A Performance Study of Remote Executions in a Wide-Area Datakit Network†

Ronald S. Arbo

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Datakit is a highly modular virtual-circuit switch that is designed to act as a component of a universal data transport system. The network architecture is structured to provide efficient data transmission in both wide-area and local-area environments. AT&T Bell Laboratories has established a nation-wide Datakit research network, called XUNET, to study Datakit performance in the wide-area context. Our study utilizes this network to characterize the performance of remote executions on UNIX‡ hosts.

We designed a number of user-level and network-level tests to identify and characterize the host and network delay factors that contribute to remote command execution delay, as seen by the user. This delay is composed of call-processing delays within the network nodes during call setup, processing time at the destination host, and call-processing delays within the network nodes during call teardown. We found that mean call-processing delay during both call setup and call teardown is roughly linear with the node count of the virtual-circuit path that connects the source and destination hosts. The mean call setup delay is 0.30 seconds per node, while the mean call teardown delay is 0.15 seconds per node.

We determined that the delay encountered at the remote host is composed of (1) the processing time of a UNIX server process, which handles Datakit call requests on XUNET hosts, and (2) the execution time of one or more processes created by the server to execute the command(s) specified for remote execution. In general, these host delays are more difficult to quantify than the network delays because host load fluctuations during testing tend to introduce unpredictable distortions in the results.

† This work was sponsored in part by AT&T Bell Laboratories and in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing official policies, either expressed or implied, of the Department of Defense or of the U.S. Government.

‡ UNIX is a trademark of AT&T Bell Laboratories

1. Introduction

Since the introduction of data networks, their use has grown at a phenomenal rate. The tremendous expansion of the ARPA Internet from less than ten hosts in 1970 to its present configuration consisting of hundreds of networks and thousands of hosts is just one example of this growth. Large networks can now be found in the financial, academic, commercial, and military environments, as well as in the original research environment. Primarily, the desire to share expensive computing resources to increase their effective utilization has fueled this growth.

Computer systems research has increasingly focused on techniques to improve utilization of distributed computing resources. Most notably, investigations are ongoing into the techniques of load balancing (LB) and distributed operating systems (DOS), which perform job allocation in a distributed system with the goal of maximizing the performance and the overall utilization of the system resources.

To effectively design a LB/DOS, one must consider the environment in which the system will be installed. For example, some LB/DOS design decisions can be influenced by the performance characteristics of the underlying data network on which the system will be implemented. One of the functions of a LB/DOS is to determine when the performance gains of migrating a job to another processor outweigh the performance losses caused by the overhead to perform the migration. Since job migration usually involves message passing between processors on a data network, the performance limitations of the network can be a significant factor in the overall job migration overhead.

In the case of a virtual circuit network, such as AT&T's Datakit network, the latency experienced by job migration messages may (depending on the design of the particular LB/DOS) include call processing time. Therefore, knowledge of the call processing characteristics of a candidate network for LB/DOS can be beneficial to the designers of such systems. For similar reasons, these characteristics can have implications in the design of a large number of other distributed applications, both at the user and system level.

To provide the designers of such applications with a characterization of Datakit call processing performance, we have implemented a number of user- and network-level tests on an experimental nation-wide Datakit network. This network, known as XUNET, connects a number of Digital Equipment Corp. (DEC) VAX hosts running the UNIX operating system.

Our initial approach was to measure remote execution delay across various segments of XUNET at the user level (as perceived by a user of the remote execution facility). We assumed that the results of this test would provide valuable insight into Datakit call processing performance; we also expected that these results might suggest additional tests that would prove beneficial in further characterizing call processing performance. To implement these initial measurements, we instrumented the UNIX command *rx*, which is run at a source host whenever a remote execution is to be performed.

We initially expected the results of our research to show that the user-level delay for a remote execution across a Datakit network is dependent on the following factors:

1) Number of nodes in the virtual-circuit path.
2) Destination host configuration (hardware and software).
3) CPU utilization of command executed.
4) Physical distance between source and destination host along the virtual-circuit path.

