July-August 1984
Vol. 63 No. 6 Part 2

AT&T BELL LABORATORIES TECHNICAL JOURNAL

A JOURNAL OF THE AT&T COMPANIES

Computing Science and Systems

AT&T BELL LABORATORIES TECHNICAL JOURNAL (ISSN0005-8580) is published ten times each year by AT&T, 550 Madison Avenue, New York, NY 10022; C. L. Brown, Chairman of the Board; T. O. Davis, Secretary. The Computing Science and Systems section and the special issues are included as they become available. Subscriptions: United States—1 year $35; 2 years $63; 3 years $84; foreign—1 year $45; 2 years $73; 3 years $94. A subscription to the Computing Science and Systems section only is $10 ($12 foreign). Single copies of most issues of the Journal are available at $5 ($6 foreign). Payment for foreign subscriptions or single copies must be made in United States funds, or by check drawn on a United States bank and made payable to the Technical Journal and sent to AT&T Bell Laboratories, Circulation Dept., Room 1E335, 101 J. F. Kennedy Pky, Short Hills, NJ 07078.

Comments on the technical content of any article or brief are welcome. These and other editorial inquiries should be addressed to the Editor, AT&T Bell Laboratories Technical Journal, Room 1H321, 101 J. F. Kennedy Pky, Short Hills, NJ 07078. Comments and inquiries, whether or not published, shall not be regarded as confidential or otherwise restricted in use and will become the property of AT&T. Comments selected for publication may be edited for brevity, subject to author approval.

Printed in U.S.A. Second-class postage paid at Short Hills, NJ 07078 and additional mailing offices. Postmaster: Send address changes to the AT&T Bell Laboratories Technical Journal, Room 1E335, 101 J. F. Kennedy Pky, Short Hills, NJ 07078.

AT&T Bell Laboratories

Technical Journal

VOL. 63 JULY-AUGUST 1984 NO. 6, PART 2

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Computer Response Time and User Performance During Data Entry

By T. W. BUTLER*

(Manuscript received October 1, 1982)

In this experiment, subjects entered data at a computer terminal while the response time of the computer was varied systematically. Long average response times were found to be associated with significantly longer subject "think times", as was an increase in the variability of the computer's response time. Six subjects entered five-character letter groups under ten different computer response time conditions. The computer response time distributions used had mean values of 2, 4, 8, 16, and 23 seconds, with two different levels of variability at each mean value. User error rate and user typing time were not significantly affected by computer response time, but computer response time was significantly related to user response time (or think time) with this task. User response time increased slowly and gradually with increases in computer response over the range of stimulus values used. Increases in computer response time variability also increased user response time.

I. INTRODUCTION

Computer system designers face a serious problem when setting response time requirements for systems that they build. They must confront the competing pressures of providing the most responsive user service possible and of providing this service at the lowest possible cost. Often, cost pressures prevail and the result is a slow, heavily loaded system. Few data are available regarding the effect computer response time has on the users of a system.

Everyone who has used an interactive computer terminal knows that the response time of a system is important for its users. Indeed,

* AT&T Bell Laboratories.

users may find response time to be a computer system's most salient general characteristic. Nevertheless, there is no firm empirical basis for setting standards of acceptable response times for computer systems. The problem is one both of user performance and of user attitudes and perceptions. The relations of these variables to computer system response time have not been established.

Many papers have noted the importance of Computer System Response Time (CSRT) for users,[1-4] but very few have presented any solid experimental data relating to the problem. Several issues and hypotheses have been put forward in this body of literature:

1. User performance may decline at very long (or very short) computer response times.
2. Variability of computer response time may be as important for users as average response time duration.
3. Different user commands have different response time requirements, depending on the cognitive load that a command imposes on users.

Most early studies of this problem failed to produce meaningful, useful results for one of several reasons. Many studies simply surveyed computer use without systematically manipulating computer response time.[5,6] Subjects in these studies were in a continuously changing environment where most of the variables of interest (e.g., mean response time, response time variability, tasks being performed) were entirely out of the experimenters' control. Other studies have used unrealistic stimulus conditions that confound their results and make their application to this problem unclear. For example, Grignetti and Miller[7] and Johnsson et al.[8] used constant computer response times within each experimental condition, rather than a distribution of response times around some specified mean value. All this research can mislead readers who are unaware of the studies' limitations.

The present experiment examined user performance in a simple data entry task while the mean duration and the variability of computer response time was being manipulated. By carefully controlling these variables over a realistic range of stimulus values, some of the confusion caused by earlier experiments has been eliminated.

II. METHODS AND APPARATUS

2.1 Task

Subjects were given a typed list of 1000 five-character letter groups to be used as input stimuli in the experiment.[9] All letter groups were printed in uppercase type. Half of the letter groups in the set were first-order approximations of English words, and the other half were zero-order approximations of English, based on the bigram frequency of the letters they contained. The zero-order and first-order letter groups were randomly intermixed. Zero-order letter groups appear to be random sets of characters, and first-order letter groups appear to be very English-like,[10] though none of the groups used here was a real word. These stimuli were chosen to approximate the mixture of English words and code values entered by data entry clerks in large operations-support systems. The same set of input stimuli was used in every data collection session.

Subjects typed these letter groups one group at a time at a video display terminal (Hewlett-Packard 2621). They were instructed to work steadily at this task, and to complete as much of the work as possible within the allotted time. No specific incentives were offered to encourage them to work quickly. A single transaction proceeded as follows: The subject, on seeing the prompt character displayed on the terminal screen, typed the next letter group on the list, and followed it with a carriage return. This carriage return marked the beginning of the computer response time interval imposed by the experiment. When the appropriate amount of time had elapsed, the prompt character was again displayed at the beginning of the next line, signaling that the computer was ready to accept the next input. Until the prompt symbol was displayed, the keyboard was locked and it was impossible for subjects to type anything into the system.

Subjects could correct typing mistakes that they noticed before striking the carriage return with a "character erase" character "#", which erased the character immediately preceding it. It was not possible for subjects to cancel an entry after they struck the carriage return key, even if they noticed that the entry was incorrect. They were told to proceed to the next letter group if this occurred.

Subjects worked at each of the ten experimental conditions for 2 hours, or until they completed the list of 1000 letter groups, for a maximum total of 20 hours of data collection per subject. (Many subjects completed the list of letter groups in less than 2 hours in the shortest computer response time conditions.) All data collection sessions for individual subjects were separated by at least a full day, and only one subject took part in the experiment at a time. Each subject worked through the ten experimental conditions in a different, random order.

2.2 Computer response times

Within each session, elements from a positively skewed distribution of response times were randomly presented to the subject. Distributions of real-world computer response time are usually strongly skewed in the positive direction. The shape of the distribution used here was approximately that of Chi-squared with 4 degrees of freedom, but its mean and standard deviation varied from condition to condition in

Table I—Computer response time distributions

Condition	Range of $\bar{x}$ (in Seconds)	Range of Standard Deviation (in Seconds)	Range of Skew	Lowest Value (in Seconds)	Highest Value (in Seconds)
2LO	2.002–2.021	0.264–0.269	0.61–0.65	1.589–1.610	2.720–2.734
2HI	2.000–2.027	0.793–0.803	0.59–0.68	0.786–0.803	4.149–4.164
4LO	4.006–4.020	0.595–0.599	0.62–0.64	3.087–3.106	5.611–5.616
4HI	3.999–4.039	1.788–1.800	0.62–0.65	1.283–1.292	8.805–8.812
8LO	8.009–8.032	1.329–1.347	0.61–0.63	5.966–5.992	11.568–11.583
8HI	8.020–8.105	4.016–4.072	0.60–0.62	1.912–1.926	18.700–18.716
16LO	15.992–16.035	2.972–3.079	0.58–0.63	11.466–11.481	23.979–23.994
16HI	16.073–16.236	9.046–9.128	0.58–0.63	2.415–2.419	39.928–39.929
32LO	32.059–32.221	6.720–6.881	0.60–0.64	21.874–21.883	49.806–49.829
32HI	31.792–32.409	20.180–20.704	0.59–0.68	1.619–1.630	85.416–85.437

Note: In the column labeled "Condition", the integer indicates the nominal mean response time and the "LO" and "HI" suffixes indicate the "low" and "high" variability conditions, respectively.

the experiment. Five different mean computer response times were used: 2, 4, 8, 16, and 32 seconds. Two different variability conditions were imposed on each mean value, for a total of 10 different experimental conditions. Table I summarizes the stimuli used in the experiment. Stimulus values varied slightly from subject to subject because of differences in the number of transactions completed in each session across subjects. However, the ranges of values shown in Table I encompass the data for all subjects. The range of mean response times used here was set after meeting with many designers of large operations systems. Though the response time design goals for such systems are normally set at about 4 seconds, the range used here is an accurate reflection of the values encountered by developers during system performance tests.

Response time variability was calibrated in psychological units across the different conditions. In the "low" variability condition, one standard deviation of the computer response time distribution was set to equal 1.0 jnd (just noticeable difference) of computer response time at each of the mean values used. In the "high" variability condition, the standard deviation of each computer response time distribution was set to equal 3.0 jnd at each mean value. This equated the perceived variability within each low and high condition across the different mean values. The ranges of computer response time within each condition resulting from this scaling are shown in Table I. Just noticeable differences of computer response time were calculated using the measurements of Butler et al.[11]

2.3 Computer system

A *PDP-8**/E laboratory computer controlled the subject's terminal.

* Trademark of Digital Equipment Corporation.

It timed the prescribed computer response time intervals, recorded these values, and also recorded the subject's response time after the prompt character was displayed, the subject's typing time, and all characters typed during each transaction. These data were sent to a host computer and filed there as they accumulated.

2.4 Subjects

The subjects in this experiment were six experienced word processing clerks employed by AT&T Bell Laboratories in Piscataway, New Jersey. All were proficient in computerized text preparation, and all had previously used a video display terminal of the same model as the one used in this study (Hewlett-Packard 2621).

III. RESULTS

Three different measures of user performance were recorded during each transaction: user response time, user typing time, and user input errors. For the purposes of this study, user response time is defined as the time between the display of the prompt character on the subject's terminal and the typing of the first character entered by the subject. User typing time is the interval between the first character typed in each transaction and the typing of the carriage return that marks the end of the transaction. User errors were compiled here by calculating the percentage of incorrect entries made by each subject in a given session. Entries were judged incorrect if they were misspelled, not entered in the prescribed sequence, or omitted. Results for each of the three measures are presented below.

Neither mean input error rate nor mean typing time per transaction was significantly affected by the computer response time variables studied here. Error rates were quite variable within each condition, but mean error rate remained approximately constant at about 2 to 4 percent for mean computer response times ranging from 2 to 32 seconds in both the high- and the low-variability conditions. Typing time was also approximately constant at about 1.25 s/transaction across all experimental conditions.

The plots of mean user response time as a function of mean computer response time shown in Fig. 1 are somewhat more interesting. While the variability in these data is quite large, a regular relationship between these two variables appears to be present. User response time, which is about 1.0 to 1.25 seconds at a mean computer response time of 2 seconds, increases steadily to about 3.5 to 4.0 seconds at a mean computer response time of 32 seconds. Also, the high-variability condition regularly incurs user response times about 0.75 second longer than those resulting during the low-variability condition.

Logarithmic curves are one way to conveniently represent the data in the two variability conditions:

$$\text{low variability: } Y = 2.179 \log X + 0.075 \qquad r^2 = 0.913$$
$$\text{high variability: } Y = 2.518 \log X + 0.465 \qquad r^2 = 0.990,$$

where Y equals mean user response time and X equals mean computer response time.

These equations were calculated by transforming the computer response time variable to a logarithmic scale, and then performing a linear least-squares regression on the transformed data. The resulting plot, including the regression lines, is shown in Fig. 2.

A repeated-measures analysis of variance showed that the effect of computer response time variability was significant ($p < 0.05$) and that the effect of mean computer response time was marginally significant ($p = 0.053$). Both of these variables accounted for a fairly small percentage of the total variance in the experimental data, however. The data were noisy, and between-subjects variability accounted for about 84 percent of the total variance.

IV. DISCUSSION

In general, the data on errors and typing time gathered in the present study concur with the results of earlier studies using similar tasks.[12] Because these measures are not affected by different computer response time conditions, they are of little interest here.

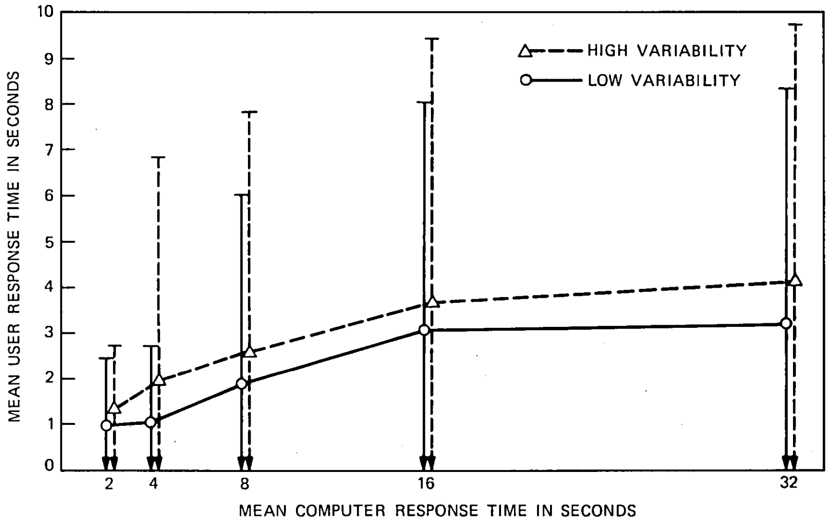


Fig. 1—The average user response time plotted as a function of mean computer response time. The error bars indicate ±1 standard error.

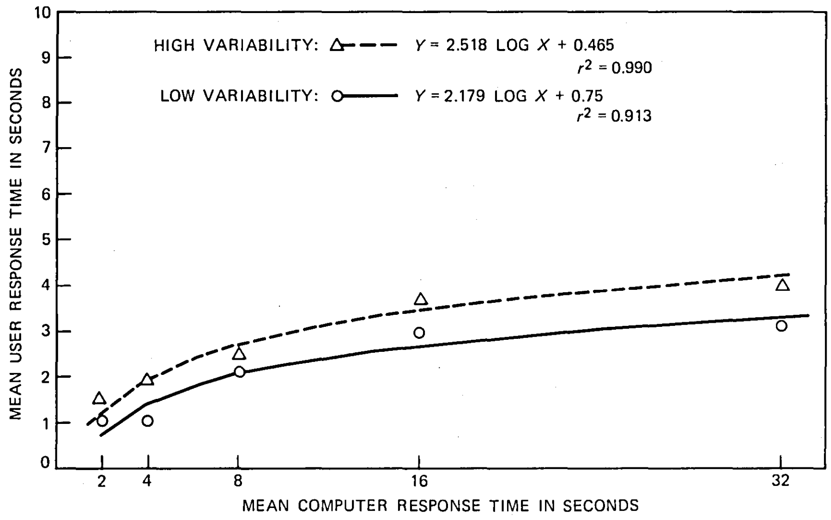


Fig. 2—The average user response time plotted as a function of mean computer response time. The lines drawn through the data points are linear-log least-squares regression lines.

The effect of different mean computer response times on user response times is another, more interesting matter. Several studies have made specific conclusions about the relationship between these two variables. Some of these conclusions are affirmed by the present study, and some are not.

One striking finding of several papers has been that user performance is optimal at computer response times of intermediate duration;[13–15] usually the values have been found to be in the 4- to 5-second range. There is no evidence of any decrement in performance at short computer response times in the data from the present experiment. Rather, performance (as measured by the user's response time) gradually worsens over the range of computer response times used. (This measure of user performance worsens over the range of average computer response times from 4 to 32 seconds. Changes in user performance over the 2- to 4-second range of stimuli are less obvious.) The important difference between the present study and these earlier experiments is that all the older papers reporting this finding were investigating specific problem-solving tasks, while the present study used a simple data-entry task. It could be that the subjects in these problem-solving studies required some amount of preparation or "think time" between transactions; they were planning their next command. With the data-entry task used in the present study, subjects needed no think time. They knew what their next command had to be and merely waited for the opportunity to enter it. The difference found here provides confirmation for Miller's speculation (see Ref. 16) that different commands have different response time requirements for users, depending on the cognitive load that they impose. It also points out the importance of considering the task type of a given command before extrapolating experimental data from other types of tasks to it.

Another widely cited set of data relating computer response time to user response time originated in a study by Boies and Gould,[5] and has also been reported by Boies[17] and Doherty and Kelisky.[18] This was a survey-type study in which all usage was monitored in a general-purpose research computer system. Doherty and Kelisky found that the expected user response time was equal to 15 seconds plus whatever the computer response time was for that transaction.[18] The results of the present study are clearly different from this, as shown in Figs. 1 and 2. This difference is not surprising, since Doherty and Kelisky described responses during a heterogeneous mixture of user command types, and because they correlated user think times with the single preceding computer response time, rather than with the average response time of the system.

An important finding of the present study is that users appear to respond to the computer's average response time, rather than to the last single response time experienced. One extreme hypothesis is that a user's response time is determined by the computer response time that immediately precedes it, and its opposite extreme is that the user's expectation of what the computer's response time should be, based on his or her total experience with the system, determines what his or her response will be. The first of these hypotheses is implicit when one uses survey data like that cited by Doherty and Kelisky.[18] One must assume that the mechanism mediating the user's response has no memory, and that each transaction is independent of all others that have preceded it.

A straightforward test of the two alternatives can be made using data from the present study. To make this test, all individual transactions in the high-variability condition with computer response times of 3.5 to 4.5 seconds were isolated, and the average user response time of each individual subject for these transactions was compared across mean computer response time conditions. The results of the test were qualitatively similar for all subjects. Data from two of them—the subject with the largest performance variation across all conditions, and the subject with the smallest overall performance variation—are shown in Figs. 3 and 4, respectively. These figures are plots of mean user response time as a function of mean computer response time. It is clear from these data that a user's average response time to these isolated transactions of 3.5- to 4.5-seconds duration is not constant across conditions, and that the important determinant of the user's

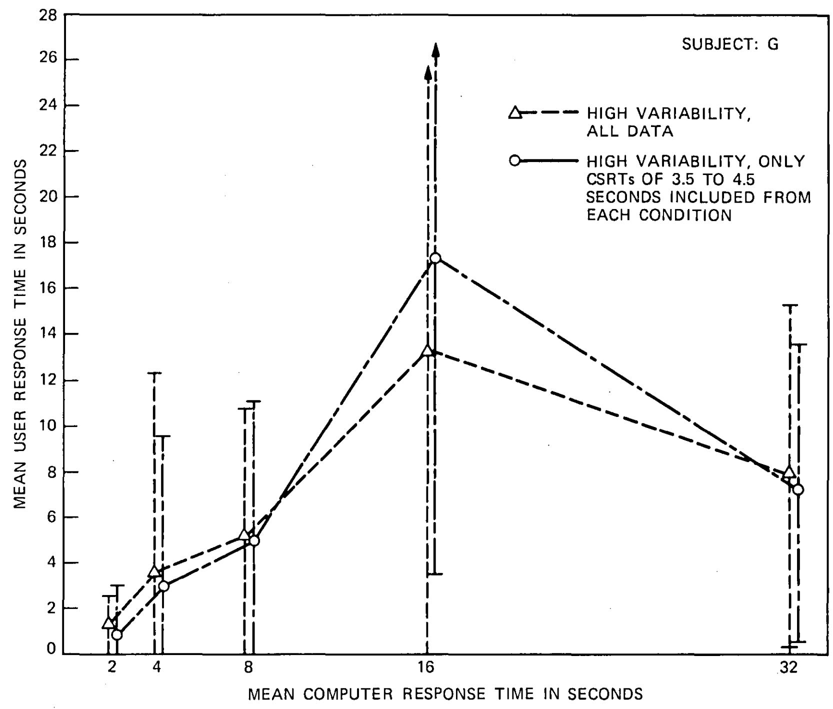


Fig. 3—Average user response time plotted as a function of average computer response time for the subject G.

response time here is the *mean* computer response time within each session.

4.1 Computer response time variability

Uniformly increasing the variability (in psychological units) of the computer response time distributions that the subjects experienced succeeded in uniformly increasing mean user response times by about 0.75 second across all conditions. Unfortunately, it is not possible to quantify this relationship with only two levels of variability in the study. This point is probably worth further examination in the future.

4.2 Conclusions

Systematically manipulating the response time characteristics of a computer system while users performed a simple data-entry task has shown that increases in the response time of the system cause a slow, gradual, and not very large degradation in user performance, as indicated by the user's response time. Also, increasing the variability of computer response time significantly degraded user performance, ac-

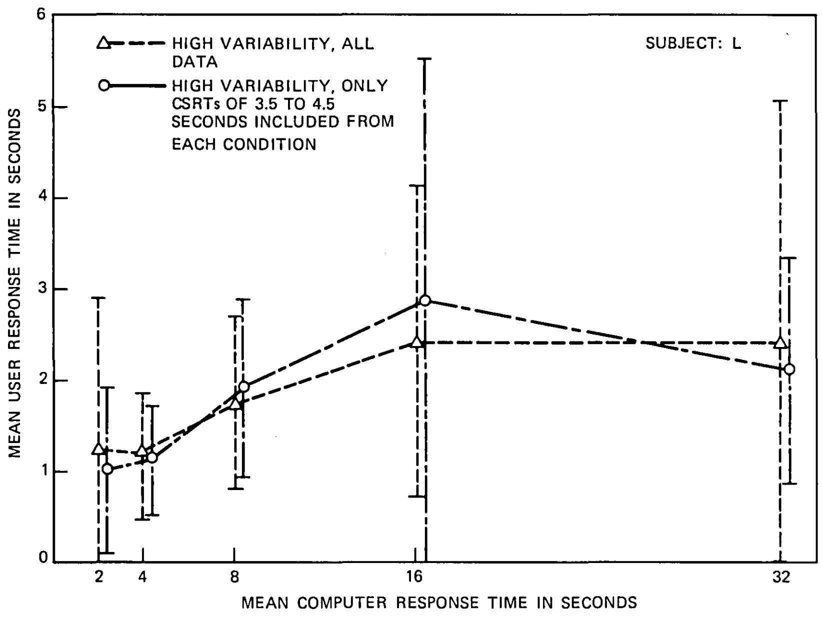


Fig. 4—Average user response time plotted as a function of average computer response time for the subject L.

cording to the same measure. There is no reason to believe that these results can be applied directly to user tasks requiring more cognitive effort, or to tasks in which the user's response depends on the content of the preceding computer output. These results should, though, set a baseline to which user performance with other, presumably more complex, user tasks can be compared.

The psychological basis for the effects seen here is not obvious. At first glance, it could be seen as only a problem of attention for the subjects; the long computer response time intervals could simply be making their responses more lethargic. Two facts, though, suggest that this simple hypothesis is not adequate: First, if attention was a problem, input error rates would probably rise along with each user's response time. This did not happen in the present experiment. Second, the high level of variability seen in these data would probably not be expected if something as simple as attentiveness were mediating subjects' responses. The subjects in the experiment were a fairly homogeneous population of experienced computer users, yet the variability of response time between users and even in each user's case was high. This suggests that some more complex factor or factors are important to the subjects' responses. Further research is needed before we will know the basis for the results of this study.

The most striking finding of this study is how little user performance at this task is influenced by computer response time. Of all the measures of user performance tested, only one was affected at all by computer response time, and the performance degradation shown by this measure as computer response time increased was quite small. The second important finding of the present study is that computer users' performance appears to be related to *average* computer response time, rather than to the duration of the immediately preceding computer response time interval. This finding has important methodological implications for future studies.

REFERENCES

1. J. C. R. Licklider, "Man-Computer Symbiosis," IEEE Trans. Hum. Factors Electron., *HFE-1* (March 1960), pp. 4–11.
2. J. C. R. Licklider, "Man-Computer Partnership," Int. Sci. Technol., *33* (May 1965), pp. 18–26.
3. J. R. Carbonell, J. I. Elkind, and R. S. Nickerson, "On the Psychological Importance of Time in a Time Sharing System," Hum. Factors, *10* (April 1968), pp. 135–42.
4. S. E. Engel and R. E. Granda, "Guidelines for Man/Display Interfaces," IBM Tech. Report TR00.2720, December 19, 1975.
5. S. J. Boies and J. D. Gould, "User Performance in an Interactive Computer System," Proc. 5th Annual Princeton Conf. on Info. Sci. and Systems, Princeton, NJ, March 25–26, 1971.
6. R. E. Barber, "Response Time, Operator Productivity, and Job Satisfaction," Ph.D. dissertation, NYU Graduate School of Business Administration, 1979.
7. M. C. Grignetti and D. C. Miller, "Modifying Computer Response Characteristics to Influence Command Choice," Proc. Conf. on Man-Computer Interactions, Cambridge, UK, September, 1970, pp. 201–5.
8. B. Johnsson, B. Andersson, and S. Wallin, "Man-Computer Communication Through Alpha Numeric Display Terminals," Int. Conf. on Cybernetics and Society, Tokyo-Kyoto, Japan, 1978, *2–3*: pp. 1278–83.
9. K. Hirata and M. P. Bryden, "Tables of Letter Sequences Varying in Order of Approximation to English," Psychonom. Sci., *25*, No. 6 (December 1971), pp. 322–4.
10. C. N. Cofer, "Properties of Verbal Materials and Verbal Learning," *Woodworth and Schlossberg's Experimental Psychology*, 3rd ed., ed. J. W. Kling and L. A. Riggs, New York: Holt, Rinehart, and Winston, 1971, pp. 896–8.
11. K. A. Butler, G. L. Felfoldy, S. E. Simms, and J. S. Swenson, unpublished work.
12. J. D. Williams, J. S. Swenson, J. A. Hegarty, and T. S. Tullis, unpublished work.
13. B. W. Boehm, M. V. Seven, and R. A. Watson, "Interactive Problem-Solving: An Experimental Study of 'Lockout' Effects," AFIPS Conf. Proc., Atlantic City, NJ, May 18–20, 1971, *38*: pp. 205–10.
14. M. A. Morfield, R. A. Wiesen, and M. Grossberg, "Initial Experiments on the Effects of System Delay on On-line Problem-solving," Lincoln Laboratory, MIT, Lexington, MA, Tech. Report No. ESD-TR-69-158, 1969.
15. M. Grossberg, R. A. Wiesen, and D. B. Yntema, "An Experiment on Problem Solving With Delayed Computer Responses," IEEE Trans. Syst. Man Cybern., *SMC-6* (March 1976), pp. 219–22.
16. R. B. Miller, "Response Time in Man-Computer Conversational Transactions," AFIPS Conf. Proc., San Francisco, CA, December 9–11, 1968, *33*, pp. 267–77.
17. S. J. Boies, "User Behavior on an Interactive Computer System," IBM Syst. J., *13*, No. 1 (1974), pp. 2–18.
18. W. J. Doherty and R. P. Kelisky, "Managing VM/CMS Systems for User Effectiveness," IBM Syst. J. *18*, No. 1 (1979), pp. 143–63.

AUTHOR

Thomas W. Butler, B.Sc. (Psychology), 1972, Michigan State University;

Ph.D. (Psychology), 1977, Brown University; University of California at Berkeley, 1976–1978; AT&T Bell Laboratories, 1978—. At the University of California Mr. Butler was a member of the Physiology–Anatomy Department. Since joining AT&T Bell Laboratories, he has worked on methods for the human estimation of work time, and on the effects of computer response time on user performance. He is currently a member of the Human Performance Engineering department, where his work includes a variety of projects related to the *UNIX*™ operating system.

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July-August 1984
Printed in U.S.A.

Program Transformations for Data Access in a Local Distributed Environment

By J. D. DETREVILLE* and W. D. SINCOSKIE†

(Manuscript received March 23, 1983)

This paper presents a set of program transformations that are useful in transforming certain sequential program schemas for use in a local distributed environment. The environment is considered to be a set of processors connected by a local area network with broadcast capability. Examples of transformed program schemas are given that implement shared data, maximization, and abstract queues in a distributed environment.

I. INTRODUCTION

The use of program transformation has been frequently proposed as an aid to program development and program structuring.[1,2] Transformations that preserve program correctness can be used to convert a clearly written program that is unfortunately inefficient or otherwise unsuited to its environment into an equivalent implementation that is more directly usable. Moreover, if transformations are performed mechanically (although, as typically proposed, guided by the user), the developmental relationship between the original program and the transformed program can be retained explicitly, aiding in later understanding of the transformed program and simplifying further changes.

1. Typical simple transformations involve operations such as moving invariant computations outside of loops, or eliminating recursion

*AT&T Bell Laboratories. †AT&T Bell Laboratories; present affiliation Bell Communications Research, Inc.

by the use of explicit stacks; these can achieve greater efficiency at a cost in simplicity.

2. Another family of transformations involves exploitation of certain dualities to transform a program into its dual. For example, Wall notes a duality between sites in a distributed environment and the messages they exchange and proposes that certain classes of programs (in particular, those programs whose structure is related to the topology of the network) be written from the point of view of the messages themselves.[3] Useful insights can be gained through the use of this approach, and since such programs can be mechanically transformed into programs written from the point of view of the processors sending the messages, the new process is easier to execute.

This paper presents a set of program transformations that can be used for programs in a distributed environment to convert references to local data relations into references to the data over a network. We first note a duality between certain looping constructs and a particular distributed communication structure. Programs iterating over a global data relation stored locally are shown to be equivalent to programs making explicit requests to external processors for data; programs written in the first form are easy to understand, but must be converted to the second form to be executed. We then present a further set of transformations that can be performed on programs in the second form to make their communications more efficient.

II. THE DISTRIBUTED ENVIRONMENT

Consider the distributed environment shown in Fig. 1, with some number of similar sites connected over a local area network. These sites are loosely coupled architecturally, with all communication achieved via messages exchanged over the network.

The assumption of a local area network suggests high-speed operation. It also suggests the ability for any one site to broadcast a message to all other sites.

Within this architectural model, assume that we wish to provide some set of shared data, accessible to all the sites and containing information related to the sites themselves. For example, consider the case of a distributed telephone system in which each site controls

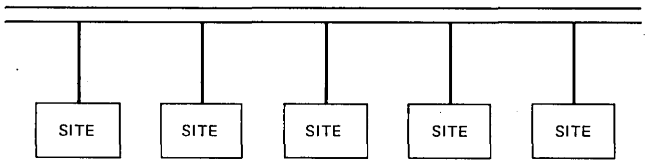


Fig. 1—Distributed environment.

some small number of telephones. A data relation mapping telephone numbers to site addresses could be used as a directory to determine which site is associated with a given telephone number. The question arises of where to store these data. There are at least three distinct approaches:

1. The data could be redundantly stored at every site, requiring quadratic total space (linear at any site); reliability would be very good. Accessing the data would be simple. Updating them could entail significant complexity and cost. For the telephone example above, each site would contain a complete directory, which would be bulky and difficult to update.

2. The data could be stored at some central site, at low cost but with poor reliability. Service would be lost if the central site were to fail. Accessing as well as updating the data would involve communication with the central site. For the telephone example, there would be a central directory server, giving good space efficiency but not allowing telephone calls to be made if the server were unavailable.

3. The data could be stored across sites, with each site holding the data pertaining directly to it. Accessing the data would typically involve broadcast communication with all sites, as could updating the data. Reliability could be very good, although this approach requires quadratic total time (linear at any site). For the telephone example, each site would know only its own telephone numbers and its own site address. Mapping a telephone number to a site address would involve a broadcast message to all sites, followed by a reply or replies. If a site were down, only calls involving it would be affected. For a moderate number of sites, this approach should be reasonably time-efficient.

This paper assumes that the third case is chosen. Thus, programs accessing shared data will need to communicate with the various sites where the data are actually stored.

We show that a program written as though these distributed data were available locally, as in the first case, can be mechanically transformed into one with explicit communication, as in the last case. We also show that this communication can often be made more efficient through the use of further transformations. The increases in efficiency occur with a reduction in the number of messages transmitted. Depending on the network being used and other particulars, there may or may not be a significant advantage to doing this.

III. TRANSFORMING SIMPLE LOOPS

The basic transformation presented here transforms looping structures over data relations into a distributed message-passing structure. The original control structure is a simple iteration over data, while the network topology is a simple iteration of sites. The transformed structure is a simple broadcast to the sites, followed by their iterative replies. We note that, as in Wall's approach, the original control structure corresponds to the topology of the network and is transformed into the communication structure of the resulting program. Consider the pseudocode program fragment:

```
loop for tuple in relation do
    "perform operation on tuple"
end loop
```

If the tuples of the data relation are distributed across sites, this can be transformed into:

```
broadcast (this_relation_id);
loop until all_replies_received do
    receive tuple;
    "perform operation on tuple"
end loop
```

where a separate process at each site performs:

```
loop do
    receive request;
    case request.type in
    . . .
    this_relation_id:
        loop for tuple in relation_here do
            reply tuple
        end loop
  . . .
  end case
end loop
```

Here, an iteration over all tuples in a local relation is transformed into a broadcast request for all sites (including this one) to transmit as replies those tuples of that relation that they remotely store (held in *relation_here*; the constant *this_relation_id* names which relation is being requested), followed by an iteration over the replies. (It is assumed that the order of iteration is unspecified for the original looping construct.)

Since all sites receive the broadcast request almost simultaneously, they could be ready to transmit their replies at about the same time. On contention networks, such as *Ethernet*,*[4] this could lead to a low transmission efficiency due to collisions and retransmissions. On such networks, we could have each site choose an appropriate random delay time to wait before transmitting its tuples.

This transformation covers most accesses to data stored in relations,

*Trademark of Xerox Corporation.

in which the particular tuple or tuples to be used are not known beforehand. If they are known and their home site is also known, we may apply the obvious additional transformations to communicate directly with that site.

In the simplest case, the access to the tuples is read-only. If a tuple is to be updated within the loop, an additional message must be returned to the site from which the tuple was sent.

The problem of synchronization of multiple processes at multiple sites accessing shared data is not considered here. Algorithms for mutual exclusion in a distributed environment are presented by Ricart and Agrawala,[5] including certain approaches similar to those later in this paper.

It may be difficult for the requesting site to determine when all replies have been received. Although the number of sites may be known, some of these may currently be inactive and thus may not send replies. The use of time-outs seems the only workable solution to this problem within the distributed framework assumed here. Since the relative speeds and response times of the sites may vary, this time-out might need to be fairly large, possibly limiting the range of applications of this approach to systems with infrequent accesses to shared data. In the case of a telephone system such accesses would be required only during the call setup, which occurs relatively infrequently and which only needs to proceed at human speeds. We note again that the failure of sites to reply when they are unavailable can be viewed as perfectly appropriate if the tuples being accessed relate to the sites themselves. If a site is down, it can be viewed as nonexistent and so its data should not be seen. Thus, the semantics of the original programs have been (unavoidably) extended to deal with site or communications failures.

In certain cases (where, for instance, there is only one tuple per site), the tuple information may be stored implicitly and regenerated on each request. If this is done, the nature of a data relation will have undergone a substantial shift. Where previously it would have been a data structure in its own right, now it would exist only as an effect of the control structure of the transformed program.

IV. MOVING FUNCTION EVALUATION TO REMOTE SITES

Consider the program fragment:

```
loop for tuple in relation do
    "perform operation on f(tuple)"
end loop
```

where f is some side-effect-free function applied to the tuple. The function can be computed at the remote sites and the value returned instead of the tuple. The program fragment above becomes:

```
  broadcast (this_relation_id);
  loop until all_replies_received do
        receive f_of_tuple;
        "perform operation on f_of_tuple"
  end loop
```
while the separate process at each site is transformed to:
```
  loop do
        receive request;
        case request.type in
        . . .
        this_relation_id:
              loop for tuple in relation_here do
                    reply f(tuple)
              end loop
     . . .
     end case
  end loop
```
Here, all sites combine the application of *f* to the tuples in parallel, which can decrease the total running time of the application by parallel computation and may reduce the length of messages if *f* maps a tuple into less total data. The sites may be heterogeneous and the computation of *f* may depend on some characteristic of the site to which the data refer. Having each site compute its own *f* does not require the requester to know how to compute *f* for the data of other sites. Each site need only know how to compute its own *f*, which could be useful in a heterogeneous network.

V. MOVING TESTS TO REMOTE SITES

The transformation of loops over tuples in a relation, turning a program that accesses data as though it were local into one explicitly requesting remote data, is the basic transformation considered here. After this is done, there are further transformations possible that can make the communication more efficient.

The first additional transformation allows us to perform certain tests remotely. For example, assume that we have transformed the program fragment:
```
  loop for tuple in relation do
        if tuple.key = value then
          "perform operation on tuple"
    end if
  end loop
```
into:
```
  broadcast (this_relation_id);
  loop until_replies_received do
```

```
        receive tuple;
        if tuple.key = value then
           "perform operation on tuple"
     end if
  end loop
```

with appropriate remote server processes receiving the broadcast messages and returning their tuples. We can now invoke a further transformation of the program fragment into:

```
  broadcast ((this_relation_id, this_test_id), value);
  loop until all_replies_received do
        receive tuple;
        "perform operation on tuple"
  end loop
```

with the remote server process transformed to:

```
  loop do
        receive request;
        case request.type in
        . . .
        (this_relation_id, this_test_id):
           loop for tuple in local_relation do
                 if tuple.key = request.value then
                   reply tuple
                 end if
              end loop
           . . .
           end case
        end loop
```

Here, we have moved a test on the tuples to the sites where the tuples are stored. The constant *this_test_id* is used to identify which test the remote sites should perform. Since the remote sites will transmit a reply only if the test succeeds, this transformation can sharply reduce the number of messages transmitted. We note that, since the variable *value* appears free in the test condition, its value must be transmitted to the remote sites for their evaluation of the test.

In the telephone example, we would broadcast a request containing the telephone number to all sites, and, assuming uniqueness of telephone numbers, the site replying would be the one controlling the telephone with that number.

VI. MOVING LOOP TERMINATION TO REMOTE SITES

Consider the program fragment:

```
  found := false;
  loop for tuple in relation do
```

```
        if tuple.key = value then
           found := true;
           break;
        end if
  end loop
```

Using the transformation in the previous section, we can move the test condition to the remote sites. Moreover, we can then make the loop termination (in this case *break*) operate remotely by having the remote servers, while they are waiting to transmit their own replies, watch the network for any prior reply. If one is seen, the site should abort its own reply (and if it had more than one reply possible, it should transmit only one).

Due to race conditions, the requesting site may still receive multiple replies to its requests. Any late replies should be identified and discarded. Of course, this was already the case with replies received after a time-out and can be implemented through the use of serial numbers on requests and their corresponding replies.

VII. MOVING MAXIMIZATION AND MINIMIZATION TO REMOTE SITES

Just as loop termination can be moved to remote sites, as shown in the last section, so can certain ongoing loop computations. One important example is maximization (or minimization) of a quantity over a relation.

Consider the program fragment:

```
  max := negative_infinity;
  largest_tuple := nil;
  loop for tuple in relation do
        if max < tuple.value then
           max := tuple.value;
           largest-tuple := tuple;
        end if
  end loop
```

Applying the transformation to this fragment would give:

```
  max := negative_infinity;
  largest_tuple := nil;
  broadcast (this_relation_id);
  loop until all_replies_received do
        receive tuple;
        if max < tuple.value then
           max := tuple.value;
           largest_tuple := tuple;
        end if
  end loop
```

The remote process would wait for requests and respond with the maximum-value tuple existing in the remote machine.

This transformation can be improved if one takes advantage of the broadcast nature of many local networks. The process at the remote site can delay its reply, while watching the network for other replies larger than its own. If a larger value is seen, the reply is aborted. In general, if the sites reply in random order, the expected value of the number of replies is reduced from N to $\log_2 N$, since about half of the sites will drop out on each reply seen. An even greater improvement can be realized, if the distribution of values is well understood, by having each remote site's delay be inversely correlated with its value, letting the sites with the larger values transmit first.

For the telephone example, there are many times when it is necessary to maximize or minimize some quantity over a relation. For example, when calling a hunt group (a set of telephones with the same number), one wants to find the nonbusy telephone with that number (a pair of conditionals, which can be distributed) that is the earliest in the list (the one that has been idle the longest, or that has not rung for the longest time).

In a distributed computer system, resource allocation is often a maximization problem. For example, if one site finds itself overloaded, it could find the site with the maximum available resource, such as available processor time or main memory, and off-load part of its work to that site. Independent derivations of this approach have been reported earlier: Farber and Heinrich applied it as a "bidding" mechanism for load sharing in a distributed computer system.[6]

Certain abstract queues can be implemented using maximization. A site would enter a queue of sites simply by setting an internal flag. The operation of finding the head of the queue would involve finding the site that has been in the queue the longest time. Here, as before, the queue itself has disappeared. Sites are in the queue if and only if they think they are in the queue; the queue is only an effect of their control structure. Such a scheme can be quite robust, since if a site in the queue fails, then it is no longer in the queue (since it will not reply to queries). Queues of any objects closely coupled to sites can be created in a similar fashion.

VIII. CONCLUSION

A number of transformations have been presented that turn some common sequential program schemas into distributed program schemas. These transformations operate by taking a data structure that exists at a single site and distributing it among a number of sites in a distributed environment. An effect of the distribution is that the explicit data structure "disappears" from the program, and thereafter exists only as an effect of the control structure in the distributed environment.

The transformations presented here have been applied manually to implement initially sequential algorithms in certain experimental distributed environments. Useful distributed program schemas included locating resources in a distributed environment and performing distributed maximization.

REFERENCES

1. R. Balzer, N. Goldman, and D. Wile, "On the Transformational Implementation Approach to Programming," Proc. Second Int. Conf. Software Eng. (October 1976), pp. 337–44.
2. M. S. Feather, "A System for Assisting Program Transformation," Trans. on Programming Languages and Systems, *4*, No. 1 (January 1982), pp. 1–20.
3. D. W. Wall, "Messages as Active Agents," Ninth Annual ACM Symp. on Principles of Programming Languages (January 1982), pp. 34–9.
4. R. M. Metcalf and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Commun. ACM, *19*, No. 7 (July 1976), pp. 395–404.
5. G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," Commun. ACM, *24*, No. 1 (January 1981), pp. 9–17.
6. D. J. Farber, and F. R. Heinrich, "The Structure of a Distributed Computer System—The Distributed File System," Proc. Int. Conf. Computer Commun. (October 1972), pp. 364–70.

AUTHORS

John D. DeTreville, B.S. (Mathematics), 1970, University of South Carolina; S.M. (Computer Science), 1972, Massachusetts Institute of Technology; Ph.D. (Computer Science), 1978, The Massachusetts Institute of Technology; AT&T Bell Laboratories, 1978—. Mr. DeTreville's doctoral thesis was in applied Artificial Intelligence. His first work at AT&T Bell Laboratories was on 5*ESS*™ switching equipment; he moved to Murray Hill in 1980, where his work has incorporated topics in distributed systems and program synthesis. Member, ACM.

W. David Sincoskie, B.E.E., 1975, M.E.E., 1977; Ph.D. (Electrical Engineering), 1980, University of Delaware; AT&T Bell Laboratories, 1980–1983; present affiliation Bell Communications Research, Inc. While at AT&T Bell Laboratories, Mr. Sincoskie was performing research in distributed computing, computer networking, and operating systems. Since 1982, he has been working with integrating voice and data switching on local area networks. In 1983, Mr. Sincoskie was appointed District Research Manager, Computer Communications Research, at Bell Communications Research, Inc. Member, Tau Beta Pi, Eta Kappa Nu, IEEE, ACM.

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July-August 1984
Printed in U.S.A.

System Sparing for Minicomputer-Based Operations Systems

By D. W. TOLLETH*

(Manuscript received March 29, 1983)

A queueing model was developed to derive guidelines for deploying system spares—complete minicomputer systems used as backup for failed minicomputer-based Operations Systems (OSs). The model is a three-dimensional set of queueing equations incorporating a time-dependent number of repairers, and exponential random variables for the time between failures for each minicomputer system, the time to repair each system, and the time to switch between failed systems and spares. Guidelines derived from numerical solution of the model are being used by the Bell Operating Companies and AT&T Communications to aid planning studies for (1) proving-in system spares, (2) meeting specific OS availability objectives, and (3) improving minicomputer maintenance staff utilization.

I. INTRODUCTION

1.1 Background

Development of minicomputer-based Operations Systems (OSs) to support Bell System operations began in the early 1970s. By the end of 1983 over 5000 such OSs were deployed by the Bell Operating Companies (BOCs) and AT&T Communications.

System spares are complete minicomputer systems that provide backup computing when minicomputer-based OSs fail. Increasing dependence on minicomputer systems to perform daily work functions has led the BOCs and AT&T Communications to deploy system spares

*AT&T Bell Laboratories.

for OSs clustered in Minicomputer Maintenance and Operations Centers (MMOCs).

This paper presents a comprehensive solution to the system sparing problem. The BOCs and AT&T Communications are using guidelines based on this solution for determining optimal system sparing levels and for meeting specific availability objectives. The analysis includes transient solutions that examine possible improvements in maintenance staff utilization with sparing.

The work presented here has potential applications in system design, as well as in BOC and AT&T Communications system and staff planning. OSs with stringent availability objectives have usually been designed in duplex or triplex arrangements, with one or two active and one backup processor. For such systems with large deployments in clustered environments, this work shows how to meet stringent availability objectives with fewer backup systems.

1.2 Approach

We achieved the desired results in three steps. First, we characterized the MMOC equipment, staff, and functions pertinent to system sparing, including staffing levels, system failure rates, and mean times to repair. Second, we developed a mathematical model of the sparing process. Third, we obtained system availability data from the model for a range of parameters representing current BOC and AT&T Communications MMOC operations.

1.3 Overview

Section II gives an in-depth discussion of the model and the parameters chosen to characterize the BOC and AT&T Communications MMOCs. Section III presents the mathematical details of the model, including the principal equations and state diagrams. Section IV presents availability data obtained from the model. The presentations of the data are designed to aid planning studies for proving-in spare systems, meeting specific availability objectives, and improving maintenance staff utilization. Section V presents some observations about system sparing.

II. DESCRIPTION OF THE SYSTEM SPARING MODEL

2.1 Model features

How does system sparing affect BOC and AT&T Communications minicomputer users and the MMOCs? To answer this question, we must first characterize the equipment, personnel, and functions involved with system sparing. We can then develop a mathematical model to quantify the effects of sparing.

Minicomputer systems in an MMOC can be located in a single building or distributed throughout a geographic area, with some systems clustered and some standing alone. Our interest in sparing suggests that we break the systems into two groups: (1) a cluster of systems with one or more spares in one location, and (2) the other, nonspared systems in the MMOC (see Fig. 1). The second group may be scattered geographically, clustered at another location, or colocated with the first group. To simplify the model, the second group is sized so that the total number of minicomputers is fixed and corresponds to the typical number of systems for which one repairer would be responsible in the field. This number of systems, 25, is based on current maintenance force staffing levels in BOCs doing self-maintenance, i.e., doing their own hardware maintenance, instead of contracting with a vendor.

Three personnel groups would be affected by system sparing. First, the minicomputer operators, who are located at the minicomputer

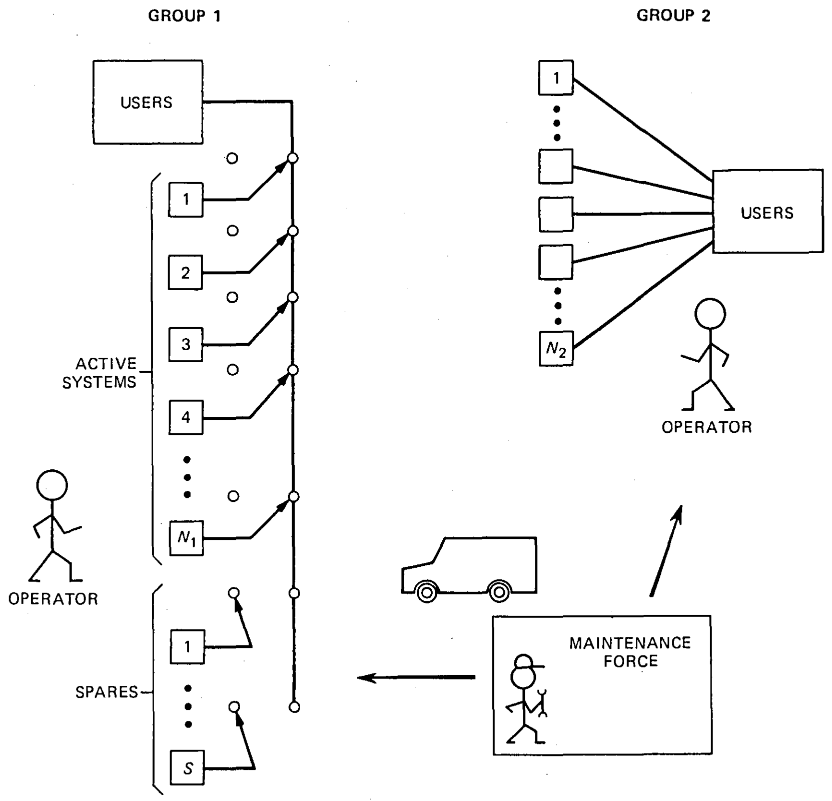


Fig. 1—MMOC systems and personnel.

sites, would be required to do the switching from failed systems to spares and back. Second, the minicomputer maintenance personnel would feel less pressure to repair a failed system if a spare were available. Finally, the minicomputer users would see improved system availability. We will see that availability can improve for all users, including those whose systems are not equipped with a spare.

Users are generally at locations remote from the minicomputer systems and communicate with them via private line circuits. These are the circuits that the operators must switch when moving from a failed system to a spare. Operators switch the data circuits and load the database of the failed system onto the spare.

The effects of sparing on maintenance are broader and more subtle than its effects on operations. When a cluster has a spare, a failed system does not demand immediate attention; the spare can be switched. Thus, when multiple failures occur, priority can be given to the systems for which no spare is available. The effect is more timely repair of the systems without a spare. Of course, users of systems equipped with a spare also enjoy improved availability. Both groups benefit.

In addition to modifying repair priorities, sparing also levels the work load of the maintenance staff: minicomputers can be queued for repair instead of maintenance staff being queued for minicomputer failures. This may allow reduction in the size of the maintenance staff or elimination of nighttime maintenance. The latter will be possible if sparing sufficiently reduces the risk of user outage at the beginning of the morning shift.

For simplicity, the model omits several common practices. First, the model considers only corrective maintenance activities and ignores the value of a spare for preventive maintenance and database management activities. Second, the model omits the common practice of maintenance personnel working overtime at the end of a shift to finish a repair.

The omission of these two features means that the model underestimates the benefits of sparing, because both practices increase the value of sparing. Spare systems can be switched for working systems, allowing preventive maintenance to be performed during the day or evening, rather than during the night shift or on weekends; and overtime at the end of a shift gets systems back on line faster, improving system availability.

The model also omits the step of "switching back" repaired systems for spares. This task is usually performed at night or at some other time when there is little or no demand for a system, so the omission should have little effect on the results.

The system sparing process was modeled mathematically, as described in Section III. The parameters used in the model are discussed below.

2.2 Model parameters—operating conditions in the MMOC

Data collected by the MMOC planning group at AT&T Bell Laboratories indicate that providing 24-hour, 7-day maintenance coverage in a clustered environment requires one repairer per shift for every 25 "COSMOS-like" systems, i.e., nominal *DEC PDP-11/70**-based systems.

Because a repairer covers 25 systems and we want the number of repairers to be an integer, we need to model 25 systems, 50 systems, etc. The minimum set—25 systems, 1 repairer and 1 spare—can be shown to satisfy the objectives of this study.

When a spare is added, the model assumes no increase in the size of the maintenance staff. The repairer who was responsible for 25 systems before sparing is thus responsible for 26 after sparing.

Our data show that systems in the field incur about 100 hours of downtime per year for corrective maintenance. One hundred hours per year corresponds roughly to a mean time between failures of 22 days and a mean time to repair of 6 hours. (The mean time to repair includes both the time to respond to and to fix the trouble.) There is considerable variation in downtime, due to such factors as the environment in which a minicomputer is operated, whether a system is maintained by on-site BOC personnel or off-site vendor personnel, and whether downtime measured is user downtime or system downtime. To allow for these variations, the model was run with 50, 100, and 150 hours of downtime per year.

Switching between a failed system and a spare requires switching the data lines and moving the database. Moving the database, which includes removing the disk packs, moving them to the spare system, and checking the files, is the rate-limiting step, and typically requires about 15 minutes. The time to move the data lines varies, depending on the switching method, but it is usually much less than 15 minutes. The model assumed 15 minutes total for switching.

To permit examination of possible improvements in maintenance staff utilization, the model allows the number of repairers on a shift to change. In particular, the model accommodates no, one, or infinite repairers. No repairers is the case used to model uncovered shifts. One repairer is the nominal case for 25 systems and 1 spare. Infinite repairers is the case used as an approximation for the two-repairer case to examine movement of the night shift personnel to the day

* Trademark of Digital Equipment Corporation.

shift. The validity of the infinite repairer approximation is discussed in Section III.

In summary, the model assumes 25 systems; 1 spare; and 0, 1, or ∞ repairers. The time required to switch a spare for a failed system is 15 minutes. Data were produced for systems with 50, 100, and 150 hours of downtime per year.

III. TIME-DEPENDENT QUEUEING MODEL

System sparing increases system availability and allows improvements in maintenance staff efficiency. System availability and the number of repairers required to maintain the systems are, therefore, the principal indicators of the effects of sparing.

We want to develop a mathematical model that predicts average system availabilities with and without sparing, as a function of time. It must keep track of three groups of systems: (1) systems with access to spares (group 1), (2) spare systems, and (3) other systems under the purview of the maintenance staff (group 2). A novel feature of this grouping is the ability to analyze sparing for differing numbers of systems per spare while maintaining a constant work load on the maintenance staff.

The model must account for changes in maintenance force staffing levels, for example at shift changes. Since the mean time to repair (6 hours) is of the order of a shift (8 hours), steady-state solutions will not be attained within a shift. So, we must develop a time-dependent model with a time-dependent number of servers.

We begin by stating assumptions for the model and defining notation. Then, state diagrams and state equations are presented. Finally, we discuss the numerical methods used and checks made on the model.

3.1 Assumptions and notation

We assume that the time between failures for each machine is an independent and identically distributed (i.i.d.) exponential random variable. Switching time and response plus repair time are also i.i.d. exponential random variables. The random variable for switching time is probably closer to deterministic. However, an exponential distribution produces more congestion in the model than a deterministic distribution. So, the model errs on the side of less recovered availability. Our conclusions, thus, are conservative.

States are described by $p(t; n_1, s, n_2)$, the probability that at time t, n_1 systems are failed in group 1, s spares are failed, and n_2 systems are failed in group 2.

We define failure rates λ_1 and λ_2 for systems in group 1 and group 2, respectively. Spares are identical to systems in group 1 and have the same failure rate, λ_1.

The model considers three cases:

1. No repairers—the case to consider for late-night and weekend shifts.

2. One repairer—the nominal case because, for purposes of analysis, the total number of systems modeled (group 1 plus group 2) is sized for one repairer on duty 24 hours per day.

3. Infinite repairers—the case used to approximate the two-repairer case.

There are three different state diagrams and three sets of state equations. The time-dependent parameter is the number of repairers, which can change at 8-hour intervals, corresponding to work shifts. In all cases the time to repair is assumed to be an i.i.d. exponential random variable, with mean $1/\mu$, which is the same for all systems. This time is assumed to include response time. Finally, the time required for an operator to switch a spare for a failed system is also an i.i.d. exponential random variable, with mean $1/\mu_s$.

To summarize, the model is characterized by the following:

$p(t; n_1, s, n_2)$—probability that at time t, n_1, s, and n_2 systems are in the failed state from the total pool of N_1 systems in group 1, S spares, and N_2 systems in group 2, respectively.

λ_1—failure rate for group 1 systems and spare systems.

λ_2—failure rate for group 2 systems.

μ—repair rate for a system (single repairer rate).

μ_s—rate to switch a spare for a failed system.

3.2 State diagrams and state equations

Consider now the three cases for repair.

