeername` must be called. It has the form:

```
getpeername(s, name, namelen)
```

The *s* argument specifies the socket for which the address is desired. The *name* argument points to a `sockaddr` structure that receives the socket address. The *namelen* argument points to an integer that receives the length of the address. The `getpeername` system call works only with connected sockets.

The `getsockname` system call allows a process to determine the local address of a socket. It has the form:

```
getsockname(s, name, namelen)
```

The *s* argument specifies the socket for which the local address is desired. The *name* argument points to a `sockaddr` structure that contains the address, and argument *namelen* points to an integer that contains the length of the address.

For additional information, see `getpeername(2)` and `getsockname(2)`.

2.1.8 Obtaining and Setting Socket Options

The `getsockopt` system call allows the application program to request information about the socket. A caller specifies the socket, the option of interest, and a location at which to store the requested information. ULTRIX examines its internal data structures for the socket and passes the requested information to the caller. It has the form:

```
getsockopt(s, level, optname, optval, optlen)
```

The *s* argument specifies a socket for which information is needed. The *level* argument identifies whether the operation applies to the socket itself or to the underlying protocol being used. In most cases, it is the *socket level* indicated by the symbolic constant SOL_SOCKET, defined in `<sys/socket.h>`. The *optname* argument specifies a single option to which the request applies. See `socket(2)` for the supported options. The *optval* and *optlen* arguments contain pointers. The first pointer points to a buffer into which the system places the requested value, and the second points to an integer into which the system places the length of the option value.

The `setsockopt` system call allows an application program to set a socket option using the same set of values obtained with `getsockopt`. The caller specifies a socket for which the option should be set, the option to be changed, and a value for the option. It has the form:

```
setsockopt(s, level, optname, optval, optlen)
```

The arguments for `setsockopt` are the same as `getsockopt` except the *optlen* argument contains the length of the option being passed to the system. The caller must supply a legal value for the option as well as a correct length for that value.

To determine the type (for example, stream or datagram) of an existing socket, the SO_TYPE socket option and `getsockopt` call can be used as follows:

```
#include <sys/types.h>
#include <sys/socket.h>

if(getsockopt(s,SOL_SOCKET,SO_TYPE,(char*)&type,&size)<0){
  perror("getsockopt");
  exit(1);
}
```

After the `getsockopt` call, *type* sets the value of the socket type, as defined in `<sys.socket.h>`. If, for example, the socket was a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

For additional information, see `getsockopt(2)` for `getsockopt` and `setsockopt` calls.

2.1.9 Handling Multiple Services

A `select` system call allows a single server process to wait for connections on multiple sockets. `select` call selects blocks waiting on one of a set of file descriptors to become ready. It has the form.

```
nfound = select(nfds, readfds, writefds, exceptfds, timeout)
```

The arguments of the `select` call are:

- *nfds* specifies how many descriptors should be examined (the descriptors checked are 0 through *nfds-1*).
- *readfds* points to a bit mask that specifies the file descriptor to check for reading.
- *writefds* points to a bit mask that specifies the file descriptors to check for writing.
- *exceptfds* points to a bit mask that specifies the file descriptors to check for exception conditions.
- *timeout*, when it is nonzero, is the address of an integer that specifies how long to wait for a connection before returning to the caller. A zero value forces the call to block until a descriptor becomes ready. Because the *timeout* argument contains the address of the timeout integer and not the integer itself, a process can request zero delay by passing the address of an integer that contains zero (that is, a process can poll to see if I/O is ready).

Issuing a `select` system call returns the number of file descriptors from the specified set that are ready for I/O. It also changes the bit masks to which *readfds*, *writefds*, and *exceptfds* point, to inform the application which of the selected file descriptors are ready. Thus, before calculating `select`, the caller must turn on those bits that correspond to descriptors to be checked. Following the call, all bits that remain set to 1 correspond to a ready file descriptor.

Example 2-1 shows reading data from two sockets, *s1* and *s2*, as it is available from each and with a 1-second timeout.

Example 2-1: Reading Data from Two Sockets

```
#include <sys/time.h>
#include <sys/types.h>

fd_set read_template;
struct timeval wait;

for(;;){
   wait.tv_sec=1;     /* one second*/
   wait.tv_usec=0;

   FD_ZERO(&read_template);

   FD_SET(s1,&read_template);
   FD_SET(s2,&read_template);

   nb = select(FD_SETSIZE, &read_template,(fd_set*)0,(fd_set*)0,&wait);
   if(nb <= 0){
     perror("select");
     exit(1);              /* An error occurred during the select, or
                           the select timed out. */

     }

   if(FD_ISSET(s1,&read_template)){
            /* Socket #1 is ready to be read from. */
     .
        .
        .
   }

   if(FD_ISSET(s2,&read_template)){
              /* Socket #2 is ready to be read from. */
        .
        .
        .
     }
}
```

The arguments to `select` point to integers instead of pointing to *fd_sets*. This type of call works as long as the number of file descriptors being examined is less than the number of bits in an integer.

The `select` call provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the `fcntl` system call. See Section 3.5 for information on asynchronous output; see `select(2)` and `fcntl`(2) for additional information.

2.2 Network Library Procedures

In addition to the system calls, ULTRIX provides a set of library procedures that provide useful network functions. The difference between system calls and library procedures is that system calls pass control to the computer's operating system, whereas library procedures are like other procedures which are part of an application program.

The network library procedures provide access to network databases. This access allows processes to obtain information about host names, protocol port numbers,

network services, and other related information.

2.2.1 Obtaining Host Information

A set of library procedures allows a process to retrieve information about a host, given either a hostname or a host's Internet address. When used with a host on the same Internet, the library procedures make the process a client of the host by sending a request to a server and waiting for a response. When used on systems that are not on the Internet, the routines obtain the desired information from a database kept locally on disk.

The `gethostbyname` procedure takes a host name as an argument and returns a pointer to a `hostent` structure for that host. It has the form:

```
ptr_host = gethostbyname(name)
```

The *name* argument points to a character string that contains a domain name for the host. The value returned, *ptr_host*, points to the following structure:

```
struct    hostent{
     char    *h_name;     /* official name of host */
     char **h_aliases;    /* alias list */
     int  h_addrtype;  /* host address type (for example, AF_INET) */
     int  h_length; /* length of address */
     char **h_addr_list; /* list of addresses, null terminated */
};
```

The `gethostbyaddr` procedure produces the same information as the `gethostbyname` procedure. The difference is that `gethostbyaddr` accepts a host address as an argument. It has the form:

```
ptr_host = gethostbyaddr(addr, len, type)
```

The *addr* argument points to a sequence of bytes that contains a host address. The *len* argument contains an integer that gives the length of the address, and the *type* argument contains an integer that specifies the type of the address (for example, AF_INET).

Three other procedures allow a user process to read the hosts database (/etc/hosts) sequentially. The client passes a nonzero argument to the `sethostent` procedure to establish a connection to the database and starts at the beginning of the database. The client then retrieves one entry at a time and finally closes the connection. It has the form:

```
sethostent(stayopen)
```

If the integer argument, *stayopen*, is nonzero, the database remains open after calling `sethostent` to search for an entry.

The `gethostent` procedure allows a process to retrieve entries sequentially, one at a time, and returns the same structure as the other retrieval requests. It has the form:

```
ptr_host = gethostent()
```

The `endhostent` procedure closes the connection to the host database. It has the form:

```
endhostent()
```

After a connection has been closed with `endhostent`, the client can call `sethostent` to create a new connection and start again at the beginning of the database.

See `gethostent(3n)` for additional information on `gethostent`, `gethostbyaddr`, `gethostbyname`, `sethostent`, and `endhostent` calls.

2.2.2 Obtaining Network Information

ULTRIX hosts keep a database of networks (`/etc/networks`). A set of network library procedures allows a process to access the network database. The `getnetbyname` procedure obtains and formats the contents of an entry from the database given the name of a network. It has the form:

```
ptr_network = getnetbyname(name)
```

The *name* argument points to a string that contains the name of the network for which information is desired. The `getnetbyname` procedure returns a pointer to a `netent` structure. The `netent` structure has the format:

```
struct    netent{
     char      *n_name;  /* official name of net */
     char      **n_aliases;    /* alias list */
     int       n_addrtype;     /* net address type */
     int       n_net;     /* network number, host byte order */
};
```

The `getnetbyaddr` procedure searches for information about a network, given its address. It has the form:

```
ptr_network = getnetbyaddr(net, type)
```

The *net* argument contains a 32-bit network address, and the *type* argument contains an integer that specifies the type of network address.

Some procedures allow sequential access of the network database. Procedure `setservent` allows the calling process to open the network database and move to the beginning. It has the form:

```
setnetent(stayopen)
```

If the integer argument, *stayopen*, is nonzero, the database remains open after calling `setnetent` to search for an entry.

The `getnetent` procedure allows a process to retrieve entries sequentially, one at a time, and returns the same structure as the other retrieval requests. It has the form:

```
ptr_network = getnetent()
```

The `endnetent` procedure closes the connection to the network database. It has the form:

```
endnetent()
```

See `getnetent(3n)` for additional information on `gethostent`, `getnetbyaddr`, `getnetbyname`, `setnetent`, and `endnetent`.

2.2.3 Obtaining Protocol Information

Some routines provide access to the protocols database (`/etc/protocols`). The `getprotobyname` procedure allows a caller to obtain information about a protocol, given its name:

```
ptr_protocol = getprotobyname(name)
```

The *name* argument points to an ASCII string that contains the name of the protocol for which information is desired. The procedure returns a pointer to a `protent` structure. The `protent` structure has the format:

```
struct   protent{
     char      *p_name;           /* official protocol name */
     char      **p_aliases;     /* alias list */
     int       p_proto;           /* protocol number */
};
```

The `getprotobynumber` procedure allows a process to search for protocol information using protocol number (*proto*) as a key:

```
ptr_protocol = getprotobynumber(proto)
```

Some procedures allow sequential access to the protocols database. Procedure `setprotoent` allows the calling process to open the protocols database and move to the beginning. It has the form:

```
setprotoent(stayopen)
```

If the integer argument, *stayopen*, is nonzero, the database remains open after calling `setprotoent` to search for an entry.

The `getprotoent` procedure allows a process to retrieve entries sequentially, one at a time, and returns the same structure as the other retrieval requests. It has the form:

```
ptr_protocol = getprotoent()
```

The `endprotoent` procedure closes the connection to the protocols database. It has the form:

```
endprotoent()
```

See `getprotoent(3n)` for additional information on `getprotoent`, `getprotobynumber`, `getprotobyname`, `setprotoent`, and `endprotoent` calls.

2.2.4 Obtaining Network Services Information

The network services routines provide information about services and protocol ports. Given the service name, the `getservbyname` procedure sequentially searches from the beginning of the database (`/etc/services`) until a matching service name (*name*) is found or until EOF is encountered. If a protocol name (*proto*) is also specified, a match is not encountered until both the service and protocol names are matched. It has the format:

```
ptr_service = getservbyname(name, proto)
```

The *name* argument specifies the address of a string that contains the name of the desired service, and the integer argument, *proto*, specifies the protocol with which the service is to be used. Normally, protocols are limited to TCP and UDP. The returned

value points to a servent structure. The servent structure has the format:

```
struct    servent{
     char      *s_name;  /* official service name */
     char      **s_aliases;    /* alias list */
     int       s_port;         /* port number, network byte order */
     char      *s_proto; /* protocol to use */
};
```

The getservbyport procedure allows the caller to obtain an entry from the services database, given the port number assigned to it. It has the form:

```
ptr_port = getservbyport(port,proto)
```

The *port* argument is the integer protocol port number assigned to the service. The *proto* argument specifies the protocol for which the service is desired.