We list these delay factors from most to least significant, in terms of the effect we expected each to have on the user-level remote execution delay. We expected the node count to be the most significant factor, with the other factors contributing to a lesser extent.

The results of the initial user-level tests were not consistent with our expectations. To further investigate, we performed three additional tests, one of which included the addition of timestamping code directly within the Datakit network controller, *Radian*, and another of which included the insertion of timestamping code within a UNIX server process that handles Datakit call requests at the destination host. This document describes the motivation, methodology and results of each of these tests.

The remainder of this document is organized as follows: Section 2 describes the configuration of the experimental network (including the attached hosts) at the time the tests were performed during the summer of 1988. Sections 3 through 6 describe the methodology and the results of each remote execution test. Section 3 describes the measurement of typical user delays. Section 4 describes the measurement of delays as a function of the virtual-circuit path length. We present Radian call-processing delays in Section 5. In Section 6, we describe our measurements of the delays encountered at the destination host. Finally, in Section 7 we summarize our findings.

Appendix A contains a description of the operating principles and implementation of Radian, the Datakit controller. A basic understanding of Radian operation is helpful in interpreting the results of the Radian call processing measurements, presented in Section 5. For those readers desiring a more detailed knowledge of Radian operation, Appendix B contains a detailed chronology of the Radian processing that occurs during a single virtual circuit setup and takedown. Finally, Appendix C contains selected measurement results in tabular form.

2. The Experimental Environment - XUNET

XUNET (the eXperimental University NETwork) is a wide-area virtual circuit network that utilizes Datakit technology. We shall describe the origin and configuration of the XUNET, while only briefly mentioning the basic Datakit operating principles. For a complete discussion of these principles, see [5].

Datakit is designed to perform efficient data transmission for a wide variety of traffic types in both the wide- and local-area environments. For example, Datakit can ideally satisfy the high-bandwidth requirements of bulk file transfers, while also meeting maximum-delay restrictions imposed for interactive traffic.

Each Datakit node consists of a cabinet that houses a shared backplane through which all attached devices communicate. Devices interface to the backplane through interface modules that plug into the cabinet and perform buffering and backplane contention procedures. Utilizing this configuration, each node performs switching and multiplexing on user data being carried on virtual circuits. In addition, a network controller at each node performs call processing procedures. In the case of a wide-area Datakit network, such as XUNET, one or more Datakit nodes are usually located at each user site. User data is carried between distant user sites through multiplexed T1 transmission lines at the rate of 1.3Mb/s.

Since the original Datakit design, a large number of Datakit networks have been installed throughout the country. Currently, AT&T Bell Laboratories uses a Datakit network to meet the majority of its everyday data communication needs between Bell Laboratories locations in several different states. In addition, Datakit networks have been installed at numerous other commercial and government organizations. Some of these networks provide an installed base on which ongoing Datakit performance testing can take place with a minimum of inconvenience and expense.

However, the vast majority of these Datakit installations are arranged in the local- or medium-area configuration. Although changes and enhancements have been made to the original Datakit design as a result of the experiences gained from these Datakit implementations, the Datakit design has not been extensively tested in the wide-area context.

The XUNET has been established by Bell Laboratories in an effort to accumulate experience using Datakit in the wide-area context and to provide a facility for wide-area Datakit network testing. The network is composed of four user sites; three are located at major universities across the country, and the fourth is located at the Bell Laboratories facility in Murray Hill, NJ (see Figure 2-1).