Case 1: No repairers—Figure 2 shows the state diagram for a general state (n_1, s, n_2) with no repairers. The corresponding state equation describing the probability at time $t + \Delta t$ of being in state (n_1, s, n_2) is, leaving out terms $o(\Delta t)$,

$$\begin{aligned} p(t + \Delta t; n_1, s, n_2) = [1 &- (N_1 - n_1)\lambda_1\Delta t - (S - s)\lambda_1\Delta t \\ &- (N_2 - n_2)\lambda_2\Delta t - \mu_s\Delta t]p(t; n_1, s, n_2) \\ &+ (N_1 - n_1 + 1)\lambda_1\Delta t\, p(t; n_1 - 1, s, n_2) \\ &+ (S - s + 1)\lambda_1\Delta t\, p(t; n_1, s - 1, n_2) \\ &+ (N_2 - n_2 + 1)\lambda_2\Delta t\, p(t; n_1, s, n_2 - 1) \\ &+ \mu_s\Delta t\, p(t; n_1 + 1, s - 1, n_2). \qquad (1) \end{aligned}$$

Switching spares for failed systems in group 1 continues until all

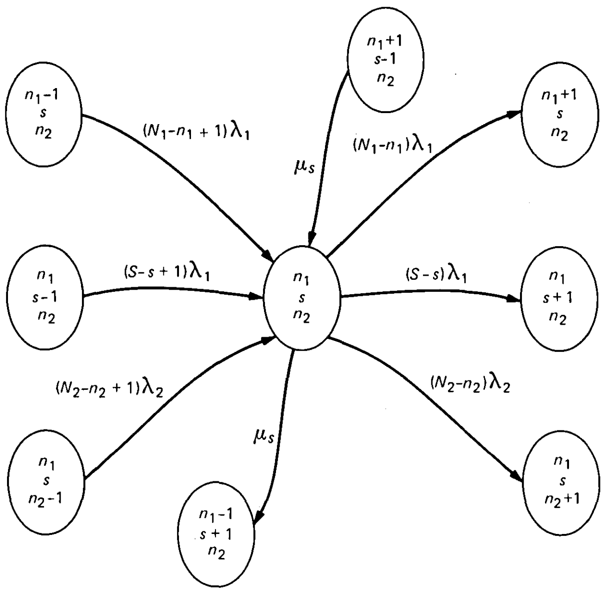


Fig. 2—State diagram for the general state (n_1, s, n_2) with no repairers.

spares are failed ($s = S$). At that point, switching stops. For all states with $s = S$, μ_s is set to zero.

There are two observations that should be made at this point. First, the group 2 part of eq. (1) is completely separable from the part for group 1 and the spares. As we discussed later, separability was used as a check on the numerical solution. We can write

$$p(t;\, n_1,\, s,\, n_2) = p(t;\, n_1,\, s)p(t;\, n_2) \tag{2}$$

and obtain a closed-form solution for $p(t; n_2)$. For ease of programming, this separation was not made in the model. Second, it is clear that the steady-state solution for this case is $p(\infty; N_1, S, N_2) = 1$. That is, with no repairer all systems are failed in the limit $t \rightarrow \infty$.

Case 2: One repairer—Adding a repairer requires four criteria in the model that determine which system the repairer will fix for a given state of the system. First, in the field the repairer probably fixes systems on a first-in first-out basis. Since all failure, repair, and switching times are i.i.d. exponential random variables, the state space is memoryless; it is impossible to determine which minicomputer failed first. For predicting minicomputer system availability with this model, the equivalent of a first-in first-out repair strategy is random repair. The probability that a given system is under repair is defined such that all failed systems have the same opportunity to be repaired. Second, the purpose of spares is to replace failed group 1 systems.

Thus, group 1 systems are not repaired if spares are available; they are switched. Third, spares are not repaired if there are any failed group 2 systems. Finally, when spares are available for failed group 1 systems, priority is given to repair of failed group 2 systems. In summary, the decision criteria for repair are:

1. Randomly repair the failed systems in group 1 and group 2 if all the spares have failed.
2. Never fix group 1 systems if spares are available.
3. Repair spares only when all group 2 systems are working and there are spares available for any failed group 1 systems.
4. Always fix group 2 systems first unless all the spares have failed.

The parameter $\chi(n_1, s, n_2)$ is defined to incorporate these criteria. χ is the probability that either (1) a system in group 1 will be fixed, when $s = S$, or (2) one of the failed spares will be fixed, when $s < S$. The only exception is that χ is the probability that a spare will be fixed, when $s = S$ and $n_1 = 0$. The probability that a system in group 2 will be fixed is $1 - \chi$. The value of χ corresponding to each of the above criteria is:

1. $\chi(n_1, s, n_2) = \dfrac{n_1}{n_1 + n_2}$, for $s = S$, $n_1 + n_2 > 0$.
2. $\chi(n_1, 0, 0) = 0$, for $S > 0$.
3. $\chi(n_1, s, n_2) = 1$, for $s < S$, $n_2 = 0$; or $s = S$, $n_1 = n_2 = 0$.
4. $\chi(n_1, s, n_2) = 0$, for $s < S$, $n_2 > 0$.

Figure 3 shows the state diagram for a general state (n_1, s, n_2) with one repairer. The corresponding state equation describing the probability at time $t + \Delta t$ of being in state (n_1, s, n_2) is, leaving out terms $o(\Delta t)$,

$$\begin{aligned} p(t + \Delta t; n_1, s, n_2) = {} & [1 - (N_1 - n_1)\lambda_1\Delta t - (S - s)\lambda_1\Delta t \\ & - (N_2 - n_2)\lambda_2\Delta t - \mu\Delta t - \mu_s\Delta t]p(t; n_1, s, n_2) \\ & + (N_1 - n_1 + 1)\lambda_1\Delta t\, p(t; n_1 - 1, s, n_2) \\ & + (S - s + 1)\lambda_1\Delta t\, p(t; n_1, s - 1, n_2) \\ & + (N_2 - n_2 + 1)\lambda_2\Delta t\, p(t; n_1, s, n_2 - 1) \\ & + \mu_s\Delta t\, p(t; n_1 + 1, s - 1, n_2) \\ & + \chi(n_1 + 1, s, n_2)\mu\Delta t\, p(t; n_1 + 1, s, n_2)^* \\ & + \chi(n_1, s + 1, n_2)\mu\Delta t\, p(t; n_1, s + 1, n_2)^\dagger \\ & + [1 - \chi(n_1, s, n_2 + 1)]\mu\Delta t\, p(t; n_1, s, n_2 + 1). \quad (3) \end{aligned}$$

* Omit when $s < S$.
† Omit when $s + 1 = S$, $n_1 > 0$.

If there are no spares and $\lambda_1 = \lambda_2$, then there is nothing different about group 1 systems compared with group 2 systems. The model is equivalent to one large group of $(N_1 + N_2)$ minicomputer systems. While the time dependence makes a solution nontrivial, this is a standard, finite source, M/M/1 queue with a well-known steady-state solution.[1] This feature was used to check the model, as discussed later.

Case 3: Infinite repairers—With an infinite number of repairers no criteria are required to determine which system gets fixed next. But, in the spirit of a two-repairer approximation, we impose the following condition on the repair process: Never fix group 1 systems if spares are available.

Figure 4 shows the state diagram for a general state (n_1, s, n_2) with infinite repairers. The corresponding state equation describing the probability at time $t + \Delta t$ of being in state (n_1, s, n_2) is, leaving out terms $o(\Delta t)$,

$$\begin{aligned} p(t + \Delta t; n_1, s, n_2) = [1 &- (N_1 - n_1)\lambda_1\Delta t - (S - s)\lambda_1\Delta t \\ &- (N_2 - n_2)\lambda_2\Delta t \\ &- (n_1{}^* + s + n_2)\mu\Delta t - \mu_s\Delta t]p(t; n_1, s, n_2) \\ &+ (N_1 - n_1 + 1)\lambda_1\Delta t\, p(t; n_1 - 1, s, n_2) \\ &+ (S - s + 1)\lambda_1\Delta t\, p(t; n_1, s - 1, n_2) \\ &+ (N_2 - n_2 + 1)\lambda_2\Delta t\, p(t; n_1, s, n_2 - 1) \\ &+ \mu_s\Delta t\, p(t; n_1 + 1, s - 1, n_2) \\ &+ (n_1 + 1)\mu\Delta t\, p(t; n_1 + 1, s, n_2)^* \\ &+ (s + 1)\mu\Delta t\, p(t; n_1, s + 1, n_2) \\ &+ (n_2 + 1)\mu\Delta t\, p(t; n_1, s, n_2 + 1). \end{aligned} \tag{4}$$

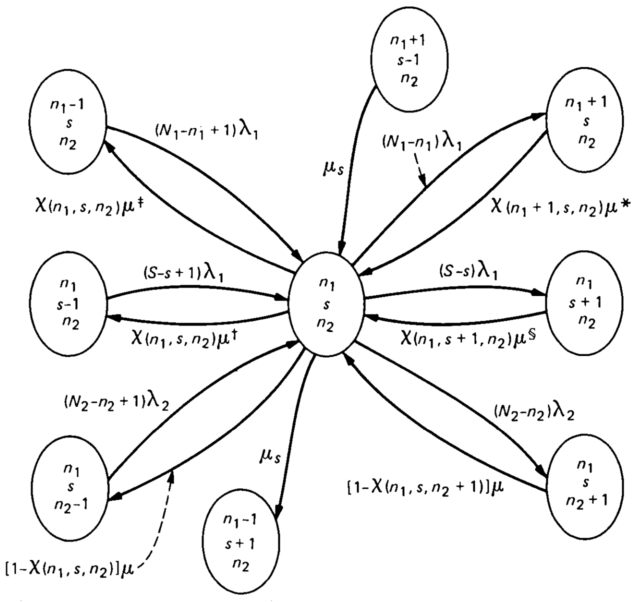
The group 2 part of eq. (4) is separable from the parts for group 1 and the spares, as is the case for no repairers [see eq. (2)]. Again, for programming ease this separation was not made.

3.3 Numerical methods and checks of the model

With the three state equations in hand we can solve for minicomputer system availabilities under different sparing and maintenance strategies.

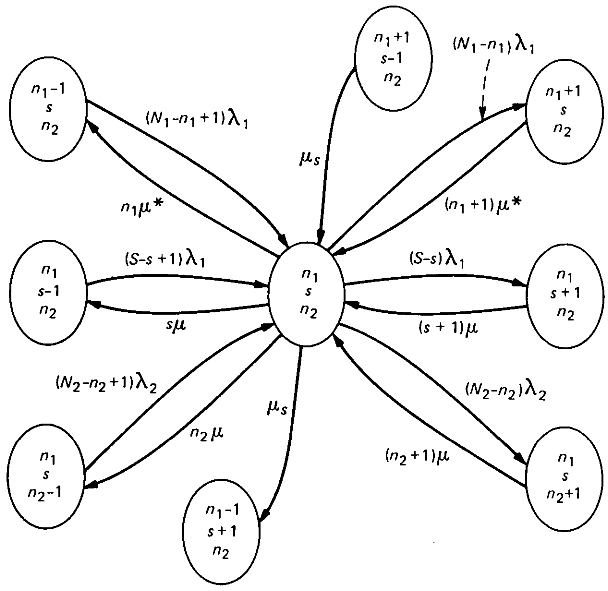
The standard procedure to arrive at analytic solutions for the state

* Omit when $s < S$.



*OMIT WHEN $s < S$.
† OMIT WHEN $s+1 = S$, $n_1 > 0$.
‡ OMIT WHEN $s < S$; OR $s = S$, $n_1 = 0$.
§ OMIT WHEN $s = S$, $n_1 > 0$.

Fig. 3—State diagram for the general state (n_1, s, n_2) with one repairer.



*OMIT WHEN $s < S$.

Fig. 4—State diagram for the general state (n_1, s, n_2) with infinite repairers.

equations is to take the limit as $\Delta t \to 0$ to obtain differential equations for $p(t;\, n_1,\, s,\, n_2)$. But analytic solutions of the differential equations are intractable.

An alternative is to take the numerical approach and integrate the difference equations (1), (3), and (4). A computer program was written to perform the numerical integration. The computer program takes the probability distribution at some initial time, T_i, and integrates in $(T_e - T_i)/\Delta t$ steps to the desired end time, T_e. The initial distribution $p(T_i;\, n_1,\, s,\, n_2)$ was chosen such that

$$p(T_i;\, 0,\, 0,\, 0) = 1 \tag{5}$$

and for all other values of n_1, s, and n_2

$$p(T_i;\, n_1,\, s,\, n_2) = 0. \tag{6}$$

To avoid nonlinear effects Δt must be small. Even so, numerical roundoff will eventually intrude with the result that

$$\sum_{n_1, s, n_2} p(t;\, n_1,\, s,\, n_2) \neq 1. \tag{7}$$

To compensate for this, the probability distribution was normalized periodically, dividing each probability by the sum of the probabilities, to ensure that the sum remains one.

To escape the influence of the initial conditions, the model must be run for a time period whose length depends on the conditions being modeled. For a constant number of repairers the model must run until the probability distribution becomes constant. For a time-dependent weekly maintenance schedule, the model must run until the probability distribution becomes periodic, repeating from week to week.

Availabilities are calculated from the probability distribution at each hour. The probability that n_1 systems are failed in group 1 at time t is

$$p(t;\, n_1) = \sum_{s=0}^{S} \sum_{n_2=0}^{N_2} p(t;\, n_1,\, s,\, n_2). \tag{8}$$

The time-dependent, average number of failed group 1 systems is then

$$\langle n_1(t) \rangle = \sum_{n_1=0}^{N_1} n_1 p(t;\, n_1). \tag{9}$$

From this the average availability of a system in group 1 is

$$n_1 \text{ avail}(t) = 1 - \frac{\langle n_1(t) \rangle}{N_1}. \tag{10}$$

Similar expressions hold for spares and group 2 systems. These expressions produced the data for the figures in this paper.

In addition to availabilities, two parameters were calculated: repairer occupancy (the probability that a repairer is busy) and the probability that three or more systems are failed. The first served as a check of the model. One expects a single repairer's work load to increase by about 4 percent (1 spare/25 systems) when the spare is added. One also expects the work load to be independent of the distribution of the 25 systems between group 1 and group 2. Calculated repairer occupancies agreed with expected values with greater than 0.1 percent accuracy.

The second parameter served to verify the infinite repairer approximation of two repairers. For all cases examined with the maintenance schedule used in the figures, the probability that three or more repairers were busy was less than 0.065 when the number of repairers was infinite. That is, for at most 6.5 percent of the time more than two of the infinite repairers are busy.

Four other checks were made to verify the calculations in the model; three of these were mentioned earlier. First, for no repairers the time dependence for the group 2 systems can be checked explicitly. Using the initial condition that all systems are working at the starting time, T_i, $p(t; n_2)$ from eq. (2) is given by

$$p(t;\, n_2) = \binom{N_2}{n_2} e^{-N_2\lambda_2 t}[e^{\lambda_2 t} - 1]^{n_2}, \tag{11}$$

which gives

$$\langle n_2(t)\rangle = N_2(1 - e^{-\lambda_2 t}). \tag{12}$$

The numeric solution agreed with this relation with greater than 0.1 percent accuracy. It is one of the best checks of the choice for Δt (0.005 hour) and the decision to normalize after every Δt interval, because it checks the time dependence as well as the steady state.

Second, for one repairer the random repair criterion was checked for the case with no spares and $\lambda_1 = \lambda_2$. As we mentioned earlier, in this case the model is equivalent to one large group of $N_1 + N_2$ minicomputers and has a well-known steady-state solution:[1]

$$\langle n\rangle = \frac{\sum_{n=0}^{N} n \dfrac{N!}{(N-n)!}\left(\dfrac{\lambda}{\mu}\right)^n}{\sum_{n=0}^{N} \dfrac{N!}{(N-n)!}\left(\dfrac{\lambda}{\mu}\right)^n}. \tag{13}$$

If we use $N = N_1 + N_2$, the average number of failed systems should be the sum of the average numbers for group 1 and group 2 in the model:

$$\langle n \rangle = \langle n_1 \rangle + \langle n_2 \rangle. \tag{14}$$

In addition, the average availabilities for systems in group 1 and group 2 should be equal and the same as that of the large group. Agreement with this relation not only checked the numerical calculation, it also verified that the random repair criterion is equivalent to first-in first-out, as far as average availabilities are concerned.

Third, for infinite repairers the steady-state solution was checked for the group 2 systems. Equation (4) separates, as we said, allowing an explicit solution for $p(\infty;\, n_2)$, which agreed with the computer calculations with greater than 0.1 percent accuracy. The steady-state solution gives

$$\langle n_2 \rangle = \frac{\sum_{n_2=0}^{N_2} n_2 \binom{N_2}{n_2} \left(\frac{\lambda_2}{\mu}\right)^{n_2}}{\sum_{n_2=0}^{N_2} \binom{N_2}{n_2} \left(\frac{\lambda_2}{\mu}\right)^{n_2}}. \tag{15}$$

Finally, special conditions exist at the boundaries of the state space. These conditions result, for example, when switching terms drop out of the state equations for $s = S$ or $n_1 = 0$, or when the repair term drops out for $n_1 = s = n_2 = 0$. For one repairer the case with the most complex boundary conditions, an explicit, steady-state solution, was derived for $N_1 = S = N_2 = 1$. This case employs all possible boundary conditions, including those for the decision parameter, χ. The steady-state computer calculation agreed with the explicit solution with greater than 0.1 percent accuracy.

IV. PLANNING FOR SYSTEM SPARING

To provide tools for studies of system sparing, system availability data from the model have been arranged to answer three questions about sparing:

1. What is the minimum number of identical systems needed for proving-in the cost of sparing?
2. Given a minimum system availability objective for a group of identical systems, what is the minimum number of spares required to meet the objective?
3. What are the operational benefits of sparing for the MMOC?

4.1 Proving-in spares

Figure 5 shows the annual recovered downtime, using one spare for the systems in group 1, as a function of the number of systems in group 1, for annual downtimes of 50, 100, and 150 hours per system without a spare, respectively. For each system,

$$\text{Downtime} = (1 - \text{availability}) \times 24 \text{ hrs/day} \times 365 \text{ days/yr.} \quad (16)$$

The solid curves in the figures show the total downtime recovered for all group 1 systems. Nonspared systems, those in group 2, also benefit from sparing, because a repairer can give them higher-priority service when a spare can be switched for a failed group 1 system. The dashed curves in the figures include the additional downtime recovered for group 2 systems.

These curves provide essential data for an economic analysis for proving-in sparing.

4.2 Meeting an availability objective

Figure 6 shows the availability per system for group 1 systems, as a function of the number of group 1 systems. The three curves correspond to annual downtimes of 50, 100, and 150 hours per system without a spare. The data assumes 1 repairer, 24 hours per day, and 1 spare.

These data can be used to determine the maximum number of systems that can be loaded onto one spare and still meet a specified availability objective. For a given availability objective on the vertical axis in the figure, read over to the curve corresponding to the average

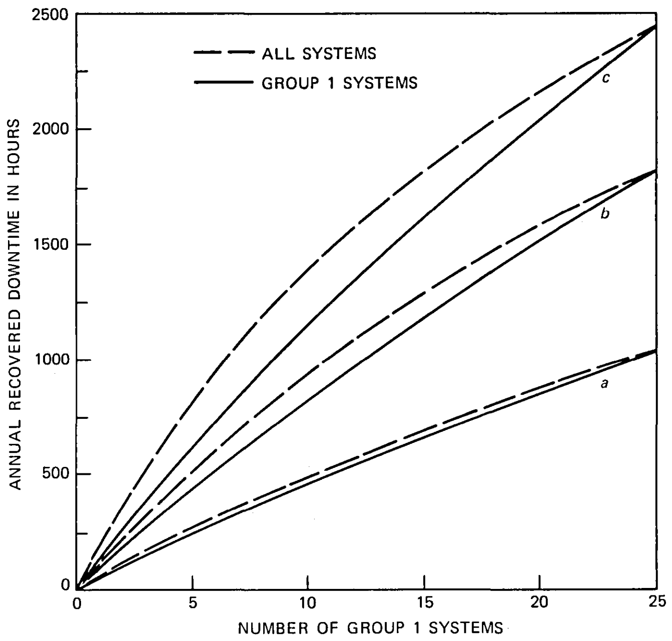


Fig. 5—Annual downtime recovered for group 1 systems and for all 25 systems, group 1 plus group 2, with one repairer, 24 hours per day, and one spare. Annual downtime is (a) 50, (b) 100, and (c) 150 hours per system without a spare.

system downtime without a spare. The maximum number of systems for which the availability objective can be met with one spare is then read on the horizontal axis.

4.3 Improving maintenance staff utilization

The solid curves in Figs. 7 and 8 show, for groups 1 and 2, respectively, system availabilities with sparing, when night shift maintenance personnel are moved to the day shift and weekend maintenance is done on a call-out basis. Such a maintenance schedule could reduce maintenance staff attrition (people do not like to work at night) and would decrease the maintenance staff size by eliminating full-time weekend coverage. These availability data are compared with system availabilities for 1 repairer and 24-hour, 7-day coverage, both with a spare (short-dashed lines in the figures) and without a spare (long-dashed lines). This is one of two potential strategies to improve maintenance staff utilization with sparing. The second is to increase the number of systems per repairer when spares are deployed. The first is recommended for clusters deploying spares. The second can be shown to be inadvisable since the value of lost system availability due to maintenance staff reduction outweighs the savings in staff salaries.

4.4 Interpolations, extrapolations, and sensitivities

Planning studies for system sparing will often require availability data for parameters that differ from those described in Section II. The differences will probably occur in three areas. First, annual system downtimes without sparing will generally not be 50, 100, or 150 hours, but instead will be somewhere within this range. Linear interpolation

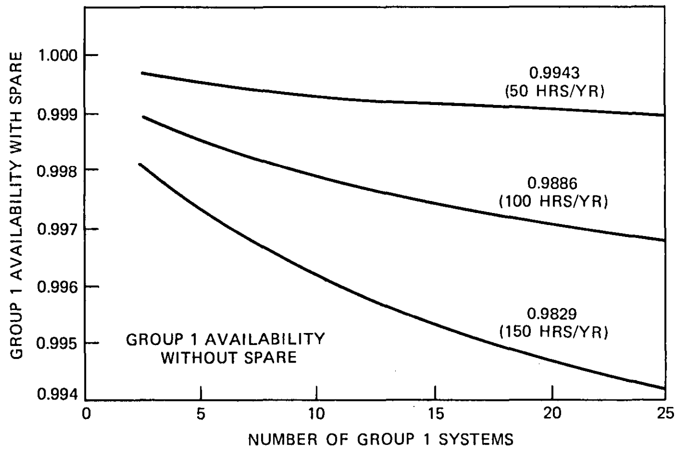
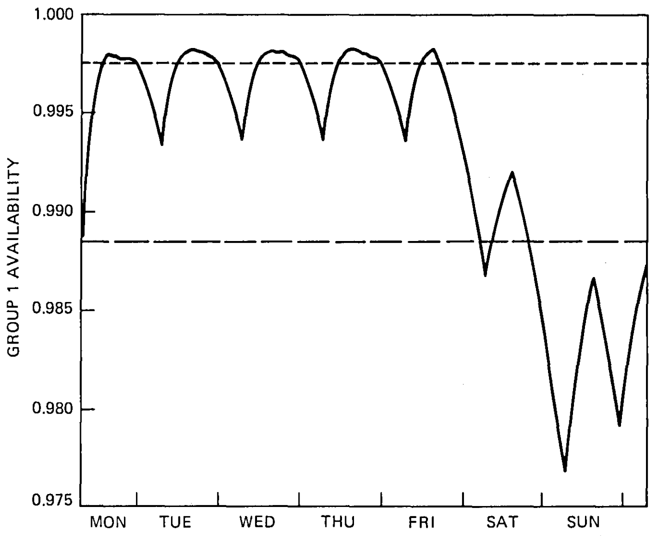


Fig. 6—System availability for minicomputers with sparing (group 1 in the model) as a function of the number of group 1 systems. Curves correspond to annual downtimes in hours per system per year without sparing.

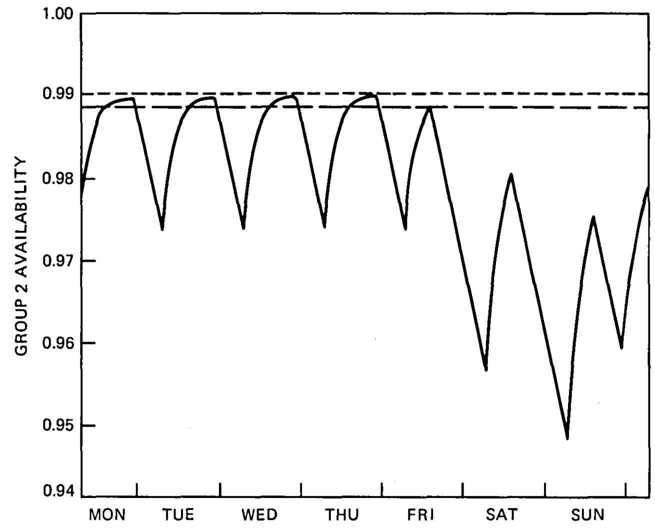


	REPAIRERS PER SHIFT						
	MON	TUE	WED	THU	FRI	SAT	SUN
DAY	2	2	2	2	2	1	1
EVENING	1	1	1	1	0	0	0
NIGHT	0	0	0	0	0	0	1

Fig. 7—Effect of one spare on system availability of 15 group 1 systems is compared for two repairers during the day, one in the evening, and weekend repair done on a call-out basis (solid curve); and for 24-hour, 7-day, single-repairer maintenance (long-dashed line). System availability without sparing is shown for 24-hour, 7-day, single-repairer maintenance coverage (short-dashed line), corresponding to 100 hours of annual down-time per system.

from the data in Figs. 5 and 6 will provide the desired system availabilities.

Second, the number of systems per repairer (group 1 plus group 2) will vary. Changes in this number have less effect on group 1 systems than on group 2 systems. For group 1 systems, as the number of systems per repairer decreases from 25 to 15, recovered downtime increases by 5.5 percent, and as the number increases from 25 to 35, recovered downtime decreases by 3.5 percent. Thus, over a broad range of systems per repairer (15 to 35), the group 1 availability data in Figs. 5 and 6 are accurate to within 5.5 percent. For group 2 systems the effect is more pronounced. For a constant number of group 1 systems, the group 2 contribution to recovered downtime (the region between the solid and dashed curves in Fig. 5) is roughly linearly proportional to the number of group 2 systems. For example, when the number of group 2 systems is one third the value used in the figures, the group 2



REPAIRERS PER SHIFT	MON	TUE	WED	THU	FRI	SAT	SUN
DAY	2	2	2	2	2	1	1
EVENING	1	1	1	1	0	0	0
NIGHT	0	0	0	0	0	0	1

Fig. 8—Effect of one spare on system availability of 10 group 2 systems is compared for two repairers during the day, one in the evening, and weekend repair done on call-out basis (solid curve); and for 24-hour, 7-day, single-repairer coverage (long-dashed line). System availability without sparing is shown for 24-hour, 7-day, single-repairer maintenance (short-dashed line), corresponding to 100 hours of annual downtime per system.

contribution to recovered downtime is roughly one third the value shown.

Finally, switching time will not be the same for every cluster. The availability data are insensitive to this parameter. Changing the switching time from 15 minutes to 30 minutes produces a negligible change in recovered availability.

V. OBSERVATIONS

5.1 Networking

Increasing dependence on minicomputer-based OSs has led the BOCs and AT&T Communications to deploy system spares for OSs clustered in MMOCs. As networking of systems continues, the impact of system downtime, and thus the value of sparing, will grow. With networking, an out-of-service minicomputer system not only fails to perform its assigned task, it also fails to provide essential data to other systems on the network.

5.2 Commonality

To be economical, system sparing requires a sufficient number of clustered OSs with identical hardware. Studies of system sparing may cause BOCs and AT&T Communications to relocate some systems to clusters with other like systems, or to purchase new systems with older model peripherals to achieve the commonality needed to spare a group of OSs. In anticipation of increasing reliance on high system availability, it is clearly desirable that future OS developments use common hardware.

REFERENCES

1. L. Kleinrock, *Queueing Systems*, New York: Wiley, 1975, Vol. 1: Theory, p. 106.

AUTHOR

David W. Tolleth, B.A. (Physics), 1974, University of California, Irvine; M.S., Ph.D. (Physics), University of California, San Diego, 1976 and 1981, respectively; AT&T Bell Laboratories, 1981—. At AT&T Bell Laboratories, Mr. Tolleth has been planning for effective operation and maintenance of minicomputer-based Operations Systems for the Bell Operating Companies and AT&T Communications.

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Build—A Software Construction Tool

By V. B. ERICKSON* and J. F. PELLEGRIN*

(Manuscript received August 19, 1983)

The `build` tool is used as a sophisticated method of generating and modifying software systems. `Build` is being used successfully by a number of *UNIX*™ software-based projects at AT&T Bell Laboratories. `Build` is an extension to the `make` program that permits several software developers to independently make a collection of software while sharing the same fully populated set of directories, with the changed files residing in their own directories. An important concept in using `build` is *software view*, which represents the selection of a particular version of software for a generation environment. For example, a developer's view of a software system generally includes all of the current "official" software perturbed by the developer's private modifications to the system. A testing team's view may be the current official software perturbed by changes that a set of developers have made and have submitted for project system testing. A system user's view is a fully tested and released version of the software. The function of `build` is to simplify the administration of the different views of the software system. The `build` tool is being used by a number of large software development projects as the primary software generation tool. `Build` plays a central role in the development strategies and standards used in these projects.

I. INTRODUCTION

The `make` tool that is available with the *UNIX*[†] operating system is used as an aid in the construction of software. A specification file, called a *makefile*, contains a description of the software targets that

* AT&T Bell Laboratories.

† Trademark of AT&T Bell Laboratories.

are to be built, a list of files that are needed to construct each specific target (the target's dependency list), and the commands to be executed to create the target. A set of directory structures populated with source files and cooperating makefiles may be used for the construction of a larger collection of software, or for the propagation of changes to the software. The `build` tool is an extension to `make` that permits several developers to independently make changes to a collection of software while sharing the same fully populated set of directories, with the changed files residing in their own directories. Through its use, support for the testing of individual developer versions of the system can be provided without the overhead of redundant storage consumption.

A basic need in a project of two or more people is to provide an environment in which they can work independently on modifications to an existing system of software. If several people wish to test their private modifications to the same portion of the system, independent testing can be achieved by replicating the complete set of populated directories for each developer and permitting them to make changes to their own copy. This is workable in a small project, but the space consumption can rapidly become prohibitive. The desire to share a single copy of the complete software among many developers, while still providing for individual developer changes and testing, was the motivation for the `build` tool. The remaining sections in this paper discuss some basic concepts used in `build`, describe its behavior in detail, provide a simple example, and describe the use of `build` in a particular large software development project.

II. BASIC CONCEPTS

2.1 The make tool

The `make` tool automates the software generation steps that come between the editing and testing phases by executing specified commands to reprocess any and all files that have been affected by the editing.[1] This eliminates manual effort required to reprocess files, potential errors caused by forgetting to reprocess files, and overhead of unnecessarily reprocessing files.

The `make` tool executes commands to generate target files as specified by the contents of a makefile description file. A makefile contains a representation of the graph of file dependencies. Each nonleaf file in the dependency graph is a target file, which may have an associated set of regeneration commands; its descendants are the files from which the target file was created. If any of the target file's descendants are modified, the target file must be regenerated to maintain a consistent system.

For example, Fig. 1 depicts a populated *UNIX* operating system

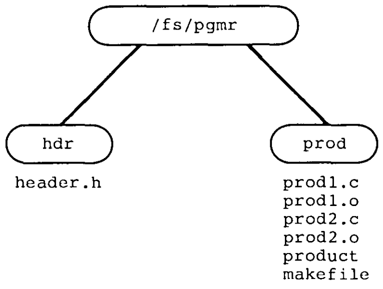


Fig. 1—Fully populated node.

directory structure, including all files necessary to generate the load module `/fs/pgmr/prod/product`. Figure 2 shows the graph of file dependencies for the load module, with file generation commands shown in brackets. The corresponding makefile is shown in Fig. 3. The file names to the left of the colons are the target files, and correspond to the nonleaf files in the graph. The files to the right of the colons are the files that the targets depend on, corresponding to the nonroot files in the graph. The `make` tool examines each file in the graph, checking to see which target files need (re)generating. If a target file does not exist or is dependent on a file that has a later modification date, then the target is (re)generated by executing the generation commands associated with it.

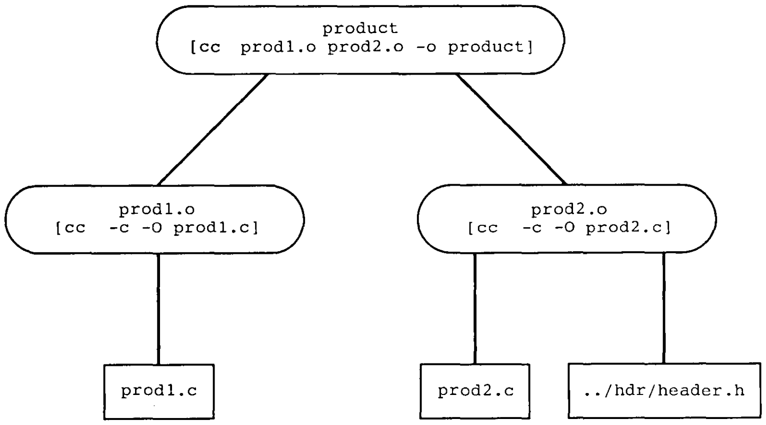


Fig. 2—File dependency.

In the above example, if a developer modified the file `../hdr/header.h` and then executed `make` from within the `/fs/pgmr/prod` directory, `make` would execute the following commands:

```
cc -c -O prod2.c

cc prod1.o prod 2.o -o product
```

The modifications to the file `../hdr/header.h` cause the target, `prod2.o`, to be regenerated. The resulting execution of `cc` causes the file, `prod2.o`, to be updated. This in turn forces the target `product` to be rebuilt.

2.2 Individual software views

Individual software views are versions of the software system that are unique to an individual developer or to a particular set of developers, such as system testers. Different views are formed by combining different collections of files from the system. These collections are stored in separate nodes. A *node* is a set of *UNIX* operating system directories, all of which share the same common ancestor directory, called the *root* directory of the node. A node for a software project is defined as a project-standard set of directories that are sufficient to contain the complete set of project files. Figure 1 is an example of a node, which is populated with all of the files in the system. The root of the node is the directory `/fs/pgmr`. Any file that can be reached from `/fs/pgmr` is contained in the node. A file in a node is identified by the relative file name describing the path from the root of the node to the file. In Fig. 1, the relative file name `hdr/header.h` identifies the file `/fs/pgmr/hdr/header.h`. Multiple instances of a node may be established by duplicating the same set of directories, each below a different root directory. Each instance may contain versions of some or all of the project files within the directories.

Individual software views for developers are established by having each developer work in a separate node containing only those files that the developer needs to change. The developer then accesses all other project files through a separate project-wide shared node that contains a complete set of all the project files.

This combining of individual and shared files in separate nodes to express a particular software view is achieved through the specification of a viewpath. A *viewpath* is an ordered list of nodes, each of which has the same directory structure. The viewpath is used to resolve

```
product:  prod1.o  prod2.o
     cc  prod1.o prod2.o -o product

prod1.o:  prod1.c
     cc  -c -O prod1.c

prod2.o:  prod2.c  ../hdr/header.h
     cc  -c -O prod2.c
```

Fig. 3—Makefile contents.

references to files. A file, identified by its relative path name within the node's directory structure, is located within the viewpath by searching in its directory within each successive node in the viewpath until it is located. Any additional versions of the file in subsequent nodes in the viewpath are ignored. In this way, the viewpath determines which version of each file in the software system is to be used in a particular software view that consists of a set of populated nodes.

To specify a viewpath, which is needed by `build`, users define the viewpath using an environment variable called `VPATH` or an option on the `build` command line. The viewpath specification consists of a list of directories, representing nodes, separated by colons.
In Fig. 4, three nodes are depicted:

`/fs/project`—complete project-released software node

`/fs/pgmr1`—private node of developer named programmer 1

`/fs/pgmr2`—private node of developer named programmmer 2.

Two different viewpaths for separate developers are indicated by

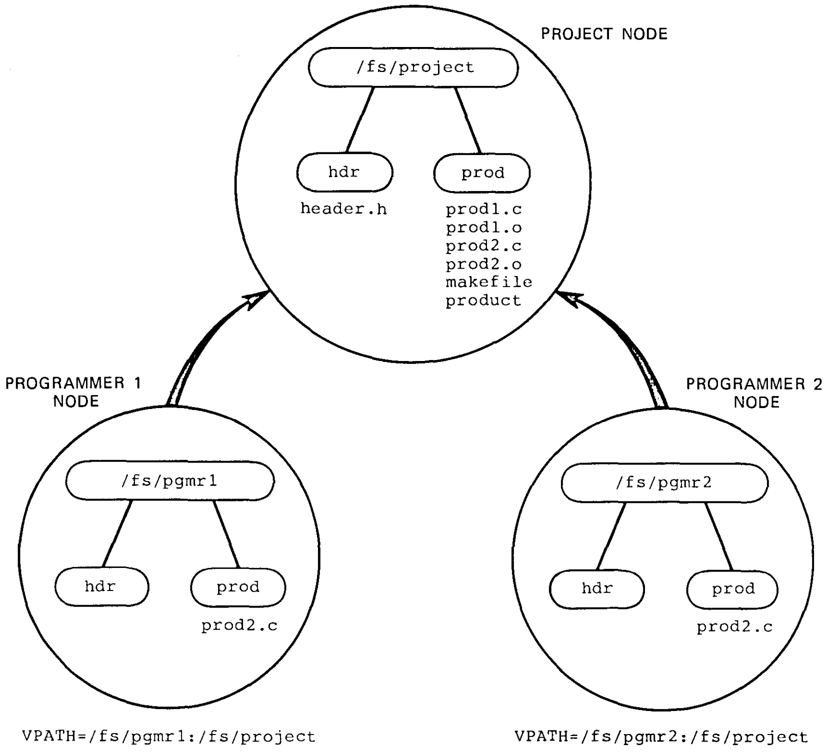


Fig. 4—Independent software views.

arrows. /fs/project contains all the files (source, objects, and intermediate objects) necessary to generate product. The other viewpath, /fs/pgmr1:/fs/project, is the viewpath for a developer named programmer 1, who wishes to generate a software view that includes personal changes in the node /fs/pgmr1 in addition to the released software in the shared node /fs/project. The only connection between the nodes is the logical one defined by the viewpath specification. Similarly, programmer 2 keeps personal changes in the node /fs/pgmr2 and includes the project node in a different viewpath specification. This illustrates how viewpaths express individual software views for two developers. Both programmer 1 and programmer 2 may have private versions of files in their respective nodes and make changes independently of each other. Each developer will have a copy of the files that they wish to change, and the file(s) will have the same relative file name as in the /fs/project node but with a different full path name. The developer's version of the file will, in effect, overlay the instance of that file in the project node. Note that one developer's changes are invisible to all other developers since the changes appear only in that developer's private node.

III. DESCRIPTION OF BUILD

The build tool makes available the capabilities of make within the context of a software view. To use build, the desired viewpath must be declared, either on the command line or using the VPATH environment variable. Build must be invoked from a directory within the first node in the viewpath. The build tool uses the viewpath to resolve all relative file references, both to the description file itself and to the target and dependency references within the description file. If build does not find a file in the first node, it looks in the same directory in successive nodes in the viewpath until the file is found or the last node in the viewpath is reached. If the viewpath contains only a single node, build resolves all file references relative only to the current directory. Hence, a build with a single node viewpath is equivalent to make.

The make tool rebuilds a target file if the file does not exist or if it is dependent on a file that has a later modification date. The build tool rebuilds a target if either of the above criteria holds, or if it is dependent on a file existing in an earlier node in the viewpath. This additional criterion is necessary for build to ensure that a correct version of the product is produced when a target file is put in a node with a modification date later than the file that is in an earlier node in the viewpath, upon which it depends.

In preparation for rebuilding targets, files that the target depends on and that are not in the first node are temporarily added to the first node. `Build` does this by using the *UNIX* link command, `ln`, if the two nodes involved are in the same *UNIX* file system; or the copy command, `cp`, if they are in different file systems. The generation of the target files depends on these files being made available in the first node. Since these files are determined from the dependency lists in the makefile, the completeness and correctness of makefiles is necessary for `build` to affect changes to the software. If there are errors of omission in makefile dependency lists, they reveal themselves during the building of a target as files that are missing—i.e., not added to the first node. After the targets are built, all files copied or linked into the first node are removed by `build`.

In addition to enabling a number of developers to share a set of files to achieve individual software views, `build` can also be useful to the individual developing a product. A developer often wants to make and test changes to a program without destroying a previous version of the program. The `build` tool allows the developer to conveniently produce a new version without having to save the old one or rewrite the makefile. The developer simply creates a new node, places it in any modified files, sets the viewpath to look first in the new node followed by the original node, and then invokes `build`.

IV. EXAMPLE

We now return to the nodes shown in Fig. 4, and consider how `build`, using the makefile shown in Fig. 3, handles the particular files in the nodes. `/fs/project` contains all the files (source, objects, and intermediate objects) necessary to generate `product`. This means that at one time `build` was used to generate `product` in this node. Since there would have been only one node in the viewpath, using `build` in this situation was identical to using `make`. Assume that the viewpath for programmer 1 is `/fs/pgmr1:/fs/project` as shown in Fig. 4.

If programmer 1 changes the file `prod2.c` and runs `build` from within the directory `/fs/pgmr1/prod`, `build` looks for a description file named `makefile` and finds `/fs/project/prod/makefile`, since there is none in the node `/fs/pgmr1/prod`. As `build` processes the makefile, it determines first that `product` depends upon `prod1.o`, which in turn depends upon `prod1.c`. `Build` locates both:

```
/fs/project/prod/prod1.o

/fs/project/prod/prod1.c
```

In `/fs/project/prod`, `prod1.o` is newer than `prod1.c`. Consequently, no regeneration of `prod1.o` occurs. `Build` then determines that `product` depends upon `prod2.o`, which in turn depends upon `prod2.c` and `../hdr/header.h`. The `build` tool locates:

```
/fs/project/hdr/header.h
/fs/project/prod/prod2.o
/fs/pgmr1/prod/prod2.c
```

Since `/fs/pgmr1/prod/prod2.c` is newer (or is in an earlier node) than `/fs/project/prod/prod2.o`, `prod2.o` needs regeneration. The `build` tool links or copies any files necessary for regeneration into the lowest node in the viewpath. In this case, `/fs/project/hdr/header.h` is linked to `/fs/pgmr1/hdr/header.h`. Now,

```
cc -c -O prod2.c
```

is invoked by `build`, producing `prod2.o` in the developer's private node. After `prod2.o` is generated, `/fs/pgmr1/hdr/header.h` is unlinked. Because `product` depends upon `prod2.o`, which was just generated, `product` is regenerated. Again, any dependents of `product` must be in the developer's node. `/fs/project/prod/prod1.o` must therefore be linked to `/fs/pgmr1/prod/prod1.o` before the following command can be executed:

```
cc prod1.o prod2.o -o product
```

This produces the executable object `product` that is located in `/fs/pgmr1/prod/product`. `Build` then unlinks `/fs/pgmr1/prod/prod1.o`.

A similar scenario occurs for programmer 2, since that developer also has a private version of `prod2.c`. Each developer is thus able to generate a private version of `product` containing only the changes they wish to test. The only files that each developer retains in their private node are `prod2.c`, `prod2.o`, and `product`, as shown in Fig. 5. Other files that they have not changed remain in the project node as shared files.

V. SAMPLE PROJECT USE OF BUILD

In the *UNIX* operating system environment, a *system of software* consists of a collection of source, object, and executable object files placed in some orderly fashion within a collection of directories (*directory structure*). A set of cooperating makefiles are placed throughout the directory structure, each specifying the instructions and the files that are needed to construct a given target file. `Build` is used to construct the targets for the first time from the original source files

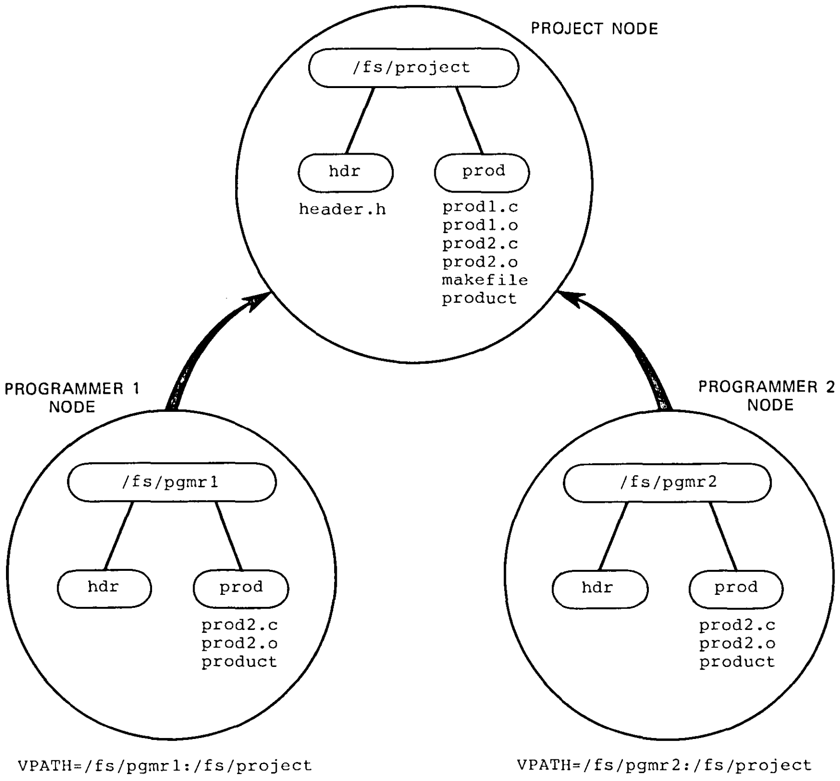


Fig. 5—Contents of nodes after executing `build`.

or to propagate changes made to a subset of the source files within the context of a complete constructed system.

There are a number of projects at AT&T Bell Laboratories that are currently using `build` as the primary construction tool. The AT&T *UNIX* Real-Time Reliability (RTR) project has over 100 developers dealing with more than 8,000 source files, for each of three major versions of the system.[2] The `build` tool is used throughout all phases of the software generation.

1. Unit testing. On each machine assigned to developers on the project, a fully populated, official node of each major version of the system is provided as a stable base for unit testing individual developer changes to the system. Using `VPATH`, developers define their viewpath to be one or more private nodes followed by the official node.

2. Integration testing. At the project level, changes constructed with the official node in the viewpath and generated by many developers are combined by accumulating them in a single test node in order to provide versions of changed products for official testing. In the case

where several developers have made changes to a single source file, the source administration system used in the project provides a source file version containing the combined changes.

3. System release generation. Changes to be combined for an incremental release of one of the system versions are accumulated in a single node (as in integration testing). These are built and system tested with the official node for the previous version of the system in the viewpath before being released.

The `build` tool is the only construction tool used in the *UNIX* RTR project, and has been extremely successful at minimizing multiple copies of files, something that can easily become a serious problem in a project of that size. Independent unit testing by developers is easily accomplished in this environment.

VI. BENEFITS

`Build` reduces the number of multiple copies of files through the use of a globally shared node containing a full set of the software, and supports independent unit testing of changes by individual developers. An additional and important benefit is that when multiple nodes participate in building software, the correctness of dependency lists in the makefiles is enforced, since `build` is otherwise unable to make all necessary files available in the first node of the viewpath for target generation. The completeness of makefiles is similarly tested, since the components necessary to construct a particular product must be made available by `build` to the developer's private node. Confirming the correctness and completeness of the makefile is absolutely necessary for a reliable software construction process.

VII. SUMMARY

`Build` is an extremely useful extension to the *UNIX* operating system `make` tool for constructing and changing software. The `build` tool automates the steps between editing and testing within the framework of individual software views. Individual software views allow a number of developers to easily make and test changes to a set of shared files without interfering with each other. `Build` also provides a check on the completeness and correctness of makefiles used to specify software construction steps. Many projects at AT&T Bell Laboratories are currently using `build` with much success.

REFERENCES

1. S. I. Feldman, "Make—A Program for Maintaining Computer Programs," Software—Practice and Experience, *9*, No. 4 (April 1979), pp. 256-65.
2. B. R. Rowland and R. J. Welsch, "Software Development System," B.S.T.J., *62*, No. 1, Part 2 (January 1983), pp. 275-89.

AUTHORS

Verlyn B. Erickson, B.A. (Mathematics/Physics), 1968, Augustana College; M.S. (Computer Science), 1970, University of Wisconsin, Madison; Mathematics Research Center, University of Wisconsin, 1969–1972; Engineering Computing Laboratory, University of Wisconsin, 1970–1977; AT&T Bell Laboratories, 1977—. Mr. Erickson has worked in various software support areas, including 1*ESS*™ Laboratory Support, AT&T 3B20 DMERT Laboratory Support, and, most recently, the Software Development environment for the 3B Processor line.

John F. Pellegrin, B.A. (Mathematics), 1965, Occidental College; M.A. (Mathematics), 1967, Arizona State University; AC Electronics Defense Research Laboratories, Goleta, CA, 1967–69; Ph.D. (Mathematics), 1972, Arizona State University; AT&T Bell Laboratories, 1972—. Mr. Pellegrin has worked in various development support areas, including hardware logic simulation, *ESS*™ software development support, software development methodology, and, currently, Software Development Systems for the AT&T 3B Processor line.

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Queueing and Framing Disciplines for a Mixture of Data Traffic Types

By A. G. FRASER* and S. P. MORGAN*

(Manuscript received September 22, 1983)

Packet-switched data networks are constructed from switching nodes interconnected by trunks. Trunk queueing delays for short messages can be reduced at the expense of long messages by having the trunk server take no more than a fixed number of bytes from each message before going on to the next message. We report analysis and simulations of two partial-service disciplines, namely Round Robin (RR) and Priority first-in first-out followed by Round Robin (PR + RR), for a mixture of traffic types. The PR + RR discipline permits short messages to experience finite mean delay at traffic levels where longer messages see infinite mean delay. Information is transmitted over the trunk in frames, where a frame may contain parts of several messages. At the far end of the trunk, the contents of a frame are not transmitted further until the end of the frame has arrived. We simulate two framing algorithms that work effectively with the PR + RR queueing discipline to achieve acceptably low frame overhead together with short delays for short messages. In addition, queueing plus framing delays for longer messages are substantially reduced, at a given overall traffic intensity if the access lines run more slowly than the trunk.

I. INTRODUCTION

Packet-switched data networks are constructed from switching nodes interconnected by trunks. For long-distance communication, 56-kb/s trunks are used. Traffic enters the network from computers

* AT&T Bell Laboratories.

and terminals that are connected to the nearest node by access lines operating at speeds ranging from 75 b/s to 1 Mb/s or more.

In a network based on the *DATAKIT** Virtual Circuit Switch,[1] traffic flows from several message sources through a node and out over a trunk line, as shown in Fig. 1. Messages generated by each source are transmitted over the access line character by character at the speed of the line. At the node the data are stored in one or more first-in first-out queues from which they are eventually taken for transmission over the trunk. The data from each source are transmitted on a different logical channel of the trunk. However, physical transmission over the trunk takes place in frames, where each frame may contain data from several sources in addition to framing information.

It is interesting to know the delay performance of the network under various operating conditions and with different choices of queue service discipline and framing discipline. Low delay is particularly critical for messages from interactive terminals and for the various control messages used in high-level protocols. Fortunately, messages that require low delay are usually only a few characters long, and we can use a queue service discipline that gives preference to short bursts of transmission. An advantage of using such a discipline is that it allows a trunk design that is appropriate regardless of user protocol.

Queueing delays for short messages generally can be reduced at the expense of long messages by using a separate first-in first-out queue for each channel, and by having the trunk server take no more than a fixed number of bytes, which we call a packet, from each queue before going on to the next queue. A short message following a long message in the same channel is necessarily delayed if we wish to preserve sequence, but this is not expected to create difficulties in practice. Two partial-service disciplines, shown in Fig. 2, are studied in this paper:

1. Round Robin (RR)—When a channel has data to transmit, the channel number is added to the end of a single list of channel numbers and works its way to the front, at which time one packet is transmitted from the given channel. If this does not empty the per-channel queue, the channel number is returned to the end of the list.

2. Priority first-in first-out followed by Round Robin (PR + RR)—This is a two-list discipline in which arriving channel numbers join a priority first-in first-out list, and then join a round robin if they require more than one packet of service. After transmitting each packet, the server checks to see if there are any channel numbers waiting in the priority list, and if so it attends to them first.

Successive packets are assembled by the trunk server into frames,

* Trademark of AT&T.

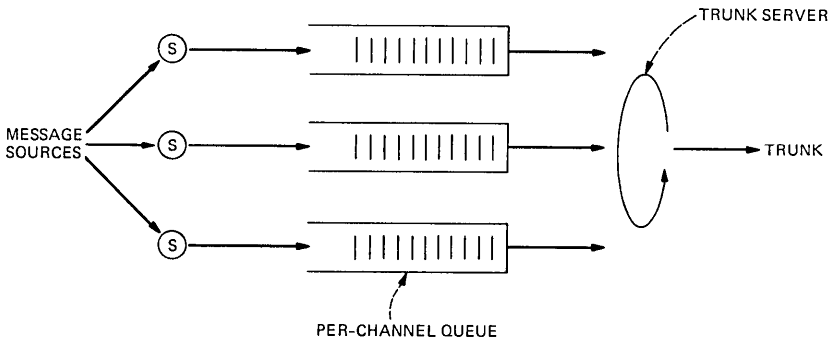


Fig. 1—Message queues at a trunk node.

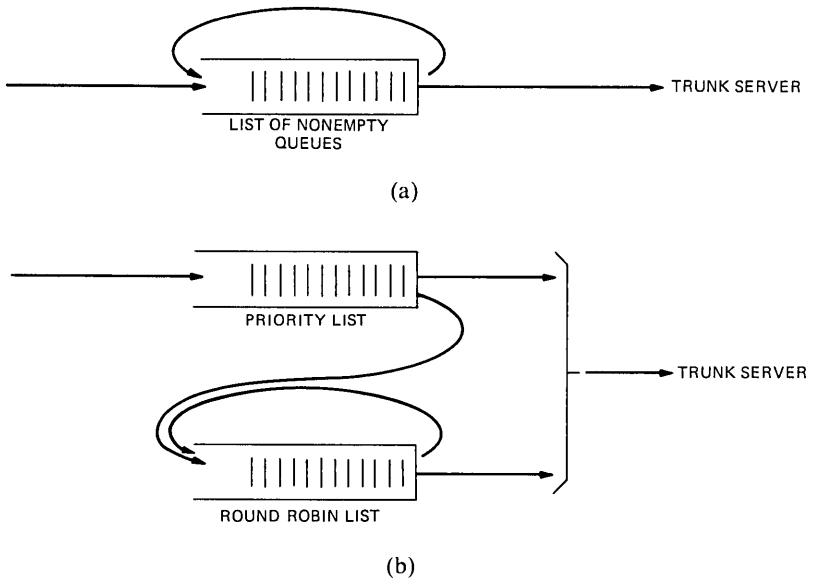


Fig. 2—Queueing disciplines. (a) Round robin. (b) Priority first-in first-out followed by round robin.

where a frame may contain data from several different channels. Each frame also includes a fixed number of framing bytes. A frame is terminated when it reaches a preset length, or when there are for the time being no data to send. At the far end of the trunk, the contents of the frame are held in a buffer until the end of the frame has arrived and the check bits have been verified. Optimal choice of frame length requires balancing the increased trunk overhead that is associated with short frames against the increased delay for short messages that is associated with waiting for the arrival of the end of long frames.

In this paper we are concerned only with the queueing plus framing delay at a trunk node. We do not include the times required to transmit the message over the access line and over the trunk; these partially concurrent times also contribute, of course, to the end-to-end delay of the network. Furthermore, we assume for simplicity that the access lines do not run faster than the trunk. The queueing plus framing delay is defined as the time that elapses between the arrival of the last character of the message in the per-channel queue (Fig. 1), and the time when the end of the frame containing the last character disappears into the trunk. In the limit of vanishing utilization and the absence of any packet or frame overhead, the delay so defined would be zero. A slightly more complicated definition of delay is needed if the access lines run faster than the trunk.

An analytical recipe for computing mean queueing delays for a mixture of message types, when the access lines run at the same speed as the trunk, has been given by Wolff[2] for the RR discipline. A similar recipe is given for PR + RR in the Appendix of the present paper. However, the analytical approach gives only mean delays and not the distribution of delay. Furthermore, it does not include the effects of framing, and it is not applicable to the practically important case where the access lines run more slowly than the trunk. Accordingly, the analysis has been augmented by extensive simulations.

In the numerical examples, we assume that the input traffic is a mixture of three types of messages, chosen to approximate single-character terminal-to-host transmissions from asynchronous terminals (10 percent), time-sharing host-to-terminal responses (40 percent), and host-to-host file transfers (50 percent). The assumed message characteristics are shown in Table I. On a 56-kb/s trunk, one character time is 0.143 ms.

In Section II we calculate mean trunk queueing delays, in the absence of framing, as a function of load when access speed is equal to trunk speed, for each message type. Most of the calculations are for a single packet size and overhead, since the qualitative effects of different packet sizes and overheads are reasonably easy to foresee. Three queueing disciplines are compared, namely First-In First-Out (FIFO), RR, and PR + RR.