Some procedures allow sequential access of the services database. Procedure setnetent allows the calling process to open the services database and move to the beginning. It has the form:

```
setservent(stayopen)
```

If the integer argument, *stayopen*, is a nonzero, the database remains open after calling setservent to search for an entry.

The getservent procedure allows a process to retrieve entries sequentially, one at a time and returns the same structure as the other retrieval requests. It has the form:

```
ptr_service = getservent()
```

The endservent procedure closes the connection to the services database. It has the form:

```
endservent()
```

See getservent(3n) for additional information on getservent, getservbyport, getservbyname, setservent, and endservent calls.

2.3 Converting Network Byte Order

Hosts can differ in the way they store integer quantities and the Internet defines a host-independent standard for byte order. The ULTRIX operating system provides library procedures that convert between the local host byte order and the network standard byte order. To be portable, programs must be written to call the conversion routines every time they copy an integer value from the local host to a network packet, or when they copy a value from a network packet to the local host.

All four conversion routines are functions that take a value as an argument and return a new value with the bytes rearranged. For example, to convert a long (4-bytes) integer from network byte order to local host byte order, you might call ntohl procedure (Network TO Host Long). See Table 2-4. It has the format:

```
hostlong = ntohl(netlong)
```

The *netlong* argument is a 4-byte (32-integer value) in network standard byte order. The result, *hostlong*, is in local host byte order.

Table 2-5: Byteorder Routines

Call	Description
htonl(hostlong)	Convert 32-bit quantity from host to network byte order
htons(hostshort)	Convert 16-bit quantity from host to network byte order
ntohl(netlong)	Convert 32-bit quantity from network to host byte order
ntohs(netshort)	Convert 16-bit quantity from network to host byte order

2.4 Manipulating Variable Length Byte Strings

The run-time library contains three routines that are used for manipulation of names. These routines are listed in Table 2-5.

Table 2-6: Manipulating Variable Length Byte String Routines

Call	Description
bcmp(s1,s2,n)	Compare byte-strings; 0 if same, non-zero otherwise
bcopy(s1,s2,n)	Copy n bytes from s1 to s2
bzero(base, n)	Zero-fill n bytes starting at base

Advanced Socket Topics 3

This chapter describes in more detail some of the facilities discussed in Chapter 2. It also describes advanced facilities that were not covered in Chapter 2.

3.1 Selecting Specific Protocols

The first two arguments of the `socket()` call (*af* and *type*) are described in Chapter 2. The third argument of the `socket` call specifies the protocol. If the third argument is 0, `socket` selects a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices may not be available. However, when using raw sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument can be important for setting up demultiplexing. For example, raw sockets in the Internet family can implement a new protocol above IP, and the socket receives packets only for the protocol specified. To obtain a particular protocol, you determine the protocol number as defined within the communication domain. For the Internet domain you can use one of the library routines discussed in Section 2.2, such as `getprotobyname`. Example 3-1 shows how a socket *s* that is using a stream-based connection can use protocol type newtcp instead of the default TCP.

Example 3-1: Specifying Another Protocol Type

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

     struct protoent *pp;

     pp = getprotobyname("newtcp");
     if((s = socket(AF_INET, SOCK_STREAM, pp->p_proto)) < 0) {
       perror("socket");
       exit(1);
     }
```

3.2 Binding an Address

The associations in binding addresses to sockets in the Internet domain are composed of local and foreign addresses and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. An association can be specified by specifying half of an association at a time. The two steps are:

1. Use the `bind` system call for a process to specify half of an association, the <local address, local port> part.

2. Use the `connect` and `accept` calls to complete a socket's association by specifying the <foreign address, foreign port> part.

Care must be exercised to ensure the association uniqueness requirements are not violated. Further, it is unrealistic to expect user programs to always know proper values for the local address and local port, because a host can reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain, a wildcard address can be used. When an address is specified in INADDR_ANY (a constant defined in `<netinet/in.h>`), the system interprets the address as *any valid address.* Example 3-2 shows how a `bind` call specifies a port number to a socket, but the local address is not specified.

Example 3-2: Specifying a Wildcard Address

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
struct sockaddr_in sin;

if((s = socket(AF_INET,SOCK_STREAM,0)) < 0){
  perror("socket");
  exit(1);
}
sin.sin_family = AF_INET;
sin.sin_port = htons(INADDR_ANY);
sin.sin_port = htons(MYPORT);
if(bind(s,(struct sockaddr*)&sin,sizeof(sin)) < 0){
  perror("bind");
  exit(1);
}
```

Sockets with wildcard local addresses can receive messages directed to the specified port number and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process accepts connection requests that are addressed to 128.32.0.4 or 10.0.0.78. If a server process only allows hosts on a given network to connect to it, it would bind the address of the host on the appropriate network.

Likewise, a local port can be left unspecified (specified as zero), in which case the system selects an appropriate port number for it. Example 3-3 shows binding a specific local address to a socket, but leaving the local port number unspecified.

Example 3-3: System Selects Local Port Number

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

struct hostent *hp;
struct sockaddr_in sin;
char *hostname="myhost";
#define NULL 0

hp = gethostbyname(hostname);
if(hp == NULL){
  perror("gethostbyname");
  exit(1);
}
bcopy(hp->h_addr, (char*)sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
if(bind(s,(struct sockaddr*)&sin,sizeof(sin)) < 0){
  perror("bind");
  exit(1);
}
```

The system selects the local port number using these rules:

- Internet ports below IPPORT_RESERVED (1024) are reserved for privileged users (that is, superusers); Internet ports above IPPORT_USERRESERVED (5000) are reserved for nonprivileged servers.
- The port number is not currently bound to some other socket.

To find a free Internet port number in the privileged range, the **rresvport** library routine can be used to return a stream socket with a privileged port number as shown in Example 3-4.

Example 3-4: Finding a Free Port Number

```
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>

int lport = IPPORT_RESERVED-1;
int s;

s = rresvport(&lport);
if(s<0){
     if(errno == EAGAIN)
          fprintf(stderr,"socket:all ports in use\n");
     else
          perror("rresvport: socket");
               exit(1);
}
```

To restrict port allocation allows processes executing in a secure environment to perform authentication based on the originating address and port number. For example, the `rlogin()` command allows users to log in to a remote host without being asked for a password. This can only happen if:

- The name of the host the user is logging in from is in the file `/etc/hosts.equiv` on the host that the user is logging in to (or the system name and the user name are in the user's `.rhosts` file in the user's home directory).
- The user's rlogin process is coming from a privileged port on the host from which the user is logging in.

The port number and network address of the host from which the user is logging in can be determined either by the *from* result of the `accept` call or from the `getpeername` call.

The algorithm used by the system in selecting port numbers for an application can be unsuitable because associations are created in a 2-step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation, the system disallows binding the local address and port number to a socket if a previous data connection's socket still exists. To override the default port selection algorithm, an option call is performed prior to address binding, as shown in Example 3-5.

Example 3-5: Overriding the Default Port Selection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
int on = 1;

struct sockaddr_in sin;

if(setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(on)) < 0) {
     perror("setsockopt");
     exit(1);
}
if(bind(s, (struct sockaddr*)&sin, sizeof(sin)) < 0){
  perror("bind");
  exit(1);
}
```

In Example 3-5, local addresses are bound that are already in use. This does not violate the uniqueness requirement, as the system checks at connect time any other sockets with the same local address and port that do not have the same foreign address and port. If the association already exists, the call returns the EADDRINUSE error.

3.3 Creating a Nonblocking Socket

Normally, sockets block. A blocking socket puts the process into suspension until the I/O request completes and returns an error. A nonblocking socket returns before the process completes. To create a nonblocking socket, a socket is created with a `socket()` call and marked with the `fcntl` call. Example 3-6 shows marking a socket as nonblocking.

Example 3-6: Marking a Socket as Nonblocking

```
#include <fcntl.h>
#include <sys/socket.h>

if((s = socket(AF_INET,SOCK_STREAM,0)) < 0) {
  perror("socket");
  exit(1);
}
if(fcntl(s,F_SETFL,FNDELAY)<0){
     perror("fcntl");
     exit(1);
}
```

When performing nonblocking I/O on sockets, you must check the global variable *errno* for the EWOULDBLOCK error. This error occurs when an operation tries to block with a nonblocking socket. In particular, `accept`, `connect`, and `write` can return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a `send` cannot be completed, but when partial writes are acceptable (for example, when using a stream socket), the data that can be sent immediately is processed, and the return value indicates the amount sent.

3.4 Notifying a Process of an I/O Request

The SIGIO signal notifies a process when a socket or file descriptor has data waiting to be read. To use the SIGIO facility, three steps are required:

1. Use `signal` or `sigvec` calls to have the process set up a SIGIO signal handler.
2. Use the `fcntl` call to set the process id or process group id that is to be notified of pending input to its own process id, or the process group id of its process group. The default process group of a socket is 0. See Section 3.5 for additional information.
3. Use another `fcntl` call to enable asynchronous notification of pending I/O requests.

Example 3-7 shows how a given process can receive information on pending I/O requests as they occur for a socket *s*. By adding a handler for SIGURG to the code in Example 3-7, it can be used to prepare for receipt of SIGURG signals.

Example 3-7: Asynchronous Notification of I/O Requests

```
#include <fcntl.h>
#include <signal.h>
int io_handler();

signal(SIGIO,io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if(fcntl(s,F_SETOWN,getpid())<0){
   perror("fcntl");
   exit(1);
}

/*Allow receipt of asynchronous I/O signals */
if(fcntl(s,F_SETFL,FASYNC)<0){
   perror("fcntl");
   exit(1);
}
```

3.5 Redefining a Process Number for a Socket

The SIGURG and SIGIO signals create an associated process number for each socket, just as it is done for terminals. This process number value is initialized to 0, but can be redefined with the F_SETOWN `fcntl`, as was done in Example 3-7. The process number indicates either the associated process id or the associated process group; it cannot specify both at the same time. To set the socket's process id for signals, positive arguments are given to the `fcntl` call. To set the socket's process group for signals, negative arguments are passed to `fcntl`. A similar `fcntl`, F_GETOWN, can be used to determine the current process number of a socket.

The SIGCHLD signal is sent to a process when any child process has changed state. Normally, servers use the signal to obtain child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Example 4-2 can be augmented as shown in Example 3-8.

If the parent server process fails to obtain its children, a large number of zombie processes can be created.

Example 3-8: Redefining Process Number for Socket

```
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <syslog.h>
#include <signal.h>

int reaper();


signal(SIGCHLD,reaper);
listen(f,5);
for(;;){
     struct sockaddr_in from;
     int g, len = sizeof(from);

     g = accept(f,(struct sockaddr*)&from,&len);
     if(g>0){
          if(errno != EINTR)
            syslog(LOG_ERR, "rlogind:accept: %m");
       continue;
      }
}

#include <sys/wait.h>
reaper()
{
     union wait status;

     while (wait3(&status, WNOHANG,0)>0)
          ;
}
```

3.6 Sending Out-of-Band Data

The ULTRIX stream socket supports **out-of-band** data. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data. The out-of-band data facilities support the reliable delivery of at least one out-of-band message at a time. This message can contain at least one byte of data, and at least one message can be pending delivery to the user at any one time. For communication protocols that support only in-band signaling (that is, the urgent data is delivered in sequence with the normal data), the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order or receiving it, out of sequence, without having to buffer all the intervening data.