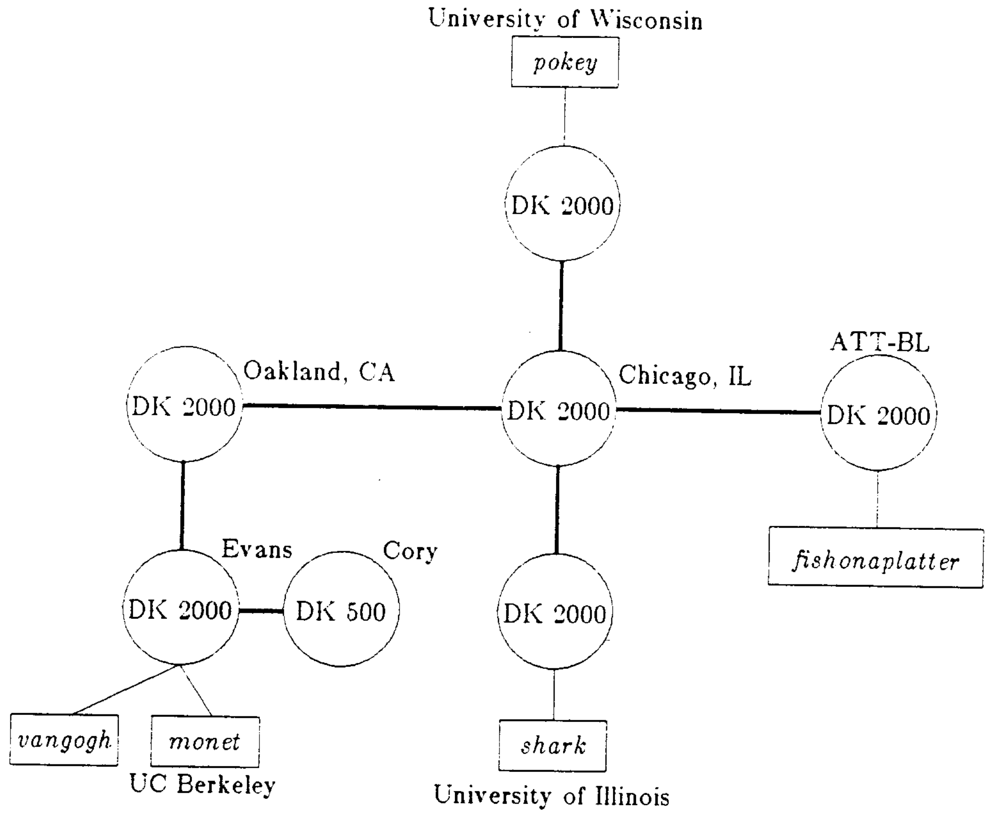


FIGURE 2-1. XUNET TOPOLOGY

In Figure 2-1, each XUNET Datakit node is represented as a circle. Two different versions of Datakit nodes are used - a Datakit 2000, and a Datakit 500. The two versions differ only in the total number of attached communicating devices that each can accommodate, the Datakit 2000 having the larger capacity. The nodes at Chicago, IL and Oakland, CA are located in AT&T central offices.

The XUNET hosts utilized in our research are represented as boxes in Figure 2-1. Relevant configuration information for each host is shown in Table 2-1. Note that all hosts are running the standard 4.3 BSD UNIX operating system except the Wisconsin host, *pokey*, which is running a version of 4.3 BSD UNIX that includes a Network File System (NFS) implementation [12].

3. Measuring Typical User Delays

TABLE 2-1. XUNET HOST CONFIGURATIONS

Name	Location	Model	Operating System	Interface
vangogh	UCB-evans	VAX 8600	4.3 BSD UNIX	KMC-11B
monet	UCB-evans	VAX 11/750	4.3 BSD UNIX	KMC-11B
shark	U of I	VAX 11/780	4.3 BSD UNIX	KMC-11B
pokey	U of W	VAX 11/750	4.3 BSD UNIX + NFS	KMC-11B
fishonaplatter	ATT-BL	VAX 11/750	4.3 BSD UNIX	KMC-11B

3.1. Methodology

This test was designed to provide a baseline of typical user delays for remote executions over various segments of the XUNET. We expected that the results would assist us in designing further tests to characterize remote executions in the more general wide-area context. Using a version of the UNIX remote execution command *rx* that includes timestamping code, we measured the user delay to perform remote executions of a number of common UNIX commands on four destination hosts, each located at a different XUNET user site. A single XUNET host, located at UC Berkeley, acted as the source machine in all test cases.