Table I—Assumed traffic characteristics

Message Type	Length Distribution	Mean Length (bytes)	Relative Arrival Rate	Traffic Fraction
1	Constant	1	1	0.099
2	Exponential	40	1/10	0.395
3	Exponential	512	1/100	0.506

In ordinary FIFO or message-at-a-time service, where entire messages are transmitted in order of arrival, all messages see the same mean first-character waiting time regardless of length. The mean last-character delay is greater for longer messages if there is nonzero packet overhead. Disciplines that break messages up into smaller parts typically treat short messages better than long messages. In the case of PR + RR, sufficiently short messages are served entirely from the PR list, and they can experience finite mean delay at traffic levels where the RR is saturated by longer messages. These results appear both from the analytic solutions and from the simulations.

In Section III we simulate combined queueing and framing delay as a function of load, assuming the PR + RR discipline with access speed equal to line speed, and the traffic mix of Table I. For PR + RR, it is possible to impose a shorter average length on frames that contain data from the PR list than on frames that do not. We simulate two framing algorithms that accomplish this in slightly different ways, and we determine mean and 95th-percentile queueing plus framing delays for each message type.

Finally, in Section IV we consider the effect of low-speed access lines on queueing plus framing delays. For a given total load on the trunk, low-speed access lines smooth out the incoming traffic flow and reduce the mean last-character delay for long messages in the per-channel queue. We obtain mean queueing plus framing delays for each message type by simulating the PR + RR queueing discipline with one of the above-mentioned framing algorithms, for a nominal trunk utilization of 60 percent excluding overhead, and for various access speeds assuming that all access lines run at the same speed.

We conclude in Section V that the PR + RR queueing discipline, followed by either of the two framing disciplines, subjects single-character messages at a 56-kb/s trunk node to mean queueing plus framing delays of less than 20 milliseconds, together with acceptably low frame overhead, even when the trunk is heavily loaded with longer messages; and low-speed access lines reduce substantially the mean queueing plus framing delays for longer messages by smoothing out the access traffic. The numerical results are for a particular line speed and a specific traffic model, since it is notoriously difficult to explore a multidimensional parameter space by simulation; but the qualitative conclusions appear to be of much broader applicability.

II. QUEUEING DELAY

In this section we consider pure queueing delay on an unframed trunk, and assume that the access lines run at the same speed as the trunk. Then each message may be regarded as ready for transmission in its entirety as soon as its first character reaches the per-channel queue. If the merged arrival process is Poisson, and if each message arrives in a separate channel so that it is not delayed behind another message in the same per-channel queue, then the messages constitute an M/G/1 queue of "customers" subject to whatever discipline the trunk server imposes.

Under FIFO service, the mean first-character waiting time is independent of message length and is given by the well-known formula

$$W = \frac{\lambda E(S^2)}{2(1 - \rho)},$$

where λ is the merged arrival rate, $E(S^2)$ is the second moment of message length in the merged message stream, and ρ is the effective traffic intensity (ratio of mean data plus overhead rate to raw trunk speed). The effect of per-packet overhead, as shown in the Appendix, is to increase ρ and $E(S^2)$ above the nominal values that are associated with the incoming message stream.

Figure 3 shows the mean first-character waiting time for message-at-a-time service on a 56-kb/s trunk, when the traffic mix of Table I arrives on 56-kb/s access lines. Mean waiting time in milliseconds is plotted against nominal utilization ρ', where ρ' is the ratio of mean user data rate to raw trunk speed. The three curves correspond, respectively, to 64-byte packets with no overhead (in which case the packet size is irrelevant), to packets with 64 bytes of data and 2 bytes of overhead, and to packets with 16 bytes of data and 2 bytes of overhead.

The curves of Fig. 3 have the form constant/$(1 - \rho'/\rho_0)$, where ρ_0 is the nominal utilization at which the sum of mean user data rate plus overhead rate is equal to the raw trunk speed. ρ_0 depends, of course, on the traffic mix and on the ratio of overhead bytes to data bytes in a packet. In the absence of overhead, $\rho_0 = 1$. For (64 + 2)-byte packets and the assumed traffic mix, $\rho_0 = 0.807$; and for (16 + 2)-byte packets, $\rho_0 = 0.757$. In general, ρ_0 will be substantially less than the ratio of data bytes to total bytes in a full packet, because if there are many short messages, there will be many packets with fewer than the maximum number of data bytes. Regardless of the number of data bytes, each packet contains a fixed number of overhead bytes.

Message-at-a-time service subjects long and short messages to the same mean first-character waiting time. Mean last-character delays depend on message length in the presence of per-packet overhead, since longer messages must wait for more overhead bytes to be transmitted; but the maximum difference is 9.3 ms in the numerical examples of Fig. 3, according to Table II of the appendix.

Rudin[3] has pointed out that breaking messages up into packets on the access lines, and then transmitting the access packets out of a

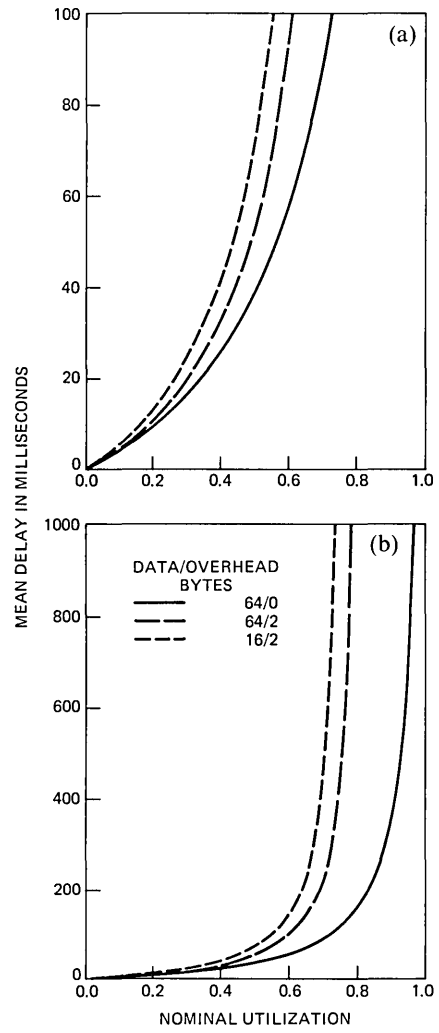


Fig. 3—Mean first-character waiting time for message-at-a-time service. (a) Light utilization. (b) Heavy utilization.

single FIFO, favors short messages to some extent; but in the present case it does not permit single-character messages to go in a few milliseconds as they should, while delaying file transfers for several seconds if necessary. To obtain the desired orders-of-magnitude discrimination, one must apparently resort to some such discipline as RR or PR + RR.

Wolff[2] has derived an infinite system of linear equations whose solution yields the mean delay for each message type in an M/G/1 queue with RR service. A similar system of equations is derived in the appendix for an M/G/1 queue with PR + RR service. In practical cases an approximate numerical solution may be obtained by truncating the infinite system to a finite system. In the unrealistic case of zero overhead, it is easy to solve the limiting case of infinitesimal packet size ("processor sharing"). Unfortunately this limit is of little interest in the present context, because data packets are not indefinitely divisible, and even if they were, the delays would go to infinity as the information per packet went to zero if the overhead per packet were a fixed nonzero amount.

Theoretical last-character mean delays are shown in Fig. 4 (dashed curves) for each message type under RR and PR + RR service, assuming (64 + 2)-byte packets and the traffic mix of Table I. As we expected, RR treats short messages better and long messages worse than ordinary FIFO, but the delays for all message types eventually saturate at the same point. PR + RR, on the other hand, pushes the delay curves farther apart; and the single-character delay does not go to infinity at the same point as the delay for longer messages. Only the RR saturates at this point, and single-character messages never see the RR.

Figure 5 shows the theoretical mean delays (dashed curves) for PR + RR with (16 + 2)-byte packets. Putting fewer data bytes in a packet with fixed overhead increases the difference in behavior of long and short messages and decreases the effective capacity of the trunk, because the average number of overhead bytes per message is increased. More generally, with the PR + RR discipline one could take different packet sizes for channels in different lists. Exploratory calculations of mean delays have not so far revealed any particular advantages to doing so, at least for the present traffic mix.

To deal with more complicated issues in the trunking of mixed data traffic, it appears necessary to simulate the desired disciplines. A simulator was therefore written to apply a variety of queueing and framing algorithms to a traffic model somewhat different from the M/G/1 queue assumed in the theoretical analysis.

The input to the trunk queueing simulator consists of a specification of one or more classes of access lines, in which n_i lines of class i each carry independent and identically distributed (i.i.d.) messages of mean length l_i separated by i.i.d. gaps of mean length g_i at a line speed s_i. The assumed configuration is like that of Fig. 1, so that under heavy loads a short message has some probability of being delayed behind a long message in the same channel. The message lengths may be deterministic (constant) or exponentially distributed; gap lengths are exponentially distributed. A number of other parameters such as trunk speed, packet sizes, overhead, queueing and framing discipline, and

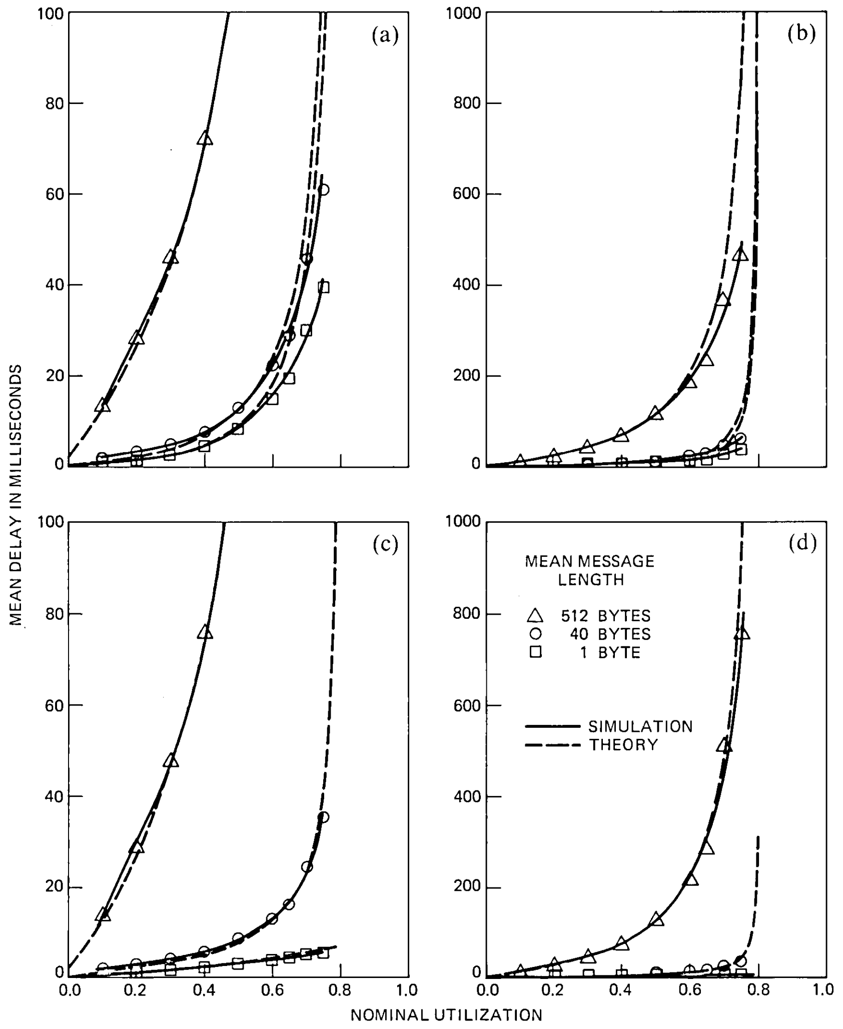


Fig. 4—Mean queueing delays for partial service disciplines. (a) Round robin low utilization. (b) Round robin high utilization. (c) Low utilization for priority first-in first-out followed by round robin. (d) High utilization for priority first-in first-out followed by round robin.

simulation time are set by flags or default options. The simulator starts with no messages in the system, and stops at the first regeneration (= empty and idle) point[4] after the specified simulation time has elapsed. Simulation outputs used in the present study are the mean delay for messages ot type *i*, and the 95th-percentile delay for messages of type *i*. The simulator can produce histograms of delay distributions for messages of each type, but we have not made use of the histograms.

A simulation is a probabilistic experiment. In order to obtain con-

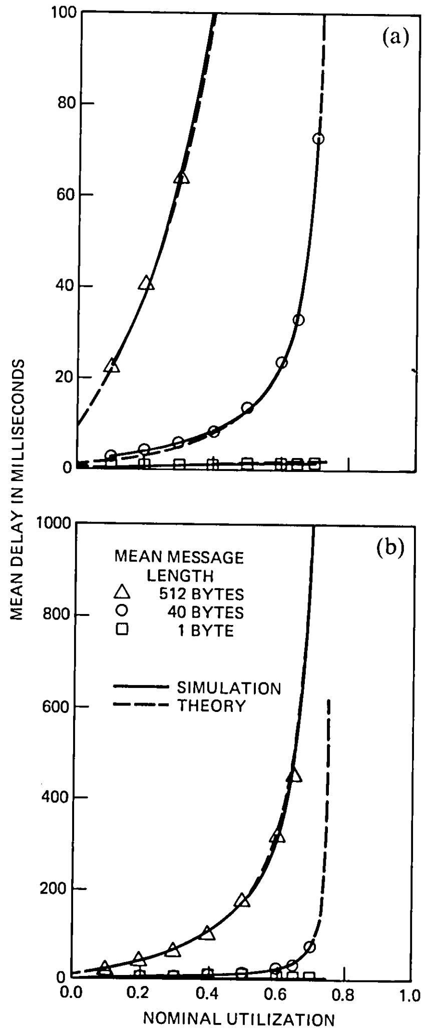


Fig. 5—Mean queueing delays for priority first-in first-out followed by round robin service with (16 + 2)-byte packets. (a) Low utilization. (b) High utilization.

fidence intervals, it is useful to divide the simulation time into a number of shorter simulations and look at the scatter of the results. We use the fact that the mean delay for a particular message type, averaged over a sufficiently large number of regeneration epochs, is asymptotically normally distributed.[5] Similarly, any particular quantile, such as the 95th percentile, of an arbitrary distribution is asymptotically normally distributed. Assuming that the mean delays from n simulations under "identical" conditions (except for the initial seeds of the random number generator) are normally distributed, we form from the n sample means and their variance a variable having a t distribution with $n - 1$ degrees of freedom. The mean of sample means is taken as the estimate of the population mean, and a confidence interval is constructed from the t distribution. An estimate and a confidence interval for the 95th-percentile delay are calculated in a similar way.

In the simulations we more or less arbitrarily assumed 100 access lines for each of the message types of Table I (300 access lines altogether), and we adjusted the gap lengths to achieve the desired ratios of arrival rates and the desired nominal trunk utilization. In the longest runs, simulation times were chosen so that approximately 4000 type 3 messages would arrive (that is, approximately 40 messages on each type 3 access line) during the course of a single run. Twenty simulations were done for each set of parameter values, and the mean of the 20 sample means was taken as the final estimate of the mean delay (similarly for the 95th-percentile delay). The 90-percent confidence interval for this estimate was constructed from the variance of the sample of 20 means.

Not surprisingly, the delays for type 3 messages have the greatest uncertainty, with 90-percent confidence intervals as wide as ± 5 percent (total width 10 percent) of the estimated value when access speed is equal to trunk speed and the trunk utilization is high. Furthermore, the absolute width of the confidence interval seems to be more or less independent of access line speed, so the relative uncertainty is greater for low-speed access lines.

To put matters in perspective, a single simulation run that includes about 4000 type 3 message arrivals takes about an hour of time on a large minicomputer, so the 20 runs necessary to get one point on the attached curves take about 20 hours and delineating the shape of a curve with 10 points may require 200 hours. Such a computation actually produces six curves, including both the mean and the 95th-percentile delays for all three traffic types, but it is still a substantial undertaking.

Why does it take so long? Basically, because the mean delay is proportional to the second moment of message length. In our case, almost the entire contribution to the second moment comes from type 3 messages, which constitute less than 1 percent of the total number of messages. We have to simulate long enough to see a substantial number of type 3 messages, while also doing all the bookkeeping for types 1 and 2 messages. As a check on our results, we have looked at published formulas[4] for confidence intervals in simulations of the mean delay in an M/G/1 queue, and have found that the predicted simulation times to achieve specified confidence intervals are quite comparable to the simulation times required in the present study.

Variance-reduction techniques, such as the method of control variates,[5] should probably be investigated if further simulations of this type are done.

How should the uncertainty in the results of the simulations be represented? One way would be to draw broken-line curves between the simulated points, and to plot the 90-percent confidence interval at each point. A more esthetic, if less scientific, visual impression is obtained by plotting the simulated points and using least squares to fit splines to the simulated points and to the upper and lower endpoints of the confidence intervals. (In cases where a function is expected to have a pole, such as queueing delays do at $\rho' = \rho_0$, the pole is taken out and least-squares fitting is applied to the numerator.) It must be emphasized that the upper and lower curves resulting from this procedure are *not* curves between which the mean delay has been proved to lie with 90-percent probability; they are only a qualitative indication of the uncertainty in the mean delay. "Bounds" are not drawn in most of the figures because they would essentially coincide with the mean value.

Results of simulating mean queueing delays, as a function of nominal utilization, for the RR and PR + RR disciplines on a 56-kb/s trunk with 56-kb/s access lines, are plotted as solid curves in Figs. 4 and 5. For those delays that saturate as a function of load, the pole is assumed to be at the same place as for message-at-a-time service.

Agreement between theory and simulation in Figs. 4 and 5 is good for light and moderate loads. For high loads, the simulated delays are less than the theoretical delays, especially for the RR discipline. This is understood qualitatively as follows: In the theory, the merged arrival process is assumed to be Poisson. In the simulations, the interarrival intervals on a single access line, being sums of an exponentially distributed or deterministic message length and an exponentially distributed gap, are not themselves exponential. The per-line arrival process is less bursty (fewer short intervals) than a Poisson process and would therefore be expected to lead to smaller queueing delays. The Palm-Khintchine theorem[6] states that the superposition of a sufficiently large number of arbitrary arrival processes approaches a Poisson process. However, Albin[7] has shown that under heavy traffic loads the Poisson limit may be approached very slowly. Some simulations with the same overall traffic divided among 600 access lines gave results closer to the Poisson values and suggested that we have encountered such an effect here.

III. FRAMING DELAY

The effect of framing on trunk delay is twofold. Framing overhead reduces the effective speed of the trunk for data transmission; and in addition, at the far end of the trunk the last byte of a given message must wait to be transmitted further until the entire frame has arrived and the check bits have been verified. If there are a constant number of framing bytes per frame, the overhead effect is reduced by long frames and the waiting effect is reduced by short frames. For a given message length and a given trunk utilization, there is evidently a frame length that minimizes the combined queueing plus framing delay. The combined delay for long messages is minimized by long frames, and the combined delay for short messages is minimized by short frames. It would be desirable, therefore, to shorten just those frames that contain short messages.

We cannot recognize short messages *per se*, but we can achieve somewhat the same effect by shortening the frames that include data from the PR list. Two algorithms that accomplish this have been simulated.

3.1 Mixed frames

In this algorithm, complete packets are transmitted until the number of transmitted bytes exceeds a preset maximum. The framing bytes are then added and the frame is closed. The current frame is also closed if there are no packets ready to send. Different maximum frame lengths are imposed depending on whether the frame contains any data from the PR list. Thus a typical frame contains a mixture of PR and RR packets, but, on the average, frames that contain some PR packets are shorter than frames that contain only RR packets.

3.2 Sorted frames

In this algorithm, a flag is set whenever a frame contains any PR packets, and no further RR packets are added to such a frame. Thus the general form of a frame is a series of packets from the RR list followed by a series of packets from the PR list. In addition, maximum frame lengths are imposed, which may differ for frames that contain some PR packets than for frames that contain only RR packets.

Figure 6 shows the mean queueing plus framing delays for the PR + RR discipline with (64 + 2)-byte packets, as obtained by simulating each of the framing algorithms just described, and Fig. 7 shows the 95th-percentile delays. For mixed frames a nominal maximum of 64 bytes is imposed on frames containing any packets from the PR list, and 256 bytes on frames containing only packets from the RR list. Note that the actual maximum frame lengths will be greater than the nominal values because the last packet is not divided, and 5 framing bytes are added at the end. For sorted frames a nominal maximum of 256 bytes is imposed on both kinds of frame.

To see the effect of frame delay, one may compare Fig. 6 with the

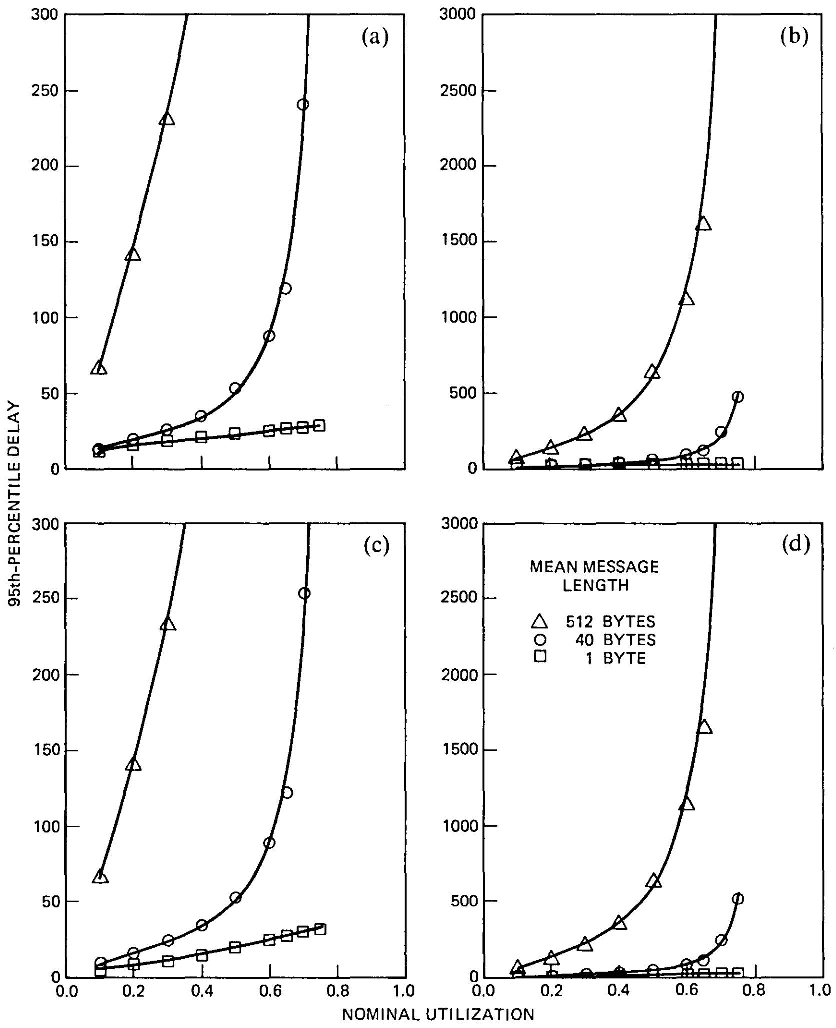


Fig. 6—Mean queueing plus framing delays. (a) Low utilization for mixed frames. (b) High utilization for mixed frames. (c) Low utilization for sorted frames. (d) High utilization for sorted frames.

PR + RR plots in Fig. 4, which show mean queueing delay separately. The addition of frame delay increases the total delay, most noticeably for single-character messages; but single-character messages still do not saturate at the same point as longer messages. (We do not have a theoretical value for the vertical asymptote in the presence of frame delay. Arbitrarily placing the pole at $\rho_0 = 0.80$ for the plots of Figs. 6 and 7 led to attractive curves using least-squares smoothing of the numerator.) The "sorted" algorithm with 256/256 limits looks about

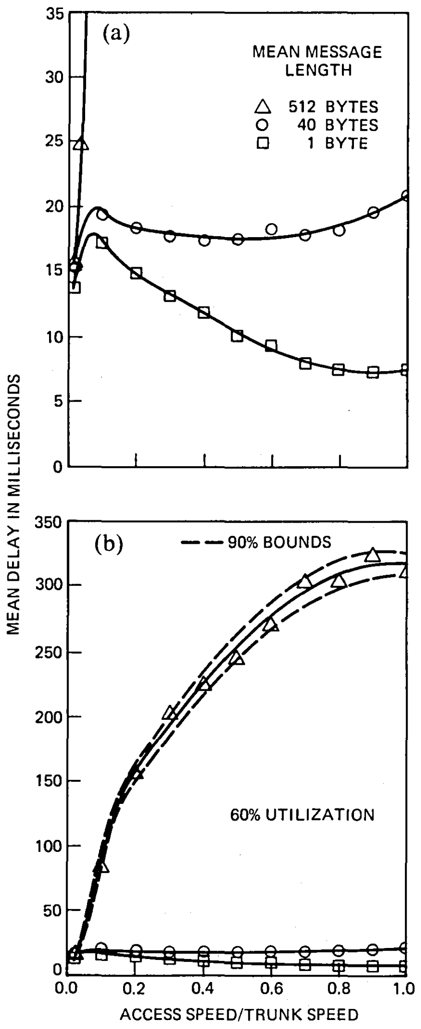


Fig. 7—95th-percentile queueing plus framing delays. (a) Low utilization for mixed frames. (b) High utilization for mixed frames. (c) Low utilization for sorted frames. (d) High utilization for sorted frames.

the same as the "mixed" algorithm with 64/256 limits for the present traffic mix. Conceptually, the sorted algorithm may be a little simpler.

IV. LOW-SPEED ACCESS LINES

Up to now we have assumed that access lines run at the same speed as the trunk. In practice, however, access lines will often be slower than the trunk, and for a given total load on the trunk the effect of low-speed access lines will be to smooth out the traffic flow and substantially reduce the delay in the per-channel queue of the last character of a typical message. However, the end-to-end message delay generally will be increased, since the access-line transmission time is increased at the same time that the trunk queueing delay is decreased. In a sense the access line itself is being used as a buffer for the trunk queueing module. Rudin[3] and Anick, Mitra, and Sondhi[8] have considered the reduction in buffering requirements permitted by low-speed access lines.

Figure 8 shows simulated mean queueing plus framing delays for the PR + RR discipline with sorted frames, assuming the traffic mix of Table I, for various access speeds when all the access lines run at the same speed. A nominal trunk utilization of 60 percent was assumed together with (64 + 2)-byte frames. Cubic splines were fitted to the simulated points by least squares, and "90-percent bounds," as discussed in Section II, are shown for the type 3 delays.

Figure 8 shows that the delay for long messages falls off with decreasing access line speed. It falls off faster with decreasing access line speed for utilizations less than 60 percent, and more slowly for utilizations greater than 60 percent. As Rudin has pointed out, the curve would be expected to have a knee near the point where the mean time for transmission of a message over the low-speed access line is equal to the mean queueing time that would exist at the trunk for full-speed access lines.[3]

At the same time, the delay for short messages rises, up to a point, with decreasing access line speed. The reason for this is that if the access lines run much more slowly than the trunk, the trunk server generally empties the per-channel queue on each pass and, not recognizing that more of the same message is coming, allows the channel number to reappear in the PR list when it does not deserve to do so. Thus, nearly everything gets served from the PR list and distinctions between long and short messages are eroded.

One could discourage the same message from reappearing in the PR list so often by making each channnel number pass through the RR list after every service. The trunk server would delete the channel number from the RR list if there were nothing in the per-channel queue the next time the server got around to looking at it; and only then would the channel number be eligible to reappear in the PR list. This "PR + RR with hysteresis" discipline would treat single-character messages somewhat better at low access speeds, but it would also cause all three message types to saturate at the same load, like ordinary RR. The choice between PR + RR and PR + RR with hysteresis may ultimately depend on more detailed knowledge of the traffic to which the trunk nodes will be subjected.

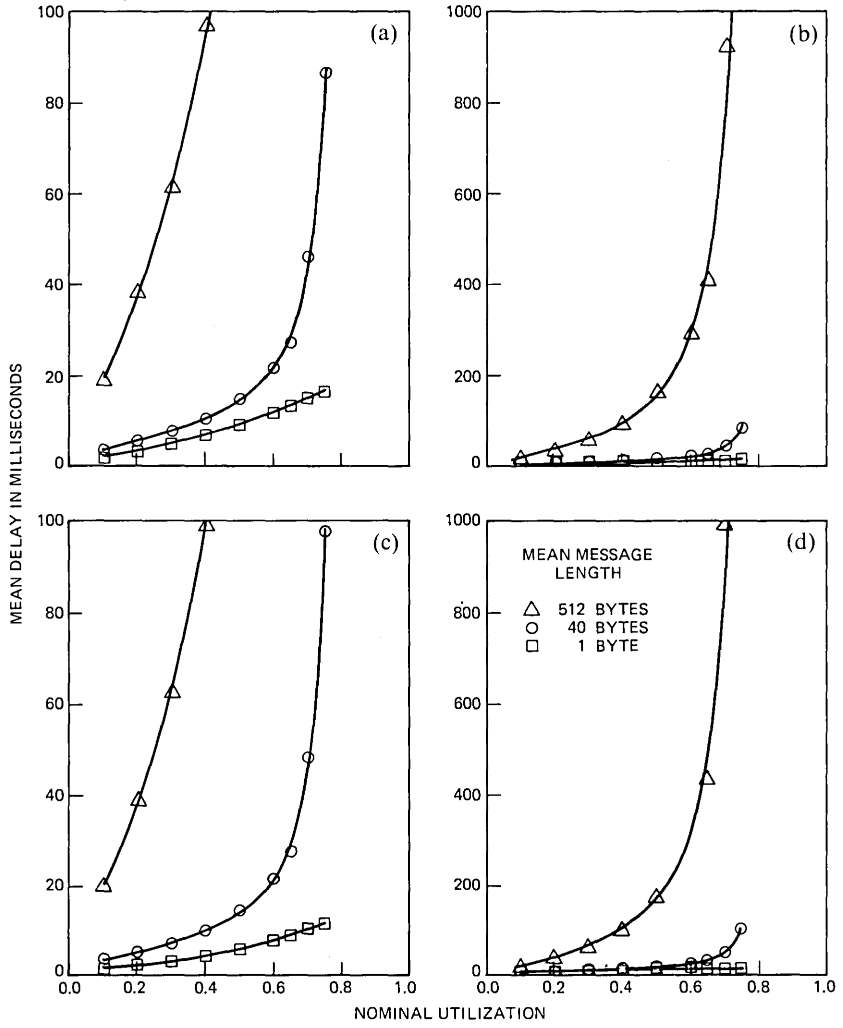


Fig. 8—Mean queueing plus framing delays for different access speeds. (a) Low utilization for sorted frames. (b) High utilization for sorted frames.

V. CONCLUSIONS

Some kind of partial-service discipline is essential if we want to transmit and receive single-character messages expeditiously in the presence of longer messages. The PR + RR discipline is particularly attractive because the RR list saturates before the PR list, so that single-character messages can still experience finite mean delay while longer messages see infinite mean delay.

Acceptable framing delays for both long and short messages can be achieved by imposing different maximum lengths on frames that do and do not contain items from the PR list. For the assumed traffic mix, the PR + RR queueing discipline with 64-byte packets, together with the sorted framing algorithm with 256-byte frames, leads to mean queueing plus framing delays for single-character messages of less than 20 milliseconds at a 56-kb/s trunk node, even when the trunk is heavily loaded.

Trunk queueing plus framing delays for longer messages are substantially reduced if the access lines run more slowly than the trunk. At 60-percent nominal utilization, the reduction amounts to a factor of about 2 if the access speed is 20 percent of the trunk speed. The relative reduction is greater if the utilization is lower. Furthermore, increasing the trunk speed while holding the access speed and the trunk utilization constant reduces the trunk queueing plus framing delay by a more than proportional factor. In a practical system it would be advantageous to have the trunks run as fast as possible.

VI. ACKNOWLEDGMENTS

We are especially indebted to G. G. Riddle, who proposed both the PR + RR and PR + RR with hysteresis disciplines for the *DATAKIT* network, and with whom we have had extensive discussions. Thanks are also due to Ward Whitt for constructive comments on an earlier draft of this paper, and to a referee for drawing our attention to Ref. 3.

REFERENCES

1. A. G. Fraser, "Towards a Universal Data Transport System," IEEE J. Selected Areas in Commun., *SAC-1*, No. 5 (November 1983), pp. 803–16.
2. R. W. Wolff, "Time Sharing With Priorities," SIAM J. Appl. Math., *19*, No. 3 (November 1970), pp. 566–74.
3. H. Rudin, Jr., "Buffered Packet-Switching: A Queue With Clustered Arrivals," Int. Switching Symp. Rec., MIT (1972), pp. 259–65.
4. S. S. Lavenberg and D. R. Slutz, "Introduction to Regenerative Simulation," IBM J. Res. Develop., *19* (September 1975), pp. 458–62.
5. A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, New York: McGraw-Hill, 1982.
6. D. P. Heyman and M. J. Sobel, *Stochastic Models in Operations Research*, Vol. I, New York: McGraw-Hill, 1982.
7. S. L. Albin, "On Poisson Approximations for Superposition Arrival Processes in Queues," Management Sci., *28*, No. 2 (February 1982), pp. 126–37.
8. D. Anick, D. Mitra, and M. M. Sondhi, "Stochastic Theory of a Data-Handling System With Multiple Sources," B.S.T.J., *61*, No. 8 (October 1982), pp. 1871–94.
9. R. W. Wolff, "Poisson Arrivals See Time Averages," Oper. Res., *30*, No. 2 (March–April 1982), pp. 223–31.

APPENDIX

Mean Delays for PR + RR Discipline

We shall derive a system of linear equations satisfied by the mean delays in the PR + RR queueing discipline. The approach is modeled on Wolff's analysis[2] of the RR discipline.

In the mathematical model, jobs arrive in a Poisson stream. There are two queues served by a single server. Queue A has nonpreemptive priority over Queue B; the server does not start a job in Queue B if work is waiting in Queue A. An arriving job joins the end of Queue A, and when it reaches the server it receives a service quantum of up to δ_1. If additional service is required, the job joins the end of Queue B, works up to the server, receives service of at most δ_2, returns if necessary to the end of Queue B, on the next pass receives service of at most δ_3, and so on. Eventually the job completes service and leaves the system.

We allow for deterministic overhead at each service by modifying the service-time distribution of the incoming jobs. Finally, we assume that the incoming stream is a superposition of K independent Poisson streams, each with its own service-time distribution, and we write down expressions for the mean delay experienced by each job stream. Numerical results for exponential and deterministic distributions appear in Figs. 4 and 5.

A.1 Notation

λ =	the arrival rate of jobs.
S =	the service time of a job.
G =	the distribution function of S: $G(t) = P\{S \leqslant t\}$.
G^c =	the complement of G: $G^c(t) = 1 - G(t)$.
μ =	the service rate, i.e., $E(S) = 1/\mu$.
ρ =	λ/μ, where we assume $\rho < 1$.
G_e =	the equilibrium distribution of G: $G_e(t) = \mu \int_0^t [1 - G(u)]du$.
S_e =	a random variable with distribution $G_e(t)$. Note that $E(S_e) = E(S^2)/2E(S)$. Assume $E(S^2) < \infty$.
δ_j =	the amount of time allocated to a job on its jth pass, $j = 1, 2, \cdots$.
Δ_j =	the total time allocated to a job on its first j-passes:

$$\Delta_j = \sum_{i=1}^{j} \delta_i \qquad \text{for} \quad j = 1, 2, \cdots, \qquad \text{and} \quad \Delta_0 = 0^-.$$

j-job =	a job that is completed on the jth pass, i.e., a job for which $\Delta_{j-1} < S \leqslant \Delta_j$.
p_j =	the probability that a job is a j-job: $p_j = P\{\Delta_{j-1} < S \leqslant \Delta_j\} = G(\Delta_j) - G(\Delta_{j-1})$.
j-pass =	a job that is either waiting in queue or in service and has completed $j - 1$ passes. Note that a j-job can be a $(j-1)$-pass, but it is impossible for a $(j-1)$-job to be a j-pass.

Q_j = the expected number of j-passes waiting in queue, in the time-average sense.

r_j = the expected delay (wait in queue) of a j-pass just prior to making the jth pass.

d_j = total expected delay (in queue) of a j-job: $d_j = \sum_{i=1}^{j} r_i$.

ν_j = the *virtual work* of a j-pass, meaning the expected amount of additional time required to complete processing a j-pass in queue (including, if necessary, time drawn from subsequent passes). Hence,

$$\nu_j = \frac{\int_{\Delta_{j-1}}^{\infty} G^c(t)dt}{G^c(\Delta_{j-1})}, \qquad j = 1, 2, \cdots . \tag{1}$$

ω_j = the expected amount of work performed on an $\{i: i \geqslant j\}$-job on the jth pass:

$$\omega_j = \frac{\int_{\Delta_{j-1}}^{\Delta_j} G^c(t)dt}{G^c(\Delta_{j-1})}, \qquad j = 1, 2, \cdots . \tag{2}$$

A.2 Equations satisfied by mean delays

Consider a particular job (the "tagged" job) arriving at Queue A. Because Poisson arrivals see time averages,[2,9] on arrival the tagged job encounters the following expected values:

Q_1 1-passes in Queue A

Q_j j-passes in Queue B, where $j = 2, 3, \cdots$.

$\rho E(S_e)$ residual service time of the job in service.

The expected delay before the tagged job reaches the server for the first time is

$$r_1 = Q_1\omega_1 + \rho E(S_e) - O, \tag{3}$$

where O, the expected overage, is the expected amount of work that will remain to be performed on the job in service when it is interrupted.

Consider the job in service at the instant the tagged job arrives ("job in service" always means this particular job). The job in service is a j-pass for some value of j. The arrival rate of jobs that will complete δ_j of service on the jth pass is $\lambda G^c(\Delta_j)$. This class of jobs receives a total of $\lambda G^c(\Delta_j)\delta_j$ service per unit time on the jth pass, so the probability that at a random instant a job is in service that is about to complete δ_j of service on its jth pass is $\lambda G^c(\Delta_j)\delta_j$. The expected overage for such a job is ν_{j+1}, as defined by (1). Hence the expected overage for the actual job in service, which can have any value of j, is

$$O = \sum_{j=1}^{\infty} \lambda G^c(\Delta_j)\delta_j \nu_{j+1} = \lambda \sum_{j=1}^{\infty} \delta_j \int_{\Delta_j}^{\infty} G^c(t)dt. \tag{4}$$

If a j-pass completes its service quantum δ_j, the probability that it will reach the kth following pass, for $k = 1, 2, \cdots$, is $G^c(\Delta_{j+k-1})/G^c(\Delta_j)$, and the expected service that it will receive on the kth following pass is, from (2),

$$\frac{G^c(\Delta_{j+k-1})\omega_{j+k}}{G^c(\Delta_j)}. \tag{5}$$

Hence the expected service that the job in service will receive on the kth following pass is

$$\sum_{j=1}^{\infty} \lambda \delta_j G^c(\Delta_{j+k-1})\omega_{j+k}. \tag{6}$$

Substituting (4) into (3) gives for the average delay of the tagged job in Queue A:

$$r_1 = Q_1\omega_1 + \rho E(S_e) - \lambda \sum_{j=1}^{\infty} \delta_j \int_{\Delta_j}^{\infty} G^c(t)dt. \tag{7}$$

Immediately after service in Queue A, the tagged job expects to find

$\lambda(r_1 + \delta_1)$	1-passes in Queue A
$Q_1G^c(\Delta_1) + Q_2$	2-passes in Queue B
Q_j	j-passes in Queue B, where $j = 3, 4, \cdots$
Job in service	(6) with $k = 1$ in Queue B.

By the time the tagged job has waited an expected additional time r_2 for its first service in Queue B, an expected λr_2 more 1-passes will have arrived and will have been served in Queue A. It follows that

$$r_2 = \lambda(r_1 + r_2 + \delta_1)\omega_1 + [Q_1G^c(\Delta_1) + Q_2]\omega_2 + \sum_{j=3}^{\infty} Q_j\omega_j + \lambda \sum_{j=1}^{\infty} \delta_j G^c(\Delta_j)\omega_{j+1}. \tag{8}$$

Now let us define

$$s_2 = r_1 + r_2, \qquad s_j = r_j \quad \text{for} \quad j \neq 2,$$

$$Q_2' = Q_1G^c(\Delta_1) + Q_2, \qquad Q_j' = Q_j \quad \text{for} \quad j \neq 2. \tag{9}$$

Then eqs. (7) and (8) become

$$s_1 = Q_1'\omega_1 + \rho E(S_e) - \lambda \sum_{j=1}^{\infty} \delta_j \int_{\Delta_j}^{\infty} G^c(t)dt, \tag{10}$$

$$s_2 - s_1 = \lambda(s_2 + \delta_1)\omega_1 + \sum_{j=2}^{\infty} Q_j'\omega_j + \lambda \sum_{j=1}^{\infty} \delta_j G^c(\Delta_j)\omega_{j+1}. \tag{11}$$

Now suppose that the tagged job has reached the server in Queue B for the lth time, where $l \geqslant 2$. Ahead of it on this circuit there were expected to be:

$\lambda(s_{l+1} + \delta_l)$	1-passes in Queue A

and the following in Queue B:

$\lambda(s_l + \delta_{l-1})G^c(\Delta_1)$	2-passes
$\lambda(s_{l-1} + \delta_{l-2})G^c(\Delta_2)$	3-passes
$\vdots$	
$\lambda(s_2 + \delta_1)G^c(\Delta_{l-1})$	l-passes
$Q_2' G^c(\Delta_l)/G^c(\Delta_1)$	$(l + 1)$-passes
$Q_3' G^c(\Delta_{l+1})/G^c(\Delta_2)$	$(l + 2)$-passes
$\vdots$	
$Q_{n+1}' G^c(\Delta_{l+n-1})/G^c(\Delta_n)$	$(l + n)$-passes
$\vdots$	
Job in service	(6) with $k = l$.

Adding all the expected times together gives, for $l \geqslant 2$,

$$s_{l+1} = \lambda \sum_{m=1}^{l} (s_{l+2-m} + \delta_{l+1-m})G^c(\Delta_{m-1})$$
$$+ \sum_{m=2}^{\infty} \frac{Q_m' G^c(\Delta_{l+m-2})\omega_{l+m-1}}{G^c(\Delta_{m-1})} + \lambda \sum_{m=1}^{\infty} \delta_m G^c(\Delta_{m+l-1})\omega_{m+l}. \tag{12}$$

The next step is to express Q_j in terms of r_j. The arrival rate of j-passes is $\lambda G^c(\Delta_{j-1})$, and so by Little's law,

$$Q_j = \lambda G^c(\Delta_{j-1})r_j, \quad j = 1, 2, \cdots . \tag{13}$$

It follows from (9) and the fact that $G^c(\Delta_0) = 1$ that

$$Q_j' = \lambda G^c(\Delta_{j-1})s_j, \quad j = 1, 2, \cdots . \tag{14}$$

Substituting (14) into (10), (11), and (12), we obtain after some rearrangement

$$s_1 = A_1 s_1 + B_1,$$

$$s_2 - s_1 = A_1 s_2 + \sum_{j=2}^{\infty} A_j s_j + B_2,$$

$$s_i = \sum_{j=2}^{i} A_{i-j+1} s_j + \sum_{j=2}^{\infty} A_{i+j-2} s_j + B_i \quad \text{for} \quad i = 3, 4, 5, \cdots , \tag{15}$$

where

$$A_i = \lambda G^c(\Delta_{i-1})\omega_i = \lambda \int_{\Delta_{i-1}}^{\Delta_i} G^c(t)dt, \quad i = 1, 2, \cdots, \tag{16}$$

and

$$B_1 = \lambda \int_0^\infty tG^c(t)dt - \lambda \sum_{j=1}^\infty \delta_j \int_{\Delta_j}^\infty G^c(t)dt,$$

$$B_i = \sum_{j=1}^{i-1} \delta_j A_{i-j} + \sum_{j=1}^\infty \delta_j A_{i+j-1}, \quad \text{for} \quad i = 2, 3, \cdots. \tag{17}$$

Note that if $\delta_j = \delta$ for all j,

$$B_i = \delta \sum_{j=1}^\infty A_j = \delta\lambda \int_0^\infty G^c(t)dt = \rho\delta, \quad i = 2, 3, \cdots. \tag{18}$$

Having solved (15) for the quantities s_j, one obtains the waiting times r_j easily from (9).

A.3 Mean delays for RR discipline

Wolff[2] has given the equations for the ordinary RR; they are

$$r_1 = \sum_{j=1}^\infty A_j r_j + B_1,$$

$$r_i = \sum_{j=1}^{i-1} A_{i-j} r_j + \sum_{j=1}^\infty A_{i+j-1} r_j + B_i \quad \text{for} \quad i = 2, 3, \cdots, \tag{19}$$

where the A's and B's are given as before by (16) and (17).

Note that one cannot get eqs. (19) by setting $\delta_1 = 0$ in eqs. (15) and renumbering. The two disciplines are slightly different even when zero service is given in Queue A of the two-queue discipline. In PR + RR, arriving jobs have to queue up in Queue A when a job is in service in Queue B, and they join Queue B immediately behind the job in service if it returns to the end of Queue B, even if the jobs that were queued in Queue A get no service there as a result of their wait. In RR, arriving jobs join the queue ahead of the job in service in case the latter has to return to the end of the queue. In practice one would hardly set up a two-queue discipline with zero service quantum in the first queue.

A.4 Overhead

Let us suppose now that the jth time the server attends to a particular job, the job gets up to δ_j' units of service and there are δ_j'' units of overhead. Define

$$\delta_j = \delta_j' + \delta_j'',$$

$$\Delta_j = \Delta_j' + \Delta_j'' = \sum_{i=1}^{j} \delta_i' + \sum_{i=1}^{j} \delta_i''. \tag{20}$$

Suppose that the job brings in the intrinsic service time distribution $F^c(t)$, where $F^c(t)$ is the probability that the service time S' excluding overhead exceeds t. If $G^c(t)$ is the probability that the effective service time S, including overhead, exceeds t, a little thought shows that

$$G^c(t) = F^c(\Delta_j') \quad \text{for} \quad \Delta_j \leqslant t \leqslant \Delta_j + \delta_{j+1}'',$$
$$G^c(t) = F^c(t - \Delta_{j+1}'') \quad \text{for} \quad \Delta_j + \delta_{j+1}'' \leqslant t \leqslant \Delta_{j+1}, \tag{21}$$

for $j = 0, 1, 2, \cdots$.

It is now straightforward to express various quantities that we need for numerical calculations. For example,

$$\begin{aligned} E(S) &= \int_0^\infty G^c(t)dt \\ &= \sum_{j=0}^{\infty} \left[\int_{\Delta_j}^{\Delta_j+\delta_{j+1}''} F^c(\Delta_j')dt + \int_{\Delta_j+\delta_{j+1}''}^{\Delta_{j+1}} F^c(t - \Delta_{j+1}'')dt \right] \\ &= \sum_{j=0}^{\infty} \left[\delta_{j+1}'' F^c(\Delta_j') + \int_{\Delta_j'}^{\Delta_{j+1}'} F^c(t)dt \right] \\ &= E(S') + \sum_{j=0}^{\infty} \delta_{j+1}'' F^c(\Delta_j'). \end{aligned} \tag{22}$$

Similarly,

$$\begin{aligned} E(S^2) &= 2 \int_0^\infty tG^c(t)dt \\ &= E(S'^2) + \sum_{j=0}^{\infty} \Big[(2\Delta_j \delta_{j+1}'' + \delta_{j+1}''^2) F^c(\Delta_j') \\ &\qquad + 2\Delta_{j+1}'' \int_{\Delta_j'}^{\Delta_{j+1}'} F^c(t)dt \Big], \end{aligned} \tag{23}$$

$$\int_{\Delta_j}^{\infty} G^c(t)dt = \int_{\Delta_j'}^{\infty} F^c(t)dt + \sum_{l=j}^{\infty} \delta_{l+1}'' F^c(\Delta_l'), \tag{24}$$

and

$$A_j = \lambda \int_{\Delta_{j-1}}^{\Delta_j} G^c(t)dt = \lambda \int_{\Delta_{j-1}'}^{\Delta_j'} F^c(t)dt + \lambda \delta_j'' F^c(\Delta_{j-1}'). \tag{25}$$

These expressions simplify somewhat if δ_j' and δ_j'' are independent of j.

A.5 Merged job streams

Suppose that the incoming job stream consists of K streams of jobs arriving according to independent Poisson processes. Suppose that the arrival rate for the ith class is λ_i, and the intrinsic service time distribution for the ith class is $F_i^c(t)$. Then the merged arrival rate is

$$\lambda = \sum_{i=1}^{K} \lambda_i, \tag{26}$$

and the merged intrinsic service time distribution is

$$F^c(t) = \frac{1}{\lambda} \sum_{i=1}^{K} \lambda_i F_i^c(t). \tag{27}$$

This expression for $F^c(t)$ can go into all the previous machinery.

Finally, we would like to compute the mean waiting time in queue for the ith job class. Since the mean waiting time for a j-job is

$$d_j = \sum_{l=1}^{j} r_l, \tag{28}$$

the mean waiting time in queue for the ith job class is just

$$W_i = \sum_{j=1}^{\infty} d_j [F_i^c(\Delta_{j-1}') - F_i^c(\Delta_j')], \tag{29}$$

where the expression in square brackets is the probability that a random job from the ith stream is a j-job.

Note that in the presence of overhead the effective mean service time $E(S_i)$ will not be the same as the intrinsic mean service time $E(S_i')$. Hence the mean delay of the ith job class due to both queueing and overhead is

$$D_i = E(S_i) - E(S_i') + W_i, \quad i = 1, 2, \cdots, K. \tag{30}$$

A.6 Numerical cases

If we assume any particular form for the intrinsic message-length distributions $F_i^c(t)$, it is straightforward to calculate the A_i's and the B_i's from (16) and (17). For example, exponential and deterministic (constant-length) streams with intrinsic mean length $1/\mu_i'$ are given, respectively, by

$$F_i^c(t) = e^{-\mu_i' t},$$

$$F_i^c(t) = \begin{cases} 1, & 0 \leq t < 1/\mu_i', \\ 0, & t \geq 1/\mu_i'. \end{cases} \tag{31}$$

The relationship between the nominal server utilization ρ' and the effective utilization ρ in the presence of overhead is given by

$$\rho' = \lambda E(S'), \quad \rho = \lambda E(S), \tag{32}$$

where $E(S')$ and $E(S)$ are related by (22).

In the numerical calculations we have assumed three message streams as in Table I, and have taken the nominal utilization ρ' as the independent parameter. Trunk packets are 64 bytes or 16 bytes, and the overhead is 2 bytes. It turns out that for these numbers:

$$\delta' = 64, \quad \delta'' = 2, \quad \rho' = 0.807\rho;$$

$$\delta' = 16, \quad \delta'' = 2, \quad \rho' = 0.757\rho. \tag{33}$$

It is perhaps less surprising that so much of the trunk capacity is consumed by overhead if we recall that every single-character message looks like a 3-character message when it is put on the trunk.

The relationship between intrinsic and effective mean message lengths in the presence of overhead is shown in Table II. Message lengths are expressed in milliseconds, using the fact that one byte time = 0.143 milliseconds on a 56-kb/s trunk.

Some words about the numerical solution of eqs. (15) and (19) are in order. The coefficient matrix of these equations is not sparse in the technical sense (that is, not mostly zeros); however, the coefficients do approach zero more or less exponentially with increasing distance from the main diagonal. Also, the contributions of high-order partial delays to the average waiting time in eq. (29) fall off exponentially. This suggests that we truncate the infinite system (15) or (19) to an $n \times n$ system where $e^{-n\mu_K'\delta'} \ll 1$, assuming that $1/\mu_K'$ is the longest average message length.

Not surprisingly, the numerical problem is easy if long messages typically fit into a few trunk packets ($\mu_K'\delta' \approx 1$), and hard if long messages require many packets ($\mu_K'\delta' \ll 1$). Since solving a system of n linear equations takes time proportional to n^3, halving the packet size multiples the solution time by 8. In the numerical examples of

Table II—Intrinsic and effective message lengths

Packet Size (bytes)			Mean Length (ms)		
Data δ'	Overhead δ''	Message Type	Intrinsic $E(S_i')$	Effective $E(S_i)$	Difference
64	2	1	0.143	0.429	0.286
		2	5.71	6.07	0.36
		3	73.1	75.6	2.4
16	2	1	0.143	0.429	0.286
		2	5.71	6.58	0.87
		3	73.1	82.4	9.3

Figs. 4 and 5, we found that $n = 75$ was an appropriate number of equations to solve for $\delta' = 64$, and $n = 300$ for $\delta' = 16$. Various checks indicate that the mean waiting times calculated with these truncations are in error by no more than 0.5 percent. One could solve larger systems if necessary, but in view of the simplified traffic model that we are using, there seems little reason to refine the computations any further.

AUTHORS

A. G. Fraser, B.Sc. (Aeronautical Engineering), 1958, Bristol University; M.A. (Computing Science), 1966, Cambridge University; Ph.D. (Computing Science), 1969, Cambridge University; Ferranti, Ltd. (now ICT Ltd.), 1959–1966; AT&T Bell Laboratories, 1969—. At Cambridge University, Mr. Fraser wrote the file system for the Atlas 2 computer. Since joining AT&T Bell Laboratories, his personal research interests have been the architecture of data communication networks and high-level languages for integrated circuit design. Since 1982 he has been Director of the Computing Science Research Center. Member, ACM, IEEE.

Samuel P. Morgan, B.S., 1943, M.S., 1944, Ph.D. (Physics), 1947, California Institute of Technology; AT&T Bell Laboratories, 1947—. Mr. Morgan is a member of the Computing Science Research Center. A research mathematician, he was originally concerned with applications of electromagnetic theory to waveguide and radar problems. From 1959 to 1967 he was Head, Mathematical Physics Department, and from 1967 to 1982 he was Director, Computing Science Research Center. His current interests include queueing and congestion theory in computer-communication networks. Member, American Physical Society, ACM, SIAM.

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Coding for a Write-Once Memory

By J. K. WOLF,* A. D. WYNER,† J. ZIV,‡ and J. KÖRNER§

(Manuscript received October 21, 1983)

A *write-once memory* (WOM) is a binary storage medium in which the individual bit positions can be changed from the 0 state to the 1 state only once. Examples of WOMs are paper tapes, punched cards, and, most importantly, optical disks. For the latter storage medium, the 1's are marked by a laser that burns away a portion of the disk. In a recent paper, Rivest and Shamir showed that it is possible to update or rewrite a WOM to a surprising degree, and that the total amount of information which can be stored in an N-position WOM in many write/read "generations" or "stages" can be much larger than N.[1] In this paper we extend their results in several directions. Let $C(T, N)$ be the total number of bits of information that can be stored in an N-position WOM using T write/read generations. We consider the four cases that result when the writer (encoder) and/or reader (decoder) know the state of the memory at the previous generation. For three of these cases, when either the encoder and/or decoder knows the previous state, we show that $C(T, N) \sim N \log(T + 1)$, with T held fixed, as $N \to \infty$. For the remaining case, when neither the encoder nor the decoder knows the previous state, we show that $C(T, N) < N \pi^2/(6 \ln 2) \approx N (2.37)$ and that this bound can be approached arbitrarily closely with T, N sufficiently large.

I. INTRODUCTION

A *write-once memory* (WOM) is a binary storage medium in which the individual bit positions can be changed from the 0 state to the 1 state only once. Examples of WOMs are paper tapes, punched cards,

* University of Massachusetts, Amherst, Massachusetts. † AT&T Bell Laboratories. ‡ Technion-Israel Institute of Technology, Haifa, Israel. § Mathematical Institute of the Hungarian Academy of Sciences, Budapest, Hungary.

and, most importantly, optical disks. For the latter storage medium, the 1's are marked by a laser that burns away a portion of the disk. In a recent paper, Rivest and Shamir showed that it is possible to update or rewrite a WOM to a surprising degree, and that the total amount of information that can be stored in an N-position WOM in many write/read "generations" or "stages" can be much larger than N.[1] In this paper we extend their results in several directions. (See Section II, "Discussion on previous work.")

To fix ideas, consider an N-position WOM which we use successively for T write/read generations. Assume that N is large. Assume that initially all memory positions are in the 0 state, and that at the t-th write/read stage ($1 \leq t \leq T$), the writer (encoder) and the reader (decoder) are aware of the state of the memory after the previous [i.e., $(t-1)$-th] write.