It is possible to read ahead (peek) at out-of-band data using the MSG_PEEK flag. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the SIGURG signal, by means of the appropriate `fcntl` call, as described in Section 3.5. If multiple sockets can have out-of-band data awaiting delivery, a `select` call for exceptional conditions can be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual

arrival of the out-of-band data, but only that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushes any pending output from the remote processes, all data up to the mark in the data stream is discarded.

To send an out-of-band message, the MSG_OOB flag is supplied to a `send` or `sendto` calls. To receive out-of-band data, MSG_OOB should be indicated when performing a `recvfrom` or `recv` call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK `ioctl` is provided:

```
ioctl(s,SIOCATMARK,&yes)
```

If *yes* returns a 1, the next read returns data after the mark. Otherwise, (assuming out-of-band data has arrived), the next read provides data sent by the client prior to transmission of the out-of-band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Example 3-9. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

Example 3-9: Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/file.h>


oob(){
     int out = FWRITE;
     char waste[BUFSIZ], mark;
     int rem;

     ioctl(1,TIOCFLUSH,(char*)&out);
     for (;;){
          if(ioctl(rem,SIOCATMARK,&mark)<0){
               perror("ioctl");
            break;
             }
          if(mark)
               break;
          (void)read(rem,waste,sizeof(waste));
          if(recv(rem,&mark,1,MSG_OOB)<0){
              perror("recv");
              exit(1);
             }
     }
}
```

A process can read the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data and only sends the notification of its presence ahead. An example of this is the TCP protocol used to implement streams in the Internet domain. With such protocols, the out-of-band byte may not yet have arrived when a `recv` is done with the MSG_OOB flag. In that case, the call returns an error of EWOULDBLOCK. There can even be enough in-band data in the input buffer to prevent the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data to allow delivery of the urgent data.

Certain programs that use multiple bytes of urgent data and that must handle multiple urgent signals (for example, telnet) need to retain the position of the urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE; see `setsockopt(2)` for usage. With this option, the position of urgent data (the *mark*) is retained, but the urgent data immediately follows the mark within the normal data stream returned with the MSG_OOB flag. Reception of multiple urgent indications moves the mark, but no out-of-band data is lost.

3.7 Broadcasting Packets

A datagram socket can be used to send broadcast packets on networks that support broadcasting. Broadcast messages place a high load on a network, because they require every host on the network to service them. For a socket to send broadcast packets, it has to be explicitly marked for broadcasting. Broadcast is used to:

- Find a resource on a local network, without prior knowledge of its address.
- Perform important functions, such as sending routing information, to all accessible hosts.

Example 3-10 shows how to send a broadcast message.

Example 3-10: Broadcasting a Message

```
#include <sys/socket.h>
#include <netinet/in.h>
int on=1;


struct sockaddr_in sin;
if((s = socket(AF_INET,SOCK_DGRAM,0)) < 0) {      1
     perror("socket");
     exit(1);
}
if((setsockopt(s,SOL_SOCKET,SO_BROADCAST,&on,sizeof(on)) < 0) {      2
     perror("setsockopt");
     exit(1);
}
sin.sin_family=AF_INET;      2
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);      3
if(bind(s,(struct sockaddr*)&sin,sizeof(sin)) < 0){
     perror("bind");
     exit(1);
}
```

1 A datagram socket is created.

2 The socket is marked for broadcasting.

3 A port number is bound to the socket.

The destination address of the message to be broadcast depends on the network on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in `<netinet/in.h>`).

3.8 Determining Network Configuration

To determine the list of addresses for all reachable hosts requires knowledge of the networks to which the host is connected. The ULTRIX operating system provides a method of retrieving this information from the system data structure. The SIOCGIFCONF `ioctl` call returns the interface configuration of a host in the form of a single `ifconf` structure; this structure contains a **data area** which is made up of an array of `ifreq` structures, one for each network interface to which the host is connected. Example 3-11 shows the `ifreq` structure.

Example 3-11: ifreq Structure

```
struct ifconf{
     int  ifc_len;       /*size of associated buffer */
     union{
          caddr_t ifcu_buf;
          struct ifreq *ifcu_req;
          }ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf   /* buffer address */
#define ifc_req ifc_ifcu_req        /* array of structures returned */

#define IFNAMSIZ    16

struct ifreq{
     char ifr_name[IFNAMSIZ];  /* if name, for example, "len0" */
     union{
          struct    sockaddr ifru_addr;
          struct    sockaddr ifru_broadaddr;
          shortifru_flags;
       caddr_t ifru_data;
     } inf_ifru;
};

#define ifr_addr    ifr_ifru.ifru_addr   /* address */
#define ifr_dstaddr ifr_ifr.ifru_dstaddr /* other end of p-to-p link */
#define ifr_broadaddr    ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags   ifr_ifru.ifru_flags  /* flags */
#define ifr_data    ifr_ifru.infru_data  /* for use by interface */
```

Example 3-12 shows the call used to obtain the interface configuration.

Example 3-12: Obtaining the Interface Configuration

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>

     struct ifconf ifc;
     char buf[BUFSIZ];

     ifc.ifc_len = sizeof(buf);
     ifc.ifc_buf = buf;      1
     if(ioctl(s,SIOCGIFCONF,(char*)&ifc)<0) {
       perror("ioctl");
       exit(1);
     }
```

1 After the call, *buf* contains one `ifreq` structure for each network to which the host is connected and modifies *ifc.ifc_len* to contain the number of bytes used by the `ifreq` structures.

For each structure there exists a set of **interface flags** that tell whether the network corresponding to that interface is up or down, point-to-point, or broadcast. The SIOCGIFFLAGS `ioctl` retrieves these flags for an interface specified by an **ifreq** structure as shown in Example 3-13.

Example 3-13: Retrieving Interface Flags

```
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>

struct ifconf ifc;
struct ifreq *ifr;

ifr = ifc.ifc_req;

for(n = ifc.ifc_len/sizeof(struct ifreq); --n >= 0; ifr++){
      /*
       * We must be careful that we do not use an interface
       * devoted to an address family other than those intended;
       */
      if(ifr->ifr_addr.sa_family != AF_INET)
          continue;
      if(ioctl(s,SIOCGIFFLAGS,(char*)ifr)<0){
                  perror("ioctl");
                  exit(1);

      }
      /*
       * Skip boring cases.
      */
      if((ifr->ifr_flags & IFF_UP) ==0||
      (ifr->ifr_flags & IFF_LOOPBACK)||
      (ifr->ifr_flags & (IFF_BROADCAST|IFF_POINTOPOINT))==0)
        continue;
}
```

After the SIOCGIFBRDADDR `ioctl` call in Example 3-13 obtains the flags, the broadcast address must be obtained. In broadcast networks the SIOCGIFBRDADDR `ioctl` call is used, while in point-to-point networks SIOCGIFDSTADDR obtains the address of the destination host. Example 3-14 shows how SIOCGIFDSTADDR can be used.

Example 3-14: Obtaining Address of the Destination Host

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>

struct sockaddr dst;
struct ifreq *ifr;
char buf[BUFSIZ];
int buflen = sizeof(buf);

if(ifr->ifr_flags & IFF_POINTOPOINT){
       if(ioctl(s,SIOCGIFDSTADDR,(char*)ifr)<0){
                perror("ioctl");
                exit(1);
       }
 bcopy((char*)ifr->ifr_dstaddr,(char*)&dst,sizeof(ifr->ifr_dstaddr));
}else if (ifr->ifr_flags & IFF_BROADCAST){
      if(ioctl(s,SIOCGIFBRDADDR,(char*)ifr)<0){
                perror("ioctl");
                exit(1);
      }
 bcopy((char*)ifr->ifr_broadaddr,(char*)&dst,sizeof(ifr->ifr_broadaddr));
}
if(sendto(s,buf,buflen,0,(struct sockaddr*)&dst,sizeof(dst)) < 0){      1
  perror("sendto");
  exit(1);
}
```

1 The `sendto` call occurs for every interface to which the host is connected that supports broadcasting or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that in Example 3-14 can be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the sender's address and port, as datagram sockets are bound before a message is sent.

3.9 Using Pseudoterminals

Many programs require a terminal for standard input and output. Because sockets do not provide the semantics of terminals, a process may have to communicate over the network through a **pseudoterminal**. A pseudoterminal is a pair of devices, master and slave, that allows a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudoterminal is supplied as input to a process reading from the master side, while data written on the master side is processed as terminal input for the slave. The process manipulating the master side of the pseudoterminal controls the information read and written on the slave side, as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this is to preserve terminal semantics over the connection, that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudoterminals for remote login sessions as follows:

- A user logs in to a remote host and is provided a shell with a slave pseudoterminal as standard input, output, and error.

- The server process handles the communication between the programs invoked by the remote shell and the user's local client process.
- A user sends a character that generates a control message for the server process.
- The server then sends an out-of-band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under the ULTRIX operating system, the name of the slave side of a pseudoterminal is `/dev/ttyxy`. In the name, the x is a single letter starting at p and continuing to w and the letter y is a hexadecimal digit (that is, a single character in the range 0 through 9 or a through f). The master side of a pseudoterminal is `/dev/ptyxy`. The x and y correspond to the slave side of the pseudoterminal.

A method to obtain a pair of master and slave pseudoterminals is to find a pseudoterminal that is not currently in use. The master half of a pseudoterminal is a single-open device; thus, each master can be opened until an open succeeds, as follows:

- The slave side of the pseudoterminal is opened and is set to the proper terminal modes, if necessary.
- The process performs a `fork` operation.
- The child closes the master side of the pseudoterminal and performs an `execs` operation for the appropriate program.
- The parent closes the slave side of the pseudoterminal and begins reading and writing from the master side.

Example 3-15 shows code for a pseudoterminal. This code assumes that a connection on a socket *s* exists, connected to a peer that wants a service, and that the process has disassociated itself from any previous controlling terminal.

Example 3-15: Creation and Use of a Pseudoterminal

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <syslog.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <netinet/in.h>
#include <netdb.h>


int gotpty = 0;
int c; char *line;
struct stat statbuf;
int i, master, slave;
struct sgttyb b;

for(c='p'; !gotpty && c<= 'w';c++){
     line = "/dev/ptyXX";
     line[sizeof("/dev/pty")-1]=c;
     line[sizeof("/dev/ptyp")-1] = '0';
     if(stat(line, &statbuf)<0)
          break;
     for(i=0;i<16;i++){
          line[sizeof("/dev/ptyp")-1] = "0123456789abcdef"[i];
       master = open(line,O_RDWR);
       if(master > 0){
            gotpty = 1;
            break;
       }
     }
}

if(!gotpty){
     syslog(LOG_ERR,"All network ports in use");
     exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line,O_RDWR); /* slave is now slave side */
if(slave < 0){
     syslog(LOG_ERR, "Cannot open slave pty %s", line);
     exit(1);
}

ioctl(slave, TIOCGETP,&b); /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP,&b);

i = fork();
if(i<0){
     syslog(LOG_ERR,"fork:%m");
     exit(1);
}else if (i){
     close(slave);
}else if (i){   /* Parent */
     close(slave);
}else{          /*Child */
     (void)close(master);
```

Example 3-15: (continued)

```
        dup2(slave,0);
        dup2(slave,1);
        dup2(slave,2);
        if(slave > 2)
                (void)close(slave);
}
```

Client/Server Model 4

This chapter contains code examples for the client/server model. In the client/server model, the client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server. However, the communications can be either asymmetric or symmetric. In asymmetric communications, one side is recognized as the master, with the other side as the slave. An example of asymmetric communications is using the FTP protocol for an Internet file transfer. In symmetric communications, either side can function as the master or slave. An example of symmetric communications is using the TELNET protocol for remote terminal emulation.