A number of network and host-related procedures contribute to the user-level delay that we measured using the timestamping *rx* code. After taking the first timestamp, the *rx* process initiates a virtual circuit setup procedure between the source and destination host. If the call is successfully established, a UNIX server process on the destination host performs various security checks on the newly established call. If the call passes these security checks, the server process sends a "proceed" message to the source host and creates a command interpreter process (shell) to handle command execution at the destination host.

When the *rx* process on the source host receives the "proceed" message, it transmits the command(s) to be remotely executed over the virtual circuit to the destination host. The command interpreter process at the destination, previously created by the server, then initiates command execution (which may include the creation of other processes to perform the actual execution of each command). All command output is transmitted back to the source host. When command execution is complete, the destination host initiates a procedure to close the connection. At the completion of call takedown, the *rx* process at the source host is notified, and the final timestamp is taken within the *rx* process. The delay is calculated based on the values of the two timestamps, and the result is stored in a log file for later processing. The timestamps within the *rx* code are implemented using the UNIX system call, *gettimeofday*, which provides an accuracy of ± 1 millisecond (ms) on the VAX 8600.

To perform one test trial, consisting of timestamped remote executions of twenty-nine different UNIX commands on four different destination hosts, we designed a shell script to execute on the source host. Shell script operation is defined using pseudo-code in Figure 3-1.

Our testing was performed in an environment that included the existence of extraneous load on the destination machines. Although some of the hosts were in general lightly utilized (namely, *fishonaplatter*, *vangogh*, and *shark*), others acted as departmental machines which were accessed daily by a large user population. In addition, we observed that some destination machines experienced significant load fluctuations as a result of the background execution of certain system processes. For instance, the XUNET host *pokey* started a number of system processes related to its network file system each night. During execution of these processes, remote execution delay increased noticeably when using this host as the destination.

FIGURE 3-1. SHELL SCRIPT USED IN TESTING

```
for (destination host 1 to destination host 4) do
    begin
        record destination host load indices

        for (typical unix command 1 to typical unix command 29) do
            begin
                timestamp #1
                perform remote execution
                timestamp #2
            end

        record destination host load indices
    end
```

Although we were not authorized to restrict user access to the involved hosts during testing, we took a number of other precautions to minimize the effects of extraneous host load on our results. First, all testing was performed during the early morning hours to increase the likelihood that the hosts would be lightly loaded. In addition, a measure of each destination host's load was recorded directly before and after each trial on a particular destination (see Figure 3-1).

These load measurements were recorded in a log file along with the test results for each trial. We performed a total of sixteen trials to each destination over a period of four nights. We considered the four trials that had the highest destination host load measurements to be the "outliers" - these trials were discarded. The load measurements consisted of the average CPU run queue lengths over the previous 1, 5, and 15 minutes.

The final precaution we took to minimize the effects of unpredictable host load changes was the use of a very large, lightly loaded machine as the source host. This is a VAX 8600, *vangogh*, located at UC Berkeley. We assumed that since *vangogh* is an order of magnitude faster than the VAX 750's and 780's used as destination machines, and it is consistently lightly loaded, it was not necessary to monitor the load as frequently as for the destination machines.

Our next decision was to choose a set of UNIX commands that was suitable for remote execution testing, and would yield the most interesting results. We followed two simple guidelines to choose the appropriate commands. First, we required that the chosen commands be representative of those commands that were most frequently executed on general-purpose UNIX systems. Second, we required that the chosen commands represent a wide variety of CPU and I/O usage characteristics (to maximize the scope of our results).

Although we had access to accounting information on each XUNET host that included a complete log of the most recently executed UNIX commands, we chose to consult the literature to determine the commands to be used for testing. We believe that the set of commands executed recently on any XUNET host is not necessarily a good representation of long term command usage characteristics on general purpose UNIX systems.

Although a number of studies have been performed on the subject (see [7] and [3]), we based our final command selection on a study performed at UC Berkeley in 1985 on a VAX 11/780 running BSD UNIX [8]. We chose to base our command selection on this study because it provided an ordered list of the most frequently executed UNIX commands. Also, the user population on the host under study was fairly diverse. As such, we believe that the activities on the system at the time represented a wide cross-section of typical activities that might be found on a "general-purpose" UNIX system.