At the first write stage, the encoder writes N independent and uniformly distributed bits, of which about half ($N/2$) will be 0. At the second write stage, the encoder writes about $N/2$ independent uniformly distributed bits using only the positions that were in the 0 state after the first write stage. The reader will be able to read the second generation information since we are assuming that it knows the state of the memory after the first generation. We continue in this way, storing $N2^{-(t-1)}$ bits at the t-th generation, for $t \leq T$. Thus, the total number of bits of information that is stored in T generations is about

$$N + \frac{N}{2} + \frac{N}{2^2} + \frac{N}{2^3} + \cdots + \frac{N}{2^{T-1}} = 2(1 - 2^{-T})N \sim 2N,$$

when T is large. Thus, we see that a total of more than N bits can be stored in the N-bit-position WOM. Actually, we can do somewhat better.

Let $\{q_t\}_{t=1}^{T}$, $0 < q_t < 1$, be arbitrary. At the first generation, write 1's on q_1N-bit positions. This can be done in $\binom{N}{q_1N}$ ways, so that we can store

$$B_1 = \log_2 \binom{N}{q_1N}$$

bits at the first generation. (All logarithms in this paper are taken to the base two.) Prior to the second write stage, there are $(1 - q_1)N$ positions in the 0 state. At the second stage, write on a fraction q_2 of these positions, storing

$$B_2 = \log \binom{N(1 - q_1)}{N(1 - q_1)q_2}$$

bits. Continuing in this way for successive stages—writing on a fraction q_t of the $(1 - q_1)(1 - q_2) \cdots (1 - q_{t-1})N$ positions, which are in the 0 state prior to the t-th write—we can store

$$B_t = \log \binom{N \prod_{j=1}^{t-1} (1 - q_j)}{Nq_t \prod_{j=1}^{t-1} (1 - q_j)}$$

$$= \log \binom{Np_1 \cdots p_{t-1}}{Np_1 \cdots p_{t-1}q_t}, \tag{1a}$$

bits at the t-th generation ($1 \le t \le T$), where

$$p_t = 1 - q_t, \qquad 1 \le t \le T. \tag{1b}$$

Using the Stirling formula for the factorial, we see that when N is large, we can store about $Nh(p_t) \prod_{j=1}^{t-1} p_j$ bits at the t-th generation, where $h(\lambda) = -\lambda \log \lambda - (1 - \lambda) \log (1 - \lambda)$ ($0 \le \lambda \le 1$) is the binary entropy function,* and $\prod_{j=1}^{t-1} p_j = 1$ for $t = 1$.

Suppose that we define the rate R_t as $1/N$ times the number of bits which are stored at the t-th generation. Our principal problem is to find the family of achievable $R_1, R_2, \cdots, R_t$, for the four situations that arise when, at the t-th generation, the encoder and/or decoder is, or is not, informed of the state of the memory after the previous [$(t - 1)$-th] write generation. (We hold T fixed, and let $N \to \infty$.) In particular we are interested in the total rate

$$C_T = \sum_{t=1}^{T} R_t, \tag{2}$$

at which information can be stored in the memory after T generations. For the case considered above (with the encoder and decoder informed),

$$C_T = \sum_{t=1}^{T} h(p_t) \prod_{j=1}^{t-1} p_j. \tag{3}$$

The choice of the $\{p_j\}$, or alternately the $\{q_j\}$, which maximizes C_T is given by

Lemma 1: Let $0 \le p_t \le 1$, *for* $t = 1, 2, \cdots, T$. *Then*

$$\sum_{t=1}^{T} h(p_t) \prod_{j=1}^{t-1} p_j \le \log(T + 1), \tag{4a}$$

* Take $h(0) = h(1) = 0$. It follows immediately from the Stirling formula that $\lim_{N \to \infty} \frac{1}{N} \log \binom{N}{\lambda N} = h(\lambda)$, $0 < \lambda < 1$.

with equality when

$$p_t = \frac{T - t + 1}{T - t + 2}, \qquad 1 \le t \le T. \tag{4b}$$

Proof: For $T = 1, 2, \cdots$, define

$$F_T(p_1, \cdots, p_T) = \sum_{t=1}^{T} h(p_t) \prod_{j=1}^{t-1} p_j, \tag{5}$$

$0 \le p_t \le 1$, $1 \le t \le T$. Observe that

$$F_T(p_1, \cdots, p_T) = h(p_1) + p_1 \sum_{t=2}^{T} h(p_t) \prod_{j=2}^{t-1} p_j$$

$$= h(p_1) + p_1 F_{T-1}(p_2, \cdots, p_T). \tag{6}$$

We now prove the lemma by induction on T. When $T = 1$, $F_1(p_1) = h(p_1) \le \log(2)$, with equality when $p_1 = 1/2$. Assume that the lemma holds for $T = T_0 - 1$. We will show (a) $F_{T_0} \le \log(T_0 + 1)$, and (b) with $\{p_t\}$ given by (4b) with $T = T_0$, $F_{T_0}(p_1, \cdots, p_{T_0}) = \log(T_0 + 1)$.

To show (a), invoke (6) and the induction hypothesis, yielding

$$F_{T_0}(p_1, p_2, \cdots, p_{T_0}) \le h(p_1) + p_1 \log T_0. \tag{7}$$

Setting the derivative of the right member of (7) (which is a concave function of p_1) with respect to p_1 equal to zero, we see that the right member is maximized at $p_1 = T_0/(T_0 + 1)$, so that

$$F_{T_0}(p_1, \cdots, p_{T_0}) \le h\left(\frac{T_0}{T_0 + 1}\right) + \frac{T_0}{T_0 + 1} \log T_0$$

$$= \log(T_0 + 1),$$

which is (a).

To show (b) let

$$\lambda_t = \frac{(T_0 - 1) - t + 1}{(T_0 - 1) - t + 2}, \qquad t = 1, 2, \cdots, T_0.$$

The induction hypothesis implies that

$$F_{T_0-1}(\lambda_1, \cdots, \lambda_{T_0-1}) = \log T_0.$$

Further, for p_t given by (4b) with $T = T_0$, $p_{t+1} = \lambda_t$. Thus, (6) yields

$$F_{T_0}(p_1, \cdots, p_{T_0}) = h\left(\frac{T_0}{T_0 + 1}\right) + \frac{T_0}{T_0 + 1} \log T_0$$

$$= \log(T_0 + 1),$$

establishing (b), and the lemma.

Applying Lemma 1 and (3) we see that for the case where the encoder and decoder are informed, we can achieve $C_T = \log(T+1)$. Thus, we can store a total of about $N \log(T+1)$ bits on an N-position WOM in T generations (with T held fixed as $N \rightarrow \infty$). In the sequel we will show that the simple scheme outlined above is essentially optimal. Quite surprisingly for two of the other cases—encoder or decoder informed—we can do just as well, i.e., achieve $C_T = \log(T+1)$. For the fourth case—neither the encoder nor decoder informed—we show that, as $T \rightarrow \infty$, the maximum achievable C_T is $(\pi^2/6)\log e \cong 2.37$, which is considerably less than $\log(T+1)$ but nevertheless significantly greater than unity.

II. FORMAL STATEMENT OF THE PROBLEM AND RESULTS

The memory consists of N cells or bit positions that can be in either the 0 or 1 state. Assume initially that all cells are in the state 0. At time (or generation) $t = 1, 2, \cdots, T$, data $\mathbf{S}^t$ is stored in the memory. Assume that $\{\mathbf{S}^t\}$ is a set of independent random K_t-vectors, and that $\mathbf{S}^t$ is uniformly distributed in binary K_t-space, $1 \le t \le T$. Denote the state of the memory at time t by $\mathbf{Y}^t = (Y_{t1}, Y_{t2}, \cdots, Y_{tN})$, where $Y_{tn} = 0$ or 1 and $\mathbf{Y}^0 = (0, 0, \cdots, 0)$.

At time t, the encoder inputs into the memory a binary N-vector $\mathbf{X}^t = (X_{t1}, \cdots, X_{tN})$ (which is a function of $\mathbf{S}^t$ and perhaps $\mathbf{Y}^{t-1}$) and the state of the memory changes to $\mathbf{Y}^t$, where

$$Y_{tn} = X_{tn} \vee Y_{t-1,n} = \begin{cases} 0 \text{ if } X_{tn} = Y_{t-1,n} = 0, \\ 1 \text{ otherwise.} \end{cases} \tag{8}$$

The contents of the memory may now be read, and an estimate $\hat{\mathbf{S}}^t$ of the data $\mathbf{S}^t$ obtained. The error rate is

$$P_e^t = \frac{1}{K_t} E d_H(\mathbf{S}^t, \hat{\mathbf{S}}_t), \tag{9}$$

where $E(\quad)$ is expectation and where $d_H(\mathbf{u}, \mathbf{v})$ is the number of positions in which the binary N-vectors $\mathbf{u}$ and $\mathbf{v}$ differ (*Hamming distance*). $\hat{\mathbf{S}}^t$ is a function of $\mathbf{Y}^t$ and perhaps $\mathbf{Y}^{t-1}$.

We now consider four cases.

Case 1 (encoder and decoder informed):

$$\begin{aligned} \mathbf{X}^t &= f_E^t(\mathbf{S}^t, \mathbf{Y}^{t-1}), \\ \hat{\mathbf{S}}^t &= f_D^t(\mathbf{Y}^t, \mathbf{Y}^{t-1}), \end{aligned} \tag{10}$$

$1 \le t \le T$. The functions f_E^t and f_D^t are the encoder and decoder functions, respectively. In this case both the encoder and the decoder are informed of the state of the memory at the previous generation.

Case 2 (encoder informed, decoder uninformed):

$$\mathbf{X}^t = f_E^t(\mathbf{S}^t, \mathbf{Y}^{t-1}),$$

$$\hat{\mathbf{S}}^t = f_D^t(\mathbf{Y}^t), \tag{11}$$

$1 \le t \le T$.

Case 3 (encoder uninformed, decoder informed):

$$\mathbf{X}^t = f_E^t(\mathbf{S}^t),$$

$$\hat{\mathbf{S}}^t = f_D^t(\mathbf{Y}^t, \mathbf{Y}^{t-1}), \tag{12}$$

$1 \le t \le T$.

Case 4 (encoder and decoder uninformed):

$$\mathbf{X}^t = f_E^t(\mathbf{S}^t),$$

$$\hat{S}^t = f_D^t(\mathbf{Y}^t), \tag{13}$$

$1 \le t \le T$.

For a given Case (1 through 4) and a given $T \ge 1$, we say that a (rate-) vector $\mathbf{r} = (r_1, r_2, \cdots, r_T)$ $0 \le r_i \le 1$, is *achievable* if, for arbitrary $\epsilon > 0$, there exists an encoder/decoder with parameters T, N, $\{K_t\}_{t=1}^T$ such that, for $1 \le t \le T$,

$$\frac{K_t}{N} \ge r_t - \epsilon, \tag{14a}$$

$$P_e^t \le \epsilon. \qquad 14b)$$

Similarly, a (total rate) R is achievable if, for arbitrary $\epsilon > 0$, there exists an encoder/decoder with parameters T, N, $\{K_t\}_1^T$ such that

$$\frac{1}{N} \sum_{t=1}^{T} K_t \ge R - \epsilon, \tag{15a}$$

$$\frac{\sum_{t=1}^{T} P_e^t K_t}{\sum_{t=1}^{T} K_t} \le \epsilon. \tag{15b}$$

The left member of (15b) is the expected fraction of the total of $\sum_1^T K_t$ bits which are decoded in error. The *capacity* C_T of the WOM is the supremum of the achievable total rates.

In each Case (1 through 4) we seek to find the family of achievable rate vectors. We now summarize our results.

Let $\mathbf{p} = (p_1, p_2, \cdots, p_T)$ be a T vector for which $0 \le p_t \le 1$, $1 \le t \le T$, and let

$$\mathscr{R}_T(\mathbf{p}) = \left\{ \mathbf{r} = (r_1, r_2, \cdots, r_T) \colon 0 \le r_t \le h(p_t) \prod_{j=1}^{t-1} p_j \right\}, \tag{16}$$

where, as in Section I, $h(\lambda)$ is the binary entropy function. Finally, define

$$\mathscr{R}_T = \bigcup_{\mathbf{p}} \mathscr{R}_T(\mathbf{p}). \tag{17}$$

In the sequel, we will establish the following two theorems, which assert that for Cases 1 through 3 (encoder and/or decoder informed), $\mathscr{R}_T$ is the family of achievable rates.

Theorem 1 (direct half): *For Cases 1 through 3, let $T \geq 1$ be given. If $\mathbf{r} \in \mathscr{R}_T$, then $\mathbf{r}$ is achievable.*

Theorem 2 (converse half): *For Cases 1 through 3, and any encoder/decoder with parameters T, N, $\{K_t\}$, and error probabilities $\{P_e^t\}$, there exists a member $\mathbf{r} = (r_1, r_2, \cdots, r_T)$ of $\mathscr{R}_T$ such that*

$$\frac{K_t}{N} \leq r_t + h(P_e^t), \qquad 1 \leq t \leq T. \tag{18}$$

It follows from the discussion in Section I that the capacity (for Cases 1 through 3) is $C_T = \log(T + 1)$. It is also a consequence of our proof of Theorem 1 for Case 2, that for the codes constructed, $P_e^t = 0$, $1 \leq t \leq T$.

For Case 4 (encoder and decoder uninformed), we cannot completely characterize the family of achievable rate vectors. We do, however, establish the following theorems.

Let $\mathbf{p} = (p_1, p_2, \cdots, p_T)$ be a T-vector for which $0 \leq p_t \leq 1$ ($1 \leq t \leq T$). Let

$$Q_0 = 0, \tag{19a}$$

$$Q_t = \prod_{j=1}^{t} p_j, \qquad 1 \leq t \leq T. \tag{19b}$$

Let $\mathscr{R}'_T(\mathbf{p})$ be the set of $\mathbf{r} = (r_1, \cdots, r_T)$ for which

$$r_t \leq h(Q_t) - p_t h(Q_{t-1}), \tag{20}$$

for $1 \leq t \leq T$. For $\mathbf{r} \in \mathscr{R}'_T(\mathbf{p})$,

$$\begin{aligned} \sum_{t=1}^{T} r_t &\leq \sum_{t=1}^{T} h(Q_t) - p_t h(Q_{t-1}) \\ &= h(Q_T) + \sum_{t=1}^{T} (1 - p_t) h(Q_{t-1}). \end{aligned} \tag{21}$$

We now state

Theorem 3 (existence): *For Case 4, let $T \geq 1$ be given. If $\mathbf{r} \in \mathscr{R}'_T(\mathbf{p})$, for some $\mathbf{p}$, then $\mathbf{r}$ is achievable.*

Theorem 4 (partial converse): *For Case 4, and any encoder/decoder with parameters T, N, $\{K_t\}$, and error probabilities $\{P_e^t\}$, then*

$$\left(\sum_{t=1}^{T} \frac{K_t}{N}\right)\left(1 - h\left(\frac{\sum_{t=1}^{T} P_e^t K_t}{\sum_{t=1}^{T} K_t}\right)\right)$$

$$\leq \sup_{\mathbf{r}\in\bigcup_{\mathbf{p}}\mathscr{R}_T(\mathbf{p})} \sum_{t=1}^{T} r_t$$

$$= \sup_{\mathbf{p}} \left\{h(Q_T) - \sum_{t=1}^{T} (1 - p_t)h(Q_{t-1})\right\} \triangleq \rho_T. \qquad (22)$$

It follows from Theorem 4 that if R is an achievable total rate, then $R \leq \rho_T$. Furthermore, we show in Section III that

$$\rho_T \leq \frac{\pi^2}{6} \log_2 e \approx 2.37318, \qquad (23a)$$

and that

$$\lim_{T\to\infty} \rho_T = \frac{\pi^2}{6} \log_2 e. \qquad (23b)$$

Discussion of previous work

An information theoretic treatment of coding for memories of this type was given by Kusnetsov and Tsybatov[2] in 1974. They studied binary memories with defective cells—typically cells that are "stuck at 1". Their work was extended and generalized considerably by Heegard and El Gamal.[3] Rivest and Shamir[1] originated the concept of rewriting on WOMs. Their problem is similar to that in previous models with the "stuck at 1" state of Ref. 3 and Ref. 2 being the result of writing on the WOM in previous generations. In the new problem, the system designer must balance the needs of memory users at all generations. In a very recent paper, Heegard[4] generalized the Rivest-Shamir results in several ways.

The models in all of the above papers correspond to our Case 2, in that it is always assumed that the encoder can read the memory before writing, and that the decoder is unaware of the state of the memory before the present write. Heegard and El Gamal[3] proved a coding theorem for the memory with "stuck at 1" defects which can be adapted to our rewriting on WOM's problem. Essentially this was done by Rivest and Shamir, although they apparently were not aware of the earlier work. Concerning Case 2, our results represent an extension of previous results in that our converse theorem (Theorem 2, Case 2) holds for codes with a small error probability, and not a zero error probability as in Ref. 4 and Ref. 1. Our results for Cases 1, 3, and 4 are new.

III. PROOF OF CONVERSES

In this section we establish the converse Theorems 2 and 4.

Proof of Theorem 2: It suffices to establish Theorem 2 for Case 1 (both encoder and decoder informed). Let $\{f_E^t, f_D^t\}_{t=1}^T$ define an encoder/decoder for Case 1 [defined by (10)] with parameters T, N, $\{K_t\}$ and error probabilities $\{P_e^t\}$. Consider the t-th generation. Since $\mathbf{S}^t$, $\mathbf{X}^t$, $\mathbf{Y}^t$, $\hat{\mathbf{S}}^t$ is a Markov chain given $\mathbf{Y}^{t-1}$, the data processing theorem yields*

$$I(\mathbf{S}^t;\ \hat{\mathbf{S}}^t \,|\, \mathbf{Y}^{t-1}) \leq I(\mathbf{X}^t;\ \mathbf{Y}^t \,|\, \mathbf{Y}^{t-1}). \tag{24a}$$

Now

$$\begin{aligned} I(\mathbf{S}^t;\ \hat{\mathbf{S}}^t \,|\, \mathbf{Y}^{t-1}) &= H(\mathbf{S}^t \,|\, \mathbf{Y}^{t-1}) - H(\mathbf{S}^t \,|\, \hat{\mathbf{S}}^t, \mathbf{Y}^{t-1}) \\ &\geq H(\mathbf{S}^t) - H(\mathbf{S}^t \,|\, \hat{\mathbf{S}}^t), \end{aligned} \tag{24b}$$

where the inequality follows from the independence of $\mathbf{S}^t$ and $\mathbf{Y}^{t-1}$, which implies $H(\mathbf{S}^t \,|\, \mathbf{Y}^{t-1}) = H(\mathbf{S}^t) = K_t$, and from the fact that conditioning decreases entropy. Further, from Fano's inequality, $1/N\ H(\mathbf{S}^t \,|\, \hat{\mathbf{S}}^t) \leq h(P_e^t)$, so that (24) yields

$$\frac{1}{N} I(\mathbf{X}^t;\ \mathbf{Y}^t \,|\, \mathbf{Y}^{t-1}) \geq \frac{K_t}{N} - h(P_e^t). \tag{25}$$

Now, writing $\mathbf{Y}^t = (Y_{t1}, Y_{t2}, \cdots, Y_{tN})$, we have for $0 \leq t \leq T$,

$$\begin{aligned} \frac{1}{N} I(\mathbf{X}^t;\ \mathbf{Y}^t \,|\, \mathbf{Y}^{t-1}) &\overset{(1)}{\leq} \frac{1}{N} H(\mathbf{Y}^t \,|\, \mathbf{Y}^{t-1}) \\ &\overset{(2)}{\leq} \frac{1}{N} \sum_{n=1}^{N} H(Y_{tn} \,|\, Y_{t-1,n}) \\ &\overset{(3)}{=} \frac{1}{N} \sum_{n=1}^{N} H(Y_{tn} \,|\, Y_{t-1,n} = 0) \Pr\{Y_{t-1,n} = 0\}. \end{aligned} \tag{26}$$

Step 1 is a standard inequality; step 2 follows from the fact that the entropy of a vector is no greater than the sum of the entropies of its components, and conditioning decreases entropy; and step 3 follows from $H(Y_{tn} \,|\, Y_{t-1,n} = 1) = 0$. Setting

* Remember $\mathbf{Y}^0 = (0, 0, \cdots, 0)$.

$$Q_{tn} = \Pr\{Y_{tn} = 0\},$$

$$Q_t = \frac{1}{N} \sum_{n=1}^{N} Q_{tn},$$

$1 \le n \le N$, $1 \le t \le T$, we have $Q_{tn} \ge Q_{t+1,n}$ and

$$\Pr\{Y_{tn} = 0 \mid Y_{t-1,n} = 0\} = \frac{\Pr\{Y_{tn} = 0,\ Y_{t-1,n} = 0\}}{\Pr\{Y_{t-1,n} = 0\}}$$

$$= \frac{\Pr\{Y_{tn} = 0\}}{\Pr\{Y_{t-1,n} = 0\}} = \frac{Q_{tn}}{Q_{t-1,n}}.$$

Hence $H(Y_{tn} \mid Y_{t-1,n} = 0) = h\left(\frac{Q_{tn}}{Q_{t-1,n}}\right)$, and (26) become

$$\frac{1}{N} I(\mathbf{X}^t, \mathbf{Y}^t \mid \mathbf{Y}^{t-1}) \le \frac{1}{N} \sum_{n=1}^{N} Q_{t-1,n} h\left(\frac{Q_{tn}}{Q_{t-1,n}}\right)$$

$$= Q_{t-1} \frac{1}{N} \sum_{n=1}^{N} \frac{Q_{t-1,n}}{Q_{t-1}} h\left(\frac{Q_{tn}}{Q_{t-1,n}}\right)$$

$$\le Q_{t-1} h\left(\frac{1}{Q_{t-1}} \frac{1}{N} \sum_{n=1}^{N} Q_{t-1,n} \frac{Q_{tn}}{Q_{t-1,n}}\right)$$

$$= Q_{t-1} h\left(\frac{Q_t}{Q_{t-1}}\right). \tag{27}$$

The second inequality in (27) follows from the concavity of $h(\cdot)$. Combining (25) and (27) we have, for $1 \le t \le T$,

$$Q_{t-1} h\left(\frac{Q_t}{Q_{t-1}}\right) \ge \frac{K_t}{N} - h(P_e^t). \tag{28}$$

Now let us define $p_t = \dfrac{Q_t}{Q_{t-1}} \le 1$, $1 \le t \le T$. Since $Q_0 = 1$, we have $Q_t = \prod_{j=1}^{t} p_j$, so that (28) is

$$\frac{K_t}{N} \le \prod_{j=1}^{t} p_j h(p_t) + h(P_e^t), \tag{29}$$

$1 \le t \le T$. Comparison of (29) with (16) yields (18) and Theorem 2.

We now turn our attention to Theorem 4. Let $f_E^{(t)}(\cdot)$ and $f_D^{(t)}(\cdot)$, $1 \le t \le T$ define an encoder/decoder for Case 4 with parameters T, N, $\{K_t\}_{t=1}^{T}$, and error probabilities $\{P_e^t\}_{t=1}^{T}$. Then

$$\sum_{t=1}^{T} K_t = \sum_{t=1}^{T} H(\mathbf{S}^t) \overset{(1)}{=} H(\mathbf{S}^1, \mathbf{S}^2, \cdots, \mathbf{S}^T)$$

$$\overset{(2)}{=} H(\mathbf{S}^1, \mathbf{S}^2, \cdots, \mathbf{S}^T, \mathbf{Y}^1, \mathbf{Y}^2, \cdots, \mathbf{Y}^T)$$

$$= H(\mathbf{Y}^T) + H(\mathbf{S}^1, \mathbf{S}^2, \cdots, \mathbf{S}^T, \mathbf{Y}^1 \cdots \mathbf{Y}^{T-1} | \mathbf{Y}^T)$$

$$= H(\mathbf{Y}^T) + U_T, \qquad (30)$$

where

$$U_t = H(\mathbf{S}^1, \cdots, \mathbf{S}^t, \mathbf{Y}^1, \cdots, \mathbf{Y}^{t-1} | \mathbf{Y}^t), \qquad 1 \le t \le T. \qquad (31)$$

Step 1 in eq. (30) follows from the independence of the $\{\mathbf{S}^t\}_1^T$, and step 2 from the fact that $\mathbf{Y}^t$ is functionally determined by $\mathbf{S}^1, \mathbf{S}^2, \cdots, \mathbf{S}^t$, $1 \le t \le T$. (Take $U_0 = 0$.)

Now, for $1 \le t \le T$,

$$U_t = H(\mathbf{S}^t, \mathbf{Y}^{t-1} | \mathbf{Y}^t)$$

$$+ H(\mathbf{S}^1, \cdots, \mathbf{S}^{t-1}, \mathbf{Y}^1, \cdots, \mathbf{Y}^{t-2} | \mathbf{Y}^{t-1}, \mathbf{S}^t, \mathbf{Y}^t)$$

$$\overset{(1)}{=} H(\mathbf{Y}^{t-1} | \mathbf{S}^t, \mathbf{Y}^t) + H(\mathbf{S}^t | \mathbf{Y}^t)$$

$$+ H(\mathbf{S}^1, \cdots, \mathbf{S}^{t-1}, \mathbf{Y}^1, \cdots, \mathbf{Y}^{t-2} | \mathbf{Y}^{t-1})$$

$$\overset{(2)}{=} H(\mathbf{Y}^{t-1} | \mathbf{S}^t, \mathbf{X}^t, \mathbf{Y}^t) + H(\mathbf{S}^t | \mathbf{Y}^t) + U_{t-1}, \qquad (32)$$

where step 1 follows from the fact that $(\mathbf{S}^1, \cdots, \mathbf{S}^{t-1}, \mathbf{Y}^1, \cdots, \mathbf{Y}^{t-2})$, $(\mathbf{S}^t, \mathbf{Y}^t)$ are conditionally independent given $\mathbf{Y}^{t-1}$, and step 2 follows from $\mathbf{X}^t = f_E^t(\mathbf{S}^t)$.

Now from Fano's inequality,

$$H(\mathbf{S}^t | \mathbf{Y}^t) \le K_t h(P_e^t),$$

and since conditioning decreases entropy,

$$H(\mathbf{Y}^{t-1} | \mathbf{S}^t, \mathbf{X}^t, \mathbf{Y}^t) \le H(\mathbf{Y}^{t-1} | \mathbf{X}^t, \mathbf{Y}^t).$$

Thus, from (32), for $1 \le t \le T$,

$$(U_t - U_{t-1}) \le H(\mathbf{Y}^{t-1} | \mathbf{X}^t, \mathbf{Y}^t) + K_t h(P_e^t).$$

Summing on t, we obtain (noting that $U_0 = 0$)

$$U_T \le \sum_{t=1}^{T} H(\mathbf{Y}^{t-1} | \mathbf{X}^t, \mathbf{Y}^t) + \sum_{t=1}^{T} K_t h(P_e^t).$$

Substituting into (30) we have

$$\sum_{k=1}^{T} K_t \le H(\mathbf{Y}^T) + \sum_{t=1}^{T} H(\mathbf{Y}^{t-1}|\mathbf{X}^t\mathbf{Y}^t) + \sum_{t=1}^{T} K_t h(P_e^t)$$

$$\le \sum_{n=1}^{N} H(Y_{Tn}) + \sum_{n=1}^{N}\sum_{t=1}^{T} H(Y_{t-1,n}|X_{tn}Y_{tn}) + \sum_{t=1}^{T} K_t h(P_e^t). \quad (33)$$

Applying the concavity of $h(\cdot)$ and Jensen's inequality, we have

$$\sum_{t=1}^{T} K_t h(P_e^t) = \left(\sum_t K_t\right) \frac{\sum K_t h(P_e^t)}{\sum K_t}$$

$$\le (\sum K_t) h\left(\frac{\sum K_t P_e^t}{\sum K_t}\right),$$

so that (33) yields

$$\left(\sum_{t=1}^{T} K_t\right)\left(1 - h\left(\frac{\sum K_t P_e^t}{\sum K_t}\right)\right)$$

$$\le \sum_{n=1}^{N}\left[H(Y_{Tn}) + \sum_{t=1}^{T} H(Y_{t-1,n}|X_{tn}Y_{tn})\right]. \quad (34)$$

Now fix n $(1 \le n \le N)$ and write $X_{tn} = X_t$, $Y_{tn} = Y_t$, $Y_{t-1,n} = Y_{t-1}$. The random variables X_t, Y_t, Y_{t-1} are binary. Consider, for $2 \le t \le T$,

$$H(Y_{t-1}|X_tY_t) = H(Y_{t-1}, X_t, Y_t) - H(X_t, Y_t)$$

$$\overset{(1)}{=} H(Y_{t-1}, X_t) - H(X_t, Y_t)$$

$$= H(Y_{t-1}) + H(X_t|Y_{t-1}) - H(X_t) - H(Y_t|X_t)$$

$$\overset{(2)}{=} H(Y_{t-1}) - H(Y_t|X_t), \quad (35)$$

where step 1 follows from $Y_t = X_t \vee Y_{t-1}$, and step 2 from the independence of X_t and Y_{t-1}. Now put back the n dependence. Letting $p_{tn} = \Pr\{X_{tn} = 0\}$, $1 \le t \le T$, $1 \le n \le N$, we have from the independence of the $\{X_{tn}\}_{t=1}^{T}$,

$$\Pr\{Y_{tn} = 0\} = \Pr\{X_{1n} = X_{2n} = \cdots = X_{tn} = 0\}$$

$$= \prod_{j=1}^{t} p_{tn} \triangleq Q_{tn},$$

and

$$\Pr\{Y_{tn} = 0|X_{tn} = 0\} = \Pr\{Y_{t-1,n} = 0\} = Q_{t-1,n},$$

and

$$\Pr\{Y_{tn} = 0 \mid X_{tn} = 1\} = 0.$$

Thus (35) is

$$\begin{aligned} H(Y_{t-1,n} \mid X_{tn} Y_{tn}) &= h(Q_{t-1,n}) + p_{tn} h(Q_{t-1,n}) \\ &= (1 - p_{tn}) h(Q_{t-1,n}), \end{aligned}$$

and the term in brackets in (34) is

$$\begin{aligned} H(Y_{Tn}) + \sum_{t=1}^{T} H(Y_{t-1,n} \mid X_{tn} Y_{tn}) \\ = h(Q_{Tn}) + \sum_{t=1}^{T} (1 - p_{tn}) h(Q_{t-1,n}) \\ \le \rho_T. \end{aligned}$$

Thus (34) is

$$\sum K_t \left[1 - h \left(\frac{\sum P_e^t K_t}{\sum K_t} \right) \right] \le N \rho_T,$$

which is Theorem 4.

Our final task in this section is to establish (23a) and (23b). We begin by establishing the following:

Proposition 1: Let $0 \le a < b < \infty$. Then

$$\int_a^b h(e^{-x}) dx \ge (1 - e^{-(b-a)}) h(e^{-a}).$$

Proof: Let $y = e^{-x}$, $y_1 = e^{-a}$, $y_0 = e^{-b}$. Then $0 \le y_0 < y_1 \le 1$ and we must show

$$\begin{aligned} \int_a^b h(e^{-x}) dx = \int_{y_0}^{y_1} \frac{h(y)}{y} dy &\ge \left(1 - \frac{y_0}{y_1} \right) h(y_1) \\ &= (y_1 - y_0) \frac{h(y_1)}{y_1}. \end{aligned}$$

But $\dfrac{h(y)}{y}$ is nonincreasing $\left(\dfrac{d}{dy} \dfrac{h(y)}{y} = y^{-2} \log(1 - y) \le 0 \right)$, so that $\dfrac{h(y)}{y}$ can be underbounded in the integral by $h(y_1)/y_1$, establishing the proposition.

Since ρ_T is nondecreasing in T, we can establish (23a) and (23b) by showing that

$$\sup \sum_{t=1}^{\infty} (1 - p_t) h(Q_{t-1}) = \frac{\pi^2}{6} \log e, \tag{36}$$

where the supremum is with respect to sequences $\{p_t\}_{t=1}^{\infty}$, where $0 \leq p_t \leq 1$, and $Q_t = \prod_{j=1}^{t} p_j$. Let $\{p_t\}$ be given, and consider

$$\psi = \sum_{t=1}^{\infty} (1 - p_t)h(Q_{t-1}). \tag{37}$$

For $1 \leq t < \infty$, let $\alpha_t = -\ln p_t$, so that $p_t = e^{-\alpha_t}$, and

$$Q_t = \prod_{j=1}^{t} p_j = \prod_{j=1}^{t} e^{-\alpha_j} = \exp\left\{-\sum_{j=1}^{t} \alpha_j\right\} = e^{-x_t},$$

where $x_t = \sum_{j=1}^{t} \alpha_j$. (Take $x_0 = 0$.) Thus,

$$p_t = e^{-\alpha_t} = e^{-(x_t - x_{t-1})},$$

and

$$\begin{aligned}\psi &= \sum_{t} (1 - p_t)h(Q_{t-1}) \\ &= \sum_{t=1}^{\infty} (1 - e^{-(x_t - x_{t-1})})h(e^{-x_{t-1}}) \\ &\leq \sum_{t=1}^{\infty} \int_{x_{t-1}}^{x_t} h(e^{-x})dx = \lim_{t\to\infty} \int_0^{x_t} h(e^{-x})dx,\end{aligned}$$

where the inequality follows from the Proposition 1. Since $h(\cdot) \geq 0$ and

$$\begin{aligned}\int_0^{\infty} h(e^{-x})dx &= \int_0^1 \frac{h(y)}{y}\, dy \\ &= (\log_2 e) \int_0^1 \left(-\ln y - \frac{(1-y)}{y} \ln(1-y)\right) dy \\ &= (\log_2 e) \frac{\pi^2}{6},\end{aligned}$$

we have shown that

$$\sup \psi = \sup \sum (1 - p_t)h(Q_{t-1}) \leq (\log_2 e) \frac{\pi^2}{6}. \tag{38}$$

Furthermore, ψ can be made arbitrarily close to the right member of (38) by setting $p_t = e^{-\delta}$ for sufficiently small $\delta > 0$. In other words,

$$\psi = \sum_{t=1}^{\infty} (1 - e^{-\delta})h(e^{-t\delta}) \to \int_0^{\infty} h(e^{-x})dx = (\log e) \frac{\pi^2}{6},$$

as $\delta \to 0$. This completes the verification of (23a) and (23b).

IV. PROOFS OF (DIRECT) THEOREMS 1 and 3

In this section we give proofs of the "direct" coding theorems (Theorems 1 and 3). Actually, we need two proofs (for Cases 2 and 3) for Theorem 1. We give these in Sections 4.1 and 4.3, respectively, and prove Theorem 3 (for Case 4) in Section 4.4.

4.1 Case 2 (encoder informed, decoder uninformed)

We begin with some definitions. The *weight*, $|\mathbf{u}|$, of a binary N-vector $\mathbf{u}$ is the number of nonzero entries in $\mathbf{u}$. Let $B_N(w)$ be the set of binary N vectors with weight w. We say that binary N-vector $\mathbf{u}$ *covers* the binary N-vector $\mathbf{v}$, denoted $\mathbf{u} > \mathbf{v}$, if $\mathbf{u}$ has 0 entries only in positions in which $\mathbf{v}$ has 0 entries. Thus, for example, when $N = 4$, $(1010) > (1000)$, but (1010) does not cover (1100).

Now consider the encoder for Case 2 with the parameters N, T, $\{K_t\}_1^T$ given. Let $M_t = 2^{K_t}$, $1 \le t \le T$. We will specify an ad hoc encoder as follows. Let $\{w_t\}_{t=0}^T$ satisfy

$$0 = w_0 < w_1 \cdots < w_T \le N.$$

The encoder will see to it that $|\mathbf{Y}^t| = w_t$, $1 \le t \le T$. It does this by setting $\mathbf{X}^t$ equal to an N-vector which covers $\mathbf{Y}^{t-1}$ (so that $\mathbf{X}^t = \mathbf{Y}^t$) and for which $|\mathbf{X}^t| = |\mathbf{Y}^t| = w_t$. The encoding is done as follows. For $1 \le t \le T$, let $\{A_m^t\}$, $1 \le m \le M_t$, be a partition of $B_N(w_t)$. Thus, for $1 \le t \le T$,

$$A_m^t \subseteq B_N(w_t), \qquad 1 \le m \le M_t,$$

$$A_m^t \cap A_{m'}^t = \phi, \qquad m \ne m',$$

$$\sum_{m=1}^{M_t} A_m^t = B_N(w_t).$$

At the t-th write, the encoder observes $\mathbf{Y}^{t-1}$, and if $\mathbf{S}^t$ corresponds to message m, it searches A_m^t to find a vector that covers $\mathbf{Y}^{t-1}$. If it finds such a vector, $\mathbf{y}$, it sets $\mathbf{X}^t = \mathbf{Y}^t = \mathbf{y}$. The decoder can recover the message m by observing that $\mathbf{Y}^t \in A_m^t$. Also $|\mathbf{Y}^t| = w_t$. An error will occur if and only if no $\mathbf{y}$ which covers $\mathbf{Y}^{t-1}$ can be found in A_m^t.

For $1 \le t \le T$, $1 \le m \le M_t$, $\mathbf{u} \in B_N(w_{t-1})$, let $F(\mathbf{u}, A_m^t) = 0$ or 1 according as A_m^t contains a vector that covers $\mathbf{u}$. Clearly, we make no error for $1 \le t \le T$, $1 \le m \le M_t$, if

$$\psi \triangleq \sum_{t=1}^{T} \sum_{m=1}^{M_t} \sum_{\mathbf{u} \in B_N(w_{t-1})} F(\mathbf{u}, A_m^t) = 0. \tag{39}$$

Now turn to Theorem 2. Let T, $\epsilon > 0$, $\mathbf{r} \in \mathscr{R}_T$ be given. We will show that with N sufficiently large and with $\{w_t\}$ suitably chosen, and with $K_t/N = \log M_t = r_t - \epsilon$, that there exists a family of partitions

$\{A_m^t\}$ for which $\psi = 0$. Thus we will have shown that not only is $\mathbf{r} \in \mathscr{R}_T$ achievable, but that P_e^t can be made equal to 0. We do this by choosing the partitions $\{A_m^t\}$ at random (according to a probability law which we will specify later) and computing the expectation $E\,\psi$. We will show that $E\,\psi \to 0$ as $N \to \infty$. Since ψ is integer valued, when $E\,\psi < 1$, there must be a family of partitions for which $\psi = 0$.

Here is how the random partitions are chosen: For $1 \le t \le T$, pick a $\mathbf{v} \in B_N(w_t)$ and place it in class A_m^t with probability $1/M_t$ $(1 \le m \le M_t)$. Do this independently for each of the members of $B_N(w_t)$, and each t. Under this random experiment, ψ is a random variable and

$$E\,\psi = \sum_t \sum_m \sum_{\mathbf{u} \in B_N(w_{t-1})} E\,F(\mathbf{u}, A_m^t). \tag{40}$$

For fixed t, m, $\mathbf{u} \in B_N(w_{t-1})$,

$$\begin{aligned} E\,F(\mathbf{u}, A_m^t) &= \Pr \left\{ \begin{matrix} \text{for all } \mathbf{v} \in B_N(w_t) \text{ such that } \mathbf{v} > \mathbf{u}, \\ \mathbf{v} \notin A_m^t \end{matrix} \right\} \\ &= \left(1 - \frac{1}{M_t}\right)^{\nu_t} \le \exp\left\{\frac{-\nu_t}{M_t}\right\}, \end{aligned} \tag{41}$$

where ν_t is the number of vectors $\mathbf{v} \in B_N(w_t)$ which cover $\mathbf{u} \in B_N(w_{t-1})$. Since in choosing $\mathbf{v}$ to cover $\mathbf{u}$ we must place w_{t-1} 1's in those positions in which $\mathbf{u}$ is 1, and we can put the remaining $w_t - w_{t-1}$ 1's in any of the remaining $N - w_{t-1}$ positions, we have

$$\nu_t = \binom{N - w_{t-1}}{w_t - w_{t-1}}.$$

We now choose the $\{w_t\}$. Since $\mathbf{r} \in \mathscr{R}_T$, there must be a vector $\mathbf{p} = (p_1, \cdots, p_T)$ such that $\mathbf{r} \in \mathscr{R}_T(\mathbf{p})$. Let

$$w_t = N - Q_t N,$$

where $Q_t = \prod_{j=1}^t p_j$ (and $Q_0 = 1$). Then, as $N \to \infty$,

$$\nu_t = \binom{Q_{t-1}N}{Q_{t-1}(1 - p_t)N} = 2^{NQ_{t-1}h(1-p_t)+O(\log N)}. \tag{42}$$

Substituting (41) and (42) into (40) and using $M_t \le 2^N$, $|B_N(w_t)| \le 2^N$ we have

$$E\,\psi \le T 2^{2N} \exp\left\{\frac{-1}{M_t}\, 2^{NQ_{t-1}h(p_t)+O(\log N)}\right\}.$$

Setting $K_t/N = 1/N \log M_t = r_t - \epsilon \le Q_{t-1}h(p_t) - \epsilon$, we have

$$E\,\psi \le T \exp\{-2^{N\epsilon+o(N)}\} \to 0,$$

which is what we had to prove.

4.2 Random coding

In this section we state the well-known random channel-coding theorem[5] in a form that will enable us to establish our direct theorems for Cases 3 and 4 with little difficulty. Consider a discrete memoryless channel with input and output alphabets $\mathscr{X}$, $\mathscr{Y}$, respectively, and transition probability $P_c(y|x)$, $y \in \mathscr{Y}$, $x \in \mathscr{X}$. A *code* $\mathscr{S}$ with parameters N, M is a subset $\mathscr{S} = \{\mathbf{x}_1, \cdots, \mathbf{x}_M\} \subseteq \mathscr{X}^N$ with cardinality $|\mathscr{S}| = M$. The *maximum-likelihood* decoder is a mapping $F_D: Y^N \rightarrow \{1, 2, \cdots, M\}$ for which $f_D(\mathbf{y})$ = the smallest m such that

$$P_c^{(N)}(\mathbf{y}|\mathbf{x}_m) \geq P_c^{(N)}(\mathbf{y}|\mathbf{x}_{m'}), \qquad m' \neq m, \tag{43a}$$

where

$$P_c^{(N)}(\mathbf{y}|\mathbf{x}) = \prod_{n=1}^{N} P_c(y_n|\mathbf{x}_n), \tag{43b}$$

$\mathbf{y} = (y_1, \cdots, y_N) \in \mathscr{Y}^N$, $\mathbf{x} = (x_1, \cdots, x_N) \in \mathscr{X}^N$. Let $\Phi_m(\mathbf{y}, \mathscr{S}) = 0$ or 1 according as $f_D(\mathbf{y}) = m$ or $\neq m$. When each of the M-code vectors in $\mathscr{S}$ are used with equal probability, the "word" error probability is

$$P_e = \frac{1}{M} \sum_{m=1}^{M} \sum_{y \in \mathscr{Y}^N} P_c^{(N)}(\mathbf{y}|\mathbf{x}_m)\Phi_m(\mathbf{y}, \mathscr{S}). \tag{44}$$

Now let $p_0(x)$, $x \in \mathscr{X}$, be a probability distribution on $\mathscr{X}$, and let I_0 be the mutual information corresponding to the distribution $p_0(x)P_c(y|x)$ on $\mathscr{X} \times \mathscr{Y}$. A random-code ensemble is constructed as follows. Let the M-code vectors $\mathbf{X}_m$ in $\mathscr{S}$ be drawn independently with $\Pr\{\mathbf{X}_m = (x_1, \cdots, x_N)\} = \prod_{n=1}^{N} p_0(x_n)$. The quantity P_e in (44) is now a random variable which depends on the choice of $\mathscr{S}$. Write it as $P_e(\mathscr{S})$, and write its expectation $E\, P_e(\mathscr{S}) = g(N, M)$. Of course, $g(\cdot)$ depends on $p_0(\cdot)$ and $P_c(\cdot|\cdot)$ too. We now state the well-known random-coding theorem.[5]

Theorem 5: Let $P_c(\cdot|\cdot)$ and $p_0(\cdot)$ be given, and let $g(N, M)$ and I_0 be as defined above. Then, with $\mathscr{R} > 0$, held fixed,

$$g(N, 2^{NR}) \rightarrow 0, \text{ as } N \rightarrow \infty,$$

provided $R < I_0$.

We conclude from this theorem that provided N is sufficiently large, there exists at least one code $\mathscr{S}$ with parameters N and $M = 2^{RN}$ such that $P_e(\mathscr{S})$ is arbitrarily small.

4.3 Case 3 (encoder uninformed, decoder informed)

For a given N, T and encoder/decoder as defined in Section II, we can think of the information K_t-vector $\mathbf{S}^t$ as an integer in $\{1, 2, \cdots, M_t\}$, where $M_t = 2^{K_t}$, and set

$$\mathbf{x}_m^t = f_E^t(m), \quad 1 \le t \le T, \quad 1 \le m \le M_t. \tag{45}$$

Thus at the t-th generation, when the message is m, the encoder writes $\mathbf{x}_m^t$. Let $\mathscr{C}_t = \{\mathbf{x}_m^t\}_{m=1}^{M_t}$ be the "code" for the t-th generation, $1 \le t \le T$. The proofs for this theorem and the next depend on a random choice of $\{\mathscr{C}_t\}_{t=1}^T$. Here is a rough and imprecise sketch of the main idea.

Let $\{p_t\}_{t=1}^T$ satisfy $0 \le p_t \le 1$, $1 \le t \le T$. The codes $\{\mathscr{C}_t\}$ are chosen randomly and independently, according to the following probability law. Each of the M_t code vectors in $\mathscr{C}_t$ is chosen independently, with the probability that the mth code vector be $\mathbf{x} = (x_1, \cdots, x_N) \in \mathscr{X}^N$ is equal to $\prod_{n=1}^N p^{(t)}(x_n)$, where

$$p^{(t)}(0) = p_t, \qquad p^{(t)}(1) = 1 - p_t. \tag{46}$$

Now let us consider the t-th write/read generation. Prior to the t-th write, the n-th bit position will be a 0, i.e., $Y_{t-1,n} = 0$, if it was not written in *each* of the $(t-1)$ previous generations. In some "average" sense, this happens with probability $\prod_{j=1}^{t-1} p_t = Q_{t-1}$. Since the decoder at the t-th generation knows $Y_{t-1,n}$ and Y_{tn}, and it is impossible for $Y_{t-1,n} = 1$, $Y_{tn} = 0$, there are essentially three possible "outputs" $(Y_{t-1,n}, Y_{tn}) \triangleq Z_{tn}$. Z_{tn} can take the values: $a \triangleq (0, 0)$, $b \triangleq (0, 1)$, $c \triangleq (1, 1)$. If, for example, the channel input $X_{tn} = 0$, then

$$\Pr\{Z_{tn} = (0, 0) = a \mid X_{tn} = 0\} = \Pr\{Y_{t-1,n} = 0\} = Q_{t-1}.$$

Thus, as far as the t-th generation is concerned, $\mathbf{Z}^t$ is the output of the memoryless channel with input X, output Z and transition probability given by Fig. 1. The random-coding theorem suggests that, in the t-th generation, we can have highly reliable transmission provided that $M_t \le 2^{N(I_t-\epsilon)}$, where I_t is the $I(X; Z)$, which results when the input X to the channel in Fig. 1 has $\Pr\{X = 0\} = p_t$. Thus,

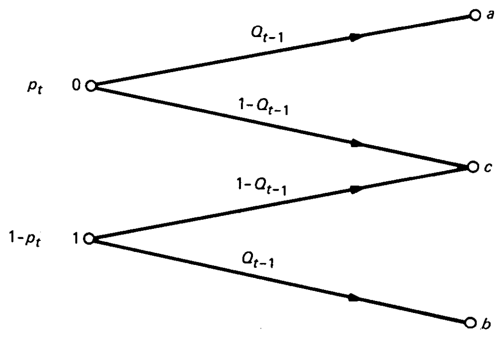


Fig. 1—Equivalent channel for Case 3.

$$I_t = I(X; Z) = H(Z) - H(Z|X)$$
$$= H(V, Z) - H(Z|X),$$

where $V = V(Z) = 1$ when $Z = a$ or b, and $V = 0$ when $Z = c$. Since $H(V) = h(Q_{t-1})$ and $H(Z|X) = h(Q_{t-1})$, we have

$$\begin{aligned} I_t &= H(V) + H(Z|V) - H(Z|X) \\ &= h(Q_{t-1}) + \Pr\{V = 0\}H(Z|V = 0) \\ &\quad + \Pr\{V = 1\}H(Z|V = 1) - h(Q_{t-1}) \\ &= \Pr\{V = 1\}H(Z|V = 1) = Q_{t-1}h(p_t). \end{aligned} \tag{47}$$

Thus we are led to conjecture that, for a given T and $\mathbf{p}$, any $\mathbf{r} \in \mathscr{R}_T(\mathrm{p})$ [given by (16)] is achievable for Case 3. We now proceed to a rigorous proof.

Again, consider an encoder/decoder with parameters N, T, $\{K_t\}$. Consider the t-th write/read generation. Suppose $\mathbf{S}^t = m$ $(1 \le m \le M_t)$, and consider

$$\Pr\{\mathbf{Y}^t = \mathbf{y}^t, \mathbf{Y}^{t-1} = \mathbf{y}^{t-1} | \mathbf{S}^t = m\}$$
$$= \Pr\{\mathbf{Y}^{t-1} = \mathbf{y}^{t-1}\}f(\mathbf{y}^t | \mathbf{y}^{t-1}\mathbf{x}_m^t), \tag{48a}$$

where $f(\mathbf{y}^t | \mathbf{y}^{t-1}, \mathbf{x}_m^t) = 1$ if

$$\mathbf{y}^t = \mathbf{y}^{t-1} \vee \mathbf{x}_m^t \tag{48b}$$

("V" is bitwise "inclusive or"), and $f = 0$ otherwise. Now $\Pr\{\mathbf{Y}^{t-1} = \mathbf{y}^{t-1}\}$ depends on the codes $\mathscr{C}_j$, $1 \le j \le t - 1$ (but not on $\mathscr{C}_t$). Let us make this dependence explicit and write

$$\Pr\{\mathbf{Y}^t = \mathbf{y}^t, \mathbf{Y}^{t-1} = \mathbf{y}^{t-1} | \mathbf{S}^t = m\}$$
$$= P(\mathbf{y}^t, \mathbf{y}^{t-1} | \mathbf{x}_m^t, \mathscr{C}_1, \cdots, \mathscr{C}_{t-1}). \tag{49}$$

Now, let $\mathbf{y}^t = (y_{t1}, y_{t2}, \cdots, y_{tN})$, and note that $(y_{t-1,n}, y_{t,n})$ cannot take the value (1, 0). Let us define $a = (0, 1)$, $b = (1, 0)$, $c = (1, 1)$, and let $\mathbf{z}^t = (z_{t1}, z_{t2}, \cdots, z_{tn})$, where

$$z_{tn} = (y_{t-1,n}, y_{tn}) \in \{a, b, c\} \triangleq \mathscr{Z}.$$

Now write (49) as

$$\Pr\{\mathbf{Y}^t = \mathbf{y}^t, \mathbf{Y}^{t-1} = \mathbf{y}^{t-1} | \mathbf{S}^t = m\} = P(\mathbf{z}^t | \mathbf{x}_m^t, \mathscr{C}_1, \cdots, \mathscr{C}_{t-1}). \tag{50}$$

Finally, let the vector $(p_1, p_2, \cdots, p_T)$ be arbitrary, $0 \le p_T \le 1$, $1 \le t \le T$. Let the codes $\mathscr{C}_1 \cdots \mathscr{C}_T$ be chosen independently according to the presciption given above eq. (46). Then (with $\mathbf{x}_m^t$ held fixed), the expectation

$$E\ P(\mathbf{z}^t | \mathbf{x}_m^t, \mathscr{C}_1, \cdots, \mathscr{C}_{t-1}) = P_c^{(N)}(\mathbf{z}^t | \mathbf{x}_m^t), \tag{51}$$

where $P_c^{(N)}(\mathbf{z}^t|\mathbf{x}_m^t)$ corresponds to the discrete memoryless channel with input alphabet $\mathscr{X} = \{0, 1\}$, and output alphabet $\mathscr{Z} = \{a, b, c\}$ and transition probability represented by Fig. 1.

Now consider the decoder at the t-th read generation. The decoder examines $\mathbf{Y}^t$, $\mathbf{Y}^{t-1}$. Let us use the following decoding rule. When $(\mathbf{Y}^t, \mathbf{Y}^{t-1}) = \mathbf{z} \in \mathscr{Z}^N$, let $f_D(\mathbf{z})$ be the smallest m, $1 \le m \le M_t$, such that

$$P_c^{(N)}(\mathbf{z}^t|\mathbf{x}_m^t) \ge P_c^{(N)}(\mathbf{z}^t|\mathbf{x}_{m'}^t), \qquad m' \ne m.$$

Let $\psi_m^t(\mathbf{z}^t, \mathscr{S}_t)$ be 0 or 1 according as $f_D(\mathbf{z}^t) = m$, $f_D(\mathbf{z}^t) \ne m$. Then the error probability* at the t-th generation (given $\mathbf{S}^t = m$) is, using (50),

$$\begin{aligned} P_{em}^t &\le \sum_{y^{t-1},y^t} \Pr\{\mathbf{Y}^{t-1} = \mathbf{y}^{t-1}, \mathbf{Y}^t = \mathbf{y}^t | \mathbf{S}^t = m\} \\ &= \sum_{\mathbf{z}^t \in \mathscr{Z}^N} P(\mathbf{z}^t|\mathbf{x}_m^t, \mathscr{S}_1, \cdots, \mathscr{S}_{t-1})\Psi_m^t(\mathbf{z}^t, \mathscr{S}_t), \end{aligned} \tag{52}$$

and the overall error probability is

$$\begin{aligned} P_e &= \sum_{t=1}^{T} \sum_{m=1}^{M_t} \frac{1}{M_t} P_{em}^t \\ &= \sum_{t=1}^{T} \sum_{m=1}^{M_t} \sum_{\mathbf{z}^t} \frac{1}{M_t} P(\mathbf{z}^t|\mathbf{x}_m^t, \mathscr{S}_1, \cdots \mathscr{S}_{t-1})\Psi_m^t(\mathbf{z}^t, \mathscr{S}_t). \end{aligned} \tag{53}$$

Taking the expectation of P_{em}^t over the random-code ensemble defined above, and noting that the random codes $\mathscr{S}_1, \cdots, \mathscr{S}_T$ are independent, we have from (51) and (53)

$$EP_e = \sum_{t=1}^{T} E\left[\frac{1}{M_t} \sum_{m=1}^{M_t} \sum_{z^t} P_c^{(N)}(\mathbf{z}^t|\mathbf{x}_m^t)\Psi_m^t(z^t, \mathscr{S}_t)\right], \tag{54}$$

where the expectation in the right number of (53) is taken with respect to $\mathscr{S}_t$. Applying Theorem 5, we conclude that for $\epsilon > 0$, this expectation $\rightarrow 0$, as $N \rightarrow \infty$, provided

$$M_t \le 2^{N(I_t - \epsilon)},$$

where $I_t = Q_{t-1}h(p_t)$. See (47). Thus for given T, $\mathbf{p} = (p_1, \cdots, p_T)$, we have established that $\mathbf{r} \in \mathscr{R}_T(\mathbf{p})$ is achievable. This is Theorem 1 for Case 3.

4.4 Case 4 (encoder and decoder uninformed)

In this section we establish Theorem 3 for the situation in Case 4 (encoder and decoder uninformed of the state of the memory at the previous generation). The proof is almost exactly the same as that of

* The right member of (52) is the so-called "word error probability", i.e., the probability that $\hat{S}^t \ne S^t$. P_e^t as defined by (9) $\le \Pr\{\hat{S}^t \ne S^t\}$.

Theorem 1 for Case 3, which was given in Section 4.3 (where only the encoder was uninformed).

Let T, $\mathbf{p} = (p_1, \cdots, p_T)(0 \le p_t \le 1)$ be given. The codes $\{\mathcal{C}_t\}_{t=1}^{T}$ are defined exactly as in Section 4.3, and we use a random-code ensemble exactly as above (46). Since the decoder at the t-th generation is uninformed of $\mathbf{Y}^{t-1}$, it must operate on $\mathbf{Y}^t$ instead of $\mathbf{Z}^t$ as in Case 3. This leads us to define the channel in Fig. 2 to replace the channel in Fig. 1, to define $P_c^{(N)}(\cdot \mid \cdot)$.

The rest of the proof parallels the proof in Section 4.3, but here

$$\begin{aligned} I_t = I(X; Y) &= H(Y) - H(Y|X) \\ &= h(p_tQ_{t-1}) - \Pr\{X = 0\}H(Y|X = 0) \\ &\quad - \Pr\{X = 1\}H(Y|X = 1) \\ &= h(p_tQ_{t-1}) - p_th(Q_{t-1}) = h(Q_t) - p_th(Q_{t-1}). \end{aligned}$$

Referring to (19a) and (19b) and (20) leads us to conclude that any $\mathbf{r} = (r_1, \cdots, r_T) \in \mathcal{R}'_T(\mathbf{p})$ is achievable, which is Theorem 3.