An alternative scheme for a service server is to use the `inetd` daemon. See Section 4.4 for a description of the `inetd` daemon.

4.1 Client Code Example

Example 4-1 shows major segments of code that can be used by the client process. In the example, the client requests the login service from the server process.

Example 4-1: Remote Login Client Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
main(argc,argv)
     int argc;
     char *argv[];
{
     struct sockaddr_in server;
     struct servent *sp;
     struct hostent *hp;
     int s;
     ...
     sp = getservbyname("login", "tcp");     1
     if(sp == NULL){
              fprintf(stderr, "rlogin: tcp/login: unknown service\n");
              exit(1);
     }
     hp = gethostbyname(argv[1]);      2
     if(hp == NULL){
              fprintf(stderr, "rlogin: %s ; unknown host\n", argv[1]);
              exit(2);
     }
     bzero((char*)&server, sizeof(server));      3
     bcopy(hp->h_addr,(char*)&server.sin_addr,hp->h_length);
     server.sin_family = hp->h_addrtype;
     server.sin_port = sp->s_port;      4
     s = socket(AF_INET, SOCK_STREAM,0);      5
     if(s<0){
            perror("rlogin:socket");
            exit(3);
     }
     ...
     /* Connect does the bind() for us */

     if(connect(s,(char*)&server,sizeof(server))<0){ 6
          perror("rlogin:connect");
          exit(5);
     }
     ...
}
```

1 The `getservbyname` call is given *login* as a service name and *tcp* as a protocol name. It uses these names to sequentially search the `/etc/services` database for a matching service and protocol name. If a match is found, the call returns a pointer to a `servent` structure that contains the information about the service. If no match is found, the error message "rlogin:tcp/login:unknown service" is returned. See Section 2.2.4 for a description of the `servent` structure.

2 The `gethostbyname` is given the name of the remote host that the server wishes to make a connection to and searches the /etc/networks database for the host entry. Once finding the entry, it returns a pointer to a `hostent` structure that contains information about that host. If the call cannot find the entry, it returns the error message "rlogin: *hostname* : unknown host". See Section 2.2.1 for a description of the `hostent` structure.

3 The address of the host and address type are placed into the socket structure. This is done by clearing the address buffer and placing the Internet address of the remote host into the socket structure. See Section 2.4 for a description of the `bzero` and `bcopy` routines.

4 The socket number of the login process that resides on the remote host is placed into the socket structure.

5 A socket is created and a `bind` operation implicitly is performed to bind the remote socket (*s*).

6 The `connect` call initiates a connection request to the server's socket *s*. The second argument, *(char*)&server*, specifies the destination address to which the call implicitly binds the socket (*s*). If the connection request is successful, data transfer can begin.

4.2 Connection Server Code Example

The server process remains dormant by listening at a well known address for service requests. When the client process requests a connection to the server's address, it responds by servicing the client's request. The major code segments of the server program are shown in Example 4-2.

Example 4-2: Remote Login Server Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <syslog.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <sys/errno.h>
#include <sys/ioctl.h>
#include <sys/file.h>

main(argc,argv)
     int argc;
     char *argv[];
{
     int f;
     struct sockaddr_in from;
     struct sockaddr_in sin;
     struct servent *sp;
     sp = getservbyname("login", "tcp");      1
     if(sp == NULL){
           fprintf(stderr, "rlogin: tcp/login:unknown service0);
              exit(1);
     }

#ifndef DEBUG
     /* Disassociate server from controlling terminal */
     detach();

#endif
    sin.sin_port = sp->s_port;      2
    if((f = socket(AF_INET,SOCK_STREAM,0)) < 0) {      3
         perror("socket");
         exit(1);
    }
    if(bind(f,(struct socketaddr*)&sin,sizeof(sin)<0)){      4
   ;
    }
    listen(f,5);      5
    for(;;){
         int g, len = sizeof(from);
      g = accept(f,(struct socketaddr*)&from,&len);      6
      if(g<0) {      7
             if(errno != EINTR)
          syslog(LOG_ERR,"rlogin:accept: %m");
            continue;
      }
      if(fork() == 0){      8
             close(f);
          doit(g,&from);
      }
      close(g);
       }
}

detach() {
     int i;
     for(i=0;i<3;++i)
          close(i);

     open("/", O_RDONLY);
     dup2(0,1);
```

Example 4-2: (continued)

```
        dup2(0,2);

        i = open("/dev/tty", O_RDWR);
        if(i>=0){
                ioctl(i,TIOCNOTTY,0);
                close(i);
        }
}
```

1 The `getservbyname` call is given *"login"* as a service name and *"TCP"* as a protocol name. It uses these names to sequentially search the `/etc/services` database for a matching service and protocol name. If a match is found, the call returns a pointer to a `servent` structure that contains information about the service. If no match is found, the error message "rlogin:tcp/login:unknown service'' is returned. See Section 2.2.4 for a description of the `servent` structure.

2 The login socket number is placed into the socket structure.

3 Allocates an open socket to the login process.

4 Binds the allocated socket to the login port to have a point to listen for incoming connections. Because the remote login server listens at a restricted port number, it must run with a user-id of root.

5 Issuing the `listen` call indicates the login server's willingness to listen for an incoming connection. Setting the *backlog* parameter to five specifies five as the maximum number of pending connections that should be queued for acceptance. See Section 2.1.3 for a description of the `listen` call.

6 The `accept` call blocks and goes into an infinite loop, until a connection is made. Upon making a connection, the system places the address of the requesting client into a `sockaddr` structure. The system creates a new socket (*g*) that has its destination connected to the requesting client and returns the new socket descriptor to the caller.

7 The returned value (*g*) is checked to ensure a connection has been established. If the call returns a failure status or is interrupted by a signal such as SIGCHLD, an error report is logged. Returning a failure status disassociates the server from the requesting client. After a server disassociates itself, it can no longer send error reports to the requesting client and must use `syslog` to log errors.

8 After making a connection, the server produces (fork operation) a child process and invokes the main body `(doit)` of the remote login protocol processing. The socket used by the parent for the queuing connection requests is closed in the child process, while the socket created from the `accept` call is closed in the parent process. The address of the client is passed to the `doit` routine to authenticate the requesting client.

4.3 Connectionless Server Code Example

An example of a connectionless service (uses datagram sockets) is the `rwho` service. The `rwho` service provides users with status information for hosts connected to a local area network. This service uses information that is **broadcasted** to all hosts connected to a particular network.

A user on any machine that is running the `rwho` server can obtain the current status of a machine with the `ruptime()` program. Figure 4-1 shows a possible output.

Figure 4-1: ruptime() Program Output

Machine Name	Status	Up Time	Current Users	1 Min	5 Min	15 Min
arpa	up	9:45,	5 users,load	1.15	1.39	11.31
cad	up	2+12:04,	8 users,load	4.67	5.13	4.59
calder	up	10:10,	0 users,load	0.27	0.15	0.14
dali	up	2+06:28,	9 users,load	1.04	1.20	1.65
degas	up	25+09:48,	0 users,load	1.49	1.43	1.41
ear	up	5+00:05,	0 users,load	1.51	1.54	1.56
ernie	down	0:24				
esvax	down	17:04				
ingres	down	0:26				
kim	up	3+09:16,	8 users, load	2.03	2.46	3.11
matisse	up	3+06:18,	0 users,load	0.03	0.03	0.05
medea	up	3+09:39,	2 users,load	0.35	0.37	0.50
merlin	down	19+15:37				
micro	up	1+07:20,	7 users,load	4.59	3.28	2.12
monet	up	1+00:43,	2 users,load	0.22	0.09	0.07
oz	down	16:09				
starvax	up	2+15:57,	3 users, load	1.52	1.81	1.86
ucbvax	up	9:34,	2 users,load	6.08	5.16	3.28

Each `rwho` service periodically broadcasts status information for each host. The same server process also receives the status information and uses it to update a database (`/usr/spool/rwho/whod`). The server process interprets the database information to generate the status information for each host. Each server operates independently, except for being connected by the local network and broadcast messages.

Example 4-3 shows a simplified example of an `rwho` server.

The server performs two major tasks:

1. The server receives status information that is broadcast by other hosts on the network. Packets received at the `rwho` port are verified as if they have been seen by another `rwho` server process. They are time-stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, it is assumed that the host is down and this is indicated on the status reports. The status report can be incorrect, for a server can be down while a host is up.

2. The server provides information regarding the host's status. This involves periodically acquiring system status information, packaging it into a message, and broadcasting it to other `rwho` servers on the local network. The acquiring of the status information is triggered by a timer and runs off a signal.

For networks that do not support broadcasting, another scheme must be used instead of the broadcasting. One possibility is to determine the hosts on the local network from the status messages received from other `rwho` servers. If all the hosts on the network have been recently booted, however, no host would know the other hosts and thus would never receive or send any status information.

The routing table management process has the identical problem when propagating routing status information. A solution is to inform one or more servers of known hosts and request that the servers always communicate with these hosts. If each host knows of at least one other host, status information can propagate throughout the local network. If a host is connected to multiple networks, however, the host can receive status information from itself, and this can lead to endless looping.

In a distributed environment, it is important that each server use the same software. The ULTRIX operating system helps to isolate host-specific information from applications by providing system calls that return the necessary information. For example, the `ioctl` call provides a way to determine the networks that a host is directly connected to. In addition, the ULTRIX operating system supports network broadcasting at the socket level. Both of these features allow a process to broadcast on any directly connected local network that supports broadcasting in a site-independent manner. This solves the problem of propagating status information in the case of the `rwho` server, for status information is broadcast to connected networks at the socket level, where the connected network have been obtained by means of `ioctl` calls.