To choose the particular UNIX commands from the study to be used for our testing, we began with the 85 most frequently executed commands (these commands accounted for 88.2% of all commands executed on the system [8]). We next removed those commands in the list that met any of the following characteristics:

- not available on all standard BSD UNIX systems
- interactive in nature (no commands such as *vi*, *man*, *mail*, etc.)
- not executable on stand-alone systems (no commands such as *ruptime*, *ftp*, *rlogin*, etc.)
- unusually large execution time[1]

After removing those commands that fit any of the above characteristics, twenty nine commands remained that were suitable for testing. We next analyze the remote execution results using these twenty-nine commands[2], shown in Table 3-1.

TABLE 3-1. COMMAND INDEX

No.	Cmd.	Function	No.	Cmd.	Function
1	**ls**	directory listing (8 entries)	16	**date**	current time
2	**cd**	change directory	17	**w**	user info
3	**echo**	display argument	18	**uptime**	system info
4	**cat**	file display (346 Bytes)	19	**tail**	file display (396 Bytes)
5	**u**	list current users	20	**ln**	link file
6	**finger**	user info (1 user)	21	**rm**	remove file
7	**grep**	file search (396 Bytes)	22	**fgrep**	file search (396 Bytes)
8	**pwd**	current directory	23	**chmod**	change file permissions
9	**mv**	move a file	24	**mkdir**	make directory
10	**cp**	copy a file	25	**rmdir**	remove directory
11	**f**	user info (all users)	26	**sleep**	process sleep (1 second)
12	**who**	user info	27	**du**	disk space used
13	**egrep**	file search (396 Bytes)	28	**wc**	word count (396 Bytes)
14	**so**	execute script	29	**head**	file display (346 Bytes)
15	**df**	disk free space			

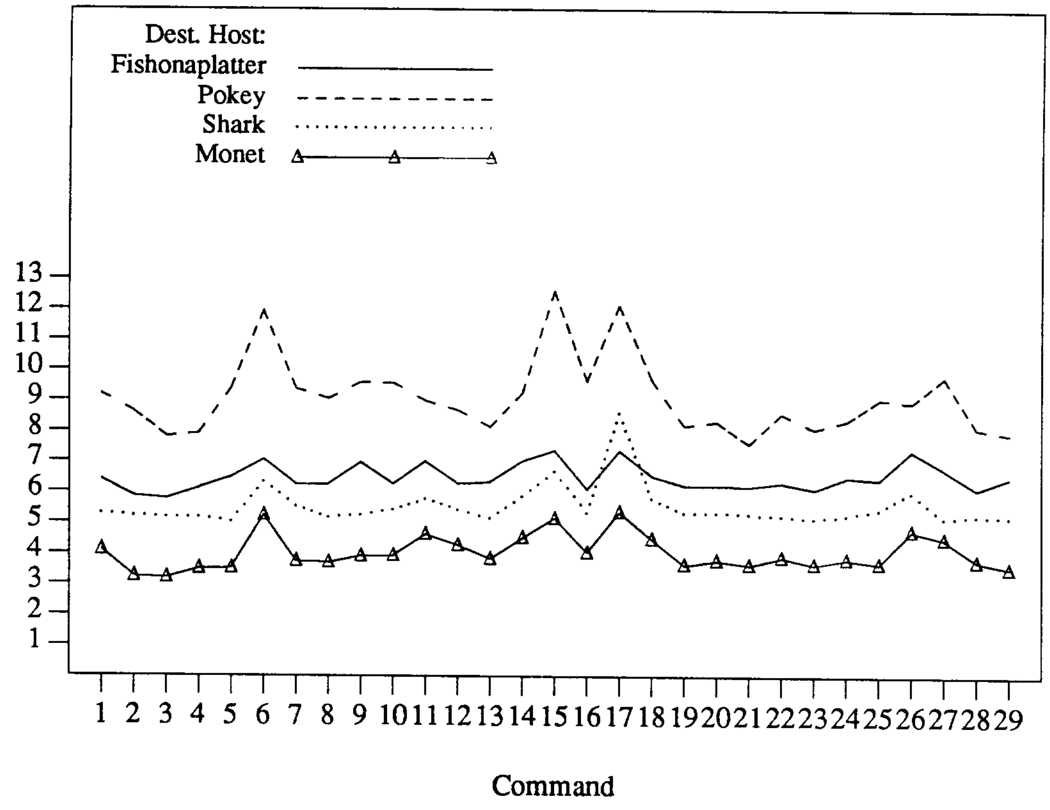
[1] This requirement is subjective, but necessary since commands with high execution time mask network performance during remote execution.