5. SOME AD-HOC RESULTS

Let us look at the family of achievable rates for Cases 1 through 3 when we impose the additional constraint that the rates at each generation be equal, i.e., that $R_t \equiv R$, $1 \le t \le T$. This is the case studied by Rivest and Shamir.[1] There is no closed-form expression for the maximum achievable R, but we can find it numerically as follows.

We seek a set $\{p_t\}$, $0 \le p_t \le 1$, $1 \le t \le T$, such that

$$R_t = h(p_t) \prod_{j=1}^{t-1} p_j \equiv R. \tag{55}$$

Thus if p_T is chosen, $R_{T-1} = R$ implies that

$$h(p_{T-1}) = p_{T-1}h(p_T),$$

for which there is exactly one solution for p_{T-1}. Further, $1/2 \le p_{T-1} \le 1$. Define $\alpha(\lambda)$, $1/2 \le \lambda \le 1$, as the unique solution of

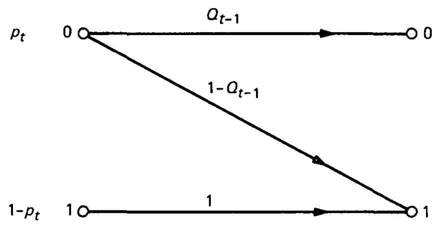


Fig. 2—Equivalent channel for Case 4.

$$h(\alpha) = \alpha h(\lambda). \tag{56}$$

We see that

$$\begin{aligned} p_{T-1} &= \alpha(p_T) \\ p_{T-2} &= \alpha(\alpha(p_T)) \\ &\vdots \\ p_1 &= \alpha^{(T-1)}(p_T), \end{aligned} \tag{57}$$

where $\alpha^{(k)}(\cdot)$ is the kth iterate of $\alpha(\cdot)$. Differentiating (56) with respect to λ yields

$$\frac{d\alpha(\lambda)}{d\lambda} = \left(\frac{d\lambda}{d\alpha}\right)^{-1} = \left(\frac{h'(\alpha) - h(\lambda)}{\alpha h'(\lambda)}\right)^{-1} > 0,$$

$1/2 \le \lambda \le 1$. Thus, $\alpha^{(T-1)}(\cdot)$ is monotonically increasing. Since $R_1 = h(p_1) = R$, we maximize R by minimizing $p_1 = \alpha^{(T-1)}(p_T)$. Thus choose $p_t = 1/2$.

The iterates $\alpha^{(k)}(1/2)$ can be obtained graphically or numerically in a straightforward manner. We obtained $\alpha^{(1)}(1/2) = 0.77291 \cdots$, $\alpha^{(2)}(1/2) = 0.83524$, $\alpha^{(3)}(1/2) = 0.86876$, $\alpha^{(4)}(1/2) = 0.89021$. Thus, under the constraint $R_t \equiv R$, we have for $T = 4$, $C = \sum_1^4 R_t = 4h(\alpha^{(4)}(1/2)) = 1.997 \cdots$, while the unconstrained total capacity is $\log(4 + 1) = 2.3219 \cdots$.

Several ad-hoc coding schemes were investigated for Case 2 where the encoder is informed of the previous contents of the WOM but the decoder is uninformed. Only the results for one of these schemes is repeated here.

The simplest case of the coding scheme to be discussed is to consider that the WOM is segmented into two equal-sized sub-WOMs, one for storing data and one for directing the decoder to the newly written data. In the first generation the encoder writes in the data-storing sub-WOM and the reader reads from that sub-WOM. For subsequent generations, the encoder does two write operations. It first copies the state of the data-storing sub-WOM into the second sub-WOM. It then writes new data, only in these positions of the data-storing sub-WOM in which there are zeros. By comparing the information in the two halves of the sub-WOM, the decoder knows in which positions (of the data-storing sub-WOM) the new data have been written.

Rather than optimize and analyze this simple case, we do this for a generalized version of this scheme. We take the N-bit WOM and subdivide it into $[N/K]$ K-bit bytes. At each generation, if a given byte is the all-zero sequence, the encoder can use it to convey new data. However, if it is any other sequence, the encoder nulls it by overwriting the all-one sequence in that portion of the memory. The decoder treats any byte other than the all-one byte as carrying new information.

The encoder uses the following scheme to write new information. On the t-th generation, $i = 1, 2, \cdots, T$, it writes the all-zero word with probability p_t and it writes any of the $(2^K - 2)$ words that are not all zero or all one with equal probability $(1 - p_t)/(2^K - 2)$. It does not use the all-one word to carry information.

The rate of information for the T generations is as follows:

$$R_1 = \frac{1}{K}\left[-p_1 \log p_1 - (1 - p_1)\log \frac{(1 - p_1)}{2^K - 2}\right] \triangleq \frac{1}{K}[k(p_1)],$$

$$R_t = \frac{p_1 p_2 \cdots p_{t-1}}{K} k(p_t), \qquad t = 2, 3, \cdots, T.$$

The total rate sum for this scheme is then

$$C_T = \frac{1}{K}[k(p_1) + p_1 k(p_2) + \cdots + (p_1 p_2 \cdots p_{T-1})k(p_T)].$$

In a manner similar to that used in Section I, one can prove that for a fixed K, the maximum C_T is obtained for

$$p_{T-i} = \frac{[(i - 1)(2^K - 2) + (2^K - 1)]}{i(2^K - 2) + (2^K - 1)},$$

resulting in a maximum C_t of

$$C_T = \frac{1}{K}\log(T(2^K - 2) + 1).$$

For $T = 2$ and 3, the largest values of C_T are obtained for $K = 3$, and for $T \geq 4$, the largest values of C_T are obtained for $K = 2$.

REFERENCES

1. R. Rivest and A. Shamir, "How to Reuse a Write-Once Memory," Inform. and Control, *55*, No. 1 (October 1982), pp. 1–19.
2. A. V. Kusnetsov and B. S. Tsybakov, "Coding in a Memory with Defective Cells," translated from Problemy Peredachi, Infromatsii, *10*, No. 2 (April–June 1974), pp. 52–60.
3. C. Heegard and A. El Gamal, "On the Capacity of a Computer Memory With Defects," IEEE Trans Inform. Theory, *IT-29*, No. 5 (September 1983), pp. 731–9.
4. C. Heegard, "On the Capacity of Permanent Memory," 1983 Conf. on Inform. Sciences and Systems, Johns Hopkins University (March 1983).
5. R. G. Gallager, *Information Theory and Reliable Communication*, New York: McGraw-Hill, 1968, Theorem 5.6.2, pp. 138.

AUTHORS

János Körner, Diploma in Mathematics, 1970, Loránd Eötvös University, Budapest. In 1970 he joined the Mathematical Institute of the Hungarian Academy of Sciences. In 1972 Mr. Körner was on leave at CISM, Udine, Italy; in the spring of 1974 he was a Visiting Professor at Ohio State University;

and in 1980 he was a Visiting Professor at Linkoping University, Sweden. From 1981–83 he was a visiting member of the Mathematics and Statistics Research Center of Bell Laboratories. He has co-authored with Imre Csiszar the book *Information Theory: Coding Theorems for Discrete Memoryless Systems* (Academic Press, 1982). His current research interests are in information theory and its interplay with combinatorics. János Körner currently is serving as an Associate Editor for the IEEE Transactions on Information Theory.

Jack K. Wolf, B.S.E.E., 1956, University of Pennsylvania; M.S.E., M.A., and Ph.D., 1957, 1958, and 1960, respectively, Princeton University; New York University 1963–65; Polytechnic Institute of Brooklyn, 1965–73; University of Massachusetts, 1973—. During the academic year, 1968–69, Mr. Wolf was a member of the Mathematics Research Center, Bell Laboratories. Presently he is a Professor of Electrical and Computer Engineering at the University of Massachusetts, Amherst. His research interests are in information theory, algebraic coding theory, communication systems, and computer networks. He is also currently International Chairman, Commission C, URSI. Editor Transactions on Information Theory, Algebraic Coding, 1969–72. Board of Governors, Information Theory Group 1970–76, 1980—.

Aaron D. Wyner, B.S., 1960, Queens College; B.S.E.E., M.S., Ph.D., 1960, 1961, and 1963, respectively, Columbia University; AT&T Bell Laboratories, 1963—. Mr. Wyner has been doing research in various aspects of information and communication theory and related mathematical problems. He is presently Head of the Communications Analysis Research Department. He spent the year 1969–70 visiting the Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, and the Faculty of Electrical Engineering, at Technion, Haifa, Israel, on a Guggenheim Foundation Fellowship. He has also been a full- and part-time faculty member at Columbia University, Princeton University, and the Polytechnic Institute of Brooklyn. Chairman of the Metropolitan New York Chapter of the IEEE Information Theory Group, Associate Editor of the Group's Transactions, and co-chairperson of two international symposia. President, IEEE Information Theory Group, 1976. Fellow, IEEE Member, AAAS, Tau Beta Pi, Eta Kappa Nu, Sigma Xi. Since September 1983, he has been Editor-in-Chief of the IEEE Transactions on Information Theory.

Jacob Ziv, B.Sc., Dipl. Eng., and M.Sc., all in Electrical Engineering, from the Technion–Israel Institute of Technology 1954, 1955, and 1957, respectively; D.Sc., 1962, Massachusetts Institute of Technology; Senior Research Engineer in the Scientific Department, Israel Ministry of Defense, 1955–9; Applied Science Division of Melpar, 1961–62. In 1962 he returned to the Scientific Department, Israel Ministry of Defense, as Head of the Communications Division and was also an Adjunct of the Faculty of Electrical Engineering, Technion–Israel Institute of Technology. Member of the Technical Staff of Bell Laboratories, 1968–70. He joined the Technion in 1970 and is a Herman Gross Professor of Electrical Engineering. Dean of the Faculty of Electrical Engineering, 1974–76, and Vice President for Academic Affairs, 1978–82. Member Israeli Academy of Science, 1981; fellow, IEEE. From 1977 to 1978, and 1982 to 1983, he was on sabbatical leave at Bell Laboratories. His research interests include general topics in information theory and statistical communication.

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Local Area Data Transport Service Overview

By M. N. RANSOM*

(Manuscript received October 25, 1983)

A new packet data communication service known as Local Area Data Transport (LADT) has been recently introduced. Combining new loop electronics technology with packet switching, LADT provides customers with low-cost data communications within local access and transport areas, as well as access to interexchange data communications networks. This paper describes the need for LADT, its goals, architecture, and services, and also serves as an introduction to the following paper on LADT system hardware and software.

I. THE NEED FOR A LOCAL PACKET-SWITCHING NETWORK

1.1 Data communications applications

The communications industry is currently experiencing a large growth in new data services. Examples are extensive. On-line credit-checking terminals are commonplace. Entrepreneurs have set up specialized information databases providing their subscribers with electronic access to economic forecasts and stock market information. Home information services, such as videotex, are beginning on a national and international scale.[1] The trend for corporate communications is to bring more users "on-line" by providing terminals for immediate access to corporate databases and to automate current paper-flow processes.[2] The current growth in personal computing also is expected to require data communications for the exchange of electronic mail and the transfer of software.[3] Current centrex customers are demanding new data communications capabilities to handle their needs more effectively.

* AT&T Bell Laboratories.

Consider some specific examples of information services already in place:

1. The *Viewtron** service, provided jointly by Knight-Ridder Newspapers, Southern Bell, and AT&T Consumer Products. This system provides videotex services in South Florida.[4]
2. The *Pronto*‡ home banking service, provided by Chemical Bank in the New York City area.[5]
3. The *Extravision*§ service, provided by CBS, AT&T Consumer Products, and New Jersey Bell, to supply home information services.[6]
4. The *Source*¶ owned by Reader's Digest, which provides news and specialized information to its subscribers.
5. *Dow Jones News/Retrieval Service*,** which provides business news and financial data to subscribers in an on-line fashion.[7]
6. The Mead Corporation, which provides bibliographical and on-line text retrieval services.

The exact nature of future applications is uncertain, but what we can expect are more applications by more information providers and, with them, a dramatic growth in data communications.[8] One common element these services will share is a need for an effective data communications network. The characteristics of data traffic are so significantly different from those of voice that use of the voice switching network is a poor match to this need. On the other hand, the cost of providing a separate data network for each type of service would be prohibitive.

Possible solutions to this problem include the use of two-way cable systems, hybrid cable and telephone systems, and use of other local distribution systems such as radio. The telephone operating companies, because of their in-place loop plant, have a unique opportunity to serve this market. LADT takes advantage of this opportunity by adding data capabilities over existing loop facilities through the addition of new loop technology. LADT is planned as part of the overall evolution of the telephone network, and is oriented toward interactive, packet-switching applications.

1.2 Data communications characteristics

As contrasted with voice, data communications has several differentiating characteristics:

1. The fundamental information content is digital in nature.

---

* Service mark of Viewdata Corporation of America Inc.
‡ Service mark of Chemical Bank of New York.
§ Service mark of CBS Inc.
¶ Service mark of The Source Telecomputing.
** Service mark of Dow Jones & Co., Inc.

2. Data communications is very sensitive to errors in this digital information.

3. Precisely defined rules and procedures are required in the form of protocols because of the machine-to-machine nature of the communications.

4. A very wide range of data traffic is typical, ranging from message sizes of a single bit, as in sensor applications, to megabits, as in facsimile or file transfers. Holding times for these types of calls can range from less than a second for a meter-reading application, to over an hour for some business applications.

5. Interactive data applications are usually characterized by bursty data traffic—long holding time calls with an average data rate a small fraction of their peak data rate.

1.3 Technical approaches to data communications: status quo

One of the early technical approaches to data communications was to treat information as if it were voice. This approach led to the development of modems (e.g., *DATAPHONE** II data sets[9]). In fact, this approach has a number of advantages. It produced an immediate, ubiquitous network for switched- and private-line applications. Data communications equipment can be placed selectively in the network, allowing data networks to be built with relatively low start-up costs. Finally, the technique is transparent to data content and format. This approach, however, has limitations. Transmission rates are limited to bit rates that can be reliably transmitted over a voiceband channel. With current technology, transmission rates are typically up to 1.2 kb/s full duplex over the switched network and up to 9.6 kb/s over four-wire private lines. Customers who opt for private-line service have the additional disadvantage of homing a particular terminal to a single-host computer. Source and destination data equipment must operate at the same transmission rate. Central computing sites that must communicate with many remote locations require a proliferation of computer front-end ports, modems, and lines. Use of a voice line for data precludes its simultaneous use for voice. This is undesirable because of the long holding time of data calls and because some applications (e.g., security) must be able to transmit data at any time. Finally, and perhaps most importantly to the telephone operating companies, many applications make inefficient use of the public switched telephone network because of very long or very short holding times and because of the long idle periods occurring in interactive data calls.

* Trademark of AT&T Technologies, Inc.

1.4 The local area data networking opportunity

Over the last ten years, a number of specialized value-added networks have emerged to address this data communications market.[10,11] Access to customers of these networks is usually provided through the local telephone network. This approach has proven attractive from a customer cost point of view because telephone operating companies have typically charged on a flat-rate basis for local calls. As measured usage is introduced, however, the cost of using this local network for data will become visible to customers.

The telephone operating companies, because of their extensive investment in the local-distribution network, are in a good position to provide effective, low-cost access to these interexchange data networks. By augmenting local loops with electronics, simultaneous voice and data service can be provided to subscribers. By combining this loop technology with packet-switching technology,[12] subscribers can not only be provided access to multiple interexchange data networks, but can also be provided access to a variety of intra-LATA (Local Access and Transport Area) data services. Such an arrangement achieves a more cost-effective way of handling bursty data applications, can eliminate the problems of speed matching the source and destination, and can provide a multiplexed access to central-site computing centers. The essential decision is whether to integrate these functions into existing voice-switching machines or to augment the existing network with new, separate components. The concept of a *packet overlay* network has been chosen to allow the telephone operating companies to initiate new data services independent of the particular voice switches currently in place. These technologies and this approach are the basis for LADT.

II. LADT SYSTEM CHARACTERISTICS

2.1 Network architecture

With the basic ingredients of subscriber loop multiplexing and packet switching, a basic network architectural plan for LADT was conceived. This architecture is depicted in Fig. 1. Two methods are provided for accessing LADT: direct access and dial-up access. The direct-access method allows voice and data service to be provided simultaneously over a single pair of wires to the subscriber. This pair of wires terminates on a Local Data Concentrator which demultiplexes the voice and data. Voice is sent in standard voice frequency format to the voice switch, while the subscriber's data are sent to a packet switch. Figure 2 shows the direct-access technique used in LADT in more detail. Data and voice are multiplexed together at Network Circuit Terminating Equipment (NCTE) at the customer premises.

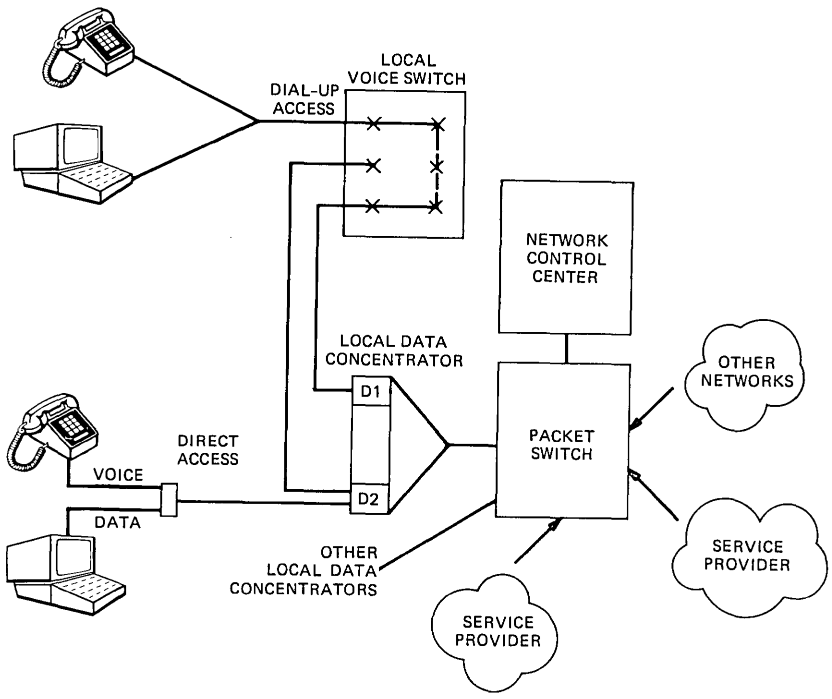


Fig. 1—LADT network architecture.

Full-duplex 4.8-kb/s data are sent above the voice spectrum over the subscriber's loop. This technique allows use of existing nonloaded loop plant up to 18 kft in length, achieves cost reductions relative to use of separate access lines with modems, and allows simultaneous use of voice and data services.

Dial-up access for low-volume users is also shown in Fig. 1. Such users access the LADT network through the current telephone network. While this results in data traffic being passed through voice switches, the number of voice switches in the transmission path is reduced relative to current value-added networks through the deployment of Local Data Concentrators. Nonetheless, customers using dial-up access cannot make normal use of their line at the same time they are using their line for data and are also limited to 1.2-kb/s service. Since this method of access may involve a longer path to the LADT network than for direct access, a more robust modulation technique is needed. Because of the widespread acceptance of the modulation technique used in the 212A data set,[13] this modem technology is used for dial-up access.

Two functions performed by the local data concentrator have just been noted: separating voice and data for direct-access subscribers,

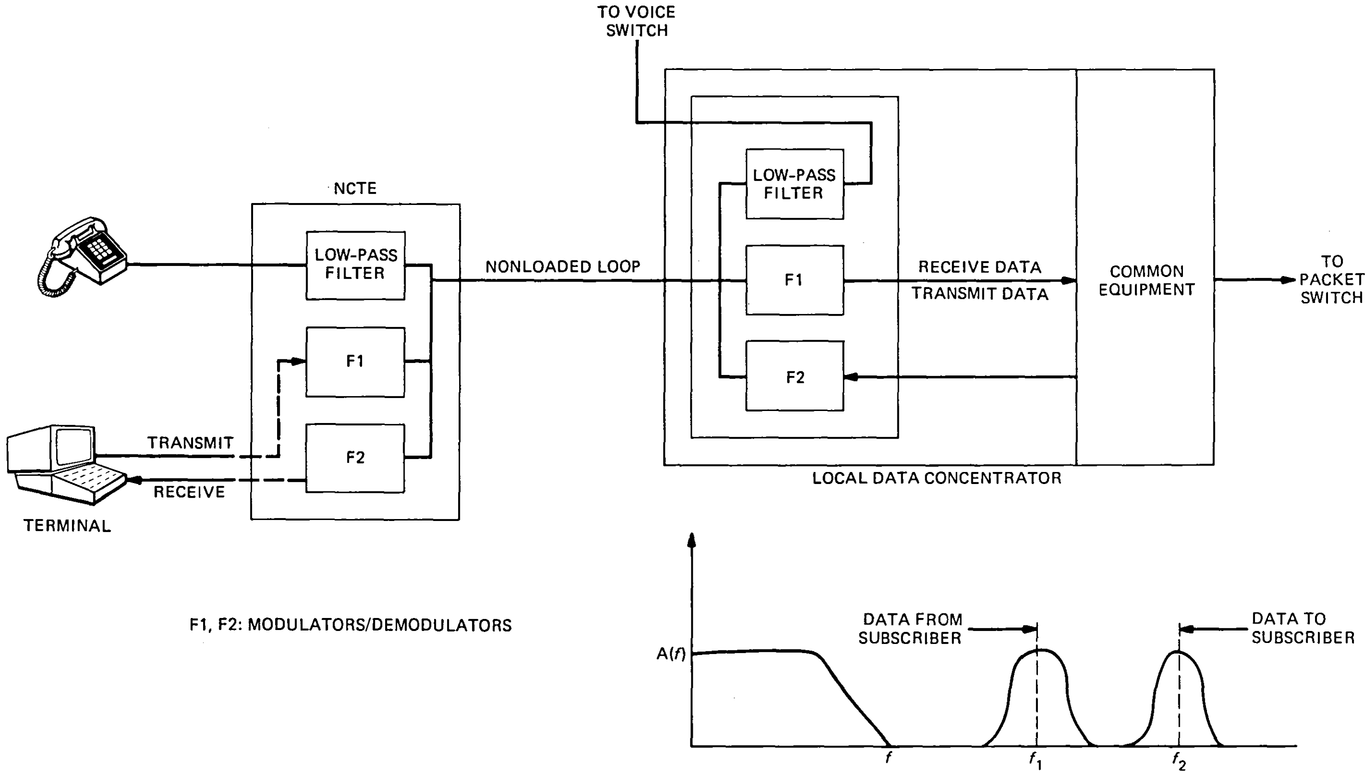


Fig. 2—Data over voice technique.

and terminating calls from dial-up access subscribers. The local data concentrator performs a number of other functions. It checks data received from subscribers for transmission errors and, if necessary, requests retransmission. To reduce transmission costs, data from many customers are concentrated by the Local Data Concentrator onto a 56-kb/s data facility to a packet switch. This greatly reduces costs relative to providing individual data facilities to the packet switch for each data call. The local data concentrator also provides a number of per call administrative functions, such as billing, traffic, and error measurements, relieving the packet switch of some of these functions.

The packet switch performs a number of important functions for subscribers and hosts. From the subscriber's point of view, the packet switch allows data calls to be connected to a large number of hosts. From a service-provider point of view, the packet switch provides a second level of subscriber concentration, allowing a single data link to the host to be accessed from subscribers throughout the LATA. The multiplexed network interface to the host allows up to 511 active terminals to be served by a single 56-kb/s *DATAPHONE* Digital Service (DDS) channel.[14] This eliminates the need for a separate modem and front-end processor port for each active line. It is expected that most LATAs will be served by a single packet switch, although large LATAs may eventually be served by multiple-packet switches.

In addition to direct connection to hosts, LADT will provide connections to inter-LATA data networks, thereby allowing subscribers to access hosts throughout the nation and, eventually, through international data networks.

A Network Control Center (NCC) will provide a centralized point for monitoring and controlling the operation of LADT throughout the LATA. It consists of a set of operations, administrative, and maintenance functions which are best located centrally in a LATA rather than distributed to each local data concentrator. In some implementations of LADT, the NCC may be combined into the packet switch.

2.2 Protocol standardization

LADT has been designed to use existing international protocol standards where available. This allows compatibility with data equipment being marketed internationally and allows internetwork and inter-LATA communications to be easily supported in the future. Current standardization activities are focused on the Open Systems Interconnect (OSI) model[15] written by the International Organization for Standardization and on the X.25 protocol[16] written by the International Telegraph and Telephone Consultative Committee (CCITT). X.25 implements the first three levels of the OSI model. LADT has chosen to support the X.25 protocol. At the terminal interface, the link level of X.25 is supported with ASCII text call progress messages. At the host interface, all three levels of X.25 are supported. Details of the LADT terminal and host interfaces are described in Sections III and IV, respectively.

2.3 Uniform numbering plan

International standards organizations, in particular the CCITT, have provided standards for data-network numbering to facilitate internetwork communications and addressing. The primary goal of the numbering plan standards is to specify the structure and coding of the international numbering plans for data networks. To do this, standard X.121 has been developed.[17] This standard specifies a numbering structure that allows a maximum of 14 digits for data-network numbering. The first four digits are termed a Data Network Identification Code (DNIC). DNICs identify the country and data network within the country. Number assignment of DNICs is administered by CCITT.

The remaining digits of the data-network number are termed a Network Terminal Number (NTN). The NTN is required to be numeric and to be at most 10 digits in length.

The LADT service has been assigned a single DNIC by the U.S. State Department/CCITT. This means that the structuring of the 10-digit NTN must allow routing to the appropriate LATA. This has two primary effects. The first is that the interexchange carriers must be aware of the NTN structure of LADT to handle inter-LATA calls. The second is that the individual telephone operating companies must use a standard NTN structure to support network-wide communications.

The LADT service has been designed to allow data communications worldwide. A 10-digit LADT numbering plan will be used, similar in structure to the today's telephone numbering plan, although independent of it. LADT NTN numbers have the following structure:

$$\mathrm{NPA}\text{-}NXX\text{-}XXXX,$$

where NPA denotes the Numbering Plan Area codes used in the public switched telephone network, N is an digit from 2 to 9, and X any digit from 0 to 9.

2.4 Performance objectives

A number of performance objectives have been set for LADT, assuring customers satisfactory service for a wide variety of data applications. LADT is to provide 24-hour-a-day, 7-day-a-week service without scheduled downtime. It is to be highly reliable with the service available to a given subscriber 99.6 percent of the time. The probability of a subscriber's request for service being denied because of lack of system resources during the busy hour is to be less than 1 percent except during the 10 busiest days of the year. The probability of an error being introduced into customer's data by the network is to be less than 1 in 10^8 packets sent. LADT introduces very little delay into the connection between the subscriber and host. The delay between the time LADT receives a packet from the subscriber and the time that packet is placed in the output queue for transmission to the host is to be less than 200 ms. This delay is defined as the time interval from the receipt of the last character of information from the source until the first character of that packet is ready for delivery to the destination. This definition eliminates from the performance objective the effects of customer choice of access line speeds and traffic loading per host X.25 line.

2.5 Administrative capabilities

It is the goal of LADT that the entire network, both local data concentrators and packet switch, be administered from a single location. Initially, billing information for dial-in subscribers, dedicated subscribers, and hosts will be collected through separate mechanisms. The collection of billing information for dial-in subscribers will not be done by LADT equipment. Instead, dial-in subscribers are billed according to the number of calls placed and holding time using standard automatic message accounting records. Billing information for direct-access subscribers is collected by the local data concentrators and sent to the NCC after each call. Billing information for hosts is collected by the packet switch. The eventual goal of LADT is for this billing information to be sent by a data link to the operating company revenue accounting office. In the initial implementation of LADT, this billing information is transferred by magnetic tape.

Traffic reports generated at the NCC allow operations personnel to monitor the functioning of the system and to engineer the system for the prevailing traffic load. Capabilities are also to be provided at the NCC to allow operations personnel to change such system data as equipment configuration data, customer profile data, and routing tables.

III. LADT TERMINAL INTERFACE

The OSI model, referred to in Section 2.2, provides a convenient and useful model for defining the interface between data equipment. It defines this interface in seven protocol layers or levels, each built on the previous level (see Fig. 3). Public networks, such as LADT, generally implement the first three levels of this model: the physical,

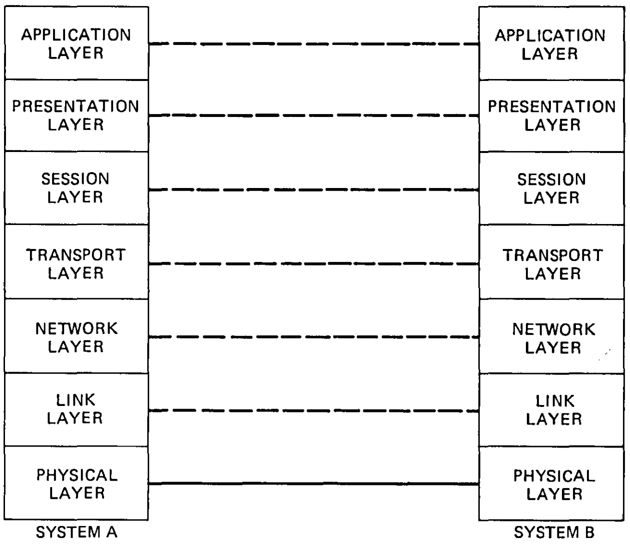


Fig. 3—Open systems interconnect model.

link, and network levels. Both the terminal and host interface to LADT[18] will be described according to this model.

3.1 Physical-level interface

The physical level defines the electrical, mechanical, and procedural control characteristics of the transmission facilities that provide access to customers. As we discussed earlier, both direct- and dial-access interfaces to LADT are provided. Two physical-level protocols are therefore supported. For direct-access subscribers, the NCTE is provided at the customer premises. This device forms the customer's interface to the network. The NCTE is depicted in Fig. 4. The NCTE voice interface is a tip and ring connection. A data interface has been chosen to be compatible with the *DATAPHONE* Digital Service and the circuit switched digital capability. This interface is provided with an 8-pin connector. The data interface provides full-duplex 4.8-kb/s synchronous transmission. Of the eight pins provided in the data connector, four are currently assigned. Two leads provide balanced transmit data and two leads provide balanced receive data. Baseband bipolar return-to-zero signaling (50-percent duty cycle) is used for transmission of data to and from the NCTE and is described by the following coding rules: A binary 0 is transmitted as 0.0 volt. A binary 1 is transmitted as either a positive or negative pulse, opposite in polarity to the previous pulse. This is the alternate polarity rule. An example of bipolar signaling is shown in Fig. 5.

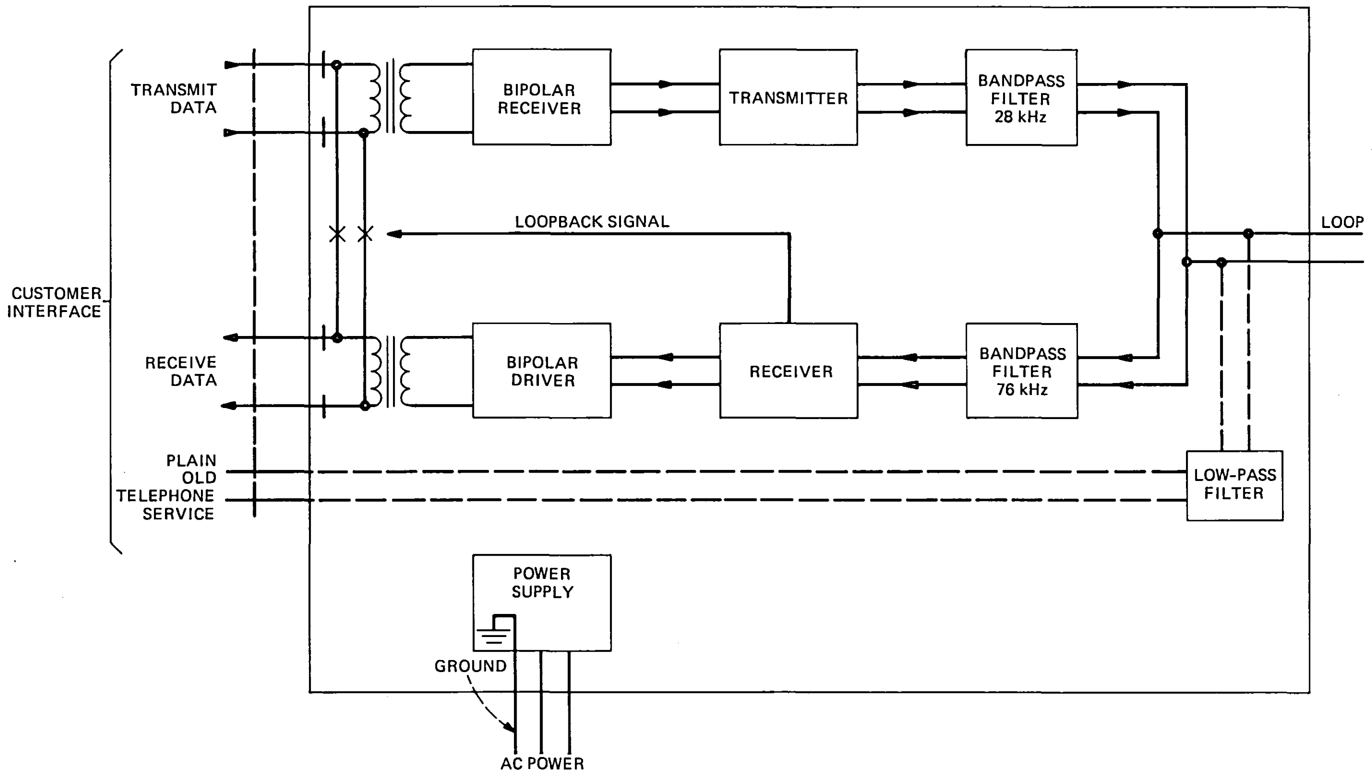


Fig. 4—LADT network circuit terminating equipment.

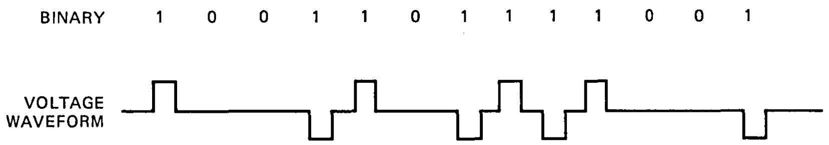


Fig. 5—Bipolar signaling.

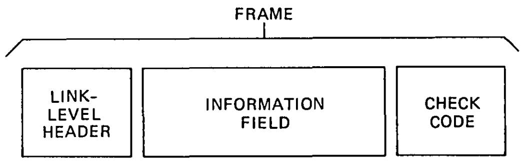


Fig. 6—LAPB frame structure.

When connected to the direct-access interface, the terminal equipment must perform the timing recovery and coding and decoding of data. Timing recovery is required to enable the terminal equipment to sample correctly the incoming synchronous data and to clock the terminal equipment's transmit data to the NCTE. Coding and decoding data involve using the bipolar coding rules.

Dial access to LADT is supported via a 212A compatible interface. Full-duplex 1.2-kb/s bit-synchronous transmission is supported. The dial-access port of LADT automatically answers incoming calls. Answer tone is provided to notify the user that a port has been connected.

3.2 Link-/network-level interface

The link- and network-level interface protocols supported by LADT are the same for direct and dial-up subscribers. The link-level protocol used in LADT is the X.25 Link Access Procedure B (LAPB) protocol. LAPB provides the subscriber an essentially error-free data channel through the use of error detection and retransmission. This is important for applications such as electronic funds transfer. Both information and control are transferred across the access link in information units called frames. The link level forms information frames, as shown in Fig. 6. A header in the frame contains such information as what type of frame it is (an information frame or a control frame). It also contains a sequence number so that lost frames can be detected. The header also contains acknowledgment information for frames received from the other end. A 16-bit cyclic redundancy code is added to each frame to detect transmission errors.

The values of the link-level parameters in the network implementation are as follows:

1. The link-level window size, LAPB parameter k, is two frames. This allows both the network and the terminal to send a second frame without having to wait for the first one to be acknowledged.

2. The acknowledgment timer, parameter $T1$, is 5.0 seconds. This is how long the network will wait for an acknowledgment before assuming the frame was lost and initiating recovery action. Timer $T1$ is started at the end of the transmission of a frame. Therefore, the terminal should not delay the response to a frame by more than $T1$ minus $T2$, where $T2$ is total of the round-trip propagation delay of the access line plus any processing time required by the network. The value of $T2$ for the network will not exceed 0.3 second. In addition, the network will not delay the response to a received frame by more than 0.3 second, including round-trip propagation delay for an access line.

3. The maximum number of attempts to obtain an appropriate response to a transmitted frame, parameter $N2$, is 4. After $N2$ unsuccessful attempts, the network will initiate the appropriate link-level recovery, as specified in X.25. Also, if the link cannot be restored in $N2$ attempts, the network will clear the virtual call on the link, if one exists.

4. The maximum number of bits in an information frame (excluding flags and 0 bits inserted for transparency), parameter $N1$, is 2080 bits (260 octets). The information field must contain an integral number of octets. If the terminal transmits an information frame whose information field exceeds this length, the network will transmit a frame reject response. If the terminal transmits an information frame whose information field is not an integral number of octets in length, the network will discard the frame without acknowledging it.

The LADT network-level access protocol provides the interface procedures required to set up, maintain (i.e., control the transfer of data), and clear virtual calls. To reduce the level of complexity of terminal protocol software, and because X.25 does not currently support dial-up protocols, LADT provides simple network-level messages in the ASCII code set. A single logical channel is supported to the subscriber. Data are transferred across this interface using information frames. Control information is transferred across the interface using several types of link-level frames.

Network-level signaling consists of signaling from the network to the terminal using signaling messages and signaling from the terminal to the network using a destination address. An information frame can contain one or two signaling messages. All signaling messages and destination addresses are coded using the ASCII code set complying with ANSI X3.4.[19] The network sets the value of the most significant (parity) bit to 0 when transmitting a signaling message to the terminal.

The terminal may set the value of the most significant (parity) bit to either 0 or 1 when transmitting the destination address to the network; the network ignores the parity bit.

Figure 7 shows an example of the signaling message which prompts for the called number and shows the ASCII coding. An example of how these signaling messages are used to set up data calls in LADT is given in Section V.

IV. LADT HOST INTERFACE

As we stated earlier, LADT will support all three levels of the X.25 protocol at the host interface. Reference 20 gives specifications of this interface.

Host subscribers interface with LADT directly to the packet switch using one or more channels. The *DDS* channels are supported at 2.4-, 4.8-, 9.6-, and 56-kb/s transmission speeds. The lower-speed interfaces may be adequate for situations where only a small number of simultaneous sessions are required. However, for interactive data applications such as videotex, the information database is typically provided on a large processor and the traffic load per active subscriber is relatively low. Therefore, there may be hundreds or even thousands of simultaneous sessions per information provider. For this kind of situation, the high-speed 56-kb/s interface would be used.

LADT supports at the host interface the same balanced LAPB supported on the terminal interface. Unlike the network-level protocol at the terminal interface, which provides only a single logical channel, the network level of X.25 at the host interface allows multiple, independent logical channels to be provided on a single physical link. This is illustrated in Fig. 8. This multiple logical channel capability allows customers to replace many individual front-end processor ports and modems with a single higher-speed digital channel and interface port

OCTETS

1	2	3	4	5	6	7	8	9	10	11
3/0	3/0	0/13	0/10	2/10	2/0	4/14	5/4	4/13	4/2	4/5
0	0	CARRIAGE RETURN	LINE FEED	*	SPACE	N	U	M	B	E

12	13	14	15	16	17	18	19	20	21	22
5/2	2/0	5/0	4/12	4/5	4/1	5/3	4/5	3/10	0/13	0/10
R	SPACE	P	L	E	A	S	E	:	CARRIAGE RETURN	LINE FEED

Fig. 7—Signaling message to prompt subscriber for called number.

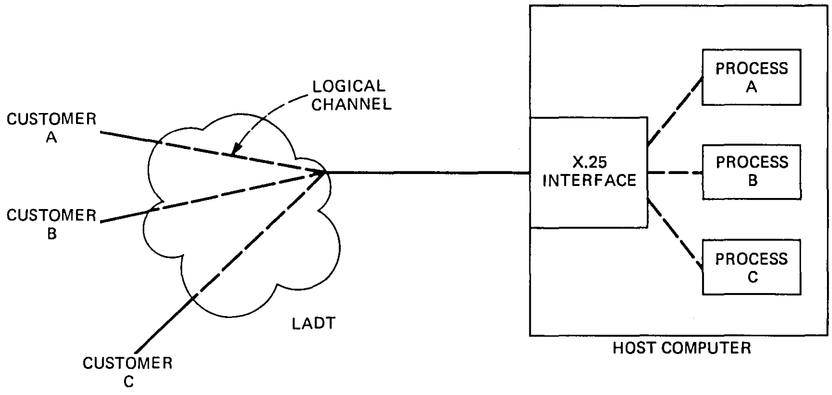


Fig. 8—Multiple logical channels on one host link.

to the host. LADT supports up to 511 logical channels over a single 56-kb/s access line.

The X.25 interface allows the customer to use these logical channels statically as permanent virtual circuits or dynamically as virtual calls. The permanent virtual-circuit interface is meant to emulate a point-to-point private line for applications that do not require switching. Virtual calls are used when the user wishes to use logical channels dynamically. A combination of permanent virtual circuits and virtual calls may be served over a single physical interface.

The LADT host interface allows customers to choose several packet-level facilities to meet particular application needs. The remainder of this section describes some of these facilities.

4.1 Hunt groups

Even though LADT allows many data calls to be set up on a single data link to a host, for very large hosts many data links will be needed to handle the volume of data traffic. For these situations LADT provides a hunt-group capability. A data network number is assigned to the data links making up the hunt group. Incoming calls are assigned to links by LADT in such a way as to try to have equal numbers of data calls on each link.

4.2 Conditional removal of data links

There may be times when a host may want to remove a data link from service but only after all existing data calls have terminated. To do so, LADT provides a conditional removal capability. When requested to conditionally remove a data link from service, LADT will then allow existing calls to remain on the link but will not allow new incoming calls to be established.

4.3 Packet size selection

Packet-switching services like LADT require network resources on a per-packet basis. The amount of processing required to switch a packet is relatively independent of the actual number of bits in the packet. Thus, users who must send large amounts of data can most efficiently transfer their data by using large-size packets. LADT allows users to select a maximum packet size of 128 octets or 256 octets of user data. Given equal cost per packet, the user with large amounts of data to send can more effectively use the switch by selecting a 256-octet maximum.

4.4 Throughput and flow control

Virtual-circuit throughput is an important measure of packet-switching service. Because the switches and trunks are statistically shared, brief storage of data from a particular virtual circuit is needed. Thus, packet buffers are allocated for this purpose. However, the system must protect itself from individual users taking an excessive share of these packet buffers. LADT does this by requiring that when an individual virtual circuit has used all of its share of packet buffers, it is prohibited from sending any additional packets until the destination accepts at least the first in sequence. The LADT X.25 interface allows users to select packet-level window sizes of two or three packets. This allows a virtual-circuit throughput of up to 9.6 kb/s.

4.5 Closed user groups

Closed user groups are supported on the X.25 interface to provide security. By means of this feature, only calls from certain locations are allowed to terminate on a host.

Closed user groups are implemented with a mechanism similar to a key and lock. The key, called a closed user group number, is sent with the call setup packet. If it matches the closed user group set allowed at the destination, the call setup packet is sent to the destination. If there is no match, the call is blocked.

4.6 Fast select

The fast-select facility is provided on the X.25 interface. It allows the customer to transmit data as the call is established or torn down. This is done by allowing up to 128 octets of user data to be sent along with a call request packet. Likewise, the called party may send up to 128 octets of data while issuing a request to clear the call.

4.7 Data connections between host and terminals

Using the LADT host interface, hosts can set up data calls not only to terminals, but to other hosts as well. When connected to another host, all X.25 capabilities that the other host supports can be used. When connected to terminals, however, certain limitations apply. This is because LADT supports a simple network-level interface to terminals, and not all X.25 capabilities can be transferred across this interface. For instance, the X.25 qualifier bit loses its significance on such connections. Reference 21 describes host requirements when connected to terminals on LADT. These are summarized in Table I.

V. CALL HANDLING

Let us now examine how data calls are handled in LADT. As an example we will describe a data call from a dial-up LADT subscriber to a host. This is illustrated in Fig. 9. A call from a subscriber with direct access occurs in the same manner except that the need for initially placing a telephone call is eliminated.

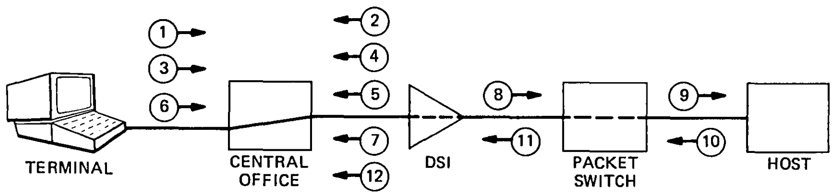
The subscriber begins by placing a telephone call to a special directory number associated with LADT. This will result in the call being routed to the nearest local data concentrator (quite likely one in the same central office as the subscriber). Upon detecting ringing, the local data concentrator answers the call and sends an answer tone to the subscriber. This tone is recognized by the subscriber's terminal (or data set), which responds by sending data carrier. This is detected by the local data concentrator, which then returns data carrier itself and begins establishing the X.25 LAPB protocol with the terminal. After this protocol is established, the local data concentrator sends the message

```
NUMBER PLEASE:
```

which is printed on the screen of the subscriber's terminal. The

Table I—Summary of use of services, facilities, and subscription items on host interface to provide compatibility with terminal interface

Service, Facility, or Subscription Item	Use
Specify the desired line transmission rate	Specify
Specify the desired number of logical channels and the range of logical channel numbers	Specify
Virtual-call Service	Mandatory
Flow-control parameter negotiation	Mandatory
Incoming calls barred	Precluded
Closed user group	Precluded
Multiple addresses on an access line	Optional
Multiple-line hunt group	Optional
One-way logical channel outgoing	Optional
Outgoing calls barred	Optional
Throughput class negotiation	Optional
Fast select	Optional
Fast select acceptance	Optional
Permanent virtual circuit service	Optional



1	SUBSCRIBER DIALS LADT
2	DSI RETURNS ANSWER TONE
3, 4	DSI AND TERMINAL EXCHANGE CARRIER
5	DSI PROMPTS FOR CALLING NUMBER
6	CUSTOMER ENTERS CALLING NUMBER
7	DSI INDICATES CALL SETUP IN PROGRESS
8 TO 11	DATA CALL SETUP TO HOST
12	DSI INDICATES CALL IS NOW SET UP

Fig. 9—Call setup example.

subscriber responds by entering the data network number of the desired host. The local data concentrator does preliminary screening on the number entered to ensure that it appears reasonable (e.g., has the right number of digits) and sends the message

```
CALL BEING ATTEMPTED
```

to the subscriber. At the same time it takes the called number and formats it into an X.25 call request packet, which it sends to the packet switch. The packet switch translates the called number and sends an X.25 incoming call packet to the called host. If the host decides to accept the call it then sends an X.25 call accepted packet to packet switch, which responds by sending an X.25 call connected packet to the local data concentrator. The local data concentrator responds by sending the message

```
CALL CONNECTED
```

to be printed on the subscriber's terminal.

At this point a connection has been established between the subscriber's terminal and the host. Thereafter, all information frames received from the subscriber are sent transparently (without interpretation by LADT) to the host. If the call had been refused by the host, or if the local data concentrator or packet switch detected an error in the data network number, an appropriate message would have been sent to the subscriber.

The call can be cleared by either the subscriber or the host. The subscriber can clear the call by sending a LAPB disconnect frame or (in the case of a dial-up subscriber) by simply going on-hook. If the host clears the call (by sending an X.25 clear request packet to the packet switch) the message is sent to the subscriber. At the end of the call, the local data concentrator will send billing information to the NCC. This information includes the number of packets sent and received, the time the call began, and its duration.

VI. SERVICE EVOLUTION

In the future additional capabilities will be added to LADT to provide expanded customer features, improve system economics, and further facilitate operations support. This section will summarize what features are currently available in LADT and what features are planned or are being considered.

6.1 Current services

The principal capabilities currently provided by LADT are the following:

1. Dial-up access at 1.2 kb/s
2. Direct access at 4.8 kb/s
3. Synchronous (LAPB of X.25) terminal access protocol
4. Billing via holding time for dial-up subscribers
5. Fixed monthly charges plus packet and holding time sensitive charging capabilities for direct access subscribers
6. All essential X.25 virtual call features on host links
7. Hunt group across host links
8. Conditional removal of host access lines.

6.2 Potential future services

A number of other capabilities are being considered for future inclusion in LADT, although no decision to include such capabilities has been made. Other access protocols may later be provided. For example, an asynchronous terminal interface capability may be added allowing asynchronous devices, including most home computers, to access LADT. Full X.25 protocol to terminals might be provided. This would permit subscribers to receive (as well as initiate) data calls and would allow the subscriber to set up multiple, simultaneous data calls. In the future, LADT customers may be able to place data calls to hosts in other LATAs. This would be accomplished using the CCITT X.75 gateway protocol[22] between the LADT and various interexchange data networks. Other access arrangements to LADT might be introduced. New billing capabilities may be included along with support capabilities in various operations support systems.

VII. INITIAL EXPERIENCE WITH LADT

The LADT service was initiated by Southern Bell Telephone Co. in the southern Florida LATA on July 1, 1983, using equipment manu-

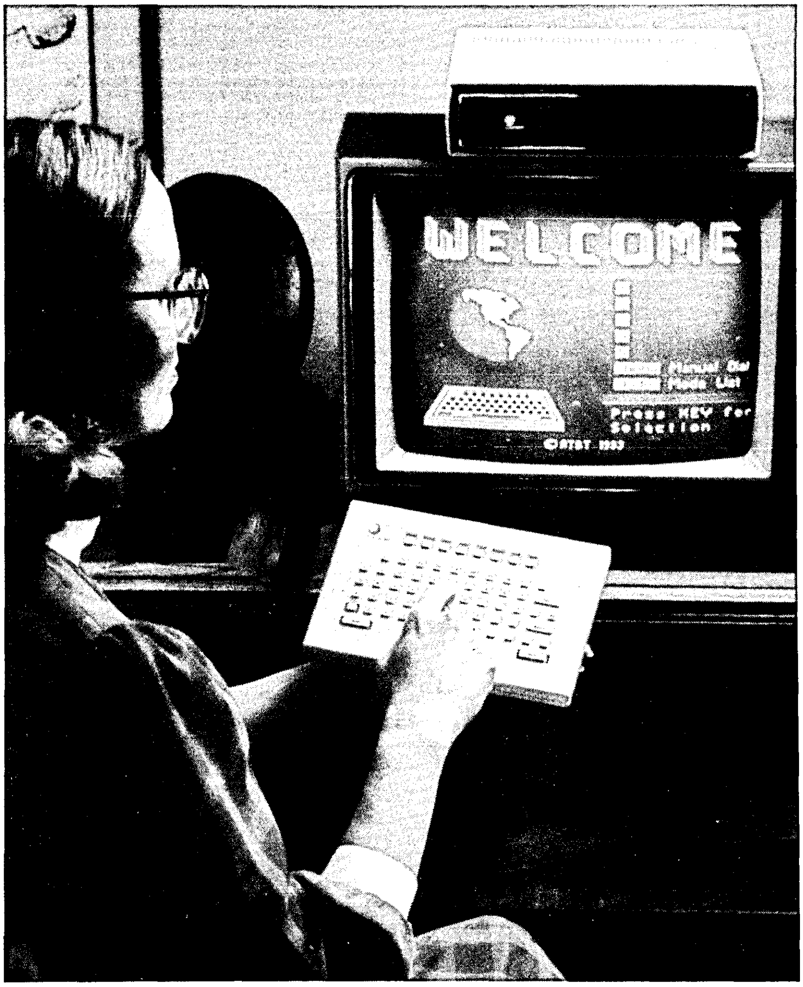


Fig. 10—The *Viewtron*™ service showing the *Sceptre*™ home terminal.

factured by Western Electric.[23] A local data concentrator called the data subscriber interface is being manufactured by AT&T Technologies, Inc. for LADT. The AT&T Technologies, Inc. packet switch available for LADT is the 1PSS packet switch[24] also used in the AT&T Communications basic packet-switching service. The initial application of LADT is the *Viewtron* service provided by Viewdata Corporation of America (see Fig. 10). Subscribers access this service using the *Sceptre** videotex terminal designed AT&T Technologies.[25] Twelve

* Trademark of AT&T Technologies, Inc.

southern Florida banks and over fifty information providers cooperate in providing the *Viewtron* service. With *Viewtron* subscribers can read up-to-the-minute news, order merchandise, post bulletin board messages, consult an electronic encyclopedia, play games, and pay bills. It is expected that by the end of 1984, 5000 subscribers will be regularly accessing this service over LADT.

VIII. SUMMARY

LADT represents an important step into the information age. By providing powerful and economic data transmission to homes and small businesses, LADT has opened the door to important new information services. Many new and varied information services may soon be provided over LADT. Our early experience with the service has shown strong customer acceptance. As a result we expect LADT service will become widely available.

REFERENCES

1. J. Tydeman, H. Lipinski, R. Adler, M. Nyhan, and L. Zwimpfer, *Teletext and Videotex in the United States*, New York: McGraw Hill, 1982.
2. H. L. Morgan, "The Interconnected Future: Data Processing, Office Automation, Personal Computing," VACN Symposium, May 1979, pp. 291–300.
3. N. Carruthers, "Personal Computers and Videotex," Viewdata 82, London October, 1982, (Middlesex: Online Conferences Limited, 1982), pp. 159–167.
4. The Viewtron Newsletter, Viewdata Corporation of America, Inc., Miami, Florida, *3* No. 2, (June 27, 1983).
5. W. B. Cornfield, "Electronic Banking: Why Its Time Has Finally Come," Viewdata 82, October, 1982, New York, New York, (Middlesex: Online Conferences Limited, 1982), pp. 343–7.
6. D. Shaider, "Taking Videotex to Market: The CBS Role In The Joint CBS/AT&T Ridgewood Trial," Videotex 83, New York, New York, June 27, 1983 (Middlesex: London Online Inc., 1983) pp. 93–8.
7. P. Sternberger, "Dow Jones News Retrieval Service, Videotex 81, Toronto, Ontario May, 1981, (Middlesex: London Online Inc. 1981), pp. 85–94.
8. I. Dorros, "Telephone Nets Go Digital," IEEE Spectrum, *20*, No. 4 (April 1983), pp. 48–53.
9. F. J. Brophy, G. Herbert Honnold, and S. J. Thayer, "DATAPHONE® II Service–New Standard for Data Communications," Bell Lab. Rec., *59*, No. 8, October 1981.
10. L. Tymes, "TYMNET—A Terminal Oriented Communication Network," 1971 Spring Joint Computer Conf. AFIPS Conf. Proc., *38*, pp. 211–6.
11. L. G. Roberts, "Telenet Principles and Practice," European Computing Conf. on Commun. Networks, London, England, 1975; pp. 315–29.
12. D. W. Davies, et al., "A Digital Communications Network for Computers Giving Rapid Response at Remote Terminals," ACM Symposium Operating Systems Problems, October 1967.
13. "Compatibility Criteria for Data Set 212A," USITA Technical Advisory No. 20–Revision No. 3, September 1977.
14. N. E. Snow and N. Knapp, Jr., "Digital Data System," B.S.T.J., *54*, No. 5 (May–June 1975), pp. 811–32.
15. H. Zimmermann, "OSI Reference Model—The ISO Model of Architecture for Open System Interconnection," IEEE Trans. Commun., *COM-28*, No. 4 (April 1980), pp. 425–32.
16. CCITT, "Interface Between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) For Terminals Operating In The Packet Mode on Public Networks," *Public Data Networks, Yellow Book, 8*, Seventh Plenary Assembly, Int. Telecommunications Union, Geneva, Switzerland, November, 1980, pp. 100–89.