Example 4-3: rwho Server Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/signal.h>
#include <sys/file.h>
#include <sys/time.h>
#include <syslog.h>
#include <netinet/in.h>
#include <stdio.h>
#include <sys/errno.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <protocols/rwhod.h>

#define RWHODIR "/usr/spool/rwho"

main()
{
     int on, s;
     struct sockaddr_in sin;
     struct servent *sp;
     struct netent *net;
     char path[BUFSIZ];

     sp = getservbyname("who","udp");
     net = getnetbyname("localnet");
     sin.sin_addr = inet_makeaddr(INADDR_ANY,net);
     sin.sin_port = sp->s_port;

     if (s = socket(AF_INET,SOCK_DGRAM,0) < 0 ) {
        syslog(LOG_ERR,"socket: %m");
        exit(1);
     }

     on = 1;
     if(setsockopt(s,SOL_SOCKET,SO_BROADCAST,&on,sizeof(on))<0){
          syslog(LOG_ERR,"setsockopt SO_BROADCAST:%m");
          exit(1);
     }
     if (bind(s,(struct sockaddr*)&sin,sizeof(sin)) < 0) {
        syslog(LOG_ERR,"bind: %m");
        exit(1);
     }

     signal(SIGALRM,onalrm);
     onalrm();
     for(;;){
          struct whod wd;
          struct sockaddr_in from;
          int cc, whod, len = sizeof(from);

       cc = recvfrom(s,(char*)&wd,sizeof (struct whod),0,
          (struct sockaddr*)&from,&len);
          if(cc <= 0){
               if(cc < 0 && errno != EINTR)
                    syslog(LOG_ERR, "rwhod:recv:%m");
               continue;

        }
        if(from.sin_port != sp->s_port){
             syslog(LOG_ERR, "rwhod: %d: bad from port",
               ntohs(from.sin_port));
```

Example 4-3: (continued)

```
            continue;

        }

        if(!verify(wd.wd_hostname)){
            syslog(LOG_ERR, "whod: malformed host name from %x",
              ntohl(from.sin_addr.s_addr));
            continue;

        }
        (void)sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
        whod = open(path, O_WRONLY|O_CREAT|O_TRUNC,0666);

        (void)time(&wd.wd_recvtime);
        (void)write(whod, (char*)&wd,cc);
        (void)close(whod);
        }
}
```

4.4 Simplifying Servers

Servers can be simplified by using the `inetd` daemon. The `inetd` daemon does the majority of IPC operations required in establishing a connection. The server invoked by `inetd` expects the socket connected to its client on file descriptors 0 and 1 and can immediately perform any operations such as `read`, `write`, `send`, or `recv`. Servers can use buffered I/O as provided by the `stdio` conventions, as long as the `fflush` call is used appropriately.

After being invoked at boot time, the `inetd` daemon does the following:

- Determines from the file `/etc/inetd.conf` which servers are to listen
- Creates a socket for each service it is to listen for, binding the appropriate port number to each socket
- Performs a `select` operation to determine which of these sockets are available for reading
- Waits for a connection to the service corresponding to the socket
- Performs an `accept` on the socket in question
- Performs a `fork` and `dups` operations on the new socket for file descriptors 0 and 1 (stdin and stdout)
- Closes other open file descriptors
- Performs `execs` operation for the appropriate server

The `getpeername` call is useful when writing server programs under `inetd`. This call returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in **dot notation** (for example, 128.32.0.4) of a client connected to a server under `inetd`, the code in Example 4-4 can be used.

Example 4-4: Returning the Address of the Peer Process

```
#include <sys/socket.h>
#include <syslog.h>
#include <netinet/in.h>
#include <netdb.h>


struct sockaddr_in name;
int namelen = sizeof(name);