[2] See [4] for more information about these commands.

3.2. Results

Figure 3-2 depicts the mean remote execution delay as a function of UNIX command for each of the four destination hosts. The delay values for each host have been connected only to identify trends. Command index numbers are shown in place of names on the command axis because of space considerations.

FIGURE 3-2. MEAN REMOTE EXECUTION DELAY (SECONDS)



As expected, we find that the remote execution delay to any single destination host is dependent on the particular UNIX command being executed. Delay peaks appear for the UNIX commands: *finger* (6), *f* (11), *df* (15), and *w* (17) on all destination hosts.

We contend that the higher delays for these commands are caused by longer execution times at the remote host, and not by network factors. Closer examination of the function of each of these commands supports this theory. The *w* command opens kernel memory and performs numerous searches to retrieve information concerning the current state of the system. The *finger* and *f* commands both open and search a number of system files to retrieve information about users on the system. Finally, the *df* command performs a read on the superblock of every file system to retrieve the number of free disk blocks. Many of these operations are CPU or I/O intensive; therefore these commands can incur higher execution time than the majority of UNIX commands.

Based on the results shown in Figure 3-2 we also see that the user-level remote execution delay does not increase linearly with respect to the virtual-circuit path length (in terms of node count). The destination hosts *pokey*, *shark*, and *fishonaplatter* are each four nodes away from the source host *vangogh*. Although the virtual-circuit path length is constant to each of these destination hosts, our measurements reveal that the mean remote execution delay varies considerably depending on the virtual-circuit path.

Although the source and destination hosts are not separated by equal physical distances in each of the above cases, we believe that the disparities in the propagation delay to each destination host are not large enough to explain the observed differences in overall user delay to each of these destinations. The disparities in propagation delay are on the order of milliseconds, while the disparities in user delay can be expressed in terms of seconds.

In Section 6, we show that this disparity is instead caused by host-related factors. One of these factors is the non-uniform loading of the destination hosts during testing. Despite the precautions taken to minimize the effects of host load on our results, we are unable to absolutely eliminate these effects. Another factor causing the disparities in user delay among the four node virtual-circuit paths, is the fact that each destination host has a different configuration, implying that command execution time may not be the same on all four hosts, even for the same command.

FIGURE 3-3. STANDARD DEVIATION OF REMOTE EXECUTION DELAY (SECONDS)