17. CCITT, "International Numbering Plan For Public Data Networks," *Public Data Networks, Yellow Book, 8*, Seventh Plenary Assembly, Int. Telecommunications Union, Geneva, Switzerland, November 1980, pp. 245–56.
18. "Local Area Data Transport Terminal Interface Specification," AT&T Preliminary Technical Reference, PUB 54200, June 1982.
19. American National Standard Code for Information Exchange, ANSI X3.4, Amer. Nat. Standards Inst., Inc., 1977.
20. "X.25 Interface Specifications," AT&T Preliminary Technical Reference, PUB 5400, August 1981.
21. "Local Area Data Transport Host Interface Specifications," AT&T Preliminary Technical Reference, PUB 54210, June 1982.
22. CCITT, "Terminal and Transit Call Control Procedures and Data Transfer System on International Circuits Between Packet-Switched Data Networks," *Public Data Networks, Yellow Book, 8*, Seventh Plenary Assembly, Int. Telecommunications Union, Geneva, Switzerland, November 1980, pp. 142–207.
23. H. J. Kafka, W. J. Paule, and D. J. Stelte, "AT&T Technologies Implementation of Local Area Data Transport—A Hardware and Software Overview," Bell Lab. Tech. J., this issue.
24. J. C. Ehlinger and R. W. Stubblefield, "No. 1 PSS Service Capabilities and Architecture," ICC '83, Boston, Mass.
25. Videotex '83, New York City, New York, June 27, 1983.

AUTHOR

M. Niel Ransom, B.S.E.E., 1970, and M.S.E.E., 1971, Old Dominion University; Ph.D., 1973, University of Notre Dame; AT&T Bell Laboratories, 1973—. Mr. Ransom is currently Supervisor of the Switching Architecture Planning Group of the Exploratory Switching Networks Department of AT&T Bell Laboratories. His group is responsible for identifying network applications of emerging switching technology. Prior to this, he held various supervisory positions with applied research and had development responsibility for AT&T products that provide Local Area Data Transport service. As a Member of Technical Staff, he had various responsibilities in switching systems engineering, applied research in voice and data switching, and development of the 5*ESS*™ switching system.

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July-August 1984
Printed in U.S.A.

AT&T Technologies Implementation of Local Area Data Transport—A Hardware and Software Overview

By D. J. STELTE,* H. J. KAFKA*, and W. J. PAULE*

(Manuscript received September 12, 1983)

AT&T Technologies has implemented hardware and software components that will provide an economical Local Area Data Transport (LADT) service. The AT&T Technologies LADT Generic 1.0 is composed of a No. 1 PSS packet switch, one or more statistical multiplexers called Data Subscriber Interfaces (DSIs), and an Administrative Processor (AP) responsible for the administrative functions of the network. This paper describes AT&T Technologies LADT Generic 1.0 hardware and software architectures of the DSI and AP components of the network.

I. INTRODUCTION

The AT&T Local Area Data Transport (LADT)[†] system is a packet-switched network that provides local exchange areas an economical data communications capability.[1] It incorporates inexpensive access mechanisms, standard interfaces, high availability, and the potential for quick and ubiquitous deployment.[2]

This system consists of three types of nodes (Fig. 1):

1. Data Subscriber Interfaces (DSIs), a statistical multiplexer that terminates up to 124 customer lines and one high-speed network access link.

2. The No. 1 PSS packet switch, a high-reliability, high-availability

* AT&T Bell Laboratories.

[†] Acronyms are defined in the Glossary at the end of this paper.

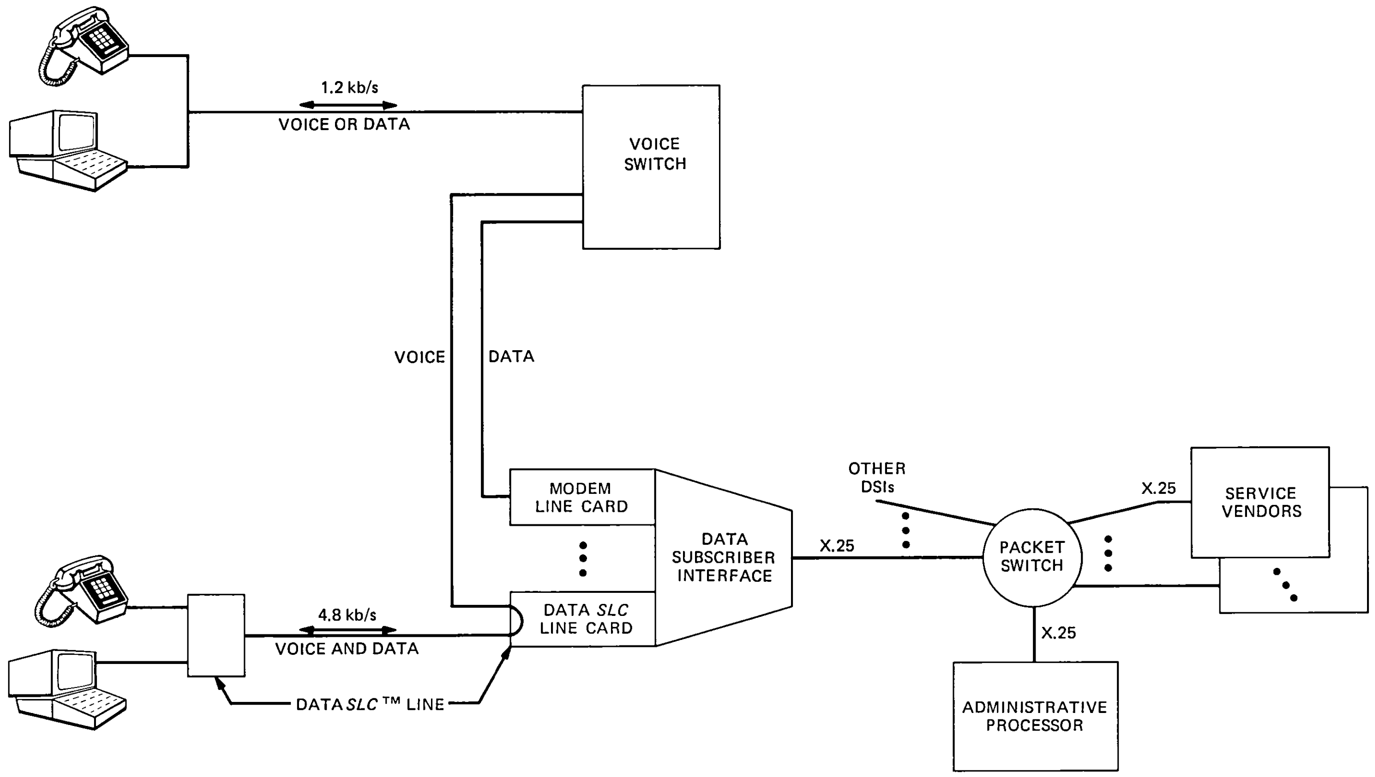


Fig. 1—LADT architecture.

X.25[3] data switch used for routing and transporting packets between DSIs and service vendors.

3. The Administrative Processor (AP), a centralized point for providing maintenance, billing, traffic, craft interface, and network-management functions for the DSIs.

The newly developed DSI provides access and multiplexing functions for subscriber lines. The DSI supports two methods of access for the customer interface. One is a switched access for either voice or 1.2-kb/s data communication through the public switched telephone network. The other is a dedicated access at 4.8 kb/s that provides simultaneous voice and data communication over a single wire pair. The DSI supports Link Access Procedure B (LAPB) as the link-level protocol for subscriber lines.

The No. 1 PSS packet switch is the hub of the network and is used for switching and transporting packets. It connects 56-kb/s lines from DSIs and 9.6- or 56-kb/s lines from service providers. The packet switch provides all essential virtual call services of the 1980 International Telegraph and Telephone Consultative Committee (CCITT) Recommendation X.25 protocol.[3]

Like the packet switch, the Administrative Processor is implemented on a high-reliability, high-availability processor. It provides network support functions in a central location for LADT.

While the preceding article provides an overview of the generic LADT services, this article gives details of the AT&T Technologies implementation of LADT.

II. DATA SUBSCRIBER INTERFACE HARDWARE

The DSI is a statistical multiplexer that concentrates data packets from up to 124 subscriber lines onto a single 56-kb/s data link to the packet switch.[4] The primary purpose of the DSI is to reduce LADT subscriber access costs by sharing packet-switch access costs among many subscribers and by providing inexpensive access to the DSI. To keep costs low while providing high availability, the DSI is implemented as a simplex system that is reliable and easily maintainable.

The DSI is divided into three major subunits, which consist of the processor-complex subunit and two line-group subunits (Fig. 2). The processor-complex subunit contains the intelligence of the DSI, the protocol-handling functions, the network interface, and the craft interface. The processor-complex centralizes the processing power of the DSI so that these functions can be shared over all of the subscriber lines. This reduces the cost by minimizing the individual line interface functions performed in the line-group subunits. Each of the line-group subunits terminates up to 64 lines and multiplexes the data streams from these lines into the processor complex. In line-group subunit 0,

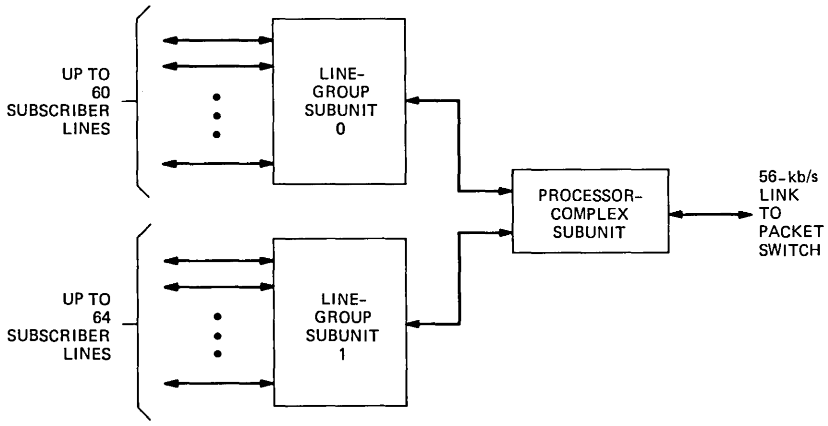

UP TO 60 SUBSCRIBER LINES
LINE-GROUP SUBUNIT 0
PROCESSOR-COMPLEX SUBUNIT
56-kb/s LINK TO PACKET SWITCH
UP TO 64 SUBSCRIBER LINES
LINE-GROUP SUBUNIT 1

Fig. 2—Hardware subunits of a DSI.

4 of the lines are used for test circuitry, while in line-group subunit 1 all 64 lines are available for serving subscribers. The following sections give physical and functional descriptions of the DSI and its components.

2.1 Physical description

LADT subscriber access costs are reduced by minimizing the amount of transmission over lines that are not concentrated. This is accomplished by locating DSIs in the same central offices with voice switches. The DSI hardware is optimized for installation and operation in the central office environment.

The DSI frame, shown in Fig. 3, uses a standard set of devices, apparatus, equipment, and design tools that are common to many AT&T Technologies products.[5] The DSI frame uses a standard framework, which is 7 feet high, 2 feet and 2 inches wide, and 18 inches deep. Each frame contains up to two DSI units and one −48V fuse-panel unit. The base of each frame also contains two −48V filter circuits, one for each of the two power buses entering the frame.

A DSI unit, shown in Fig. 4, occupies 24 inches of vertical space in the frame. Each unit consists of three 8-inch shelves equipped with backplanes. The upper shelf contains the line-group 0 subunit, the middle shelf contains the processor-complex subunit, and the bottom shelf contains the line-group 1 subunit. Each shelf contains its own power converters for the circuit packs in that level so that a power failure in one of the line-group subunits will not result in the failure of the entire unit. A fully equipped DSI unit contains 43 circuit packs.

Four types of cables connect the DSI to other equipment in a central

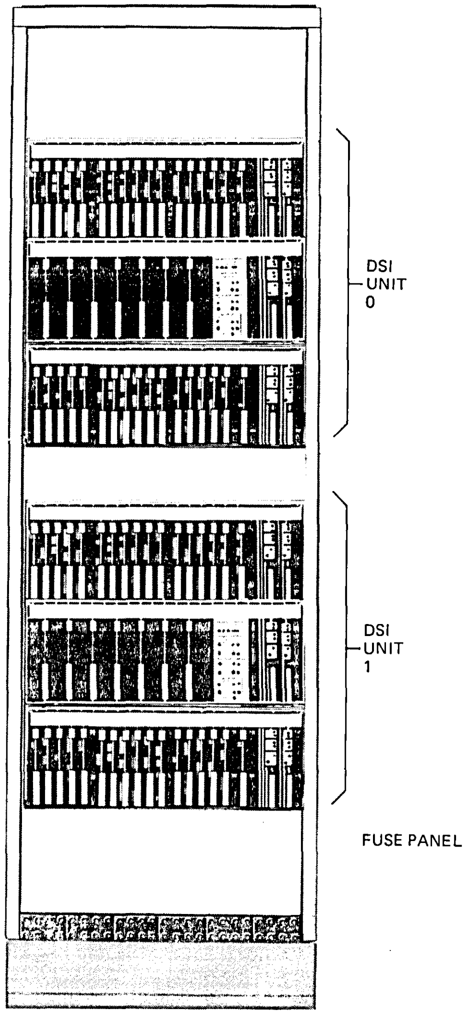


Fig. 3—DSI frame configuration with two DSI units.

office. These cables are installed when the DSI is installed. The cables include:

1. Power cables, which deliver −48V power to the frame from the central office supply.

2. Tip and ring cables, which connect the subscriber-line interfaces on the DSI to the main distributing frame in the central office.

3. 56-kb/s link cables, which provide the interface between the DSI and the packet switch.

4. Central office cables, which connect to alarm and scan points of the central office through the main distributing frame.

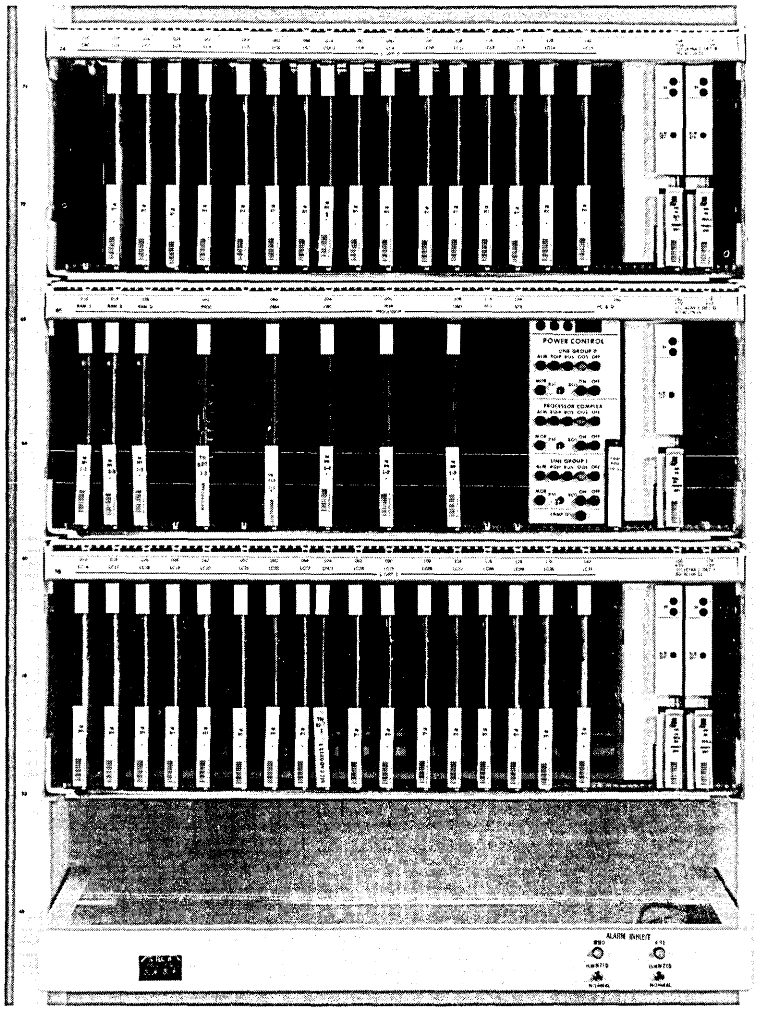


Fig. 4—DSI unit.

2.2 Processor-complex subunit

The processor-complex subunit is designed to provide all of the protocol handling and processing power required in the DSI. The tasks that must be performed are functionally partitioned between specialized hardware components and two general-purpose microprocessors in order to provide optimal performance and flexible service capabilities at a low cost.

Figure 5 shows the functional components of the DSI processor complex. The main-processor circuit pack contains a general-purpose

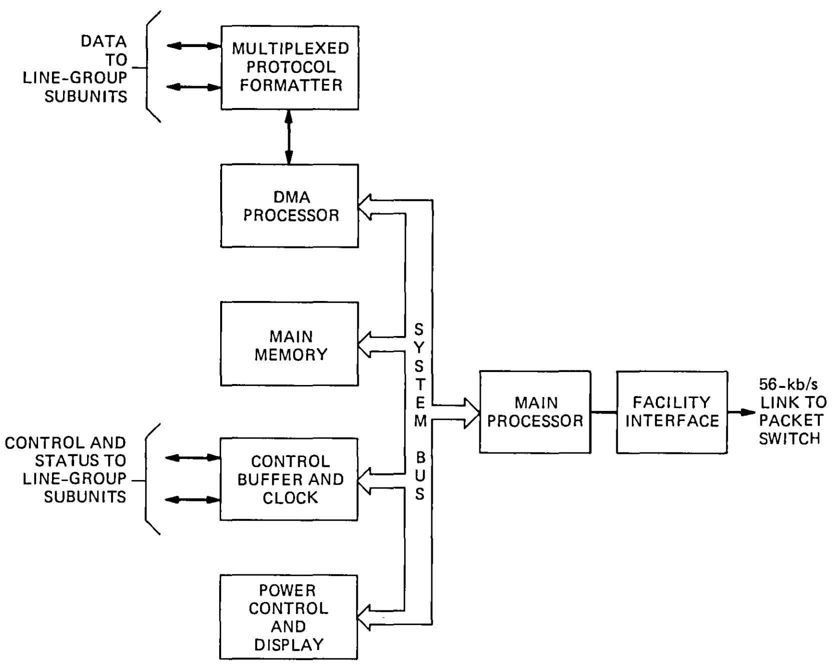

DATA TO LINE-GROUP SUBUNITS
MULTIPLEXED PROTOCOL FORMATTER
DMA PROCESSOR
MAIN MEMORY
SYSTEM BUS
CONTROL AND STATUS TO LINE-GROUP SUBUNITS
CONTROL BUFFER AND CLOCK
POWER CONTROL AND DISPLAY
MAIN PROCESSOR
FACILITY INTERFACE
56-kb/s LINK TO PACKET SWITCH

Fig. 5—DSI processor-complex subunit.

processor that handles call processing, the higher levels of protocols, and general control functions. The main-memory circuit packs provide memory for the main processor's programs and for data buffers. The power-control-and-display circuit pack controls all of the power converters in the DSI and provides the local craft interface for the DSI. The facility-interface circuit pack provides electrical interfaces for the 56-kb/s data link to the packet switch. The control-buffer-and-clock circuit pack generates clocks for the DSI and provides the main processor with convenient access to the line-group subunits. The Direct Memory Access (DMA)-processor circuit pack converts between buffered subscriber data packets and data bytes, and the multiplexed-protocol-formatter circuit pack converts between data bytes and data bits as it handles the lower levels of the protocol on the subscriber lines. The following sections provide more details on these seven types of circuit packs in the processor-complex subunit and on the system bus that interconnects them.

2.2.1 System bus

As Fig. 5 illustrates, the processor complex is organized around a general-purpose microprocessor bus called the system bus. Most of the communication between various components of the processor-complex subunit occurs over this bus. The system bus consists of a 20-bit address bus and a 16-bit data bus. The data and address signals are both protected by parity bits. Other signals that are considered to be a part of the bus include read and write strobes, circuit-pack selects, interrupt requests, bus arbitration signals, and bus timing signals.

2.2.2 Main processor

The main-processor circuit pack contains the controlling intelligence of the DSI in the form of a microprocessor. This microprocessor is responsible for the higher levels of protocols, maintenance, and line control commands, and in general the rest of the DSI. As shown in Fig. 6, the main-processor circuit pack also contains general-purpose microprocessor support circuitry, such as bootstrap Read-Only Memory (ROM), scratch-pad Random Access Memory (RAM), bus controls, interrupt controls, direct-memory-access controls, memory and I/O controls, a sanity timer, and an AT&T Technologies X.25 Protocol Controller[6] (XPC) integrated circuit.

The main-processor circuit pack is based on an Intel 8086 microprocessor operating at 5 MHz, with associated clock and reset circuitry. The main-processor circuit pack also includes 64K bytes of bootstrap ROM and 4K bytes of static RAM that are used during system initializations. The bootstrap ROM stores initialization programs, diagnostic programs, and a download program. These programs enable the DSI to obtain its operational software by placing an X.25 call to the AP through the packet switch. The RAM serves as a temporary scratch-pad memory that is used until the main processor can verify the integrity of the main-memory circuit packs. After the main-memory diagnostics are completed, the RAM on the main-processor circuit pack is disabled and logically replaced by a section of the main memory.

Large amounts of customer data pass between various components of the processor-complex subunit and the data buffers in main memory. To make these data transfers as efficient as possible, the main processor permits other components of the DSI to access the main memory directly by granting them control of the system bus. The main processor includes circuitry to control these direct-memory-access transfers and to arbitrate control of the system bus.

In any system with the hardware and software complexity of the DSI, major faults or illegal states may occur that prevent normal recovery actions from taking place. To facilitate rapid recovery in these situations, the main-processor circuit pack contains a hardware sanity timer. After this timer is initialized, it must be periodically reset by the main processor to keep from expiring. If the main processor

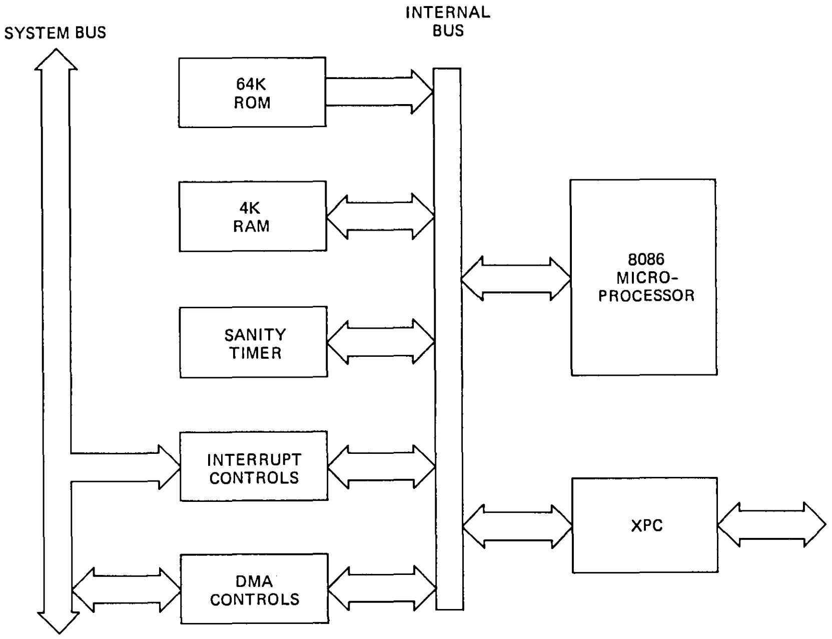


Fig. 6—DSI main processor.

allows the sanity timer to expire, the main-processor circuit is reset by an unmaskable interrupt so that appropriate recovery action can be taken.

The main-processor circuit contains an XPC integrated circuit,[6] which autonomously handles the complete X.25 level-2 (link level) protocol for the 56-kb/s link to the packet switch. The XPC chip handles communications with the packet switch by transferring packets in and out of main-memory data buffers through direct-memory access. The XPC notifies the microprocessor when significant events on the link occur, such as when packets are successfully received or when the packet switch acknowledges the reception of a packet. By autonomously handling the lower levels of the X.25 protocol, the XPC chip enables the microprocessor to concentrate on higher-level functions.

Unusual or significant system events are reported to the main processor by system interrupts. The main processor contains 15 independently maskable interrupt controls with programmable priorities.

2.2.3 Main memory

The processor complex has three identical circuit packs that form the DSI main memory. Each of these circuit packs has 256K bytes of

RAM. To increase the reliability of the DSI, the main-memory circuit packs include additional circuitry for write protection and for the detection and correction of both hard and soft memory errors. The DSI main memory provides storage for main-processor operational software, main-processor data, and packet-data buffers.

The memory array on each circuit pack consists of 128,000 words of 22 bits each. Sixteen of the bits store the normal memory word, and the six additional bits provide error detection and correction using a modified Hamming code. The memory array uses AT&T Technologies 64K-bit dynamic RAM chips. Each circuit pack also contains circuitry to refresh the dynamic RAM chips and to arbitrate between memory access cycles and refresh cycles.

A modified Hamming error detection and correction code is used to correct all single-bit memory errors and to detect all double-bit errors. Any errors also cause the generation of an interrupt to the main processor. The Hamming circuitry supports both byte and word reads and writes.

The main-memory write-protection circuitry allows the main processor to write-protect any section's main memory in 512-word (1K-byte) blocks. When an attempt to write into a protected area occurs, the main memory prevents a write from occurring and causes an interrupt to the main processor. Write-protecting sections of memory that contain the main-processor program text help to ensure the integrity of the DSI. When either Hamming or write-protect errors occur, an error register traps the address where the error occurred so that the main processor can take appropriate corrective action.

2.2.4 Power control and display

The power-control-and-display circuit pack in the processor complex contains the local craft interface and the relay contacts for the central office alarm interface. All of the power converters in the DSI are controlled from this circuit pack through the use of switches and Light-Emitting Diodes (LEDs), and other LED displays are used for maintenance and debugging purposes. All of the switches and LED displays are mounted on the faceplate to facilitate access by craft personnel.

The faceplate, shown in Fig. 7, is divided into four areas of indicators and switches. These areas correspond to the three subunits (line group 0, line group 1, and the processor complex) and the craft diagnostic display section. The three subunit areas give the craft control of the DSI and its subunits. By observing the indicator lights and by operating the appropriate switches, the craft can take a subunit out of service, remove power from a subunit, restore power to a subunit, and request restoration to service of the subunit. The lights will also reflect

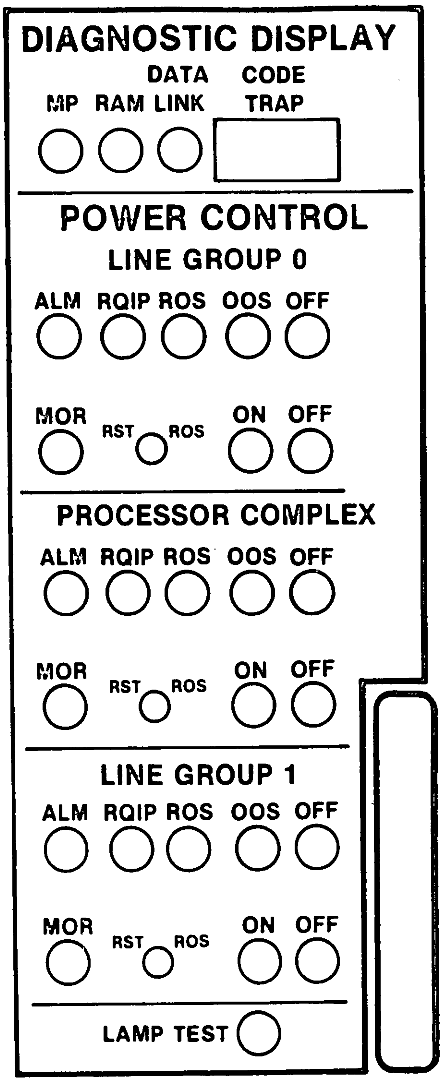


Fig. 7—Faceplate of the power-control-and-display circuit pack.

the status of each subunit when remove/restore service requests are initiated remotely by the AP.

The diagnostic display section of the faceplate contains three LED lights and two 7-segment displays. This diagnostic display section is used to provide fault localization when the link between the DSI and the AP is not functional. During initialization of the DSI, the diagnostic display indicates the correct operation of the DSI main processor, the DSI main memory, and the data link to the packet switch as determined by the ROM-based diagnostics in the main processor. Thus, if a failure occurs during initialization, the craft can make circuit-pack replacements based on the state of the diagnostic display.

2.2.5 Facility interface

The facility-interface circuit pack provides for several electrical interfaces at the 56-kb/s link between the DSI and the packet switch. The DSI can be located either remotely from the packet switch or in the same building as the packet switch. The facility-interface card permits the choice of a transmission mechanism that is cost-effective for each installation of a DSI.

For the case when both the DSI and the packet switch are not collocated, the physical link can be provided by a Digital Data System[7] private digital line in which the DSI looks electrically like Data Terminal Equipment (DTE) and connects to a Data-Service Unit.[8] This general-purpose interface permits the DSI to use standard data transmission equipment to communicate with the packet switch.

For the case when the DSI and packet switch are located in the same building, a direct-link option is provided to reduce the need for data communication equipment external to the DSI. This interface allows the packet switch and the DSI to be directly connected when they are located close together. For slightly longer distances, limited distance modems can be used.

The above two DSI interfaces are supplied as standard equipment and can support any DSI placement currently envisioned. However, since many DSI installations will be in central offices with digital facilities, the facility-interface circuit pack provides an interface that can bypass a substantial amount of the standard customer-interface hardware. This interface, called the DS-O interface because it connects to the digital facilities at the DS-O level,[9] can be used to alleviate the need for both the ac-powered data service unit and the office channel unit. The DSI's ability to directly meet the DS-O interface reduces the overall cost of the packet-network interface.

2.2.6 Control buffer and clock

The control-buffer-and-clock circuit pack contains a clock section, which transmits clock and synchronization signals to many of the other circuit packs in the DSI, and a control-buffer section, which provides a convenient means for the main processor to access registers in the line-group subunits that control and reflect the state of the subscriber lines.

The clock section includes a crystal-controlled phase-locked-loop oscillator, divide chains, and buffers, which provide most of the clocks used throughout the DSI. The crystal-controlled oscillator can free-run or it can be phase-locked to one of several external sources. This synchronization capability permits the subscriber-line data rates to be locked to network clocks. When the DSI is configured to be phase-locked to an external clock source, the control-buffer-and-clock circuit pack reports an error condition whenever the phase lock is lost.

The control-buffer-and-clock circuit pack also contains a control buffer section with a buffer memory and a scanning mechanism. This gives the main processor a simple memory-like interface to registers on the line-interface circuit packs (line cards) that terminate the subscriber lines, without the reliability problems that would result from extending the system bus to all of these circuit packs. This interface permits the main processor to carry out line control and maintenance functions on each subscriber line. The control buffer and clock controls the distribution of the line control information from the control buffer memory to the line cards, and controls the return of status information from the line cards to the status buffer memory. The main processor accesses the control and status information simply by writing to the control memory and reading from the status memory.

2.2.7 DMA processor

The Direct-Memory-Access (DMA)-processor circuit pack and the multiplexed-protocol-formatter circuit pack together handle low-level protocol functions for each of the 128 lines (124 subscriber lines plus 4 test lines) supported by a DSI. The multiplexed protocol formatter (described below) converts individual data bits from the subscriber lines to bytes of information frames. The DMA processor transfers these bytes into and out of main-memory packet-data buffers and controls the multiplexed protocol formatter.

The DMA-processor circuit pack contains an Intel 8086 microprocessor with specialized hardware and ROM-based firmware to put data bytes from the subscriber lines into the DSI main memory, and to take data bytes from the DSI main memory for transmission to the subscriber lines. The DMA processor contains bus interface hardware that enables it to transfer data bytes in and out of main memory with direct-memory-access techniques. The DMA processor also contains hardware for direct communication with the main processor. This hardware includes First-In First-Out (FIFO) memories for commands and responses, a shared memory for line states and packet-buffer addresses, and special registers. Other hardware on the DMA-processor circuit pack includes local static RAM, and specialized hardware to interface to the multiplexed protocol formatter. The DMA processor participates in certain maintenance and line control functions because it acts as the interface between the main processor and the multiplexed protocol formatter. The firmware that controls the DMA processor is described later in this paper.

2.2.8 Multiplexed protocol formatter

The multiplexed-protocol-formatter circuit pack is a time-shared state machine that uses specialized hardware and memory to handle frame-sublevel protocol functions, including flag and frame recognition, bit stuffing and unstuffing, byte assembly and disassembly, and Cyclic Redundancy Check (CRC) generation and checking, for up to 128 lines. The multiplexed protocol formatter handles all 128 lines on time-shared hardware. This reduces costs and simplifies maintenance by using less hardware and by eliminating the complexity of having 128 independent protocol-handling devices. The multiplexed protocol formatter does not interface directly to the system bus. Instead, it is directly controlled by the DMA processor (and therefore indirectly controlled by the main processor). The multiplexed protocol formatter also interfaces to each line-group subunit through time-multiplexed data streams, which contain data bits for all of the subscriber lines in the line-group subunits.

To provide the above functions, the multiplexed protocol formatter consists of two finite-state machines that are time-shared among all customer channels. The basic structure of the multiplexed protocol formatter is shown in Fig. 8. One state machine performs transmission functions, and the other state machine performs reception functions. Each of these state machines is configured for a particular channel immediately before action is required to service that channel. This configuration, or state information, is stored for each channel's receiver and transmitter in the state memory sections shown in Fig. 8. After the channel has been serviced, the new state is stored in memory and the present state of the next line is obtained. By performing these operations at a rate equal to the combined line rates of all channels, the multiplexed protocol formatter handles its portion of the protocol for all lines.

The multiplexed protocol formatter also contains command memories and FIFOs that interface with the DMA processor. The DMA processor passes information to the multiplexed protocol formatter by writing into the command memories. The information in these memories includes line control information and data bytes to be transmitted to the subscriber lines. These memories have separate locations for each line, so the DMA processor can effectively control all of the lines simultaneously. The multiplexed protocol formatter passes information to the DMA processor by writing into FIFO memories. The information includes line-status information and data bytes received

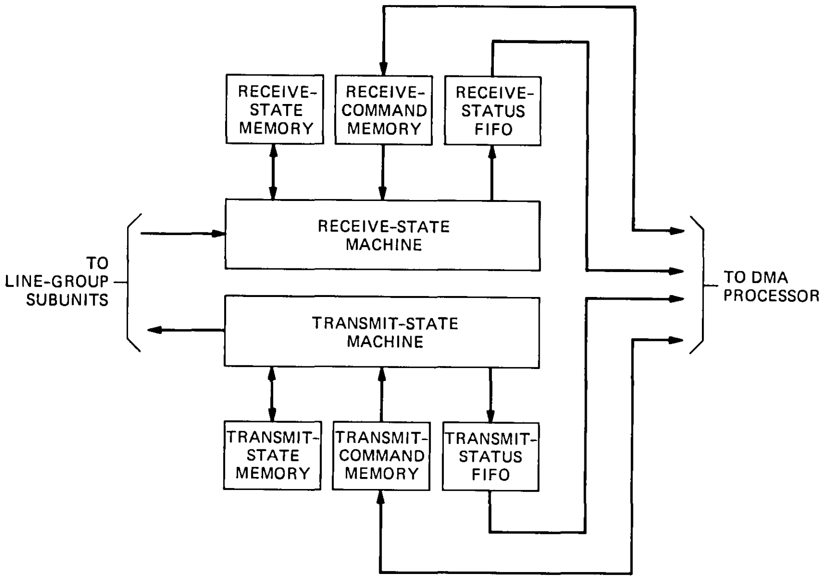


Fig. 8—DSI multiplexed protocol formatter.

from the customer lines. Each entry also includes a line number, which tells the DMA processor the subscriber line to which the entry applies.

2.3 Line-group subunit

Figure 9 is a functional representation of a DSI line-group subunit. The line group consists of one group-distributor-circuit circuit pack and up to 16 line-interface circuit packs, or line cards. The line-group subunits of the DSI may be configured to support various mixtures of line interfaces because the line cards that terminate subscriber lines meet the same backplane interface. The mixture of subscriber lines that terminate on a DSI is determined by the types of line cards that appear in the line-group subunits.

The line cards contain the interfaces to the subscriber lines. Each subscriber line terminates on one of the line circuits on a line card. These line circuits convert the modulated or multiplexed data format on the subscriber line to a digital data stream. Each line circuit also responds to control bits and supplies status bits that are used for line control and maintenance functions. Each line card contains three or four line circuits and a line-card common circuit, which converts the data, control, and status information into the proper format for the backplane interface to the group-distributor circuit. The initial types of line cards for the DSI include modem line cards with four modem

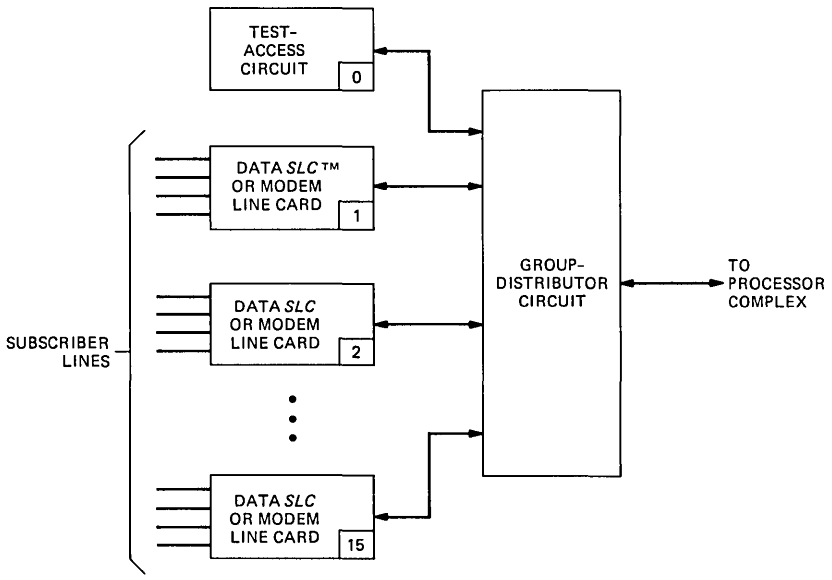

TEST-ACCESS CIRCUIT
0
DATA SLC™ OR MODEM LINE CARD
1
DATA SLC OR MODEM LINE CARD
2
DATA SLC OR MODEM LINE CARD
15
SUBSCRIBER LINES
GROUP-DISTRIBUTOR CIRCUIT
TO PROCESSOR COMPLEX

Fig. 9—DSI line-group subunit.

line circuits per circuit pack, and data *SLC** (Subscriber Line Concentrators) line cards with three data *SLC* line circuits per circuit pack.

2.3.1 Group-distributor circuit

The group-distributor circuit pack distributes clocks, data information, and control information from the processor complex to the individual line cards. Since the multiplexed protocol formatter requires the data bits from the line cards to be time-multiplexed, the group-distributor circuit performs the necessary multiplexing/demultiplexing between the information format for the processor complex and the information format for the line cards. The group-distributor circuit gives each line card a separate set of signals. This permits all of the line cards to have the same interface to the group-distributor circuit. It also increases the maintainability and reliability of the line group by isolating the effects of line-card failures. The group-distributor circuit also contains circuitry to assist in the maintenance of the control information paths to the line cards.

2.3.2 Modem line card

The modem line card is currently expected to be the most common

* Trademark of AT&T Technologies, Inc.

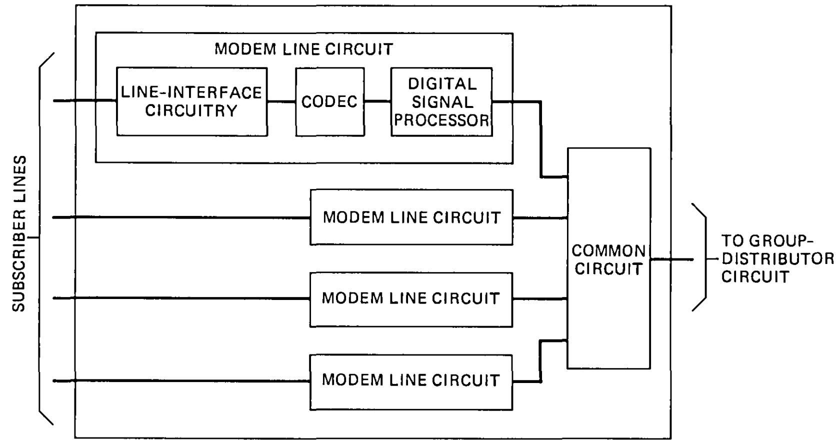


Fig. 10—DSI modem line card.

of the DSI line cards. The modem line card has four line circuits, each of which interfaces to one subscriber line. As Fig. 10 indicates, the modem line card consists of a common circuit, which is shared over all of the subscriber lines, and four modem line circuits (one for each subscriber line). Each modem line circuit performs the functions of an answer-only, 1.2-kb/s 212A modem[10] operating in the synchronous mode. The common circuit includes clock circuitry for the individual modem line circuits and interface circuitry for meeting the common backplane interface to the group-distributor circuit. This common circuit is implemented in a custom Metal Oxide Semiconductor (MOS) device.

Each of the modem circuits on a modem line card includes a Digital Signal Processor (DSP) integrated circuit[11] made by AT&T Technologies, which does the actual modulation and demodulation; a codec, which converts between the analog signals on the subscriber line and the digital format used by the DSP; and circuitry to interface to the subscriber line. The DSP performs most of the data functions of the 212A modem, including high-order digital filters for modulation and demodulation, frequency generation for modulation, carrier detection, clock recovery, and data scrambling/descrambling. The line-interface circuitry includes a ringing detector (since the modem line card is alerted to incoming calls by the presence of ringing on the subscriber lines), a 2-wire to 4-wire hybrid, and line-control relays.

Each line-card slot in the line-group subunit backplane is equipped with shorting contacts, which close whenever a line card is removed from the slot. In the case of modem lines, these contacts make the lines appear busy to the central office so that they will be skipped in

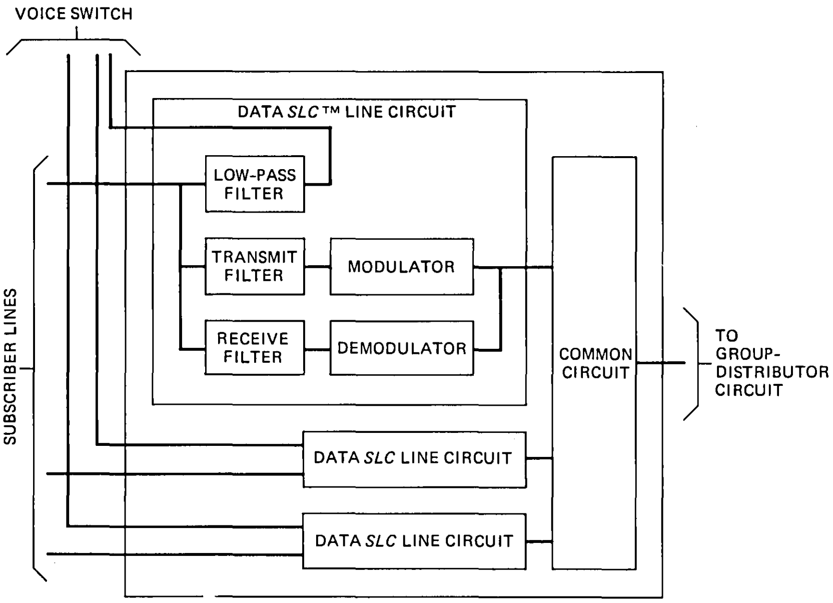


Fig. 11—DSI data *SLC*™ line card.

the line-hunting sequence. This prevents calls from terminating to lines that are temporarily unequipped in the DSI.

2.3.3 Data SLC line card

Data *SLC* system provides simultaneous voice and data channels on a single pair of wires through a technique similar to the *SLC*-1 subscriber carrier system, except that the derived voice channel of the *SLC*-1 system is replaced by a data channel in the data *SLC* system. As Fig. 1 indicates, the data *SLC* system loop is terminated by a data *SLC* line card at the DSI and by a data *SLC* remote terminal at the customer premises. The data *SLC* line card and remote terminal separate the data and voice channels in order to maintain two independent paths: a voice path between the central office and the normal subscriber telephone equipment, and a data path between the subscriber data terminal and the DSI.

As Fig. 11 indicates, the data *SLC* line card consists of a common circuit, which is shared over all three of the subscriber lines, and three line circuits, each of which is dedicated to one subscriber line. The data *SLC* line-card common circuit includes interface circuitry for meeting the common backplane interface to the group-distributor circuit, and clock circuitry that generates the clocks that are needed by the line circuits. The line circuits include passive filters to keep the high-frequency data carrier signals from being sent to the central office voice switch. Since these filters are passive, power failures in the line-group subunit do not affect the integrity of the voice path. Other filters in the line circuit separate the data channels from the voice channel and from each other. The data channel uses Frequency Shift Keying (FSK) modulation with carrier frequencies of 76 kHz and 28 kHz for data transmitted to the subscriber and to the DSI, respectively. The line circuits include circuitry to send a message to the data *SLC* remote terminal in order to invoke "far end loopback", as well as circuitry to perform a digital loopback on the data *SLC* line card itself. These loopback points permit the main processor to verify the integrity of the data paths.

Since the data *SLC* line card and the remote terminal are involved in both the data path and the voice path, they have been carefully designed to ensure the integrity of the voice channel in the event of power failures and failures in the data channel. Occasionally, failures in the data *SLC* line card require its replacement. To avoid disruption of the voice paths while repairs are being made, the line-group subunit backplanes contain shorting contacts. These contacts close whenever the data *SLC* line card is removed from the DSI, connecting the subscriber loop directly to the central office. This ensures the continuity of the voice path between the subscriber and the central office voice switch when the data *SLC* line card is removed from the DSI.

2.3.4 Test-access circuit

Each DSI contains one test-access circuit pack, which is used for diagnosing and maintaining line cards. The test-access circuit facilitates fault isolation by simulating customer premises equipment in order to determine whether a fault on a subscriber line is caused by a line card in the DSI.

The test-access circuit occupies the first line-card position in line-group subunit 0. It contains circuitry that simulates dial-up and direct subscriber lines, including an originate-only modem, a data *SLC* remote terminal, a ringing generator, and loss insertion circuits. Under command of the main processor, any line circuit in the DSI can be temporarily disconnected from a subscriber line and connected to the test-access circuit. This permits the DSI to thoroughly diagnose the line circuit and to localize faults within the DSI.

2.4 Data SLC remote terminal

As Fig. 1 shows, the data *SLC* remote terminal terminates the data *SLC* loop on the customer premises. Although the remote terminal draws ac power from the customer premises, it is part of the network and serves as the Network Channel Terminating Equipment. It uses a passive low-pass filter to pass the voiceband to standard telephones. The modulation and demodulation circuitry is similar to the circuitry of the data *SLC* line circuit. Other circuitry in the data *SLC* remote terminal performs clock recovery and provides the interface to the subscriber data terminal. This is a 4-wire interface using baseband, bipolar, return-to-zero signaling.[12] The data *SLC* remote terminal also contains circuitry to loop back the data path for maintenance purposes.

2.5 DSI hardware provisions for maintenance

The DSI hardware design incorporates many features to detect and isolate hardware faults. These features help to ensure high availability by detecting and isolating minor faults before major service disruptions occur and by localizing faults to circuit packs so that rapid repairs can be made by replacing the circuit pack. The circuitry that implements these features falls into two main classes: operational error detection circuitry and diagnostic error detection circuitry. The operational error detection circuitry continuously and automatically checks for errors while the DSI is operating normally. If an error is found, it is reported to the main processor so that corrective action may be taken. The diagnostic error detection circuitry provides thorough operational checks and fault localization. Since this diagnostic error detection circuitry could interfere with normal DSI operations, it must be explicitly requested by diagnostic software.

2.5.1 Operational error detection circuitry

Several types of error detection circuitry operate automatically and notify the main processor only when errors occur. The parity bits sent over the processor-complex bus fall into this category. During every bus cycle involving a transfer of information between two circuit packs, the circuit pack that is driving the address bus generates the address parity bit, and the circuit pack driving the data bus generates the data parity bits. The circuit pack that reads the bus also checks bus parity. If any error is detected, an interrupt notifies the main processor. This permits the rapid detection of bus faults.

Parity bits are also sent between the control buffer and clock and the line cards in order to verify the integrity of the control and status data path. The method of generating and checking these parity bits allows the control buffer and clock to verify the integrity of this path both to and from the line cards. The control buffer and clock generates a parity bit over the control data, and sends these data through the group-distributor circuit to a line card. The line card determines the sense of the parity bit (odd or even), and then generates a parity bit over the status information that has the same sense (odd or even) as the parity bit that it received. The line card sends the status information and parity bit back to the control buffer and clock through the group-distributor circuit. The control buffer and clock checks the sense of the parity over the status information to confirm that it matches the sense of the parity that it sent out with the control information. If an error is detected, the main processor is alerted.

By varying the sense of the parity in a particular manner, the control buffer and clock uses the parity mechanism to verify other aspects of the control and status scanning procedure. In each cycle of scanning the 128 lines, the control buffer and clock sends out odd parity to 127 of the lines and even parity to the other line. The line that receives the even parity is changed from one cycle to the next, so that all lines receive even parity once in every 128 cycles. With this procedure, the control buffer and clock confirms the integrity of the control and status path, confirms the parity checking and generating hardware on the line cards, and confirms that it is transmitting to and receiving from the proper line card at each point in the cycle.

The multiplexed protocol formatter and the line cards exchange parity over the data bits in a manner similar to the exchange between the control buffer and clock and the line cards. If the multiplexed protocol formatter detects any parity errors, it notifies the DMA processor, and the DMA processor notifies the main processor. In this way, all of the major data paths in the DSI are continuously monitored for integrity through the use of parity bits. This permits the main processor to detect and respond to faults very rapidly.

2.5.2 Diagnostic circuitry

The DSI contains diagnostic error detection circuitry, which enables the main processor to verify the proper operation of the circuitry, to localize any detected faults to a particular component, to verify the repair of the DSI, and to reconfigure the DSI so that it can operate in the presence of certain faults.

The localization of faults is accomplished principally through loop-back points. At a loopback point, the main processor can order the transmit data to be looped back to the received data. This enables the main processor to compare the received data to a known pattern that it is transmitting in order to verify that the circuitry up to the loopback point is operating properly. Loopback points occur at strategic locations in the DSI in order to permit the detection of faults and to permit the determination of the location of faults. Since the data coming from the customer are ignored when a loopback is activated, loopback tests are only performed when customers are not being served by the portion of the circuitry under investigation.

The DSI contains diagnostic circuitry, which permits the main processor to inject faults in order to verify that fault detection circuitry is operating properly. As an example, the main processor can force bad parity to be generated by the multiplexed protocol formatter in order to confirm that the bad parity is sent to a line card, returned to the multiplexed protocol formatter, and reported back to the main processor.

III. LADT SOFTWARE OVERVIEW

To make the network more flexible and cost-effective, functions in the LADT system are divided between two different types of processors. The functions that provide the interface to subscriber lines tend to be real-time intensive and are performed in small, microprocessor-based concentrators called Data Subscriber Interfaces (DSIs). The DSIs were kept simple to make them both reliable and cost-effective.

Functions such as billing and traffic-data processing, required to administer an LADT network, were centralized in the AP, the highly reliable duplex computer. Since only one AP is required per LADT network, it can have resources such as terminals, printers, and disk storage that cannot be economically provided on each DSI. The disk storage in the AP is used to store the operational software of the DSIs so that they do not need to have local mass storage. The AP also provides a convenient central location from which the network can be administered.

Many LADT functions are divided between the DSIs and the AP. For example, traffic data for subscriber lines are collected in the DSIs, then sent to the AP for processing into printed reports. Table I shows how functions in LADT are partitioned between these two network nodes. The remainder of this section will discuss some major LADT functions and how they are divided between the DSIs and the AP.

3.1 AP/DSI communication

Since many logical functions are split between the AP and the DSIs, a mechanism was created to allow the two parts of each function to communicate. During normal operation, each DSI always has one X.25 virtual call up to the AP. The Remote Internal Communication Handler (RICH) implements another layer of protocol above the X.25 level 3 to handle the multiplexing of the many logical data streams between the AP and the DSI. It allows individual processes within the AP and DSI to communicate by providing an addressing scheme compatible with the LADT internal architecture and adds another layer of error checking.

3.2 Craft interface

An LADT network can be distributed over an area at least the size of a major metropolitan area. The AP provides a centralized craft

Table I—Division of functions between AP and DSIs

Function	AP	DSI
Subscriber call processing		X
Billing data collection		X
Billing data processing	X	
Traffic data collection		X
Traffic report generation	X	
DSI diagnostics		X
DSI trouble-locating procedures	X	
Craft interface	X	
DSI generic storage	X	
DSI recent change craft interface	X	
Central LADT recent change database	X	
Local copies of DSI database		X

interface for the geographically distributed DSIs, with only a limited subset of the craft interface available at the DSI. Locating the craft interface for the network in the AP lowers the number of craft needed to administer the network, makes the DSIs less expensive, and allows the DSIs to be located in unattended offices.

DSIs and DSI subunits can be restored to service, removed from service, and diagnosed from the AP. All recent changes in data for the AP and the DSIs are entered at the AP.

In addition to the maintenance terminal and recent change terminal, which are standard on the AT&T 3B20D computer, LADT provides remote terminals for interfacing to other telephone operating company work centers such as the network administration center, the recent change and memory administration center, and the switching control center. Each center has a command set tailored to run only the LADT commands needed at that center, with only the switching control center and the local terminals allowed to run the entire set of LADT commands.

3.3 Data call processing

The data-call-processing functions of LADT are performed by the DSIs and the packet switch. The packet switch provides a standard CCITT recommendation X.25 interface for direct 9.6- and 56-kb/s lines, while the DSI provides an X.25 level-2 (LAPB) interface for 4.8-kb/s direct subscriber lines and 1.2-kb/s dial-up subscriber lines. With this configuration, the DSI is treated as a host DTE (Data Terminal Equipment) by the packet switch, while the DSI provides an X.25 LAPB DCE (Data Circuit-Terminating Equipment) interface to the subscriber terminal.

The DSI data-call-processing software is responsible for supporting the X.25 protocol on a single 56-kb/s access line to the packet switch and supporting the LAPB protocol for up to 124 subscriber lines. The DSI establishes an X.25 virtual circuit through the packet switch to a remote X.25 host for each call initiated on a LAPB access line. Once the data call setup has been performed, subscriber data flow transparently on this virtual circuit until either end disconnects the call.

3.4 Billing

Billing data collection for subscribers with direct access is done in the DSIs. Single-entry billing is done for calls that cross less than two midnight boundaries. At the end of one of these "standard" calls, the DSI sends a data-call record to the AP that contains connect and disconnect times, packet counts for up to four different rate periods, and a variety of "per-call" data that may be used for detailed traffic studies. If a call extends past two midnights, at each midnight after the first, the DSI sends an interim record to the AP and clears its internal packet counters. A final record is then sent at the end of the call. This method prevents counter overflow on very long calls and prevents all the data from a long call being lost if a DSI failure occurs. The DSI can buffer several hours of billing records if the link to the AP is temporarily unavailable.

3.5 Maintenance

The LADT maintenance functions are divided between the AP, the packet switch, and the DSIs. Each of these units has local diagnostics to detect and localize hardware faults, along with local audits to verify the integrity of the operational software. The AP provides a unified craft interface for maintenance of the AP and of all of the DSIs that communicate with the AP. While the DSI has its own local diagnostics and audits, the DSI reports the results of its diagnostics and audits to the AP, where they are displayed to the craft. The AP also provides a means for the craft to control the execution of diagnostics in the DSI. In order to permit the craft to localize faults that may prevent a DSI from communicating with the AP, the DSI hardware includes a basic display and craft interface at the unit itself. This permits the evaluation of the results of diagnostics that the DSI executes in order to determine why the link to the packet switch cannot be established.

3.6 DSI software generic download

The DSI is a RAM-based processor. The software generic needed by the DSI is downloaded from the AP to the DSI during DSI start-up. The DSI contains only enough ROM to accommodate start-up diagnostics and the software needed to download the generic. Storing the DSI generic in the AP allows the code in the DSIs to be updated without having to change the ROM in each DSI. The generic download function is the only AP-to-DSI communication that does not use the RICH, since the code for the RICH function is part of the generic to be downloaded. Instead, generic download uses its own layer on top of X.25 level 3, optimized to transfer large amounts of data efficiently and with high reliability.

3.7 Traffic

The DSIs collect extensive traffic measurements, which are sent to the AP every 5 minutes for processing. Any errors occurring at the DSI will be reported to the AP immediately. The AP produces two sets of traffic reports, one for normal DSI traffic measurements, and one for DSI error counts. Both types of reports can be generated for 5-minute, 30-minute, and 24-hour periods. The printing of the 5-minute reports is normally inhibited, but they are printed automatically any time a threshold in a report is exceeded. In addition, the craft can manually request a printing of any of the last six 5-minute reports, any of the last 48 30-minute reports, or the last 24-hour report. The content of each of the reports can be tailored through the LADT recent change system.