if(getpeername(0,(struct sockaddr*)& name, &namelen)<0){
     syslog(LOG_ERR,"getpeername:%m");
     exit(1);
}else
     syslog(LOG_INFO,"Connection from %s", inet_ntoa(name.sin_addr));
```

X/Open Transport Interface 5

5.1 Description

The X/Open transport service interface (XTI) consists of a set of transport-independent C library functions that conform to the X/Open Transport Interface specifications. XTI applications can be written to support both BSD sockets and System V streams. A network application that uses the XTI calls is portable across systems, as long as both systems incorporate the XTI calls and support the same underlying transport provider. The transport provider is defined as the entity that provides the services of the transport service interface. At present, the ULTRIX operating system supports Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) services using XTI.

The TCP service is circuit-oriented and enables data to be transmitted over an established connection in a reliable, sequenced manner. It also provides an identification mechanism that avoids the overhead of address resolution and transmission during the data transfer phase. This service is useful for applications that require relatively long-lived, data stream-oriented interactions.

In contrast, UDP service is message-oriented and supports data transfer in self-contained units, with no logical relationships required among multiple units. This service requires only a preexisting association between the peer users involved, which determines the characteristics of the data to be transmitted. All the information required to deliver a unit of data (for example, the destination address) is presented to the transport provider, together with the data to be transmitted, in one service access that need not relate to any other service access. Each unit of data transmitted is entirely self-contained. UDP service is useful for applications that:

- Involve short-term request/response interactions
- Exhibit a high level of redundancy
- Are dynamically reconfigurable
- Do not require guaranteed, in-sequence delivery of data

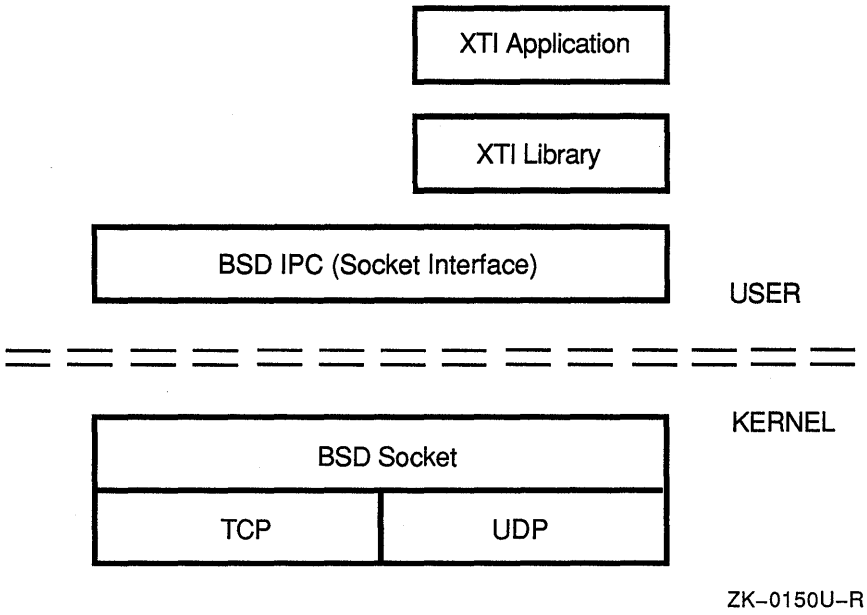
To access the services of the transport provider, the transport user issues the appropriate service requests. The transport user is defined as the entity that requires these service. A transport user may be a network application or session layer protocol.

5.2 XTI Software Components

XTI is similar to the existing Berkeley Software Distribution (BSD) socket-based interprocess communications (IPC). Like IPC, XTI provides a programming interface to access the underlying transport services and uses a file descriptor to identify the

endpoint for communication. The file descriptor is known as the transport endpoint. Figure 3-1 shows how XTI is layered into the networking architecture.

Figure 5-1: XTI Software Components



The XTI applications layer represents the set of application programs making XTI calls to the XTI library. The XTI library, a C library (`/lib/libxti.a`), provides all the function calls specified by the XTI specification. The library exists in the user space and can be bound with both Digital-supplied and user-written applications.

The XTI library is layered over the existing (kernel) BSD socket mechanism. As was described in Chapter 2, sockets provide the basis for interprocess communication on BSD systems. Access to the IPC services is provided by the set of BSD IPC primitives, which is known as the socket interface. The XTI library accesses kernel services only through the BSD IPC primitives. The lowest layer in Figure 3-1 represents the Transport Providers (TCP and UDP).

5.3 XTI Documentation

The *Guide to X/Open Transport Interface* describes how to write network application programs using XTI. The guide not only contains a description of XTI fundamentals, but it also contains descriptions of client and server examples using TCP and UDP.

Writing Distributed Applications with DECrpc 6

This chapter briefly describes the programming interface to DECrpc, the remote procedure call mechanism supported by the ULTRIX operating system. The chapter also lists the manuals that provide complete programming information.

DECrpc Version 1.0 is based on and is compatible with the RPC component of the Network Computing System (NCS) Version 1.5, which is a set of tools for heterogeneous distributed computing.

6.1 Distributed Applications

Using remote procedure calls, software applications can be distributed across heterogeneous collections of computers, networks, and programming environments. Distributed applications can take advantage of computing resources throughout a network or internet, with different parts of each program executing on the computers best suited for the tasks.

There are many applications that can be distributed among multiple systems. For example, one program might perform graphical input and output on a workstation while it does intense computation on a supercomputer. A program that performs many independent calculations on a large set of data could distribute these calculations among any number of available processors on the network or internet.

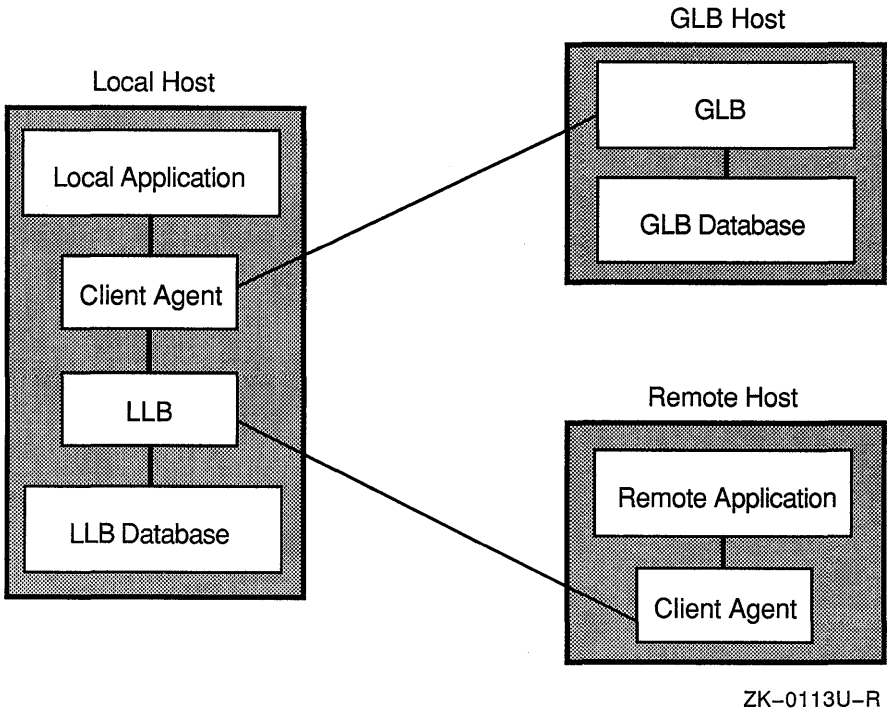
6.2 RPC Software

The software for writing distributed applications is written in portable C wherever possible. The components are:

- Remote Procedure Call (RPC) runtime library, which provides runtime support for distributed applications
- Network Interface Definition Language (NIDL) Compiler, a tool for developing distributed applications
- Location Brokers, which provide runtime support for distributed applications

Figure 6-1 illustrates the relationship between a distributed application and the software components of DECrpc. The rest of this section describes the components.

Figure 6-1: A Distributed Application



6.2.1 RPC Runtime Library

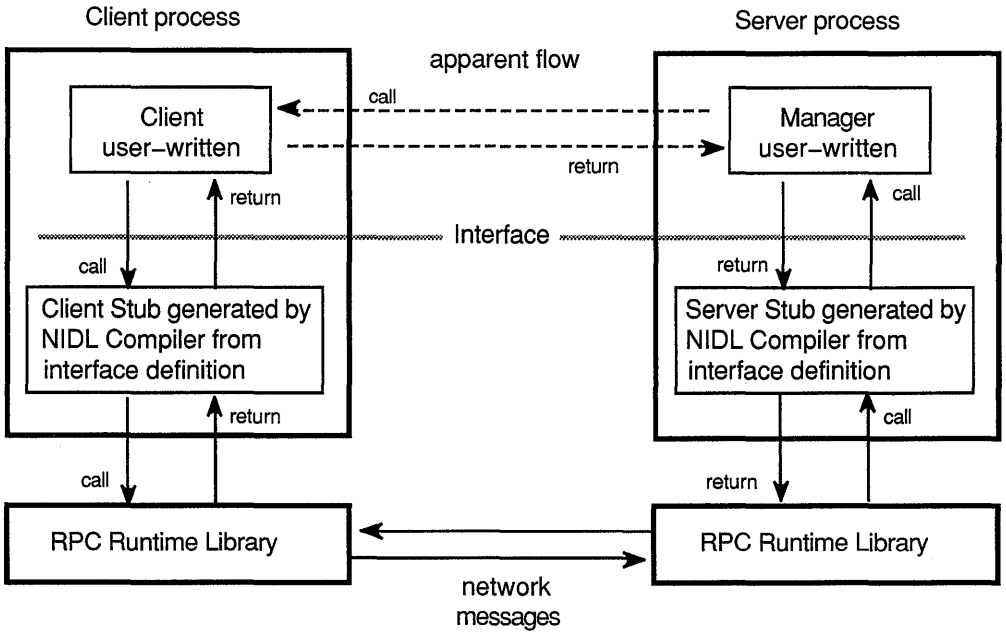
The DECrpc runtime library provides the routines that enable local programs to execute procedures on remote hosts. These routines transfer requests and responses between the programs calling the procedures and the programs executing the procedures.

When you develop distributed applications, you usually do not use many runtime routines directly. Instead, you write interface definitions in NIDL and use the NIDL Compiler to generate most of the required calls to the runtime library.

Figure 6-2 shows the flow of remote procedure calls and illustrates how the RPC paradigm hides the remote aspects of a routine from the calling client. The client application uses ordinary calling conventions to request a procedure, as if the procedure were a part of the local program, but the procedure is executed by a remote server. The client stub acts as the ''local representative'' of the procedure.

A stub is a program module generated by the NIDL Compiler from a user-written interface definition. The stub uses runtime library calls to communicate with the server. Similar activities occur within the server process.

Figure 6-2: Remote Procedure Call Flow



6.2.2 Network Interface Definition Language Compiler

The NIDL Compiler takes as input an interface definition written in NIDL. From this definition, the NIDL Compiler generates client and server stub programs. An interface definition specifies the interface between a user of a service and the provider of the service. The definition describes how a client application "sees" a remote service and how a remote server "sees" requests for its services.

The stubs produced by the NIDL Compiler contain nearly all of the "remoteness" in a distributed application. The client stub program performs data conversions, assembles and disassembles packets, and interacts with the RPC runtime library. The server stub program provides similar support for the server. It is easier to write an interface definition in NIDL than it would be to write the stub code that the NIDL Compiler generates from your definition.

6.2.3 Location Broker

A broker is a server that provides information about resources. The Location Broker enables clients to locate specific objects, such as a database or a specialized processor, or specific interfaces, such as a data retrieval interface or a matrix arithmetic interface. In addition to the Location Broker specific interfaces, DECrpc includes a name-service independent interface. The interface lets you write programs that will be portable across future versions of DECrpc and other naming services. The *DECrpc Programming Guide* describes the name-service independent interface.

Location Broker software includes the Local Location Broker (LLB) that manages information about resources on a local host, the Global Location Broker (GLB) that

manages information about resources available on all hosts, a client agent through which programs use the Location Broker facilities, and the `lb_admin` administrative tool.

The GLB stores in a database the locations of objects and interfaces in a network or internet. Clients can use the GLB to access an object or interface, without knowing its location beforehand. The LLB also implements a forwarding facility that provides access by way of a single address to all of the objects and interfaces at the host.

6.3 DECrpc Documentation

Table 6-1 lists the software components for developing distributed applications and the manual that provides information about the component.

In addition to the manuals listed in Table 6-1, the ULTRIX Reference Pages contain reference pages for each utility, special file, or library routine.

Table 6-1: DECrpc Documentation

Component	Manual	Description
RPC	*DECrpc Programming Guide*	Programming information and examples for developing distributed applications.
NIDL Compiler	*DECrpc Programming Guide*	Descriptions of interface definitions written in NIDL. Information on compiling definitions and the output of the compilations.
Location Broker	*Guide to the Location Broker*	Describes `lb_admin`, the Location Broker administrative tool and the procedures for setting up and maintaining the local and global Location Broker daemons, `llbd` and `nrglbd`.

ULTRIX Extended SNMP Agent 7

This chapter tells how to write an Extended SNMP Agent, an extension of the Simple Network Management Protocol (SNMP) Agent. The Extended SNMP Agent allows you to manage a private Management Information Base.

The Simple Network Management Protocol is used widely in the Internet community for network management. The protocol defines the role of a Network Management Station (NMS) and an SNMP Agent. The NMS exchanges messages with the SNMP Agent by means of the User Datagram Protocol (UDP) over the underlying Internet Protocols (IP).

The SNMP Agent allows remote network managers on an NMS to monitor and manage TCP/IP network entities specified in a Management Information Base (MIB). The MIBs are defined in the Internet Request for Comments (RFC) 1066. The Extended SNMP Agent implemented in the ULTRIX operating system allows you to define a private MIB of objects not defined by RFC 1066.

Introduction to Networking and Distributed System Services describes the SNMP Agent and the procedures for configuring the SNMP Agent daemon and an Extended SNMP Agent daemon in an ULTRIX system.

RFCs in the Internet Request for Comments series provide complete information about the SNMP and the MIB. You can obtain copies of the RFCs from the Network Information Center, SRI International, Menlo Park, CA 94025, or with the `ftp`(1) command. The RFCs that define the SNMP and the MIB are:

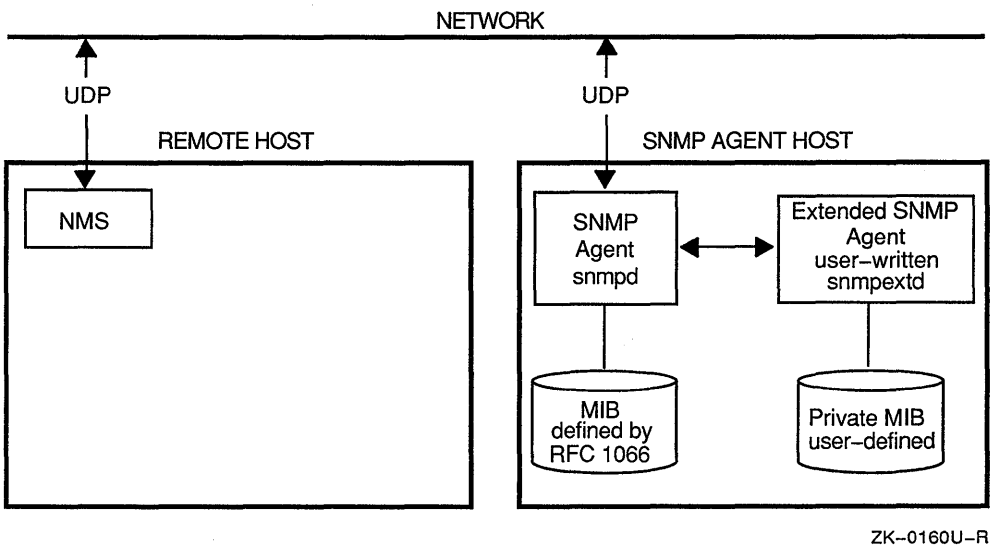
- RFC 1065—Structure and Identification of Management Information for TCP/IP-based Internets
- RFC 1066—Management Information Base for Network Management of TCP/IP-based Internets
- RFC 1067—A Simple Network Management Protocol

7.1 The Extended SNMP Agent

The Extended SNMP Agent allows users to add private MIBs to their environment and to provide information about the private MIB to the SNMP Agent. An Extended SNMP Agent is a logical extension of the SNMP Agent. To the Network Management Stations, both an agent and an extended agent appear as a single entity and cannot be distinguished by the network manager.

The Extended SNMP Agent must physically reside on the same host as the SNMP Agent, as illustrated in Figure 7-1.

Figure 7-1: SNMP Configuration in a Network



The Extended SNMP Agent conforms to the "Form and Meaning of References to Managed Objects," as defined in RFC 1067. An Extended SNMP Agent can manage a single private MIB or multiple private MIBs. More than one Extended SNMP Agent can be configured into a system and registered with the SNMP Agent.

The program you write is a daemon that sets up your private MIB and provides information from it. The daemon extracts MIB information from kernel memory, from static variables from the SNMP configuration file, `/etc/snmpd.conf`, or from wherever your application specifies. The daemon uses the library routines in `libsnmp.a` to respond to the SNMP Agent daemon, `snmpd`, on requests for information about your private managed objects.

For example, in the application described in Section 7.2, the managed objects in the private MIB provide virtual memory information and disk information. A network manager at an NMS requests information about the disk MIB from the SNMP Agent. The SNMP Agent knows there is an Extended SNMP Agent from an entry in the `/etc/snmpd.conf` file and sends a request to the Extended SNMP Agent. The Extended Agent gets the information from the MIB and returns it to the SNMP Agent, which returns it to the NMS.

7.2 Online Example Directory

The directory `/usr/examples/snmp/snmpext` contains code for an Extended SNMP Agent daemon, `snmpextd`. The daemon manages two private MIBs, one that supplies information about disks and another that supplies information about virtual memory.

Files in the directory are:

`defs.h`
: Header file that contains internal data structure definitions for the Extended SNMP Agent. Includes the required header files and defines values for the two supported MIBs.

`diskdef.h`
Header file that contains definitions for the disk database.

`vmdef.h`
Header file that contains definitions for the virtual memory database.

`main.c`
Main program module. Contains code for the daemon that handles the private MIBs. The module calls the library routines in `libsnmp.a` to provide to the `snmpd` daemon information about the managed virtual memory and disk objects. The main program module constructs the object identifier in a given structure and calls the appropriate register routine, either disk or virtual memory, to register the MIB with `snmpd`.

`snmp.c`
Module that processes requests for information from the SNMP Agent daemon, `snmpd`. The module contains a routine for determining the MIB for the request.

`disk_grp.c`
Module that looks up variables in the disk MIB. The code defines the disk MIB and returns a disk MIB variable to `snmpd`. Includes the routine that registers the disk MIB.

`vm_grp.c`
Module that looks up variables in the virtual memory MIB. The code defines the virtual memory MIB and returns a variable to the `snmpd` from the virtual memory MIB. Includes the routine that registers the virtual memory MIB.

`Makefile`
Makefile that builds and installs the example `snmpextd` daemon.

The next sections describe the header files and the library routines, using code from the program modules to illustrate the discussion.

7.3 Header Files

Header files that contain required definitions for the Extended SNMP Agent are listed here:

`/usr/include/protocols/snmp.h`
Contains data structures and type definitions defined by the SNMP. Defines internal constants.

`/usr/include/protocols/snmperrs.h`
Contains error return codes.

In the SNMP example directory, the file `defs.h` includes both these header files, as well as other files required for network calls.

7.4 Defining Objects in a Private MIB

Figure 7-2 illustrates the tree layout of the SNMP MIB as defined in RFC 1066 and shows the private MIB from the online example program under the private heading. You can define all the objects under the private and enterprise identifiers as needed for your application. The example MIB uses the ''Your Organization'' and ''Me'' identifiers to illustrate one way of organizing a private MIB. The ''Your Organization'' identifier is a number assigned to your organization by the Internet.

The MIBs in this chapter and in the online example directory use the number fifteen (15) as an example. The *Introduction to Networking and Distributed System Services* includes expanded MIB diagrams that list each object identifier in the Management MIB.

Note

In this implementation of RFC 1065, the object identifier must have the prefix 1.3.6.1 to be registered. If it does not, the SNMP Agent rejects the registration.

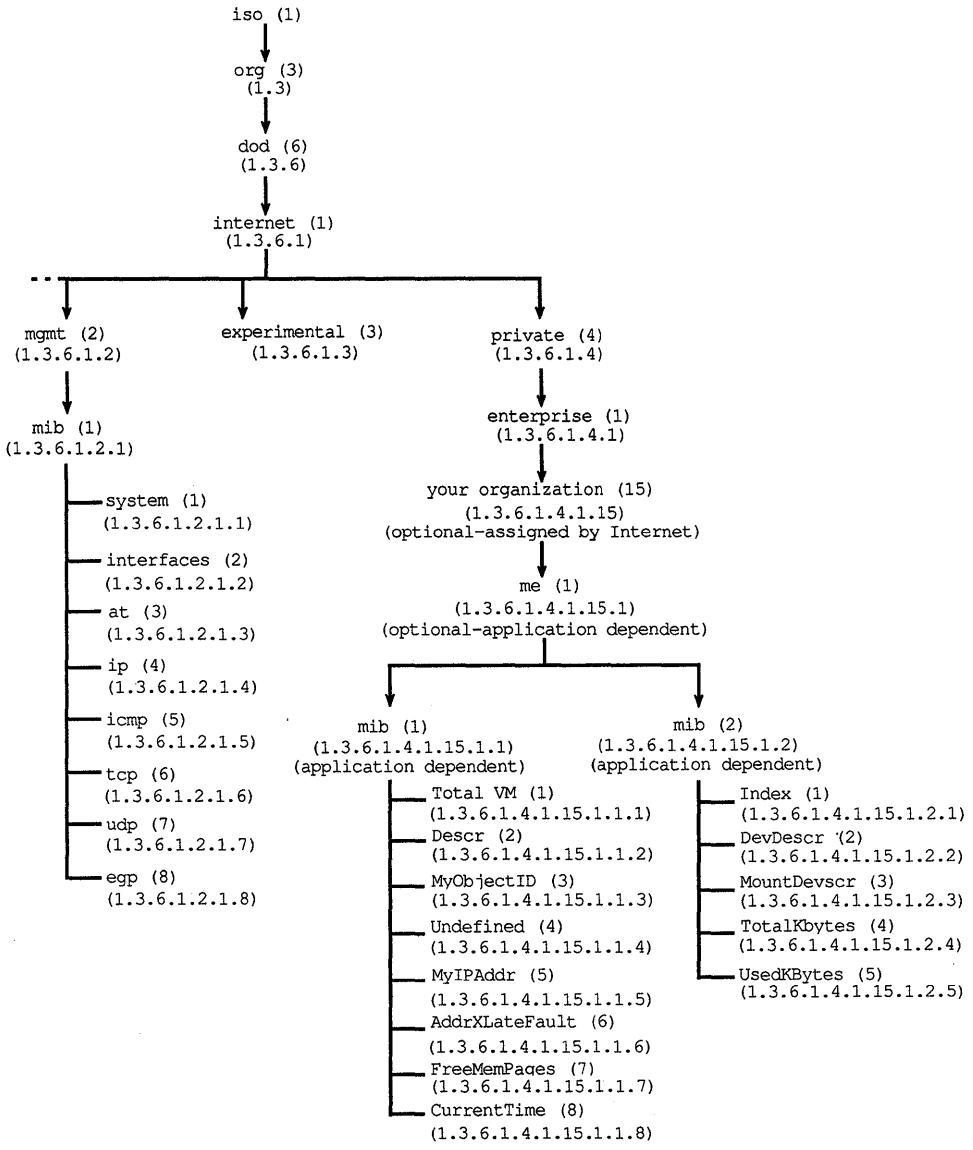
The code in Example 7-1, from `disk_grp.c` in the example directory, defines one of the private MIBs illustrated in Figure 7-2.

Example 7-1: Defining a Private MIB (disk_grp.c code)