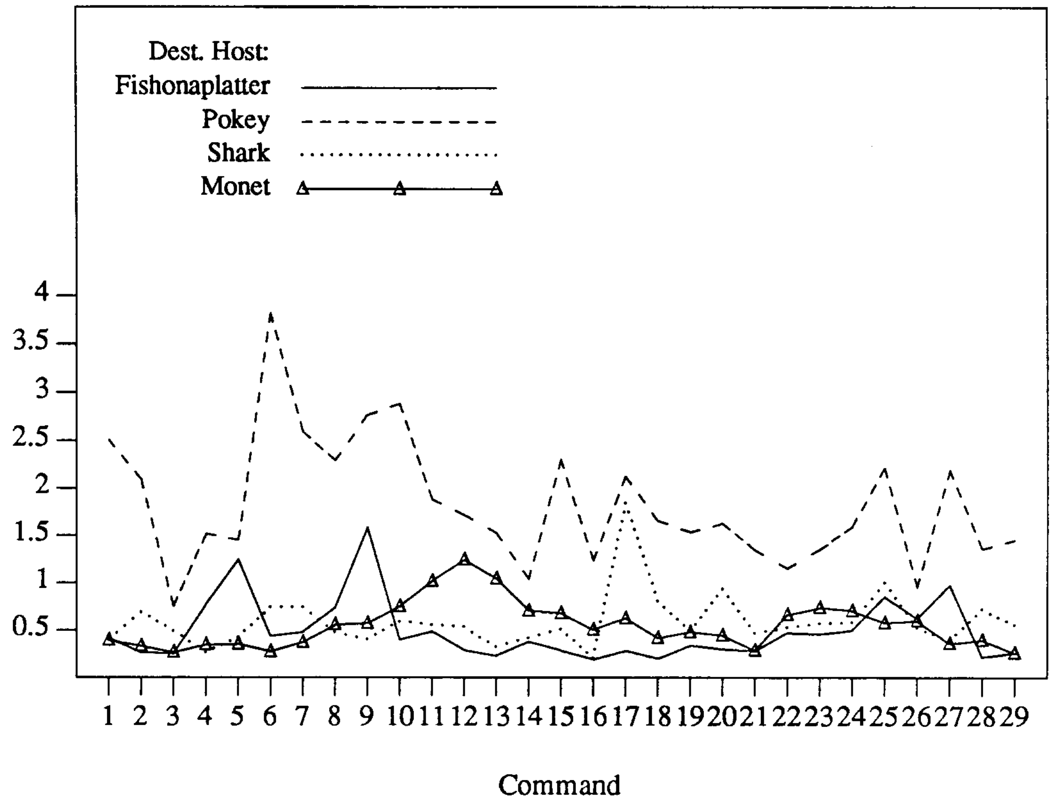


Figure 3-3 shows the standard deviation of the command delay results. The command index is shown in Table 3-1. We see that the standard deviation of command delays on *pokey* are larger than those of any other destination. This is especially true for the long execution time commands mentioned above. Since *pokey* was the most heavily loaded destination host, we conclude that the variance is caused by unpredictable extraneous host load during testing. Load index measurements taken during testing verify this conclusion.

4. Measuring Network Delays

4.1. Methodology

The results of the user-delay test described in the previous section show that the user delay for remote executions is composed largely of delays incurred at the remote host as well as delays incurred within the network. In addition, although we had taken precautions against load fluctuations at the destination host, their effects can still be seen in the high variance of the results.

This test was designed to minimize the effects of the host-related delay factors during remote execution in order to study the network factors independently. Our approach was to select a single UNIX command to be remotely executed in all test cases that would place minimum demands on the remote CPU and I/O subsystem, thereby making execution time at the remote host a smaller fraction of the total user delay.

In addition, we increased the number of remote execution trials to fifty for each destination host. We then analyzed only the *minimum* measured delay for each destination. We assumed that this represented the ideal case when the command is executed immediately after arrival at the remote CPU, thereby reducing the host related delays in our results to a minimum.

To enable further study of the dependence between the number of nodes in the virtual-circuit path and the user-level delay, logical loops were installed in the Cory, Oakland, Chicago, and New Jersey XUNET nodes (see Figure 2-1). These loops enabled us to remotely execute commands from the source host, *vangogh*, through various virtual-circuit paths consisting of one, three, four, five, or seven nodes, instead of being restricted to the one or four node path lengths available from *vangogh* without using logical loops.

A string of ASCII characters called a "dialstring" specifies a destination device within a Datakit network. During call establishment, the destination dialstring, provided by the call originator, is passed to each subsequent node in the virtual-circuit path. Each node routes the call based on the dialstring information. In the case of a "looped" virtual-circuit path, the node containing the loop translates the dialstring so that the incoming virtual circuit is routed back to the original node at which the call request originated.