IV. DSI SOFTWARE ARCHITECTURE

The DSI software system, which executes on the main processor, is divided between ROM-based firmware, which is responsible for bootstrapping the DSI from power-up or system recoveries, and the RAM-based operational software, which is responsible for normal operations of the DSI. The operational software generic is remotely stored on the AP disk file system and is downloaded into DSI main memory by the firmware.

The firmware is designed as a collection of special-purpose programs that are executed sequentially from a single controlling subroutine. Sequential execution is performed at base level while external events such as timer interrupts and packets received from the packet switch access line are processed at interrupt level. Common library routines are available for setting software timers and communicating with the AP. After start-up or system recoveries, the bootstrap firmware is responsible for placing the hardware into a known state, running a minimal set of diagnostics on essential hardware such as the main processor and main memory, establishing an X.25 virtual circuit to the AP for downloading the operational software, and installing the downloaded software into DSI main memory. Once this has been successfully performed, execution will be passed to a known entry point in the operational software. When the operational software begins execution, the real-time DSI Operating System (DSIOS) is initialized; a master-control process is created, which performs data initialization and synchronizes system-process creation; and execution control is turned over to DSIOS.

The core of the DSI operational software is the data-call-processing system, which contains the X.25 protocol programs and access line interface software. Administrative programs collect and maintain LADT administrative data such as billing and traffic measurements. The DSI software includes resident maintenance programs that minimize the impact of failures on system performance and provide the craftsperson with the information required to locate and repair any troubles quickly. Management of the execution of these programs and of system resources is controlled through DSIOS.

The operational software programs execute as processes under DSIOS. A process can be either a system process or a nonsystem process. System processes are permanent, created during system initialization, and should never terminate unless a system reinitialization is performed. Most administrative and maintenance programs execute as system processes. Nonsystem processes are temporary, created upon demand, and normally terminate when the task they are designed to perform is complete. Transient tasks such as hardware diagnostics and call processing execute as nonsystem processes. Except for hardware interrupts, task execution is strictly nonpreemptive. Processes run until they voluntarily return control of the main processor to DSIOS.

4.1 Operating system

The DSI has a general-purpose real-time operating system (DSIOS) designed to meet the specific needs of the operational software. DSIOS manages processes, memory allocation/deallocation, software timers, and intra-DSI process communication. Access to these resources is controlled through a set of DSIOS primitives that have been designed to provide a powerful and efficient, yet simple, interface. A list of the most frequently used DSIOS primitives and their functions is shown in Table II.

Programs execute as processes under DSIOS. Process creation is controlled by the `oscreate` primitive. A program table defines the characteristics of each system program, such as scheduling priority, stack size, and main entry point. At process creation, a stack and process-control block is allocated to the process. The process-control block is used to store the state of the process when it relinquishes the main processor. A process-ready queue is associated with each process priority level. There are three process-ready queues—high, medium, and low. A program's priority is determined by the needs of the entire system. Critical tasks such as DSI clock synchronization are given highest priority, call processing receives medium priority, and deferrable jobs such as maintenance have lowest priority.

When a process is ready to run, it is appended to its respective

Table II—List of frequently used DSI Operating System (DSIOS) primitives

Primitive	Operation
`OScreate`	Create a new process and schedule it to run.
`OSget1type`	Get a message of a specific type. If no message of desired type is waiting, caller can request either immediate return or suspension until message arrives.
`OSsendmsg`	Send a message to a process. If the receiver is waiting for a message of that type, schedule it to run.
`OSrtimer`	Set a timer using the relative clock. A relative timer is used to schedule a process after a specific delay.
`OSatimer`	Set a timer using the absolute clock. Absolute timers are used to schedule a process at a specific time.
`OSktimer`	Kill a timer. This is used by the call processing software to stop protocol timers.
`OSsuspend`	Take a real-time break. The process is placed on the bottom of its priority queue.
`OSexit`	Exit the current process and remove it from the system.

process-ready queue. The DSIOS process scheduler visits the process-ready queues in a fixed pattern. The high-priority queue is visited most frequently and the low-priority queue least frequently. When a nonempty process-ready queue is found, the first process on the queue is removed, the state of the processor is updated by its process-control block, and processor execution is turned over to the process.

Processes within the DSI communicate by passing messages. The `OSsendmsg` primitive allows any process to send a message to any other process, while the `OSget1type` primitive allows a process to receive messages. A process can request a specific message type or any message type. A request for any message will return the first message waiting for the process. Message reception can be blocking or nonblocking. When a blocking `OSget1type` request is made and the requested message is not immediately available, the process relinquishes the main processor. A nonblocking request returns immediately with an indication of whether or not a message was found.

DSIOS memory management is simplified by partitioning memory into preallocated buffer pools. Initializing the DSIOS creates a linked list of free buffers for each buffer type. Allocation and deallocation routines are provided for each buffer pool. When a buffer is allocated, it is removed from its respective free list, and the buffer is tagged with the process identification number for software auditing purposes. This scheme eliminates memory fragmentation, simplifies software audits, and provides a simple environment for software testing and debugging.

Access routines are provided for starting and stopping software timers. Timers are maintained in a circular queue that is serviced on a periodic basis. The circular queue is composed of head cells that point to a linked list of timers. When a timer is set, DSIOS inserts the time on the linked list associated with the appropriate head cell. When the time expires, it is removed from the circular list and placed on the destination process-control block. When a timer stop request is issued, the timer is removed from the circular list, or if the timer has expired, it is removed from the process-control block. Timer delays can be set relative to the current time of day with the `osrtimer` primitive, or for an absolute time of day with the `osatimer` primitive.

4.2 DSI-system start-up

The DSI-system-start-up subsystem is responsible for initializing the DSI to an in-service state for processing subscriber calls. The system is initialized upon start-up or whenever an unrecoverable error is detected. The software for this subsystem is split between the DSI bootstrap firmware that brings the system up from a power-up, and RAM-based operational software that is responsible for initializing the system to an operational state after the operational software has been downloaded into DSI main memory.

4.2.1 DSI boot sequence

Whenever the DSI starts up or the integrity of the operational software is suspect, the DSI must download a copy of the operational software from the AP disk into the DSI main memory. The start-up firmware is responsible for DSI hardware initialization, running out-of-service hardware diagnostics, setting up an X.25 virtual circuit to the AP for receiving the operational software, and downloading the operational software into the DSI main memory.

A goal of the system start-up firmware is to bring up those components of the hardware that are absolutely necessary to communicate with the AP in order to download the operational software. The DSI peripheral hardware associated with interfacing to subscriber access lines (control buffer and clock, DMA processor, group distributor circuits, and subscriber line interface cards) is disabled. Hardware diagnostics are executed on the main processor, main memory, power-control-and-display (PCD), and facility-interface circuit packs to determine whether hardware faults exist that will prevent the DSI from coming into operation. If diagnostics fail, the system cannot be brought into service. The main processor will be halted and a hardware alarm will be asserted. The PCD panel lights will identify the circuit pack and the diagnostic test that failed.

Once the hardware has been initialized and diagnostics have passed, the DSI attempts to establish an X.25 virtual circuit to the AP. If this is successful, main processor control is turned over to the generic

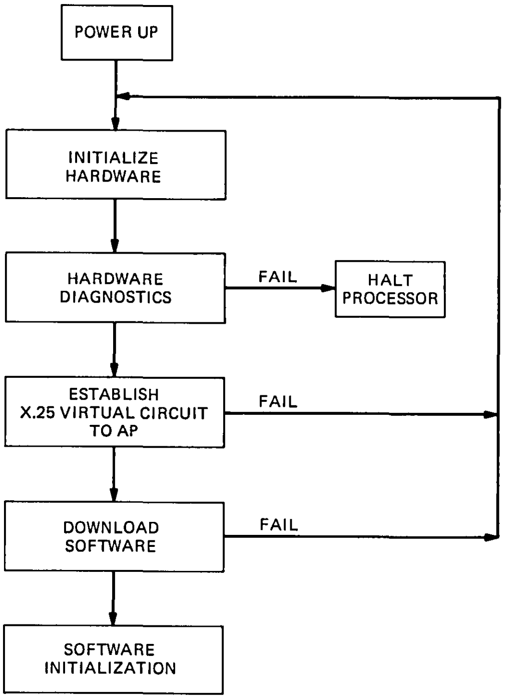


Fig. 12—Flow diagram of DSI boot sequence.

download program, which sends a download request message to the AP. The operational software is transferred from the AP over the X.25 virtual circuit. The generic download program numbers and checksums each message as an added measure of protection. Once the operational software is successfully received, control is passed to a fixed entry point in the operational software. A software initialization will be performed to place the machine in an in-service state for servicing subscriber calls. A flow diagram of a DSI boot sequence is shown in Fig. 12.

4.2.2 Software initialization

Software initialization is the lowest level of DSI recovery. This action is taken immediately after completing a DSI boot or whenever unrecoverable software or hardware faults are detected. A threshold is placed on the number of software initializations allowed during a given period of time. When this threshold is exceeded, a DSI boot is initiated. This causes hardware diagnostics to run and a fresh copy of the operational software to be downloaded.

The entry routine of the operational software initializes DSIOS, creates a process called start-up control, and turns main processor

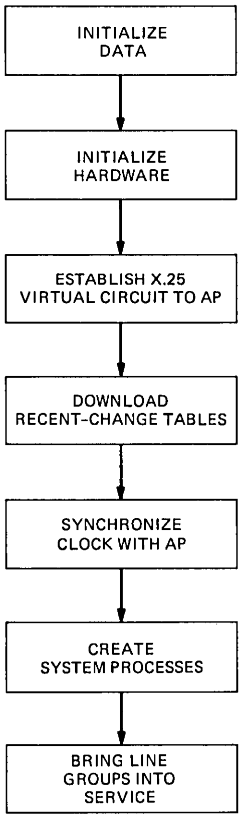

INITIALIZE DATA
INITIALIZE HARDWARE
ESTABLISH X.25 VIRTUAL CIRCUIT TO AP
DOWNLOAD RECENT-CHANGE TABLES
SYNCHRONIZE CLOCK WITH AP
CREATE SYSTEM PROCESSES
BRING LINE GROUPS INTO SERVICE

Fig. 13—DSI start-up flow sequence.

control over to DSIOS. The control flow sequence for the start-up control process, which coordinates bringing the DSI into service, is shown in Fig. 13. The start-up control reinitializes the DSI hardware, initializes shared system tables, and creates all of the system processes. After hardware initialization, the X.25 protocol over the packet-switch access link is reestablished. An AP/DSI communication process is created to establish an X.25 virtual circuit to the AP for receiving critical data and craft requests. Once a communication path is established to the AP, a process is created for downloading the recent change tables from the AP. A process is then created to retrieve the current time-of-day from the AP. The DSI needs an accurate time-of-day clock for call billing purposes and for scheduling time-of-day events. After updating the time-of-day clock, all other system processes are created. The start-up control waits for completion messages from each process before it brings the line groups into service. Once the line groups have been brought into service, the DSI is ready to receive subscriber calls.

4.3 Administrative software

The administrative functions of LADT such as billing, traffic measurements, and recent change are split between the DSI and AP. The DSI administrative software programs serve to collect and update the administrative data. Data are collected and sent to the AP for processing. The processes that perform these operations in the DSI communicate with their counterpart routines in the AP through the Remote Internal Communications Handler (RICH) software subsystem.

4.3.1 DSI/AP communication

Data exchanged between the DSI and the AP are multiplexed over a single X.25 virtual circuit. A DSI process called the RICH process manages this data stream. An access routine sends messages to AP processes. The DSIOS message-sending primitive `ossendmsg` routes messages from the AP to the DSI processes. To simplify the DSI/AP RICH process interface, no flow-control or acknowledgment procedures are used. The features of X.25 level 3 accomplish this. The user process must provide additional data integrity procedures.

The DSI/AP virtual circuit is established during software initialization. The DSI-RICH process attempts to reestablish this connection upon detecting a failure without having the system recovery software intervene. This allows the DSI to provide subscriber service when communication with the AP fails. All transmission requests to the AP are denied when the DSI/AP communication path is not operational.

4.3.2 Billing

The DSI collects billing data for each subscriber call. Data such as holding time and packet counts are maintained in a virtual call record for each call. When a call terminates, the call processing software sends the virtual call record to a DSI per-call billing process. This process then sends the virtual call record to the AP for storage on a magnetic tape.

Because of the importance of the billing information, the design of the per-call billing process prevents the loss of billing data if the communication path to the AP is lost. During an outage, virtual call records are stored in the DSI, and the per-call billing process periodically attempts to retransmit the stored records to the AP.

Calls that extend over two or more consecutive midnight boundaries are given special attention. A special process called long duration billing handles these calls. This process runs every midnight and scans each virtual call record to determine whether it extended over two midnight boundaries. If so, a long duration record is created from the virtual call record and is sent to the AP for storage.

Periodic checks are made to ensure that billing records are not lost between the DSI and the AP. A billing-tracer process sends counts to the AP every hour, allowing the regional accounting office to keep track of billing records. If any billing records are lost, the regional accounting office can detect this by comparing the number of billing records received with the tracer counts.

4.3.3 Traffic

The DSI collects measurements that are used for off-line evaluation of system demand, subscriber calling patterns, service, and DSI component utilization. Counters are pegged by the operational software and are collected and sent to the AP for storage every 5 minutes by a DSI traffic process. Certain measurements, such as queue lengths and the number of active calls, are sampled on a more frequent basis by the DSI traffic process and are also sent to the AP.

4.3.4 Recent change

Certain information in the DSI changes independently of the operational software and is referred to as recent-change data. The recent-change subsystem maintains these data in system tables and provides access routines for other programs. Because of the critical importance of these data, checksums are included as part of the data tables, and the tables are write-protected.

Recent-change information can be altered by the craft at any time. The recent-change data are managed by the recent-change-system process that is created immediately after the DSI/AP virtual circuit is established during software initialization. The recent-change process sends a download message to the AP asking for its tables that are stored on AP disk. If the AP recent change has reason to suspect that the recent-change information is not correct, it will ask DSI recent change to verify that the tables at a DSI are up to date. If these tables are out of date, the AP recent change will begin a download of all current information.

An update interface exists between the DSI recent change and its counterpart in the AP. When the craft enters a change at the AP, the updated recent-change tables are automatically sent to the DSI. The same procedures are followed as those for the original recent-change table insertion. Equipment status updates may require that an equipment remove or restore to service request be sent to the DSI equipment maintenance manager software.

4.3.5 DSI/AP clock synchronization

Accurate billing requires that the DSI be synchronized to the AP time-of-day clock. Time changes may be necessary because of DSI clock drift or changes to or from daylight savings time. A DSI time-change-system process manages corrections to the DSI time-of-day clock. Messages that exchange the time of day between the DSI and AP are used to update the DSI clock. The time-change process must note these changes in active billing records when the time of day is updated. Special attention must be given to changes that cross midnight boundaries. Timers that were set to expire based on the time of day must also be given special attention. A DSIOS primitive is provided to adjust these timers. The DSI clock can also be adjusted by craft command from the AP.

4.3.6 Emergency program update

Facilities are built into the DSI that allow the craft to examine and modify DSI data and text during in-service operation. Formatted octal or hexadecimal memory dumps of DSI memory or a snapshot of system process status information can be requested. A memory-patch facility allows emergency modifications to the operational software. This allows field updates without having to remove DSIs from service and without having to issue new software. Software patches are stored on AP disk and downloaded after the operational software is downloaded. Patches may be installed by the craft any time after the DSI is brought into service.

A software patch is installed by downloading the replacement software into a block of memory reserved for software patches. An assembly language jump instruction is inserted in the program text that is in error. This will transfer execution to the replacement software. The last executable statement in the replacement software will jump back to the executing program.

4.4 Data call processing

The data communication functions of the DSI are provided by the data-call-processing software. Data call processing controls the interfaces to subscriber access lines, the packet-switch access line, and provides text messages to subscribers during call setup. Each subscriber call is handled by a separate process called a call process. The call process performs both the data transfer functions specified in X.25 and LAPB, as well as handling the call setup and termination procedures. Each call process executes the same collection of shared protocol access programs. Per-call state information is kept in a data block that is allocated by DSIOS upon the call process creation. Each call process allocates a virtual call record for storing billing information. A call process is created when a subscriber initiates a request for service over a subscriber access line. Receipt of an X.25 LAPB frame initiates a request for service. This process will exist until the call is torn down. At that time, the call process will return any system resources to DSIOS and terminate itself.

Routing tables are used for associating packets received on an access line with the call process that handles the packet. A subscriber-line routing table is used to route all X.25 LAPB frames received on subscriber access lines to call processes. One entry exists for each line. Likewise, a network-channel routing table is used to route X.25 level-3 packets received on the packet-switch access line to the appropriate call process. One entry exists for each X.25 level-3 logical channel supported on this interface. Figure 14 shows a simplified view of packet routing and call-process architecture.

4.4.1 Call-control software

Coordination of call setup and termination procedures is performed by a program called call control, which is executed by the call process. Its responsibilities include allocating resources for the call when the call process is created, prompting the subscriber for a destination address, originating a request to X.25 level 3 to set up the X.25 virtual circuit, transmitting call progress messages to the subscriber terminal, coordinating call termination, and deallocating any system resources before the call process terminates.

4.4.2 Subscriber-access-interface software

The subscriber-access-interface software manages the software interface to the subscriber access lines. The main processor interface to subscriber lines is through the DMA processor. Interactions with the DMA processor are made through a hardware-control FIFO, a hardware status FIFO, and memory that is shared by both processors. The DMA processor interrupts the main processor when an entry is placed into an empty status FIFO. A DMA-processor-interface package provides a software interface to the DMA processor. Access routines are provided for transmitting control and data to the DMA processor. These routines are robust enough to allow use by both call processing and DMA processor diagnostics. Per-line status information is routed to the call process identified in the subscriber-line routing table.

The state transitions of dial-up line modems are controlled by DSI software. Modem status information is placed in shared memory by the control-buffer-and-clock circuit pack. A system process called the dial-up control process periodically scans this information looking for modem status transitions, such as ringing and carrier detected. When the dial-up control software detects that a subscriber is calling into

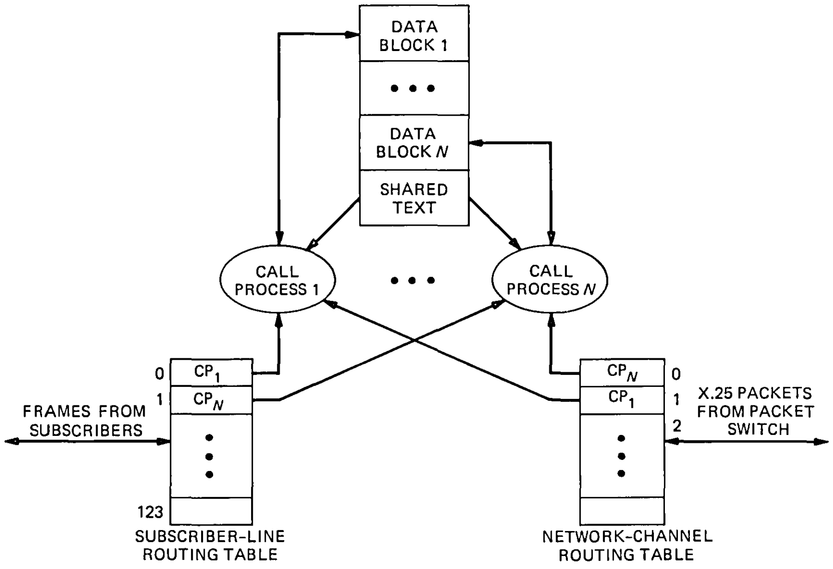


Fig. 14—Simplified diagram of call-process architecture.

the DSI, it takes the modem through the necessary handshake procedures for establishing an X.25 level-1 connection. Access routines are provided for external control of dial-up line states by other software systems.

A subscriber data-link manager called the customer-data-link status system process is responsible for serving events that may affect the status of subscriber access lines. For example, call waiting tones on a telephone line should not affect subscriber data service. The customer-data-link process must differentiate between the loss of the level-1 connection and the call waiting tones. If the link remains idle for more than a predetermined period of time, the customer-data-link status process assumes that the level-1 connection is lost and notifies the call process.

4.4.3 X.25 protocol controller interface software

The DSI supports an X.25 interface on the packet switch access line. A specially designed Large-Scale Integration (LSI) device called the XPC performs the entire X.25 level-2 protocol procedures. The X.25 level-3 protocol processing is performed by the call process. Because the X.25 level-3 processing is distributed across many call processes, a special system process, called the level-3 status process, globally monitors the status of the entire X.25 level-3 interface. Events such as packet-switch link failure, which affects all X.25 virtual circuits, are reported to this process. Handling of X.25 level-3 packets on the logical channel 0 is also performed by this process.

An XPC-interface software package, similar to the one used for the DMA processor, provides a software interface to the XPC device. Access routines are provided for transmitting level-3 packets and controlling the operation of the level-2 protocol. Priority queues are used to buffer packets sent by level 3 to the XPC device for transmission. The priority of a packet is specified in the packet buffer that contains the level-3 packet that is passed to the XPC-interface software. When the XPC device requests a packet for transmission, the highest-priority queue is serviced first.

The logical channel number of a received packet is used to index into the network-channel routing table for routing purposes. Any packet received on an inactive channel is routed to the level-3 status process.

4.5 Craft interface

The DSI craft-interface subsystem includes those programs that allow operations personnel to monitor and control the operations of the DSI. There are two methods that the craft can use to interact with the DSI. The Maintenance Terminal (MTTY) at the AP provides the craft with remote control of the DSI. Requests to execute hardware diagnostics, software audits, and requests to remove and restore DSI hardware can be sent to the DSI from the MTTY. The front panel on the power control and display circuit pack contains switches and LED indicators, which provide the craft with local control of the DSI and an indication of its current state.

4.5.1 Remote interface

Remote craft requests from the AP are routed to the DSI through the AP/DSI RICH communications package. There is no central DSI craft-input program. Craft requests are routed to the destination process by DSI-RICH software. Each destination process must then parse the input message to extract the information that is needed.

4.5.2 DSI power control and display panel administrator

Craft inputs can be made directly to the DSI through the Power Control and Display (PCD) panel circuit. Requests to remove or restore a line group or the entire DSI are made by activating switches on the panel. The PCD circuitry generates a hardware interrupt to the main processor when a switch is activated or a line group is powered on or off. When a PCD interrupt is detected, a PCD nonsystem process is created to handle the request in a background mode. This allows the DSI to continue processing calls.

When a line group is turned on, a PCD line-group diagnostic is automatically requested by the DSI maintenance software. Should the diagnostic fail, an alarm light is enabled on the panel and the line group is not brought into service. Otherwise, if the toggle switch associated with the line group is in the "restore" position, a request to restore the line group to service is made. Conversely, when a line group is turned off, a line group "remove request" is made by the PCD panel administrator.

The PCD software enables the "Request In Progress" (RQIP) light on the panel associated with the equipment. When the request has been completely processed the RQIP light is disabled, and the panel switch is examined to determine whether another request is pending.

The DSI has two 7-segment displays that are used to inform the craft of DSI status. Software-access routines are provided for controlling these displays. During system start-up, the PCD display is used to track the progress of DSI software initialization. After start-up is completed and the DSI is ready for normal operation, the display indicates the occurrence of certain errors.

4.6 Equipment maintenance

Equipment maintenance is the name given to those programs designed to control and monitor the operation and configuration of the DSI hardware. Equipment maintenance minimizes the impact of failures by providing the operating personnel with specialized hardware, software, and human interfaces to allow rapid detection, isolation, and repair of troubles. The specific tasks involved with equipment maintenance are restoring, removing, and diagnosing the DSI hardware, routinely exercising the system to ensure proper operation, and analyzing the system for problems.

4.6.1 Maintenance request administration

The central controller of equipment maintenance is a system process called the Maintenance Request Administrator (MRA). MRA coordinates equipment removal and restoral requests and hardware diagnostic requests. It also responds to inquiries on the state of equipment in the DSI. These requests can be made remotely from the MTTY, from the PCD panel, or from other DSI processes.

For nonstatus requests, MRA spawns nonsystem processes to handle each request. These processes are responsible for validating the request message parameters, determining whether the request can be honored given the current state of the machine, and returning a result message to the requesting process.

Requests to remove or restore equipment can be conditional or unconditional. Conditional requests for bringing equipment into service require running a diagnostic on the equipment. If the diagnostic fails, the request will be denied. Conditional requests to remove equipment from service will cause the MRA software to remove idle equipment and to camp-on busy lines. A camp-on procedure requires that the line be placed in a maintenance busy state. The MRA software will wait for the call to terminate before removing the line from service.

Unconditional requests for restoring equipment to service are handled immediately without running diagnostics. Unconditional requests to remove equipment from service cause any active calls on the affected lines to be terminated. Call termination messages are broadcast to the affected call processes to allow graceful call termination. After a brief delay to allow the broadcast messages to be transmitted, the lines will be removed from service.

When a diagnostic request is issued, the MRA determines which set of diagnostic phases are to be run. It then sends a diagnostic request message to the diagnostic software, which defines what phases to execute and how to execute them. Only one diagnostic request can be serviced at a time. If an additional diagnostic request is received, it will be queued until the currently executing diagnostic has completed.

4.6.2 Diagnostics

DSI diagnostics are those programs that test the DSI hardware and report detected faults. The goal in executing these diagnostics is to isolate errors to a single circuit pack as quickly as possible so that the faulty circuit pack can be replaced and the unit restored to full service.

To locate faults quickly, the diagnostics are organized so that they exercise the DSI hardware in layers. During the first stage, the first layer (the main processor and main memory) is diagnosed. Next, the other circuit packs in the processor complex are exercised, and finally, the group-distributor circuits and line cards are tested. By running diagnostics in this order, the core of the DSI is verified before any peripheral circuit packs are tested. Therefore, any errors uncovered are probably located in the circuit under test and are not the result of faults in a more central circuit pack.

Because the DSI is a simplex system, the diagnostics are divided into two categories: those that can execute without affecting the normal operation of the DSI and those that must be run when the DSI is out of service. Because the DSI downloads its operational software (including most of the diagnostics), it must have the capability to verify the minimal system needed to complete the download process before the download occurs. Therefore, some diagnostics are present in the bootstrap ROM for execution during a DSI boot sequence.

The diagnostic programs in the ROM- and RAM-based software systems consist of a two-part structure. A diagnostic controller schedules individual tests, reports errors, and performs the supervisory tasks needed to execute the diagnostics. The second part of the diagnostic structure consists of the diagnostic functions, or phases. Each phase contains multiple tests and is designed to exercise a portion of a particular circuit or subsystem. The first test of each phase, called the prologue, sets the environment for the hardware to be exercised. The last test of each phase, called the epilogue, restores the environment for normal operation when the diagnostic phase is complete. The phases contained in the operational software are organized so that a subset may be performed with the DSI in operation without affecting normal call-processing activities.

Diagnostic requests are received by the MRA, which subdivides the request into logical parts that exercise sections of the hardware at given intervals. After determining the logical breakdown, MRA creates the diagnostic-control process, which controls the execution of the diagnostic phases. Diagnostic failure messages are sent back to the AP MTTY. A completion message is sent back to the MRA software.

There are various modes of diagnostic operation that can be enabled in the diagnostic request at the AP MTTY. Running a diagnostic in "raw mode" causes each diagnostic failure to be reported back to the MTTY. Normally only the first failure is reported. The repeat mode allows the diagnostic to be executed more than once by a single craft command. The trouble-location-procedure mode causes diagnostic failures to be routed to software in the AP, which will output additional information about the probable cause of the diagnostic failure on the MTTY.

4.6.3 Routine exercises

The goal of the routine-exercises subsystem is to thoroughly verify the integrity of all hardware units of the DSI on a periodic basis without disrupting service to the subscriber. It detects hardware failures before they affect system performance. Routine testing is done during periods of low traffic. The routine-exercises subsystem includes line tests and the nondisruptive diagnostics of the common equipment. During routine exercises only lines without active calls are diagnosed.

Since the low traffic period of each DSI varies (observed by the operating personnel), the run time of the routine exercises can be changed depending on the low traffic period of the office. The routine-exercises trigger time is altered via recent change by the craft. Routine exercises can be disabled by recent-change input.

Routine exercises will first run in-service main-memory and main-processor diagnostics. Failures are reported to the craft. Next, routine exercises diagnoses each idle line. If a line diagnostic fails, a request is sent to MRA to remove the line from service. Craft will be notified that the line was removed. The number of lines that can be removed during a routine-exercises cycle is limited by a recent-change parameter.

4.7 System integrity

Service to the subscriber may be affected by hardware failures, software errors, data irregularities, and system overload. The system integrity software is designed into the DSI to minimize the impact of these failures on system performance.

4.7.1 Sanity detection

Basic system sanity is monitored by a hardware sanity timer. The timer is initialized by the start-up software and generates an unmaskable interrupt when it expires. Should the timer expire, software initialization is initiated. During normal operation, the system process called the sanity-timer reset process is responsible for periodically resetting the sanity timer. Should the system stop cycling for any reason, the sanity timer will automatically cause the DSI to recover without manual intervention.

4.7.2 Exception handler

Error handling is centralized in the DSI by providing a software interface through which all software and hardware errors are reported. This software subsystem, called the exception handler, is used to report and, when necessary, correct errors. The implication of each exception error may have different meanings for different processes. For example, certain processes may be able to recover from a temporary inability to allocate a system buffer, while others may not. To accommodate this flexibility, the exception-handler interface allows each process to pass a parameter that suggests the recovery action that should be taken. The final decision about error recovery is built into the exception handler for errors that have global system impact.

Each exception error has a unique error code. There are three categories of errors. The least severe are report errors. Software errors that do not affect system performance fall into this category. Report errors are reported to the AP and, in certain cases, a software audit is scheduled to run. Threshold errors can affect system performance if they happen too frequently. Each threshold error has a counter associated with it, which is incremented when the error occurs. Another system process, the error-analysis subsystem, periodically looks at these counters to determine whether further recovery action is required. The third category, called recovery errors, are serious errors that affect system performance and must be handled immediately. The DSI recovery subsystem is notified immediately to take corrective action.

4.7.3 Recovery

Recovery software is resident within the DSI to handle critical hardware and software faults that affect the entire system. Recovery takes immediate action to contain the failure before it becomes widespread. Since the DSI is a simplex system, hard faults cannot be circumvented and the affected equipment may have to be removed from service. Recovery actions taken include DSI boot, software initialization, packet-switch link reinitialization, line-group removal, or DSI removal. Errors are reported through the exception-handler software.

For each reset action, a report and alarm is issued to alert the craft about the failure. Should a recovery action require a DSI boot, the recovery software will store system status information in protected memory. This system snapshot, referred to as the postmortem, is transmitted to the AP for off-line analysis after the DSI boot has completed.

Neither the DSI recovery nor the DSI initialization invokes diagnostics directly. To find out more about the failing device, the AP craft must initiate a diagnostic session.

4.7.4 Error analysis

The main function of error analysis is to detect DSI hardware errors or abusive subscriber terminal conditions and isolate them before they can degrade DSI performance. Error counters maintained by error analysis are incremented by the operational software or through calls to the exception handler. The error counters are treated as "leaky bucket" integrators. Error analysis decrements them periodically and also checks to see if they have exceeded threshold. The decrement frequency and threshold of each counter is a function of the type of error condition which that counter is tracking.

Two types of error counters are maintained by error analysis. General errors are the result of faults from common equipment. When the threshold for any of the errors is exceeded, the recovery software is notified immediately to remedy the situation. Line errors can be isolated to a single subscriber access line or a line-card circuit. Line errors are pegged directly by the call-processing software. Depending on the type of line equipment and the error condition, a request to terminate a call can be sent to a call process if the line is active, or a request can be sent to the MRA to remove the line from service.

4.7.5 Audits

System audits protect against the loss of system integrity due to corrupted transient data. Transient data include routing tables and system resources such as packet buffers and message buffers. Audits are performed on a routine basis and upon demand. Data irregularities are corrected at the point of discovery.

The audit-controller process administers audit requests and schedules audits to run. Demand audits, requested by the DSI software through the exception-handler software, receive immediate attention by the audit controller. These requests are made when certain errors indicate a possible loss of resources or when the number of available key resources drops below a system threshold. Manual audits, requested by the craft, can be scheduled at any time but are queued behind demand audits. The lowest-priority audits are routine audits. They are scheduled frequently enough to support the main-line execution, but not so frequently as to affect system capacity.

For every irregularity detected, a report is generated that provides enough information about the irregularity for off-line analysis. Because a single data irregularity may have multiple side effects that could flood the AP with many reports, the number of audit reports is limited for each invocation of the audit controller.

4.7.6 Overload control

The objectives of the overload-control subsystem are to smooth variations in traffic loads, to provide normal service to existing subscribers, and to prevent an overall system outage due to overload. The overload controls are aimed primarily at controlling packet-buffer congestion. The purpose of packet-buffer congestion procedures is to avoid the negative consequences of running out of packet buffers. Running out of packet buffers could result in calls that are deadlocked or prematurely dropped. Overload monitors are built into the main-line code. When critical resources drop below defined thresholds, overload actions are taken. Several levels of control are allowed by having two overload states. Hysteresis is used to smooth the transitions between overload states. Overload controls include flow controlling access lines and blocking new call originations.

4.8 DMA-processor firmware

In addition to the software that runs on the DSI main processor, the DMA processor is controlled by its own firmware, which resides in a separate ROM on the DMA-processor circuit pack. This firmware is designed to operate strictly in response to external events, and it includes both operational and maintenance programs. The DMA-processor operational software is an extremely compact and efficient set of assembly language routines. These routines are invoked by a main routine, which cyclically scans the FIFO memory interfaces to the main processor and the multiplexed protocol formatter for entries. The scanning procedure has been designed to optimize the DMA-processor's response time to critical events. When an entry is detected in one of the FIFOs, the DMA processor reads the entry, takes appropriate action, and then returns to the main scanning routine. In addition to the operational software, the DMA processor contains certain limited diagnostics and maintenance software. Many of the maintenance routines are divided into short sections, each of which executes very quickly. This permits these routines to run while the DMA processor is processing customer information without affecting service.

4.9 Data flow through the DSI

In order to help clarify the functions of many previously described DSI software subsystems, and in particular the data-call-processing subsystem, this section traces the progress of a data call through the DSI. The treatment is developed for a dial-up data call, although the sequence of events for a dedicated access data call is similar, but does not require a level-1 connection to be established.

4.9.1 Establishment of a level-1 connection on dial-up lines

When a subscriber dials the LADT access number, the public telephone network routes the call to a line in a line hunt group to which a DSI is connected. The terminating voice switch applies ringing to that line. The modem line circuit detects the presence of ringing and sets an internal-status bit. When the dial-up control-unit software scans these bits and detects ringing, it commands the modem line circuit to go off-hook and answer the incoming call.

After two seconds, the dial-up control-unit software commands the modem line circuit to send an answer tone. When the subscriber modem receives an answer tone, it should respond by sending carrier. If the subscriber terminal does not send carrier within 18 seconds, the modem line circuit will go back on-hook. When the dial-up-control-unit software detects carrier, it commands the modem line circuit to send an answering carrier.

After 1 second, the multiplexed protocol formatter will start transmitting flags on that line. A packet buffer is allocated for receiving the first LAPB frame from the subscriber terminal. If the LAPB protocol is not established within a specified period of time after flags are sent, the dial-up control-unit commands the line to go back on-hook.

4.9.2 Call set-up/termination flow

For a dial-up subscriber line, the DSI must go through a modem handshake sequence to establish the level-1 (circuit-switched) connection. For in-service dedicated lines, the DSI is always sending flags to the terminal indicating a willingness to establish a level-2 connection. After a level-1 connection is established, the subscriber may request service by establishing the X.25 level-2 LAPB protocol. When a frame is received on a subscriber line whose entry in the subscriber-line routine table is free, the DMA processor interface software creates a call process. The routing table is updated to point to the newly created call process, and a message identifying the subscriber line number and the received frame are forwarded to the call process.

The first frame expected by the DSI LAPB protocol is a Set Asynchronous Balance Mode (SABM) command for establishing the level-2 protocol. Non-SABM frames are discarded and the call process is terminated. Otherwise, an acknowledgment is transmitted to the terminal and the call-control program is invoked. Call control transmits an ASCII prompt to the terminal requesting a destination address. These signaling messages between call control and the terminal are transmitted in LAPB information frames. If a destination address is not received within a specified period of time, call control will terminate the call process.

When a valid address is received from the user, call control selects a free logical-channel number entry from the network routing table. The routing table is updated to associate the call process with the selected logical channel. Call control creates an X.25 level-3 *call-request* packet and invokes the X.25 level-3 program to set up a call through the packet switch. A *call-attempt signaling message* is transmitted to the terminal informing the user that the call is being placed. Call control waits for an acknowledgment from level 3 that a virtual call has been set up and periodically sends the *call-attempt signaling message* to the terminal. When the call-connect packet is received, it is passed to call control for a facility field verification. If the parameters are valid, a *call-connected signaling message* is sent to the user's terminal. After the call is set up, all data are passed transparently on the virtual circuit.

A call can be terminated by the subscriber terminal, remote host DTE, or by the LADT network due to internal errors. A subscriber terminal can terminate a call by transmitting a LAPB Disconnect Command (DISC) to the DSI or by terminating the level-1 protocol (hanging up). The remote host DTE terminates the call by clearing the virtual circuit. The DSI can initiate call termination by either of the above methods.

When the DSI receives a DISC command from the subscriber terminal, an acknowledgment is sent to the terminal and the subscriber-line routing table is restored to the free state. Call control commands level 3 to clear the virtual call. The call process will wait for an acknowledgment (or time-out) from the packet switch. When the acknowledgment is received, the network-channel routing-table entry is restored to the free state. The billing record is sent to the billing process and the call process terminates.

4.9.3 Call flow for packets received from terminals

The largest percentage of the DSI main-processor real time is used to switch ordinary information packets after the data call has been established. The DSI data-call processing has been designed to be real-time efficient in effecting such packet transfers. The paragraphs that follow describe the logical steps of the hardware and software interaction for a packet received from a subscriber terminal.

When the multiplexed protocol formatter detects the start of a packet, it builds up the first byte of the packet by accumulating the data bits. After it has assembled an entire byte, it transfers the data byte to the DMA Processor. The DMA processor and the main processor communicate by using bidirectional hardware FIFOs for time-critical events, and by using a shared memory accessible by both the DMA processor and main processor for routine events. When the DMA processor receives the first byte of a packet, it looks into its shared memory for the address of the packet buffer that has been previously allocated for that line. If a valid packet-buffer address is present, the DMA processor transfers the byte into main memory at the address specified by the packet buffer. The DMA processor also puts a message into its FIFO for the main processor to indicate that it has started to receive a packet from the subscriber line. As the multiplexed protocol formatter assembles additional bytes, it passes them on to the DMA processor, which in turn transfers the bytes into the packet buffer in main memory.

While the packet is being transferred into main memory, the DMA-processor interface software reads the FIFO from the DMA processor and discovers that the subscriber line has started to receive a packet. The DMA processor will need a new packet buffer after the packet reception is complete, so a new packet buffer is allocated and its address is written into the DMA processor's shared memory. This preallocation of packet buffers is necessary so that another packet can be received immediately after the end of the current packet is detected.

When the multiplexed protocol formatter detects the end of the packet, it confirms that the reception was accurate by checking the CRC at the end of the packet and then informs the DMA processor that the packet reception has been completed. The DMA processor then writes an entry into the FIFO to the main processor indicating that the subscriber line has just received a packet with a good CRC.

After the DMA-processor interface software reads this entry from the FIFO, it routes the packet to the call process associated with the subscriber line. All inputs to a call process are funneled through a single entry point that is waiting for messages. When a LAPB frame is received, the LAPB protocol is invoked to process the frame. The LAPB program will transmit responses to control frames while data frames are passed on to the X.25 level-3 program for transmission to the packet switch.

The X.25 level-3 program queues data received from the subscriber terminal if the virtual circuit to the remote DTE has been flow-controlled or the flow-control window is full. Otherwise, it inserts an X.25 level-3 data-packet header and calls a packet-switch access routine to transmit the packet. The XPC-interface software queues the packet for transmission based on the priority of the packet specified in the packet buffer by level 3. The XPC device notifies the main processor when X.25 level-2 acknowledgments for a packet are received form the packet switch. This allows the XPC-interface software to deallocate the packet buffer.

4.9.4 Call flow for packets received from packet switch

The packet flow through the DSI from the packet switch is similar to that for packets received from a subscriber terminal and is not covered in as much detail. The hardware, firmware, and software interaction for transmitting a packet is also similar to that required for receiving a packet and is not elaborated in detail. The XPC device interrupts the main processor after an incoming packet has been placed into main memory by DMA. The XPC-interface software driver extracts the X.25 level-3 logical-channel number from the level-3 header for routing purposes. If the routing-table entry is free or invalid, the packet is routed to the level-3 status process. Otherwise, the packet is routed to the appropriate call process. The call process will invoke the level-3 program when it receives a packet from the packet switch. When level 3 determines that a valid data packet has been received, it invokes the LAPB protocol program to transmit the data out to the subscriber terminal.

LAPB will queue the data if it is flow-controlled or the flow-control window is closed. Otherwise, it inserts a LAPB header and invokes a DMA-processor-interface software transmit routine which queues the frame for transmission. Each subscriber access line has an output transmit queue. If the transmit queue is empty, the packet buffer address is placed in shared memory and a command is sent to the DMA processor to start transmission. If other frames are queued, the frame is placed at the end of the transmit queue. The DMA processor will interrupt the main processor when it has completed a transmit request. The DMA-processor-interface software will then initiate another transmission on that line if a frame is queued.

V. ADMINISTRATIVE-PROCESSOR SOFTWARE ARCHITECTURE

The AP is implemented under the DMERT (Duplex Multi-Environment Real Time)[13,14] operating system that runs on the AT&T 3B20D computer.[15] The 3B20D computer was chosen for its high reliability. All AP software is written in C programming language, most of it residing at the user level of the operating system, with only a few real-time critical functions running at the kernel-process level. Figure 15 shows the internal architecture of the AP software.

5.1 X.25 handler

The AP communicates with the rest of the LADT network over one 9.6-kb/s link running levels 2 and 3 of the X.25 protocol. The AP-X.25 implementation is based on the DMERT-supplied X.25 package. The DMERT package was modified to support dynamic virtual calls and to conform to the version of X.25 supported by the packet switch.[16] The LADT implementation of X.25 retains the basic file interface for the *UNIX** operating system provided by DMERT. However, a new interface was added to notify AP processes that use X.25 about changes in the status of X.25 calls. Each of these AP processes attaches to its own DMERT message port, to which the X.25 handler sends a DMERT message whenever a call to that process is set up or cleared.

Since the DSIs all use identical copies of the DSI software, a DSI has no way of knowing which DSI it is. However, the packet switch can identify a DSI by the physical link on which the DSI call originates. If the DSI does not fill in the originating address in the call request packet, the packet switch will fill in an originating address unique to that DSI. The AP can then use this originating address to identify the DSI. The table of legal DSI addresses is stored in the DMERT-supplied equipment configuration database, and the AP will refuse any call that is not from one of these addresses.

5.2 Generic download

Since the DSIs are RAM-based, their operational generics must be downloaded from the AP. There is a ROM-based start-up program in the DSI that includes some hardware diagnostics, a partial X.25 level-3 implementation, and the DSI portion of the generic download

* Trademark of AT&T Bell Laboratories.

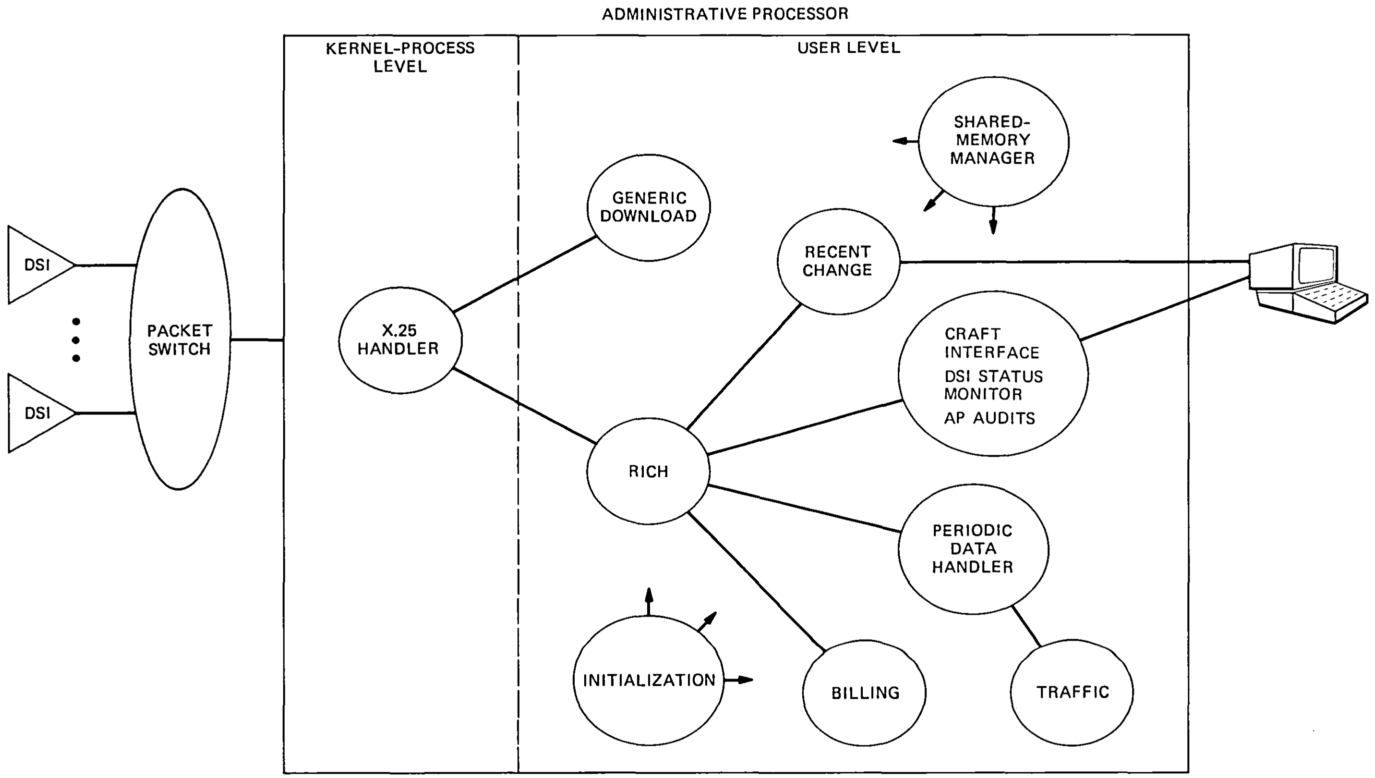


Fig. 15—Internal architecture of the administrative-processor software.

program. At start-up, the DSI first runs its internal diagnostics and then sets up a call to the generic download process in the AP.

When the AP receives a request from a DSI for a download, it first checks a history file to see when the DSI was last downloaded. If the DSI has requested an excessive number of downloads within a specific period of time, the AP will alert the craft and refuse to download the DSI. If the download request is accepted, the AP generic download function checks the LADT database to determine which DSI generic should be sent. The generics are divided into 1792-byte blocks, with a checksum over each block. When the DSI receives each block, it computes the checksum, and then sends an acknowledgment back to the AP for that block. The AP end of generic download will only transmit blocks to the DSI as long as there are fewer than five unacknowledged blocks outstanding. This windowing mechanism is used so that the AP does not have to always idly wait for the acknowledgments, but will also not needlessly send several bad blocks after an error has occurred. If the DSI does report that there was an error in receiving a block, the AP will resume sending the generic starting at that block. If the download is interrupted for external reasons such as a failure of the DSI-to-packet switch link, the DSI generic download program keeps track of the last good block that it received and notifies the AP to start at that block number when it re-establishes the call.

5.3 Shared-memory manager

When large amounts of data must be moved between two processes, one of the most efficient methods is through the use of shared memory segments. The shared-memory manager is a process that centralizes the handling of shared-memory segments in the AP. When the AP is initialized, the shared-memory manager reserves all the shared memory segments using DMERT's *UNIX* system calls. Each shared memory segment has a process associated with it that is considered to be the master of the segment. For example, the RICH process controls the shared memory that is used to pass RICH transactions between it and application processes.

When the AP reinitializes, it is not known which processes will need to be restarted. Because of this, it is conceivable for user processes to request to be attached to shared memory before the memory has been initialized by the master process. To circumvent this possibility, the shared-memory manager waits until the master process for each segment notifies it that the segment has been initialized before it allows the other processes to attach to the shared-memory segment.

Besides controlling the initialization sequence, the shared-memory manager also allows a greater degree of fault tolerance in the AP architecture. If each master process reserved its own segment using DMERT's *UNIX* system calls and the master process terminated, it would lose track of the memory segment and all the data stored in it would be lost. With the shared-memory manager, if a process terminates, it can ask the shared-memory manager for the correct memory segment when it reinitializes. Since the shared-memory manager is a simple process, it is considered less likely to terminate than some of the more complicated processes.

5.4 Remote internal communications handler

The only way the AP can communicate with the DSIs is over X.25 virtual circuits. However, there are many processes in each DSI that the AP needs to address, such as maintenance, recent change, billing, and traffic. If one virtual circuit was assigned between the AP and each DSI for each individual process that needed to be addressed, the AP would have to terminate several hundred virtual circuits.

To avoid having to terminate many virtual circuits in the AP, a new internal protocol, RICH, was developed, which allows most of the AP-to-DSI communication to share one virtual circuit between each DSI and the AP.

A RICH transaction consists of a 12-byte header and up to 244 bytes of user data. RICH addresses are 4-byte integers that allow processes to be addressed by a combination of their function (such as billing) and the type and instance of their host processor (such as DSI number 3). Within a given process address, up to 100 RICH transaction types may be locally defined. For instance, the maintenance processes could use different transaction types to distinguish between different types of diagnostic messages that are returned from the DSIs. A destination process can ask to be given either the oldest message of a given type or just the oldest message of any type.

The RICH in the AP is implemented using a dedicated memory segment that is shared among all the processes that use the RICH. The RICH provides function calls to send and receive packets which are implemented as a *UNIX* software library that is built into every process that uses the RICH.

One complication in the RICH is that the byte and word orders of the AP and the DSI processors are different, making it impossible to pass data structures directly from one processor to the other. To pass data structures other than arrays of characters between the two processors, every data structure must have the bytes "swabbed" to the correct order for the other processor.

5.5 Periodic data handler

Every 5 minutes, the DSIs each send traffic counts to the AP in a RICH transaction. Since these counts are needed by several AP processes, another shared-memory segment was created to allow all the processes to access the same copy of the data. The periodic data handler is the process that reads the traffic transactions from the RICH and puts them into the shared-memory segment. The periodic data handler uses data in the recent-change table and from the DSI status monitor to determine from which DSIs it should receive transactions. When transactions have been received from all the DSIs, it sends a signal to all the AP processes that use the data. If the transactions from all the DSIs have not been received within a short period of time after they are expected, the periodic data handler informs the DSI status monitor that there may be problems with a DSI and signals all the user processes to read the data.

5.6 Billing process

The DSIs send billing transactions to the AP using RICH transactions. While several transactions may be sent for very long calls, the AP does not do any call record assembly. The DSIs also send hourly tracer records that contain counts of the number of billing records of each type that the DSI has sent. The AP billing program checks the number of records that it has received against the counts given in the DSI tracer records. Any errors in the counts are then reported in tracer records generated by the AP for the billing processing center. For most telephone operating companies, the bill generation will be done by revenue accounting offices.

The processing of billing records in the AP is done in two stages. As records arrive from the DSIs, they are processed to check their validity and to calculate holding times. Then that information along with other billing data such as packets sent and received based on rate periods is written to a daily billing file stored on disk. Both dial-up and dedicated records are read by this process, but only dedicated records are processed. Both types of records are also written to a log file that allows off-line analysis of the per-call data. The last five daily billing files are stored on disk, allowing several days worth of billing data to be written to tape at the same time. If errors are encountered during the reading of the tape at the revenue accounting office, a new copy of the tape can be generated from the disk data.

5.7 Traffic data processing

Every 5 minutes, each DSI sends a block of traffic data to the AP in a RICH transaction, which is then read into the periodic data handler. The traffic report generator then reads the traffic data out of the periodic data handler to generate reports. The traffic process prepares reports at 5-minute, 30-minute, and 24-hour intervals with identical content, though it is possible to tailor the printed output of the different reports through the use of recent-change forms. For instance, the output printed in the 30-minute reports can be limited to the subset of the reports that is needed on a timely basis. The entire report could then be printed only at midnight. The 5-minute reports can be turned on conditionally so they will be printed only if the value in one of the report fields exceeds a set threshold. Thresholds can be set on any field in the report through recent change. Crossing a threshold also causes an error message to be printed. Threshold exceptions and any other DSI errors are summarized in 5-minute, 30-minute, and 24-hour error reports that can also be tailored through the recent-change system.

5.8 Recent change

The AP recent-change system stores all configuration data for the AP and the DSIs. While the AP stores the master copy of all configuration data, the recent-change data for each DSI is sent to the DSI when the DSI is initialized or when the copy in the AP is changed. The craft interface to the AP recent-change system is through the DMERT-provided Online Data INtegrity (ODIN) system, a forms-based data entry system that provides several levels of data validation. Many new ODIN forms were created for LADT, including forms for traffic report formats, DSI equipment forms, DSI subunit equipment forms, and customer data forms.

The AP recent-change database is implemented using a shared memory segment and a *UNIX* software library of recent-change function calls. For the simple types of data retrieval needed in LADT, a shared-memory implementation is faster than a database built using the more powerful DMERT-supplied database tools.

5.9 DSI status monitor

While the recent-change system stores data on the DSIs that are equipped, a different mechanism is needed to store the maintenance state of the DSIs. The DSI status monitor provides the current maintenance status of the DSIs to both internal AP processes and to the craft. The DSI status monitor classifies each DSI in one of the following states:

1. Active
2. In the process of being downloaded
3. Manually out of service
4. Link down
5. Out of service.

The DSI status monitor gathers data on the state of the DSIs from many sources. First it determines which DSIs are equipped from the recent-change system. When a DSI starts a generic download, the

X.25 handler notifies the DSI status monitor that the DSI is in the download state. When the download is finished and the DSI operational, the DSI sends a message to the DSI status monitor stating that it is active. The DSI status monitor checks the periodic data handler every 5 minutes to see which of the DSIs have sent data for that 5-minute period. If the periodic data handler informs the DSI status monitor that data were not received from a DSI that should be active, the DSI status monitor sends a RICH transaction to the DSI querying the DSI status. If the DSI status monitor does not get the proper response from the DSI, it marks the DSI out of service and triggers an alarm. The X.25 handler also alerts the DSI status monitor any time a call from a DSI to the AP is torn down.

5.10 DSI craft interface

While some local craft capabilities are provided by the DSI power control and display panel, most of the craft interface for the DSIs is provided through the AP. All LADT craft commands conform to the Program Documentation Standards (PDS) formats used for many types of AT&T switching systems. Some commands execute locally on the AP, while other commands are entered on the AP and then sent to a particular DSI for execution. Where the commands actually execute is transparent to the craft who enter the commands.

5.11 AP maintenance

Besides the LADT maintenance functions for DSIs, the AP has some internal maintenance functions to augment the maintenance functions provided by DMERT. The AP uses the User Level Automatic Restart Process (ULARP) provided by DMERT to handle internal initializations. This process has a list of AP processes that should be started at initialization and restarted if any of them terminate. To prevent thrashing, the application integrity monitor checks how often a process is restarted. If any process is restarted more than twice in 6 minutes, the application integrity monitor will cause an AP software initialization.

The AP has one level of application initialization defined below the DMERT levels of initialization. If fatal errors are encounted in running the AP application code, this application level of initialization restarts all the LADT specific processes without restarting DMERT. If two iterations of user-level initialization occur within a 5-minute period, then the initialization level is escalated to DMERT initializations.

5.12 AP audits

Since many AP processes use shared memory segments, it is important to prevent a bad process from corrupting the data stored in the segments. The AP has internal audits for the shared memory buffers that run periodically. These audits attempt to correct any errors that are found, or cause a software initialization if the errors cannot be corrected.

VI. SUMMARY

The LADT system has been designed to provide data-transport services within a Local Access and Transport Area (LATA). AT&T Technologies implementation of LADT divides the necessary functions among already existing and newly designed components in order to provide economical access for various types of customers. The network has been designed to be both cost-effective and reliable.