```
/*
 * My private MIB (this information can also be obtained
 * with the df utility):
 *
 * iso(1) org(3) dod(6) internet(1) private(4) enterprise (1)
 *               your org(15) me(1)
 *
 *                      disk MIB(2) ..
 *      INT             diskIndex(1)        1.3.6.1.4.1.15.1.2.1
 *      STR             diskDevDescr(2)     1.3.6.1.4.1.15.1.2.2
 *      STR             diskMountDescr(3)   1.3.6.1.4.1.15.1.2.3
 *      GAUGE           diskTotalKbytes(4)  1.3.6.1.4.1.15.1.2.4
 *      GAUGE           diskUsedKBytes(5)   1.3.6.1.4.1.15.1.2.5
 */
unsigned long Disk_Var[] = {            /* df */
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x02 [1]
};
unsigned long Disk_Index[] = {
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x02, 0x01
};
unsigned long Disk_DevDescr[] = {
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x02, 0x02
};
unsigned long Disk_MountDescr[] = {
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x02, 0x03
};
unsigned long Disk_TotalKbytes[] = {
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x02, 0x04
};
unsigned long Disk_UsedKbytes[] = {
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x02, 0x05
};
```

[1] The value 0x02 in the MIB field identifies the disk MIB. The file `diskdef.h` defines constants for each attribute value (the last field in the object identifier).

Figure 7-2: SNMP MIB Layout



Example 7-2 shows code from `vm_grp.c` that defines the virtual memory MIB.

Example 7-2: Defining a Private MIB (vm_grp.c code)

```
unsigned long Vm_Var[] = {              /* vm */
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x01 [1]
};
unsigned long Vm_TotalVM[] = {
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x01, 0x01
};
unsigned long Vm_Descr[] = {
        0x01, 0x03, 0x06, 0x01, 0x04, 0x01, 0x0f, 0x01, 0x01, 0x02
};
.
.
.
```

[1] The value 0x01 in the MIB field identifies the virtual memory MIB.

7.5 Library Routines

The Extended SNMP Agent library, `libsnmp.a`, includes these routines that communicate with the SNMP Agent:

`snmpextregister`(3n)
: Registers the MIB of the extended agent with the SNMP Agent.

`snmpextgetreq`(3n)
: Receives a request for a MIB variable from the SNMP Agent.

`snmpextrespond`(3n)
: Returns the requested variable to the SNMP Agent.

`snmpexterror`(3n)
: Returns an error to the SNMP Agent.

The next sections describe the library routines. See also the reference page for each routine.

7.5.1 The snmpextregister(3n) Library Routine

The `snmpregister` routine registers the Extended SNMP Agent's MIB with the SNMP Agent. This example shows the syntax:

```
snmpextregister (reg, community)
struct snmpareg *reg;
char *community;
```

The caller provides the `reg` parameter, which contains the object identifiers to be registered.

The `snmpareg` structure is defined in `<protocols/snmp.h>`, as shown in this example:

```
struct  snmpareg {                      /* Registration Data Structure */
        short           oidtype;        /* database type of object id */
        objident        oid;            /* object id */
};
```

Section 7.5.3 describes the procedure for defining objects in a private MIB.

The routine in Example 7-3, from `/usr/example/snmp/snmpext/main.c`, constructs the object identifier structure and then calls `snmpextregister` to register the MIB.

Example 7-3: Registering a Private MIB with the SNMP Agent, snmpd

```
char     *community = "public";  /* community name */ [1]
.
.
.
register_disk() [2]
{
    register int    e = -1;

    if (reg_oid(Disk_Index, DISK_SIZE, OIDTYP_INST) == e) return(e);
    if (reg_oid(Disk_DevDescr, DISK_SIZE, OIDTYP_INST) == e) return(e);
    if (reg_oid(Disk_MountDescr, DISK_SIZE, OIDTYP_INST) == e) return(e);
    if (reg_oid(Disk_TotalKbytes, DISK_SIZE, OIDTYP_INST) == e) return(e);
    if (reg_oid(Disk_UsedKbytes, DISK_SIZE, OIDTYP_INST) == e) return(e);
    return(0);
}
.
.
.
reg_oid(oidbuf, numoid, oidtyp) [3]
u_long  *oidbuf;
short   numoid;
int     oidtyp;
{
        static struct snmpareg  reg;

        reg.oidtype = oidtyp;
        reg.oid.ncmp = numoid;
        bcopy(oidbuf, reg.oid.cmp, numoid*sizeof(u_long));
.
.
.
        if (snmpextregister(&reg, community) != GENSUC) { [4]
        return(0);
}
```

[1] The `community` parameter is defined in the `snmpd.conf` file and can be any NULL-terminated string of characters known to both the SNMP Agent and the NMS.

[2] This is a user-written routine that works with the user-defined MIB in the example programs. The `register_disk` and `reg_oid` routines fill the `reg` structure with the disk MIB information.

[3] The `reg_oid` routine constructs the object identifier for the example MIB.

[4] The first argument to `snmpextregister` is a pointer to the `reg` array, which contains information about the disk MIB. For `DISK_DEVDESCR`, defined as 2 in `diskdef.h`, for example:

```
reg.oidtype = OIDTYP_INST;              /* This object has instance */
reg.oid.ncmp = 10;                      /* The number of components
                                           in the object identifier */
reg.oid.cmp[] = 1,3,6,1,4,1,15,1,2,2; /* The object identifier    */
```

If successful, the call to `snmpextregister` returns `GENSUC` (generic success), defined in `<protocols/snmperrs.h>`.

7.5.2 The snmpextgetreq(3n) Library Routine

The Extended SNMP Agent uses the `snmpextgetreq` library routine to get a request for a MIB variable. The syntax of the call is shown in this example:

```
snmpextgetreq(reqoid, reqinst)
objident *reqoid;
objident *reqinst;
```

The arguments are described here:

reqoid — The requested object identifier, as defined in the private MIB. For example, 1,3,6,1,4,1,15,1,2,2 is the `reqoid` for DISK_DEVDESCR.

reqinst — The object instance associated with the requested variable. Each disk in the private disk MIB is an instance. The NMS requests information about a specific instance of an object.

The `snmpextd` blocks until a request arrives from the SNMP Agent, as shown in the code in Example 7-4, from `main.c` in the example directory.

Example 7-4: Waiting for a Request from the SNMP Agent, snmpd

```
while (1) {
          status = snmpextgetreq(&reqoid, &reqinst); 1
          if (status != GENSUC) {
                  syslog(LOG_ERR, "main: snmpextgetreq failed: %d", status);
                  continue;
          }
          status = procreq(&reqoid, &reqinst);
          if (status != 0) {
                  syslog(LOG_ERR, "main: procreq failed: %d", status);
                  if (notdaemon) printf("main: procreq failed: %d0,
                  continue;
         }
     }
}
.
.
.
procreq(reqoid, reqinst) 2
objident        *reqoid;
objident        *reqinst;
{
        int  error = 0;

        switch (whichmib(reqoid->cmp)) {
        case MIB_VM:
                return( ret_vm(reqoid) );
        case MIB_DISK:
                return( ret_disk(reqoid, reqinst) );
        default:
                syslog(LOG_ERR, "procreq: unknown MIB");
          return(1);
        }
}
```

1 The routine snmpextgetreq contains the object identifier for the requested variable. For example, if the NMS requests the DISK_DEVDESCR for disk number 1, reqoid contains these values:

```
reqinst.cmp = 1;                        /* The instance                */
reqoid.ncmp = 10;                       /* The number of components
                                           in the object identifier */
reqoid.cmp[1] = 1,3,6,1,4,1,15,1,2,2; /* The object identifier     */
```

The NMS numbers instances starting with one, not zero.

If successful, the snmpextgetreq routine returns the generic success variable GENSUC, which is defined in <protocols/snmperrs.h>.

2 The procreq routine, a user-written routine, processes a request for information from these example MIB–disk or virtual memory. The routine determines which MIB the request is for and calls the routine that returns information for that MIB.

7.5.3 The snmpextrespond(3n) Library Routine

The snmpd uses the snmpextrespond library routine to return the requested variable to the Extended SNMP Agent. The syntax for the call is shown in this example:

```
snmpextrespond(reqoid, rspinst, rspdat)
objident *reqoid;
objident *rspinst;
struct snmparspdat *rspdat;
```

The arguments for the call are listed here:

reqoid
: The requested object identifier. This is the same object identifier used in the snmpextgetreq call.

rspinst
: The object instance associated with the returning variable. If there is no object instance associated with the requested variable, you must supply a null value. (See Example 7-5.)

rspdat
: The returned variable.

The response data type can be one of the SNMP data types listed in Table 7-1. The file /usr/include/protocols/snmp.h. includes the definitions for the data types.

Table 7-1: Response Data Types

Data Type	Description
CNTR	Counter
GAUGE	Gauge
INT	Integer
IPADD	Internet address
OBJ	Object identifier
STR	Octet string
TIME	Time

The SNMP Agent maintains a configurable timer for outstanding requests to the Extended Agent. Therefore, the Extended Agent must be able to respond within the SNMP Agent's timeout interval to prevent a premature timeout in the SNMP Agent.

See `/etc/snmpd.conf` for the default timeout value for the system.

The code in Example 7-5, from `disk_grp.c` in the example directory, illustrates a call to `snmpextrespond`. If the NMS had requested the DISK_DEVDESCR, for example, the case statement shown in the example would execute.

Example 7-5: Returning a MIB Variable to the SNMP Agent, snmpd

```
ret_disk(reqoid, reqinst) [1]
.
.
.
case DISK_DEVDESCR:
      if (get_diskstat(attrtag, reqinst, &reqmet, &rspinst) != 0) { [2]
              error = BADVAL;
              break;
      }
      rsp.type = STR; [3]
      rsp.octets = strlen(reqmetstr); [4]
      rsp.rspdat = (char *)malloc(strlen(reqmetstr)); [5]
      bcopy(reqmetstr, rsp.rspdat, strlen(reqmetstr)+1);    /* ' ' */
      if (snmpextrespond(reqoid, &rspinst, &rsp) != GENSUC) { [6]
              syslog(LOG_ERR,"disk_grp: respond failed");
              error = GENERRS;
              break;
      }
.
.
.
```

[1] The `ret_disk` routine returns the requested disk MIB variable to the SNMP Agent.

[2] If the request is for the `DISK_DEVDESCR` attribute, this code executes. The `get_diskstat` routine returns the requested MIB variable to the `ret_disk` routine, which then fills the `rsp` array.

[3] The type of the `DISK_DEVDESCR` attribute is declared as `STR`, one of the allowable types.

4 The `octets` member contains the number of octets in `rsp.rspdat`.

5 The `rspdat` member of `rsp` contains the returned variable.

6 The `snmpextrespond` routine then returns the information to the SNMP Agent, `snmpd`. The `reqoid` parameter is the object identifier from the `snmpextgetreq` library call. The `rspinst` parameter specifies the instance (which disk) of the object.

Example 7-6 shows a call to `snmpextrespond` in the `vm_grp.c` example program, in which the requested object has no instance.

Example 7-6: Returning a MIB Variable with NULL Instance

```
case VM_TOTALVM:
      if (snmpextrespond(reqoid, NULL, &rsp) != GENSUC) { 1
            syslog(LOG_ERR,"vm_grp: respond failed");
            break;
      }
      return(0);
```

1 The second parameter, `rspinst`, is specified as NULL. Because there is only one virtual memory, there is no instance for the vm MIB.

7.5.4 The snmperror(3n) Library Routine

The library routine `snmperror` returns errors to the SNMP Agent. This example shows the syntax of the call:

```
snmpexterror(error)
long error;
```

The `snmpexterror` routine returns one of these errors:

NOERR
: Indicates a successful SNMP *get-next-request* end-of-table. This happens when the requested instance does not exist.

NOSUCH
: Indicates that the requested object identifier was unknown.

GENERRS
: Indicates a generic error.

BADVAL
: Indicates a bad variable value.

The code in Example 7-7, from `disk_grp.c` in the example directory, illustrates the call.

Example 7-7: Returning an Error to the SNMP Agent, snmpd

```
    default:
        /* Log the error and respond to snmpd with an error.  */

        syslog(LOG_ERR, "disk_grp: invalid MIB request attribute");
               (void) snmpexterror(NOSUCH); 1
               return(-1);
    } /* switch */
}
```

Example 7-7: (continued)

1 If the request was for an unknown MIB variable, the call to `snmpexterror` returns `NOSUCH` to the SNMP Agent.

7.6 Compiling and Installing an Extended SNMP Agent

This section describes the procedure for compiling and installing an Extended SNMP Agent. Refer to the *Introduction to Networking and Distributed System Services* for instructions on configuring the Extended Agent daemon in your system.

The `Makefile` in the online example directory provides an example of the compilation and installation.

Extended SNMP Agents must be compiled with the `libsnmp.a` library.

After compiling the extended agent, use the `install`(1) command to install the Extended SNMP Agent. In this example, the Extended Agent is installed in `/etc` so you must be superuser to perform the installation shown for an Extended Agent named `snmpextd`:

```
# install -c -s snmpextd /etc
```

Add an entry for the Extended SNMP Agent to the `/etc/snmpd.conf` file, as described in *Introduction to Networking and Distributed System Services.*

Packet Filter Programming 8

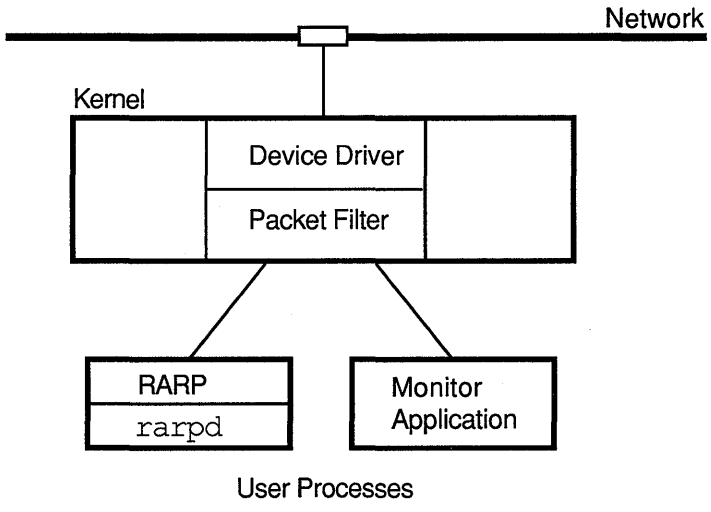
This chapter briefly describes the packet filter pseudodevice driver, a kernel-resident network packet demultiplexer supported by the ULTRIX operating system. The chapter also lists the documentation that provides programming information.

The packet filter provides a raw interface to Ethernets and similar network data link layers. Packets received that are not used by the kernel (for example, to support the IP and DECnet protocol families) are available through this mechanism.

8.1 Packet Filter Software

The packet filter driver is kernel-resident code provided by the ULTRIX operating system. The code appears to applications as character special files. The relationship between packet filter application programs and the packet filter driver is illustrated by Figure 8-1.

Figure 8-1: Packet Filter Software Structure



ZK-0168U-R

Each packet filter application process uses one or more of the character special files to gain access to the network. To open one of these special files, an application should use the `pfopen` library routine. Packet filter special files are created with the `MAKEDEV` script as described in Section 8.2.

Associated with each open instance of a packet filter character special file is a user-settable packet filter "program" that is used to select which incoming packets are delivered by means of the special file. The filter program returns "true" or "false," depending on whether the application wants to see the specific packet. Whenever a packet (not otherwise consumed in the kernel) is received from the network, the

packet filter driver successively applies the filter programs of each open packet filter special file to the packet, until one filter program "accepts" the packet. When a filter accepts the packet, the packet is placed on the packet input queue of the associated special file. (Optionally, a packet might be delivered to more than one application.) If no filters accept the packet, it is discarded. The format of a packet filter is described in detail in `packetfilter`(4).

Reads from the character special files return the next packet from a queue of packets that have matched the filter. Writes to the special files transmit packets on the network, with each write generating exactly one packet.

The packet filter driver provides direct access to the packet format of the underlying network; the packet filter mechanism does not know anything about the data portion of the packets it sends and receives. The application must supply the data-link level headers for transmitted packets (although the system makes sure that the source address is correct), and the headers of received packets are delivered to the application. The packet filter programs treat the entire packet, including headers, as uninterpreted data.

8.2 Configuring Packet Filters

Before you can program a packet filter application, you need to configure the packet filter option into the kernel as described here:

- Make sure the system configuration file includes entries for the packet filter. (The configuration file is `/sys/conf/{`*mips*`,`*vax*`}/`*HOSTNAME*`.`)

  ```
  optionsPACKETFILTER
  .
  .
  .
  pseudo-device packetfilter
  ```

  If you selected the packet filter as a configuration file option during an advanced installation, the entries should be in the configuration file.

  If you need to configure the packet filter into your system, add the entries to the system configuration file and then rebuild the kernel with the command `/etc/doconfig -c`.

  Refer to `doconfig`(8) and *System Configuration File Maintenance* for information on rebuilding the kernel.

- If `/dev` does not include the `pf` directory, run `MAKEDEV` with the `pfilt` option to create packet filter character special files, as illustrated in this example:

  ```
  # cd /dev
  # MAKEDEV pfilt
  ```

  The command creates the directory `/dev/pf`, which contains the packet filter character special files.

  A single call to `MAKEDEV` with an argument of `pfilt` creates 64 character special files in `/dev/pf`, which are named `pfilt`*nnn*, where *nnn* is the unit number. Successive calls to `MAKEDEV` with arguments of `pfilt1`, `pfilt2`, and `pfilt3` make additional sets of 64 sequentially numbered packet filters to a maximum of 256, which is the maximum number of minor device numbers allowed for each major device number.

  Refer to `MAKEDEV`(8) for more information on making special files.

The next section lists the documentation that describes packet filter programming.

8.3 Packet Filter Documentation

Table 8-1 lists the documents in the ULTRIX documentation set that provide information about the packet filter.

Table 8-1: Packet Filter Documentation

Document	Description
The Packet Filter: An Efficient Mechanism for User-level Network Code	Describes the implementation, uses, and performance of the packet filter. Provides background information about the development of the packet filter.
`packetfilter`(4)	Tells how to create packet filter special files. Describes application code needed to set up a packet filter. Describes `ioctl`(2) requests used to control a packet filter.

Index

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

W

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to Network Programming
AA-PBKWA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

_____

What do you like best about this manual? _____

_____

_____

What do you like least about this manual? _____

_____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line

Reader's Comments

ULTRIX
Guide to Network Programming
AA-PBKWA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

What do you like best about this manual? _____

What do you like least about this manual? _____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line