REFERENCES

1. H. J. Kafka et al., "Local Area Data Transport—A Packet-Switched Network For Exchange Area Services," IEEE Int. Conf. on Commun. ICC '83, June 1983.
2. G. J. Handler, "Networking—Bit by Bit," Conf. Record of Videotex '82, pp. 453–63.
3. C.C.I.T.T., Data Communication Networks Services and Facilities, Terminal Equipent and Interfaces, Vol. VII–Fascicle VIII.2, Geneva, 1981.
4. M. N. Ransom, "Local Area Data Transport System Overview," AT&T Bell. Lab. Tech. J., this issue.
5. W. L. Harrod and A. G. Lubowe, "The *BELLPAC*™ Modular Electronic Packaging System," B.S.T.J., *58*, No. 10 (December 1979), pp. 2271–88.
6. A. B. Glaser et al., "The XPC—A VLSI Link-Level Controller for X.25 LAPB," IEEE Int. Conf. on Circuits and Computers ICCC '82, September–October 1982.
7. N. E. Snow and N. Knapp, Jr., "Digital Data System: Sytem Overview," B.S.T.J., *54*, No. 1 (May 1975), pp. 811–32.
8. E. C. Bender, J. G. Kneuer, and W. J. Lawless, "Digital Data System: Local Distribution System," B.S.T.J., *54*, No. 1 (May 1975), pp. 919–42.
9. P. Benowitz et al., "Digital Data System: Digital Multiplexers," B.S.T.J., *54*, No. 1 (May 1975), pp. 893–918.
10. "Data Set 212A Interface Specification," AT&T Technical Reference, PUB 41214, January 1978.
11. J. R. Boddie et al., "Digital Signal Processor: Architecture and Performance," B.S.T.J., *60*, No. 7, Part 2 (September 1981), pp. 1449–62.
12. "Local Area Data Transport Terminal Interface Specifications," AT&T Technical Reference, PUB 54200, June 1982.
13. J. R. Kane, R. E. Anderson, and P. S. McCabe, "Overview, Architecture, and Performance of DMERT," B.S.T.J., *62*, No. 1, Part 2 (January 1983), pp. 291–302.
14. M. E. Grzelakowski, J. H. Campbell, and M. R. Dubman, "DMERT Operating System," B.S.T.J., *62*, No. 1, Part 2 (January 1983), pp. 303–22.
15. W. N. Toy and L. E. Gallaher, "Overview and Architecture of the 3B20D Processor," B.S.T.J., *62*, No. 1, Part 2 (January 1983), pp. 181–90.
16. "X.25 Interface Specifications," AT&T Preliminary Technical Reference, PUB 54010, August 1981.

GLOSSARY

AP	administrative processor
CRC	cyclic redundancy check
DMA	direct memory access
DISC	LAPB disconnect command
DMERT	duplex multi-environment real time

DSI	data subscriber interface
DSIOS	DSI operating system
DSP	digital signal processor
DTE	data terminal equipment
FIFO	first-in first-out
FSK	frequency shift carrier
LADT	local area data transport
LAPB	link access procedure B
LATA	local access and transport area
MOS	metal oxide semiconductor
MRA	maintenance request administrator
MTTY	maintenance terminal
ODIN	on-line data integrity
PCD	power control and display
PDS	program documentation standards
RAM	random access memory
RICH	remote internal communication handler
ROM	read-only memory
RQIP	request in progress
SABM	set asychronous balance mode
SLC	subscriber loop carrier
XPC	X.25 protocol controller

AUTHORS

Henry J. Kafka, B.S. (Electrical Engineering), 1979, Northwestern University; M.S. (Electrical Engineering), 1980, University of Illinois, Urbana; AT&T Bell Laboratories, 1979—. At AT&T Bell Laboratories, Mr. Kafka's responsibilities have included hardware and software design, as well as applied research in new digital services and advanced packet-switching concepts. Mr. Kafka is presently Supervisor of the LADT System Evaluation Group and is responsible for software development and field support for AT&T's Local Area Data Transport products. Member, Eta Kappa Nu, Tau Beta Pi, IEEE.

W. Joseph Paule, B.S. (Computer Science), 1976, Iowa State University; M.S. (Computer Science), 1977, University of California, Berkeley; AT&T Bell Laboratories, 1976—. Upon joining Bell Laboratories, Mr. Paule worked on the development of the 1A Voice Storage System (1A VSS), participating in tool development, system integration, system test, field support, and feature design. He was promoted to Supervisor in 1980. From 1981 to 1983, Mr. Paule worked on the LADT project, where he supervised software development, system integration, and field support. He is currently in the Data Packet Switching Department, where he is Supervisor of the 1PSS System Test and Field Support Group. Member, Phi Beta Kappa, Phi Kappa Phi, ACM.

David J. Stelte, B.S., Electrical Engineering (summa cum laude), 1972, University of Notre Dame; M.S. (Electrical Engineering), 1973, University of Illinois; GTE Network Systems Digital Switching Development Laboratory, 1974–1978; AT&T Bell Laboratories, 1978—. At GTE, Mr. Stelte was involved in the design and implementation of Pulse Code Modulation systems as well as system architectural work on class 5 digital switching systems. At AT&T

Bell Laboratories, Mr. Stelte's responsibilities have included applied research in new digital services, and work on ISDN standards and their implementation in digital switching systems. He has also worked in the areas of data set design and mobile telecommunications. Mr. Stelte is presently Supervisor of the LADT System Design Group and is responsible for architectural planning and software development for AT&T data switching products. Member, Eta Kappa Nu, Tau Beta Pi, IEEE.

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Optimum Scan-Width Selection Under Containment Constraints

By M. R. GAREY* and R. Y. PINTER*

(Manuscript received December 20, 1983)

We consider the following algorithmic problem, which arises in connection with optimally choosing beam widths and positions for electron exposure of integrated circuit wafers. Let $H > 0$ be a fixed real number, and let c be a fixed, positive-valued, nondecreasing cost function defined on $(0, H]$. An instance of the problem consists of a given range $R = [a, b]$ and n given intervals $I_i = [a_i, b_i]$, $1 \leq i \leq n$, each contained in R and of length not exceeding H. A solution (or feasible solution) for such an instance is a collection of segments, $S_1, S_2, \cdots, S_k$, each of length at most H and contained in R, such that each given interval is contained in at least one segment and the union of all the segments is R. The goal is to find an *optimal solution* with respect to c, i.e., a solution for which the sum of the costs of the individual segment lengths is as small as possible. The segments in a solution describe the beam positions and widths, projected on one side of the wafer, and the given intervals correspond to particularly sensitive regions on the layout mask, each of which must be entirely exposed by a single scan. The cost function gives the time required for a single scan of given width, including alignment overhead. Using dynamic programming techniques, we give efficient algorithms and data structures for solving this problem for several natural classes of cost functions, the most general of which is the class of all concave increasing functions, solved by an algorithm that runs in time $O(n^2)$.

I. INTRODUCTION

Electron lithography systems[1] are used in the fabrication of integrated circuits to expose areas of the manufactured device specified

* AT&T Bell Laboratories.

by layout masks. In most such systems an electron beam repeatedly sweeps across the mask in parallel horizontal movements, exposing one horizontal stripe of the material at a time, so that each point of the mask is included in at least one stripe. The height of the stripe can vary on successive sweeps and is subject to a physically determined upper bound, H, with the time required for a sweep being a nondecreasing function of height. In general, it is desirable to process each mask in the minimum possible amount of time in order to maximize throughput and minimize the risk of failure.

The quality of the etching made by a single sweep is highly reliable, and the generated stripe can be regarded as an atomic piece of the process for purposes of quality control and cost evaluation. However, the machinery cannot be realigned between sweeps with absolute precision, and errors can arise from the resulting imperfect positioning of the stripes. Usually, conservative design rules prevent such alignment errors from being harmful, but some parts of a circuit mask may be too sensitive to tolerate these (otherwise acceptable) errors. These more sensitive regions ("islands") cannot tolerate either small unexposed gaps between stripes or multiple exposures created by overlapping stripes, although an island can be shielded during some sweeps to prevent exposure during those sweeps. Thus, the islands impose containment constraints on the scanning process, in that each island must be completely contained within a single stripe. This paper focuses on how to minimize the cost (time) for exposing a two-dimensional region in the presence of such containment constraints.

Notice that the upper bound, H, on the height of any stripe implies that no island can have height greater than H. It also implies that, in general, some parts of the region may have to be scanned more than once. For example, if there are two islands of height $3H/4$ arranged as in Fig. 1, the stripe exposing the first must overlap and be distinct from the stripe exposing the second. Hence, each of the islands would need to be shielded during the sweep that exposes the other island.

The total cost of the scanning process is the sum of the costs for the individual sweeps, where the alignment overhead is incorporated into the cost for the sweep. Various cost functions can be used to approximate the actual cost of scanning a stripe of given height. We will examine several classes of such functions and present efficient algorithms that minimize the covering cost for them. All cost functions considered will be monotonically nondecreasing in the stripe height.

Section II gives the mathematical formulation of the scan-width selection problem and makes some preliminary observations. Section III describes our optimization algorithms for the various cost functions. Section IV concludes the paper by mentioning several open problems.

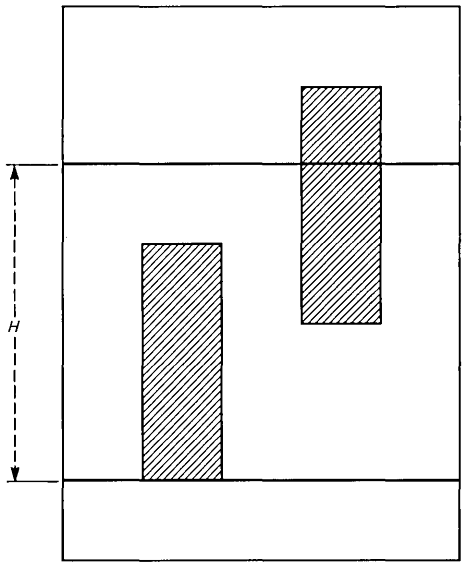


Fig. 1—Layout mask showing that two islands of height $3H/4$ require overlapping stripes.

II. PROBLEM FORMULATION

We first observe that the problem we have described is essentially one-dimensional. We can project the region to be exposed and the islands contained in the region onto a vertical line, transforming the region into a *range* $[a, b]$ and each island into an *interval* $[a_i, b_i]$ contained in this range. (See Fig. 2.) A stripe of height $h \leq H$ then becomes a *segment* of length $h \leq H$, and containment of an island within a stripe corresponds to containment of the associated interval within the appropriate segment. We will deal exclusively with this one-dimensional formulation in the remainder of the paper, rotating it 90 degrees for convenience of description.

Problem definition: Let $H > 0$ be a fixed real number, and let c be a fixed, positive-valued, nondecreasing cost function defined on $(0, H]$. An instance of the problem consists of a given range $R = [a, b]$ and n given intervals $I_i = [a_i, b_i]$, $1 \leq i \leq n$, each of length at most H and contained in R. A solution (or feasible solution) for such an instance is a collection of segments, $S_1, S_2, \cdots, S_k$, each of length at most H and contained in R, such that each given interval is contained in at least one segment and the union of all the segments is R. Our goal is to find an *optimal solution*, i.e., a solution for which the sum of the costs of the individual segment lengths is as small as possible.

We will assume that the specified intervals are given in a particular

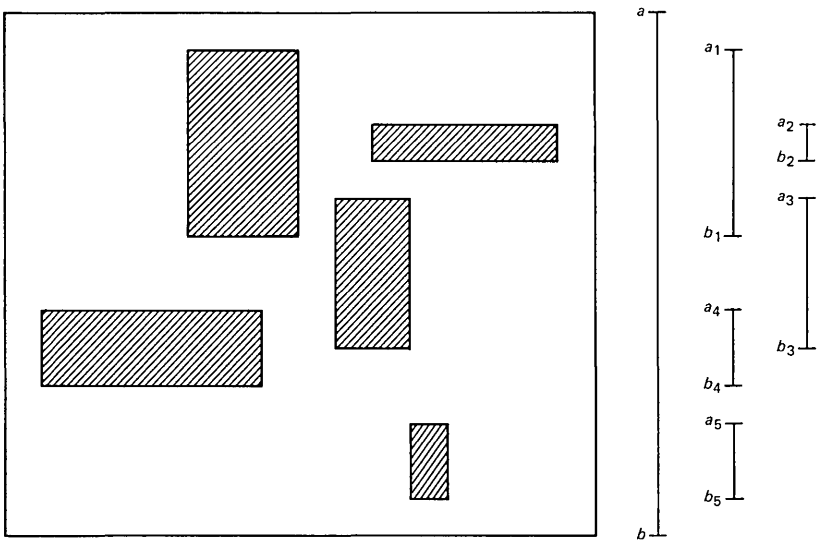


Fig. 2—A covering problem and its one-dimensional interpretation.

sorted and reduced format. First, we assume that the intervals are sorted by their left end points, so that $a_i \leqslant a_{i+1}$ for $1 \leqslant i < n$. Second, we assume that no given interval is contained in another given interval; any segment that contains the larger one must also contain the smaller, so the smaller can be ignored without loss of generality. Thus, the intervals will also be sorted by their right endpoints, i.e., $b_i \leqslant b_{i+1}$ for $1 \leqslant i < n$. We shall not account for the complexity of the preprocessing needed to meet these assumptions, which is $O(n \log n)$ if sorting is necessary or $O(n)$ if the intervals are presented in sorted order.

We will consider the following types of cost functions (all are assumed to be positive and nondecreasing on $(0, H]$):

1. $c(h)$ is constant, i.e., $c(h) = \beta$.
2. $c(h)$ is proportional to h, i.e., $c(h) = \alpha h$.
3. $c(h)$ is linear, i.e., $c(h) = \alpha h + \beta$.
4. $c(h)$ is concave, i.e., if $h_1 < h_2 < H$ and $0 < \epsilon \leqslant \min\{h_1, H - h_2\}$, then $c(h_1) + c(h_2) \geqslant c(h_1 - \epsilon) + c(h_2 + \epsilon)$.

We will also consider the variant on the constant cost function in which all segments must have length exactly H. This is not a nondecreasing function, since segments shorter than H have essentially infinite cost, but we shall see that the solution for this case follows directly from the solution for the case of constant cost functions.

In the general case of a concave cost function, we will assume that the function is given simply by a "black box" subroutine for computing $c(h)$. We will also assume that the running time for the subroutine is bounded by some constant, γ; if the calculation is more complex and a more detailed timing analysis is needed, this should be easily obtainable from our analysis.

We conclude this section with a general normalization lemma for optimal solutions. Let us define a *gap* in a given problem instance to be a maximal subinterval of $R = [a, b]$ that is disjoint from all the given intervals. We regard a gap as an open subinterval, i.e., the two endpoints of a gap do not belong to the gap. A maximal subcollection of the given intervals such that no two are separated by a gap will be called a *block*; by our assumption that the intervals are sorted, each block consists of intervals whose indices are consecutive. Thus, the given problem instance is partitioned into an alternating sequence of gaps and blocks, with the indices of all intervals in each block being less than the indices of all intervals in blocks to its right. If no interval starts at the left end, a, of the range, we will regard a as both the left end of the corresponding gap and as the right end of an empty block. Similarly, if no interval ends at b, we will regard b as both the right end of the corresponding gap and as the left end of an empty block. Our "normalization lemma", which will be used for all but the case of constant cost functions (and its fixed length variant), follows.

Lemma 1: For any nondecreasing cost function, c, and any given problem instance, there always exists an optimal solution in which:

(1.1) No segment is contained in another segment.

(1.2) All segments start at left ends, a_i, of intervals, right ends of blocks, or in gaps; and all segments end at right ends, b_i, of intervals, left ends of blocks, or in gaps.

(1.3) No point in a gap is in the interior of more than one segment.

Proof: For (1.1) we simply observe that any segment that is contained in another can be deleted with no increase in cost and without destroying the solution. Thus, any optimal solution that also has a minimum number of segments (among all optimal solutions) must satisfy (1.1).

For (1.2) consider any optimal solution with a minimum number of segments [hence, satisfying (1.1)]. If the left end of any segment is not of the specified form, we can shorten the segment to start at the leftmost point of that form that it contains, without changing the set of intervals contained in the segment and without leaving uncovered any portion of a gap. Thus, the new solution is still a solution, and since the cost cannot have increased, it remains an optimal solution. Similarly, we can shorten the right end of any segment whose right endpoint fails to have the specified form. Repeating these operations to the given optimal solution results in a new optimal solution satisfying (1.2). Furthermore, it must still satisfy (1.1), since the number of segments in the solution has not been increased (if the shortening of any segment were to make it contained in another segment, we could delete the smaller, which would be a contradiction of our assumption that the number of segments was originally minimal—hence, this cannot happen).

For (1.3) consider any optimal solution obtained as in the preceding paragraph, and suppose some point in a gap is in the interior of two segments. Let the segments be $[x, y]$, $[z, w]$, with $x < z < y < w$. Since some point, u, in the range (z, y) is in a gap, we can replace these two intervals by the two intervals $[x, u]$, $[u, w]$ without destroying the solution and without increasing the solution cost, since both intervals have been shortened. Moreover, this operation clearly preserves (1.1), because the number of segments remains minimum, and preserves (1.2), because the only new point to start or end a segment is u, which is in a gap. Thus, repeating this operation will eventually result in an optimal solution satisfying (1.1), (1.2), and (1.3). □

We will restrict attention to finding optimal solutions of the form given in Lemma 1. Notice that (1.1) of the lemma allows us to order the segments in a solution from left to right in the same way intervals were ordered; if the left end of a segment is less than the left end of another segment, then its right end must also be less than the right end of the other segment. Thus, there will be no confusion when we say a segment is to the left of another segment.

III. OPTIMIZATION ALGORITHMS

We now consider each of the four types of cost functions, in order of increasing difficulty.

3.1 Constant cost function

In this case we are simply trying to minimize the number of segments, so we may, without loss of generality, restrict all segments to the maximum possible length H. (We will not be using Lemma 1 here.) The solution then follows immediately by observing that we can always start the first segment at the left endpoint, a, of the range $[a, b]$. This leaves a remaining problem that includes exactly those intervals not contained in the first segment, i.e., those intervals $[a_i, b_i]$ for which $b_i > a + H$; and the range for the new problem can be taken to be $[a', b]$, where a' is the minimum among $a + H$ and the left endpoints, a_i, of the remaining intervals. Because we can always start the first segment at the left end of the range, we can repeat this "greedy" approach until we reach the end of the range $[a, b]$. However, the last segment must be started at the point $b - H$ to keep it within the original range. The following algorithm implements this approach and runs in time linear in the larger of the number of given intervals and the number of segments in the optimal solution. The variable s_j

denotes the left endpoint of the jth segment, and the final value of j is the number of segments in the optimal solution.

Algorithm 1:

```
initialize $i := 1$; $j := 0$; $newa := a$;
while $newa \leqslant b - H$ do
  $j := j + 1$; $s_j := newa$;
  while ($b_i \leqslant newa + H$) do $i := i + 1$;
  $newa := \min(a_i, newa + H)$;
if ($newa < b$){$j := j + 1$; $s_j := b - H$};
```

3.2 Proportional cost function

For the sake of simplicity, we shall assume that the constant of proportionality α has been normalized to 1, so the cost of a solution is just the sum of the lengths of its segments. We first observe that the gaps can be covered independently of the blocks, with each gap being covered simply by a sequence of adjacent segments whose lengths sum to the length of the gap. The lengths of these segments can be chosen arbitrarily, but all must be H or less. Thus, we need only show how to solve problem instances that consist of a single block, starting at the left end of the range and running all the way to the right end.

So, suppose the given problem consists of just a single block. Since there are no gaps, Lemma 1 tells us that we can restrict attention to solutions in which all segments start at left ends of intervals and run to right ends of intervals. This sets the stage for the use of a dynamic programming approach. Let $C(i)$, $1 \leqslant i \leqslant n$, be the cost of an optimal covering for the range $R(i) = [a_i, b]$ and the intervals $[a_j, b_j]$, $i \leqslant j \leqslant n$, contained in that range. We want to find $C(1)$. Then we can write

$$C(i) = \min_{\substack{j \geqslant i \\ b_j - a_i \leqslant H}} \{b_j - a_i + C(j+1)\}, \qquad (1)$$

where we artificially set $C(n + 1) = 0$. The solution corresponding to a particular value of j consists of the segment $[a_i, b_j]$ followed by an optimal solution for the range $[a_{j+1}, b]$. (Notice that $a_{j+1} \leqslant b_j$ since there are no gaps). By using (1) to solve for the $C(i)$ in order of decreasing i, we can then find $C(1)$. The segments that realize the solution can be recorded in a one-dimensional trace vector T, where $T(i)$ is set to that value of j for which $C(i)$ is minimized. From this information we can easily reconstruct the optimal solution.

The obvious algorithm based on this approach can require time proportional to n^2. We will show how to reduce this to $O(n)$, by using some simple algebraic transformations and a carefully chosen data structure. First, we can rewrite (1) as

$$C(i) = -a_i + \min_{\substack{j \geq i \\ b_j - a_i \leq H}} \{b_j + C(j+1)\}, \tag{2}$$

which leaves the minimization independent of i, except for defining the relevant j values. Let us define $C^*(i) = b_{i-1} + C(i)$. Then, from (2) we have

$$C^*(i) = b_{i-1} - a_i + \min_{\substack{j \geq i \\ b_j - a_i \leq H}} \{C^*(j+1)\}, \tag{3}$$

and by defining $b_0 = 0$, we have $C^*(1) = C(1)$. We also extend the definition of C^* so that $C^*(n+1) = b_n + C(n+1) = b_n$. We will use (3) as the basis for our improved dynamic programming algorithm.

The $C^*(i)$'s will be computed in order of decreasing i. The key idea is to keep accessible only those $C^*(i)$ values that can be useful for subsequent minimizations and, at the same time, to make it easy to find the new minimum for each $C^*(i)$. At each stage (new value of i), we need to delete those previous $C^*(j)$ values that are too large to be of further use or that are "too far away" to be used because $b_j - a_i > H$. We do this by storing the so-far-computed C^* values in a *deque* (double-ended queue), a data structure that allows lookup, insertion, and deletion at both ends, but allows no direct access to interior elements. The elements stored in the deque are pairs $(C^*(i), i)$ with i monotonically increasing from left to right, although the i values need not be consecutive. The deque operations are (since there is only one deque, we omit explicit reference to it):

push-left (x, i), *push-right* (x, i)	to insert element (x, i) at designated end
pop-left ( ), *pop-right* ( )	to delete element at specified end
C-of-left ( ), *C-of-right* ( )	to return C^*-value on specified end without changing deque
i-of-left ( ), *i-of-right* ( )	to return i-value on specified end without changing deque.

These are easily implemented in standard ways.[2]

Now we can present the linear-time algorithm for computing $C(1) = C^*(1)$ and the trace vector, T. The input for the algorithm is the sequence $a_1, a_2, \cdots, a_n$ of interval left endpoints and the sequence $b_1, b_2, \cdots, b_n$ of interval right endpoints, both in increasing order.

Algorithm 2:

```
initialize deque := $\langle (b_n, n+1) \rangle$; $b_0 := 0$;
for $i := n$ to 1 by $-1$ do
   while ($b_{i\text{-}of\text{-}right(\,)-1} - a_i > H$) do pop-right ( );
```

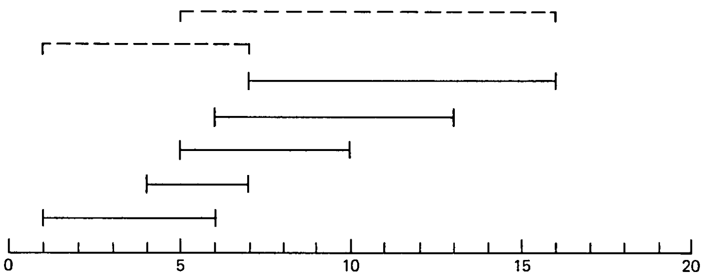


Fig. 3—Intervals (solid lines) and segments (dashed lines). There are two segments in the solution and their total length is 17.

```
$C^*(i) := b_{i-1} - a_i +$ C-of-right ( );
$T(i) :=$ i-of-right ( ) $- 1$;
while (C-of-left ( ) $\geq C^*(i)$) do pop-left ( );
push-left ($C^*(i), i$);
```

The preceding discussion and the following easily verified observations directly imply the correctness of the algorithm and the fact that it runs in time $O(n)$.

Observation 1. Algorithm 2 maintains the deque in such a way that if (x, i) is to the left of (y, j), then $x > y$ and $i < j$.

Observation 2. For each i, $1 \leq i \leq n$, $(C^*(i), i)$ is inserted exactly once, and in each iteration of the loop the number of elements that are examined but not deleted is exactly two.

Theorem 1: Algorithm 2 correctly computes $C(1) = C^*(1)$ *and the trace vector,* T, *in* $O(n)$ *steps.*

Example. Consider the set of intervals [1, 6], [4, 7], [5, 10], [6, 13], [7, 16], with $H = 11$. Then the deque will assume the following values at the end of each iteration:

init:	(16, 6)	
$i = 5$:	(22, 5), (16, 6)	$T(5) = 5$
$i = 4$:	(20, 4), (16, 6)	$T(4) = 5$
$i = 3$:	(18, 3), (16, 6)	$T(3) = 5$
$i = 2$:	(20, 2), (18, 3)	$T(2) = 2$
$i = 1$:	(17, 1)	$T(1) = 2$.

The intervals and the optimal covering are shown in Fig. 3.

3.3 *Linear cost function*

In this case, gaps become significant and can no longer be treated separately. However, we shall see that we can restrict our attention to a limited set of potential segment starting points in the gaps. This will let us use a slightly more complicated, but still very efficient, dynamic programming approach.

Lemma 2: For any linear cost function, c, and any given problem instance, there exists an optimal solution of the form given by Lemma 1 such that, for any segment [x, y] in the solution:

(2.1) If $x + H$ lies in a gap, then $y = x + H$, i.e., the segment has length H.

(2.2) If $x + H$ does not lie in a gap, then y does not lie in a gap.

Proof: Consider any optimal solution of the form given in Lemma 1 and suppose also that it has a minimum number of segments among all such solutions. Then we know that no two adjacent or overlapping segments in the solution have lengths summing to H or less, since by the linear cost function they could then be combined into a single segment at no increase in cost.

Consider the leftmost segment $[x, y]$ that violates either (2.1) or (2.2). If no such segment exists, we are done. Otherwise, let $[z, w]$, with $x < z \leq y < x + H < w$, denote the next segment in the ordering. [The inequalities on x, y, z, and w follow from Lemma 1 (1.1) and our observation at the beginning of the proof]. If $x + H$ is in a gap but $y \neq x + H$, we can replace the two segments $[x, y]$ and $[z, w]$ by the two segments $[x, x + H]$ and $[x + H, w]$ without destroying the solution, since the new segments cover the same span, and any interval contained in one of the original two is contained in $[x, x + H]$ if it was to the left of $x + H$ and is contained in $[x + H, w]$ otherwise. If $x + H$ is not in a gap but y is in a gap, let u be the right endpoint of the gap containing y. Then we can replace $[x, y]$ and $[z, w]$ (where in fact $z = y$) with $[x, u]$ and $[u, w]$ without destroying the solution. In neither case have we increased the cost of the solution, because the sum of the lengths of the two new segments is no greater than the sum of the lengths of the original two. Moreover, the new segment starting at x retains the position of $[x, y]$ in the left-to-right ordering of segments in the solution and no longer violates (2.1) or (2.2). In addition, it is easy to verify that all segments starting to the left of x continue to satisfy (2.1) and (2.2) and no violations of the conditions of Lemma 1 have been introduced. Thus, we can repeatedly apply the appropriate one of these two transformations to the leftmost violator of (2.1) or (2.2), and we will eventually obtain an optimal solution of the stated form. □

We will restrict attention to optimal solutions of the form given by Lemma 2.

Consider what this tells us about solving any subproblem consisting of a range $[x, b]$ and all intervals contained in that range. If $x + H$ is not in a gap, then, by Lemma 2 (2.2) and Lemma 1 (1.2), we need only consider solutions in which the leftmost segment runs from x to the right endpoint of some interval or to the left end of some block. The remaining subproblem in each such case will have a range starting at the left endpoint of the leftmost uncovered interval from the original subproblem, as in the proportional case, so we do not need to know about any subproblems that start with points in gaps. If $x + H$ does lie in a gap, then, by Lemma 2 (2.1), we know that there is an optimal solution in which the leftmost segment has length exactly H, and we can, without loss of generality, choose to start with such a segment. However, to compute the cost of that solution, we need to know the optimal solution cost for the subproblem that then remains, and the range for that remaining subproblem begins at the point $x + H$, which does lie in a gap. This suggests that we will indeed need to solve certain subproblems whose ranges begin with points that lie in gaps. The key lies in finding a small number of such points that will suffice.

For any point x, define $fwd(x)$ to be the rightmost point $y \geqslant x$ such that y is congruent to x modulo H and such that all points in the sequence $x + H, x + 2H, \cdots, y$ lie in gaps. If $x + H$ does not lie in a gap, we let $fwd(x) = x$. Now, again consider the above situation of solving a subproblem with range $[x, b]$, where $x + H$ belongs to a gap. Then, by repeated application of Lemma 2 (2.1), we know that there is an optimal solution for this subproblem that begins with a sequence of adjacent segments of length exactly H ending at the points $x + H$, $x + 2H, \cdots, fwd(x)$, and we can, without loss of generality, choose to start this way. The remaining subproblem, whose solution cost we need for computing the optimal cost for the initial subproblem, still begins with a point in a gap (namely, the point $fwd(x)$), but it is a point of very special form. In particular, by the definition of $fwd(x)$, we have that $fwd(fwd(x)) = fwd(x)$.

It follows directly from the preceding discussion that we need only solve subproblems for ranges of the form $[x, b]$, where x is the left end of some interval, the right end of some block, or $fwd(z)$ for z one of those two types of points. We can describe the computation as follows: Let $\{z_1, z_2, \cdots, z_m\}$ be the set of all such starting points, sorted so that $a = z_1 < z_2 < \cdots < z_m = b$. For each z_i, let $b(z_i)$ be defined as b_j if $z_i = a_j$ for some interval $[a_j, b_j]$ (there can only be one such j by our assumption that no interval is contained in another) and as z_{i+1} otherwise. Notice that $b(z_1) < b(z_2) < \cdots < b(z_m)$ and that $b(z_i) \geqslant z_{i+1}$ for all i, $1 \leqslant i < m$. Let $C(z_i)$ denote the cost of an optimal solution for the subproblem with range $[z_i, b]$ consisting of all intervals contained in that range. Then, if $z_i \neq fwd(z_i)$, we have

$$C(z_i) = \frac{fwd(z_i) - z_i}{H}(\alpha H + \beta) + C(fwd(z_i)). \quad (4)$$

The solution in this case consists of a sequence of intervals of length H starting at the points z_i, $z_i + H, \cdots, fwd(z_i) - H$, followed by an optimal solution for the range $[fwd(z_i), b]$. If $z_i = fwd(z_i)$, we have

$$C(z_i) = \min_{\substack{j \geq i \\ b(z_j) - z_i \leq H}} \{\alpha(b(z_j) - z_i) + \beta + C(z_{j+1})\}. \tag{5}$$

The solution corresponding to a particular choice of j here consists of the segment $[z_i, b(z_j)]$ followed by an optimal solution for the range $[z_{j+1}, b]$. It is not difficult to see that the sequence $b(z_i) < b(z_{i+1}) < \cdots < b(z_k)$, where k is the greatest index such that $b(z_k) - z_i \leq H$, includes all possible right endpoints that need be considered for a segment starting at z_i, although it may also include some points in gaps that could have been ignored.

In addition, we can still use an algebraic transformation like that of the preceding section to simplify the computation. Accordingly, define $C^*(z_i) = \alpha b(z_{i-1}) + C(z_i)$. Then, rewriting (4), we have, if $z_i \neq fwd(z_i)$,

$$C^*(z_i) = \alpha(b(z_{i-1}) - fwd(z_i)) + \frac{fwd(z_i) - z_i}{H}(\alpha H + \beta) + C^*(fwd(z_i)). \tag{6}$$

Here we used the observation that if $fwd(z_i) = z_k$, $k > i$, then $b(z_{k-1}) = z_k$. Then, rewriting (5), we have, if $z_i = fwd(z_i)$,

$$C^*(z_i) = \alpha(b(z_{i-1}) - z_i) + \beta + \min_{\substack{j \geq i \\ b(z_j) - z_i \leq H}} \{C^*(z_{j+1})\}. \tag{7}$$

We also have $C^*(z_m) = b(z_{m-1})$, and we define $b(z_0) = 0$. These equations will provide the basis for our algorithm, which again uses the deque data structure of the previous subsection.

We assume that the algorithm is given as input the integer m, the sequence $z_0 = z_1, z_2, \cdots, z_m$, and, for $0 \leq i \leq m$, the corresponding values for $b(z_i)$ and $fwd(z_i)$. The algorithm computes $C(z_1) = C^*(z_1)$ and the trace vector T, as before. However, the value $T(i)$ of the trace vector T at z_i will be undefined whenever $z_i \neq fwd(z_i)$, since we already know that the solution for the subproblem starting at z_i begins with a sequence of length H segments from z_i to $fwd(z_i)$.

Algorithm 3:

```
initialize deque := $(b(z_{m-1}), m)$;
for $i := m - 1$ to 1 by $-1$ do
  while $(b(z_{i\text{-of-right}(\,)-1}) - z_i > H)$ do pop-right ( );
  if $z_i \neq fwd(z_i)$
    then $C^*(z_i) := \alpha(b(z_{i-1}) - fwd(z_i))$
    $+ ((fwd(z_i) - z_i)/H)\,(\alpha H + \beta) + C^*(fwd(z_i))$
    else {
      $C^*(z_i) := \alpha(b(z_{i-1}) - z_i) + \beta + C\text{-of-right}(\,)$;
      $T(i) := i\text{-of-right}(\,) - 1$};
```

```
while ($C$-of-left ( ) $\geq C^*(z_i)$) do pop-left ( );
push-left ($C^*(z_i)$, $i$);
```

The correctness of this algorithm and the fact that it runs in time $O(m) = O(n)$ follow from the preceding discussion and observations analogous to those made in the previous subsection.

Theorem 3: Algorithm 3 correctly computes $C(z_1) = C^*(z_1)$ and the trace vector T in $O(n)$ steps.

It remains for us to show how to compute the input for Algorithm 3. The computation of the $b(z_i)$ values is straightforward and can be accomplished easily in linear time. However, computing $fwd(z)$ for z, the left end of an interval or the right end of a block, from which the sequence $z_1, z_2, \cdots, z_m$ is determined, requires some care. It is easy to do this in linear time for each such z, simply by comparing z to each of the given intervals in left to right order, searching for the first interval to the right of z that contains a point congruent to z modulo H, and then setting $fwd(z)$ equal to that point minus H. Unfortunately, this would require $O(n^2)$ time overall, substantially worse than Algorithm 3 itself. We now describe a method for computing the values of $fwd(z)$ for all such z in time $O(n \log n)$, still worse than linear but comparable to the preprocessing time for originally sorting the intervals.

The basic idea of the method follows: Suppose the range $[a, b]$, and all the given intervals, are cut at all points that are exact multiples of H and the resulting pieces are arranged into "shelves", as shown in Fig. 4, so that the multiples of H increase as we go down the shelves. For any point x, consider a vertical cutline through the shelves that passes through x, i.e., a line at distance $x \bmod H$ in from the left ends of the shelves. Then it is easy to see that $fwd(x)$ is the furthest point not in any interval that can be seen from x by looking downward along this cutline, where the presence of an interval along the cutline blocks the view of shelves below it. To prevent "looking" beyond the last shelf, we include a dummy interval $[b, b + H]$ at the end of the original range.

Let us define the H-value of a shelf, or any point on that shelf, to be the integer t such that tH is the left endpoint of the shelf, and let us call a shelf *empty* for a particular cutline if the point at which the cutline passes through the shelf does not belong to any interval on that shelf. Now suppose we start with a cutline through the left ends of the shelves and gradually move it to the right, always keeping track of the H-values for the shelves that are nonempty at the current cutline position. This set of H-values changes whenever the cutline encounters the left or right endpoint of some interval, and we maintain this information in a data structure that can be updated easily when-

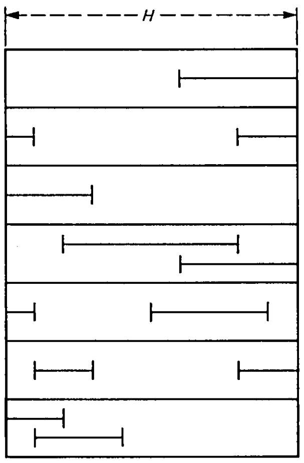


Fig. 4—Intervals arranged in shelves. Each shelf has width H.

ever such an endpoint is encountered. In addition, we choose our data structure so that it is easy to find, for any given integer t, the least H-value in the data structure that is larger than t. This is used for computing the value of $fwd(x)$ each time the moving cutline hits a point x that is the left endpoint of an interval or the right end of a block (it may hit more than one such point simultaneously). To compute $fwd(x)$ at this point, we simply find the least H-value q in the data structure that is larger than the H-value for x and set $fwd(x) = (q - 1) H + x \bmod H$. Thus, the values of $fwd(x)$ for all such points x will have been computed by the time the cutline has moved all the way to the right end of the shelves.

By using a balanced binary tree for storing the H-values, we can add or delete an individual H-value, or find the least H-value larger than a given integer, in time $O(\log n)$. Since at most $O(n)$ H-values will be added to or deleted from the data structure, regardless of the number of shelves, and since at most $O(n)$ requests for the least H-value larger than a given integer will occur, the entire procedure will thus require only $O(n \log n)$ time. The input $z_1, z_2, \cdots, z_m$ for Algorithm 3 is then obtained by combining the set of points x for which fwd was computed with the corresponding set of points $fwd(x)$ computed for them, and sorting the resulting collection of points (also in time $O(n \log n)$).

To clearly show how this scanning process can be implemented, we will describe it in more detail. The balanced tree data structure is quite standard and we will not go into detail here (see Ref. 3, for example). For our purposes, the entries in the tree can be regarded as pairs consisting of an H-value and an integer *multiplicity* for that H-value. The insertion of an H-value into the tree requires following the appropriate path from the root to find the node for that H-value, increasing its multiplicity by one if it is in the tree, and otherwise creating and inserting a new node for it with multiplicity 1. The deletion of an H-value from the tree requires following the appropriate path from the root to the node for that H-value, decreasing its multiplicity by one, and deleting the node if the multiplicity becomes zero. Finding in the tree the least H-value larger than a given integer t also involves a simple traversal from the root (the details depend on the exact implementation of the tree structure) and can be facilitated by storing in each interior node the smallest H-value occurring in its right subtree. Notice that the tree must be rebalanced only when a new node is inserted or deleted. Standard methods for implementing these operations ensure that each requires time at most logarithmic in the number of H-values in the tree and hence time $O(\log n)$.

The first step of the procedure is to cut the given intervals into shelves. It is convenient to begin by adding two dummy intervals, $[a - H, a]$ and $[b, b + H]$, at the beginning and end of the range, so that the points a and b do not have to be treated exceptionally. We then cut each interval that contains a point congruent to 0 modulo H in its interior (not as an endpoint) into two adjacent intervals at that point, i.e., replacing $[a_i, b_i]$ by $[a_i, tH]$ and $[tH, b_i]$, where $a_i < tH < b_i$ (there can be at most one such t). Next we combine all the endpoints of the resulting set of intervals into a single set (retaining repetitions), associating with each point the type of point it was in the interval it came from, with the choices being from among left end of a block (*lb*), right end of a block (*rb*), left end of an interval (but not of a block) (*li*), and right end of an interval (but not of a block) (*ri*). We also remember which points were introduced as dummies to split intervals. Finally, we replace each of these points x by the ordered pair $(\lfloor x/H \rfloor, x \bmod H)$, where $\lfloor w \rfloor$ denotes the largest integer not greater than w, and we sort them into nondecreasing order according to their second components. Notice that the first component for a point is its H-value (designating the shelf it is on), the second component is its position on that shelf, and the ordering corresponds to the sequence in which the points will be encountered as the cutline is moved from left to right.

We initialize the binary tree data structure to contain the H-values for all points with shelf position (second component) 0 and type either *rb* or *ri*. We then repeat the following steps until all points in the list have been processed (initially none have been processed):

1. Let δ be the shelf position (second component) for the next

unprocessed point on the list. (Notice that all points on the list with shelf position δ occur consecutively on the list.) Add to the tree the H-values of all points with shelf position δ and type either *lb* or *li*.

2. For each point x with shelf position δ and type *lb*, *li*, or *rb*, except for those that were added as dummies, compute $fwd(x)$ by finding in the tree the least H-value q larger than the H-value for x and setting $fwd(x) = (q - 1)\,H + \delta$.

3. Remove from the tree the H-values of all points with shelf position δ and type either *rb* or *ri*. All points with shelf position δ have now been processed.

It is easy to check that this procedure correctly computes $fwd(x)$ in each instance and that, except for the points $a - H$ and $b + H$, the points x for which $fwd(x)$ has been computed are exactly those that are left endpoints of intervals or right ends of blocks for the original set of intervals, as required. Thus, we simply need to delete those two inappropriate points, combine the remaining points x for which $fwd(x)$ was computed with the corresponding set of points obtained as values of $fwd(x)$ for them, and sort the resulting collection to form the sequence $z_1, z_2, \cdots, z_m$.

Combining Algorithm 3 with this method for computing the sequence $z_1, z_2, \cdots, z_m$, we then have Theorem 4.

Theorem 4: The proportional cost function case can be solved in time $O(n \log n)$.

3.4 Concave cost function

In this case we will again show that we can restrict attention to a limited set of potential segment starting points in gaps. The key lemma is the following:

Lemma 3: For any concave cost function c and any given problem instance, there exists an optimal solution of the form given by Lemma 1 such that, for any segment $[x, y]$ in the solution:

(3.1) if $x + H$ lies in a gap, then $y = x + H$, i.e., the segment has length H, and

(3.2) if y lies in a gap and $y \neq x + H$, then for some integer $k \geqslant 1$ the points $y + H, y + 2H, \cdots, y + (k - 1)H$ all lie in gaps, the point $y + kH$ does not lie in a gap (and, hence, by Lemma 1, $y + kH$ is either the right endpoint of some interval or the left end of some block), and the segments $[y, y + H], [y + H, y \pm 2H], \cdots, [y + (k - 1)H, y + kH]$ all belong to the solution.

Proof: Consider any optimal solution of the form given in Lemma 1, and suppose also that it has a minimum number of segments among all such solutions. Then, as in the proof of Lemma 2, we know that no two adjacent or overlapping segments in the solution have lengths summing to H or less, since by the concave cost function they could then be combined into a single segment at no increase in cost.

We first deal with violations of (3.1). Consider the leftmost segment $[x, y]$ that violates (3.1). If no such segment exists, we are done with this portion of the proof. Otherwise, let $[z, w]$, with $x < z \leqslant y < x + H < w$, denote the next segment in the ordering. (The inequalities on x, y, z, and w follow from Lemma 1 (1.1) and our observation at the beginning of the proof.) If $x + H$ is in a gap but $y \neq x + H$, we can replace the two segments $[x, y]$ and $[z, w]$ by the two segments $[x, x + H]$ and $[x + H, w]$ without destroying the solution, since the new segments cover the same span, and any interval contained in one of the original two is contained in $[x, x + H]$ if it was to the left of $x + H$ and is contained in $[x + H, w]$ otherwise. Moreover, since the sum of the lengths of the two new segments is no greater than the sum of the lengths of the original two, and since the length of the longer of the two is now as large as possible, we cannot have increased the cost of the solution. It is easy to see that this transformation can cause no violation of (3.1) to the left of $[x, y]$ and that the properties of Lemma 1 are preserved, so repeated application to the leftmost violator of (3.1) will eventually remove all such violations.

So suppose we now have a solution satisfying Lemma 1 in which there are no violations of (3.1). Consider the leftmost segment $[x, y]$ that violates (3.2). If none exists, we have a solution of the form given by the lemma. Otherwise, since $[x, y]$ violates (3.2), $x + H$ is not in a gap and y is in a gap. Letting k be the least positive integer such that $y + kH$ does not lie in a gap, all the segments $[y, y + H]$, $[y + H, y + 2H]$, $\cdots$, $[y + (k - 2)H, y + (k - 1)H]$ belong to the solution, since (3.1) is not violated, but $[y + (k - 1)H, y + kH]$ does not belong to the solution. Let $[y + (k - 1)H, z]$ denote the segment in the solution that does start at $y + (k - 1)H$. We propose to shift the entire sequence of length H intervals starting at y to either the left or the right in a way that lengthens the longer of $[x, y]$ and $[y + (k - 1)H, z]$, correspondingly shortening the shorter of the two, until either one of the points $y + iH$ (for the new value of y), $0 \leqslant i < k$, no longer belongs to a gap or one of the two extreme segments achieves length H. By the concave cost function, this cannot increase the solution cost; it also introduces no violations to (3.2) to the left of $[x, y]$ and no violations to the conditions of Lemma 1. If the shifting terminates because one of the points $y + iH$, $0 \leqslant i < k$, ceases to belong to a gap, then the segment $[x, y]$ no longer violates (3.2). However, in this case it is possible that the new segment $[y + (k - 1)H, z]$ now violates (3.1), but we can then reapply the transformation described in the preceding paragraph until there are no such violations of (3.1) without affecting any intervals to the left of the point $y + (k - 1)H$. If the shifting terminates with all points $y + iH$, $0 \leq i < k$, still in gaps, then it must have terminated with the new segment $[y + (k - 1)H, z]$ having length H, since the point $x + H$ does not belong to a gap. In this case we have not introduced any violations to (3.1), and if z is not in a gap, $[x, y]$ no longer violates (3.2). On the other hand, if z is in a gap, we now have a new value of k for our new value of y and a longer sequence of length H segments starting at y. Hence, we can continue as above, and since the length of such a sequence of length H segments is finite, eventually we must obtain a solution in which $[x, y]$ no longer violates (3.2). Therefore, continued application of such transformations to repeatedly eliminate the leftmost segment violating (3.2) will eventually produce a solution satisfying the lemma. □

We will restrict attention to optimal solutions of the form given by Lemma 3.

Consider what this tells us about solving any subproblem consisting of a range $[x, b]$ and all intervals contained in that range. If $x + H$ lies in a gap, then, by Lemma 3, we know that there is an optimal solution that begins with a sequence of segments of length exactly H ending at the points $x + H$, $x + 2H$, $\cdots$, $fwd(x)$, where $fwd(x)$ is defined exactly as in the preceding subsection. If $x + H$ does not lie in a gap, then, by Lemma 1 (1.2), we need only consider solutions in which the leftmost segment runs from x to the right endpoint of some interval or the left end of some block, or from x to some point y in a gap. Furthermore, in the latter case, by Lemma 3 (3.2), we can restrict attention to points y such that, for some integer $k \geq 1$, $y + kH$ is the right endpoint of some interval or the left end of some block, and all the points $y + H$, $y + 2H$, $\cdots$, $y + (k - 1)H$ lie in gaps.

As in the preceding subsection, let $a = z_1 < z_2 < \cdots < z_m = b$ be the sorted collection of all interval left endpoints, block right endpoints, and points of the form $fwd(x)$ for x one of those two types of points; for $1 \leq i \leq m$, let $b(z_i)$ be defined in the same way as before. In addition, for $1 \leq i \leq m$, define $bkwd(z_i)$ to be $b(z_i)$ if $b(z_i)$ is in a gap or $b(z_i) - H$ is not in a gap; otherwise, define $bkwd(z_i)$ to be the leftmost point $y \leq b(z_i)$ congruent to $b(z_i)$ modulo H such that all points $b(z_i) - H$, $b(z_i) - 2H$, $\cdots$, y lie in gaps. Notice that, for $b(z_i)$ not in a gap, $bkwd(z_i)$ is defined in the same way as $fwd(z_i)$ except that the jumps of length H are made to the left from $b(z_i)$ instead of to the right from z_i, i.e., we reverse our sense of direction and use the other end of the interval or block. Thus, $bkwd(z_i)$ for all i can be computed in time $O(n \log n)$ using a method analogous to that of the previous subsection for computing $fwd(z_i)$. Also as before, let $C(z_i)$ denote the cost of an optimal solution for the subproblem with range $[z_i, b]$ consisting of all intervals contained in that range.

Now, once again consider the consequences of Lemma 3. If

$z_i \neq fwd(z_i)$ (or, equivalently, $z_i + H$ lies in a gap), we have

$$C(z_i) = \frac{fwd(z_i) - z_i}{H} c(H) + C(fwd(z_i)). \tag{8}$$

The corresponding solution consists of a sequence of length H intervals ending at the points $z_i + H$, $z_i + 2H$, $\cdots$, $fwd(z_i)$, followed by an optimal solution for the range $[fwd(z_i), b]$. On the other hand, if $z_i = fwd(z_i)$, then, from the fact that every right endpoint of an interval or left end of a block in this subproblem is included among $\{b(z_j): j \geq i\}$, $C(z_i)$ will be the smaller of

$$\min_{\substack{j \geq i \\ b(z_j) \leq z_i + H}} \{c(b(z_j) - z_i) + C(z_{j+1})\} \tag{9}$$

and

$$\min_{\substack{j \geq i \\ b(z_j) > z_i + H \\ bkwd(z_j) \leq z_i + H}} \left\{ \left\lfloor \frac{b(z_j) - z_i}{H} \right\rfloor c(H) + c((b(z_j) - z_i) \bmod H) + C(z_{j+1}) \right\}. \tag{10}$$

Formula (9) covers all the possibilities in which the leftmost segment does not end in a gap (but includes some potential ending points in gaps that could have been ignored), and (10) covers all the possibilities in which the leftmost segment does end in a gap and is followed by a sequence of segments of length H that terminates at the right endpoint of some interval or left end of some block (the corresponding $b(z_j)$). The solution corresponding to a particular choice of j is, in (9), the segment $[z_i, b(z_j)]$ followed by an optimal solution for the subproblem with range $[z_{j+1}, b]$ and, in (10), is the segment $[z_i, z_i + (b(z_j) - z_i) \bmod H]$, followed by a sequence of length H intervals starting at the points $z_i + (b(z_j) - z_i) \bmod H$, $\cdots$, $b(z_j) - 2H$, $b(z_j) - H$, followed by an optimal solution for the subproblem with range $[z_{j+1}, b]$. Notice that the condition that $bkwd(z_j) \leq z_i + H$ will be satisfied if and only if the left endpoints of all those length H segments belong to gaps, as required. Since $bkwd(z_j) \leq b(z_j)$, we can combine (9) and (10) to obtain, for the case of $fwd(z_i) = z_i$,

$$C(z_i) = \min_{\substack{j \geq i \\ bkwd(z_j) \leq z_i + H}} \left\{ \left\lceil \frac{b(z_j) - z_i}{H} \right\rceil c(H) + c((b(z_j) - z_i) \bmod H) + C(z_{j+1}) \right\}. \tag{11}$$

Notice that the cost for the interval starting at z_i is accounted for in the first term of the minimization if that interval has length H and otherwise is accounted for in the second term of the minimization.

Equations (8) and (11) will serve as the basis for our algorithm. (The further simplifications used in the previous cases do not apply here, since they hold only for cost functions that are linear). We assume that the algorithm is given as input the integer m, the sequence $z_1, z_2, \cdots, z_m$, and, for $1 \leq i \leq m$, the corresponding values for $b(z_i)$, $fwd(z_i)$, and $bkwd(z_i)$. These can be computed using the methods of the previous subsection in time $O(n \log n)$. The algorithm computes $C(z_1)$ and the trace vector T. The value of $T(i)$ will be that value for j for which the cost expression for $C(z_i)$ is minimized and from which the segments in the optimal solution can be reconstructed. $T(i)$ will be undefined whenever $z_i \neq fwd(z_i)$, since then we already know that the solution for that subproblem begins with a sequence of adjacent length H segments running from z_i to $fwd(z_i)$.

Algorithm 4:

```
initialize $C(z_m) := 0$;
for $i := m - 1$ to 1 by $-1$ do
  if $z_i \neq fwd(z_i)$
    then $C(z_i) := ((fwd(z_i) - z_i)/H)c(H) + C(fwd(z_i))$
    else {
      initialize $C(z_i) := \infty$; $T(i) := \infty$;
      for $j := m - 1$ to $i$ by $-1$ do
        if $bkwd(z_j) \leq z_i + H$
          then {
            $hmult := \left\lfloor \frac{b(z_j) - z_i}{H} \right\rfloor$;
            $resid := (b(z_j) - z_i) \bmod H$;
            $newC := hmult \cdot c(H) + c(resid) + C(z_{j+1})$;
            if $newC < C(z_i)$
              then $\{C(z_i) := newC; T(i) := j\}$;
            };
    };
```

It is straightforward to verify that the algorithm runs in time $O(m^2) = O(n^2)$. The correctness of the algorithm follows from the preceding discussion and Lemma 3.

Theorem 5: Algorithm 4 correctly computes $C(z_1)$ and the trace vector T in $O(n^2)$ steps.

IV. OPEN PROBLEMS

The obvious open problems are to ask for improvements on and simplifications of the algorithms derived in this paper. Given the apparent need for sorting, it appears unlikely that the speed of our first three algorithms can be improved in any major (i.e., asymptotic) way. However, the algorithm for the concave cost function case offers some room for improvement here, even though it intuitively seems that the arbitrary nature of such functions can force one to evaluate the cost function for $O(n^2)$ different segment lengths in general. Further simplification of our algorithms, on the other hand, seems feasible and certainly would be useful.

There is also a generalization of our problem that is of great potential interest. Suppose that the problem definition is augmented to include an additional integer K as input, with the additional restriction on solutions that no more than K segments can ever overlap at a single point. This overlap constraint arises in situations where extensive multiple exposures are likely to arise and cannot be tolerated anywhere on the layout. It is easy to see that all of our normalization lemmas continue to hold under such a constraint, i.e., the transformations done in their proofs do not increase the maximum number of overlapping segments in the solution. However, it is no longer the case that in an optimal solution the solutions to subproblems (of the types considered in this paper) need be optimal solutions for those subproblems, since they may need to satisfy certain derived (and possibly complicated) additional constraints on the maximum overlap in various subregions. At present we do not see an appropriate, more restrictive, definition of subproblem that is both sufficient to allow these problems to be solved via dynamic programming and, at the same time, leads to a small enough class of subproblems for any given problem instance that all can be solved in a reasonable amount of time. We would be very interested in either efficient algorithms for problems of this sort or convincing demonstrations that these problems are inherently intractable.

V. ACKNOWLEDGMENT

We would like to thank Dave Alles for suggesting the problem and for helpful discussions.

REFERENCES

1. D. R. Herriott et al., "EBES: A Practical Electron Lithographic System," IEEE Trans. Electron Dev., *ED-22*, No. 7 (July 1975), pp. 385–92.
2. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Reading, MA: Addison-Wesley, 1968, Chap. 2.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Reading, MA: Addison-Wesley, 1983, Chap. 5.

AUTHORS

Michael R. Garey, B.S. (Mathematics), 1967, M.S. (Computer Science),

1969, and Ph.D. (Computer Science), 1970, University of Wisconsin; AT&T Bell Laboratories, 1970—. Mr. Garey has been Head of the Mathematical Foundations of Computing Department since 1981. He has done research in various areas of mathematics and computer science, including combinatorics, graph theory, design and analysis of algorithms, and computational complexity. He was awarded the 1979 Lanchester Prize of the Operations Research Society of America, and from 1979 through 1982 he served as editor-in-chief of the Journal of the Association for Computing Machinery. Member, ACM, SIAM, ORSA.

Ron Y. Pinter, B.S. (Computer Science), 1975, Technion—Israel Institute of Technology; S.M. and Ph.D. (Computer Science), The Massachusetts Institute of Technology, 1980 and 1982, respectively; AT&T Bell Laboratories, 1982–1983; IBM Israel Scientific Center, 1983—. Mr. Pinter was a member of the Principles of Computing Research Department, where he had been studying layout algorithms for integrated circuits, computational geometry, and the design of programming languages. Recently, he returned to Israel after spending 5 years in the United States as a Fulbright-Hayes grantee. Member, ACM, IEEE Computer Society.

AT&T BELL LABORATORIES TECHNICAL JOURNAL is abstracted or indexed by *Abstract Journal in Earthquake Engineering, Applied Mechanics Review, Applied Science & Technology Index, Chemical Abstracts, Computer Abstracts, Current Contents/Engineering, Technology & Applied Sciences, Current Index to Statistics, Current Papers in Electrical & Electronic Engineering, Current Papers on Computers & Control, Electronics & Communications Abstracts Journal, The Engineering Index, International Aerospace Abstracts, Journal of Current Laser Abstracts, Language and Language Behavior Abstracts, Mathematical Reviews, Science Abstracts (Series A, Physics Abstracts; Series B, Electrical and Electronic Abstracts; and Series C, Computer & Control Abstracts), Science Citation Index, Sociological Abstracts, Social Welfare, Social Planning and Social Development*, and *Solid State Abstracts Journal.* Reproductions of the Journal by years are available in microform from University Microfilms, 300 N. Zeeb Road, Ann Arbor, Michigan 48106.

AT&T
Bell Laboratories