FIGURE 4-1. SHELL SCRIPT USED IN TESTING

```
for (virtual-circuit path 1 to virtual-circuit path 8) do
    begin
        record destination host load indices

        for (repetition 1 to repetition 50) do
            begin
                timestamp #1
                perform remote execution
                timestamp #2
            end

        record destination host load indices
    end
```

The shell script to perform this test is shown in Figure 4-1 using pseudo-code. We chose to use the UNIX command *echo* in all test cases. Because the *echo* command is built directly into the csh command interpreter, a time consuming process creation is avoided when executing the command; the csh process performs the command execution directly. Also, since the command is executed without an argument, minimum command output is generated for transmission back to the source host. This minimizes the effects of delay factors that may be dependent on the size of the command output returned to the source host.

4.2. Results

In this section, we first describe the final results of the network delay measurements, then we briefly discuss two network problems that originally corrupted the results of our first attempts at these measurements.

FIGURE 4-2. MINIMUM REMOTE EXECUTION TIME VS. VIRTUAL CIRCUIT PATH LENGTH (SECONDS)

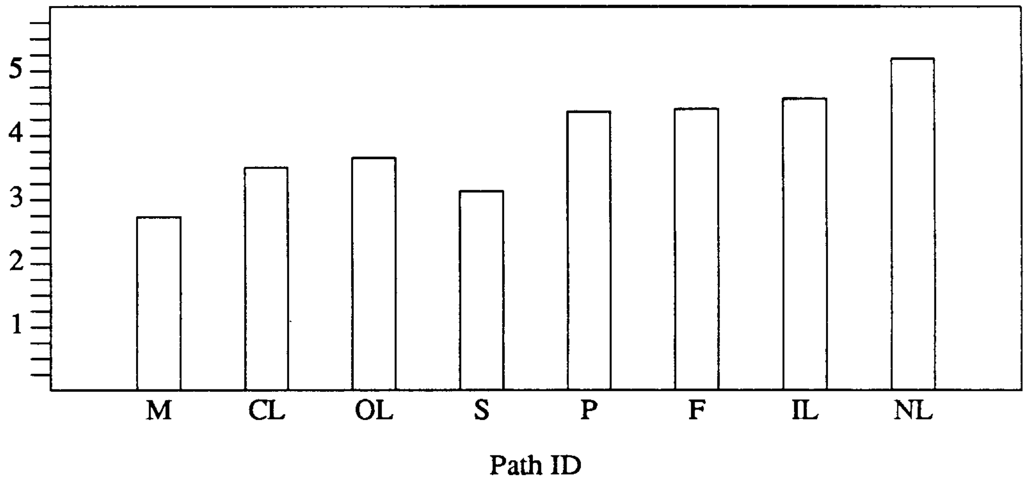


Figure 4-2 illustrates the minimum remote execution delay for each virtual-circuit path. To be concise, the virtual-circuit paths have been labelled on the graph ordinate with an ID that can be referenced in Table 4-1.

TABLE 4-1. PATH INDEX

Path ID	Node Count	Path
M	1	*evans*
CL	3	*evans - cory - evans*
OL	3	*evans - oakland - evans*
S	4	*evans - oakland - chicago - illinois*
P	4	*evans - oakland - chicago - wisconsin*
F	4	*evans - oakland - chicago - nj*
IL	5	*evans - oakland - chicago - oakland - evans*
NL	7	*evans - oakland - chicago - nj - chicago - oakland - evans*

We see from Figure 4-2 that although there is a generally increasing trend in user delay as a function of path length, not all paths conform to this trend. Most notably, the S path, which is four nodes long, yields a minimum delay that is smaller than those of the three node paths CL and OL. In addition, the five-node path, IL, shows only a slight delay increase over paths P and F, which are only four nodes in length. However, despite these inconsistencies, the results are close to our original expectations of a linear dependence between user delay and virtual-circuit path length.

We should explain that the reason we expect to see a user-delay characteristic that is linear with the node count of the virtual-circuit path is that each node in the path must perform a similar procedure to set up the virtual circuit. This procedure includes call routing, channel allocation, window size negotiation, and a modification to switch memory to establish the virtual circuit. (We explain this procedure in great detail in Appendix B). Since the procedure is the same for each node, we assume that each node would incur the same average delay.

With regard to our efforts to minimize the standard deviation of the results through higher repetition counts and the use of a single low CPU utilization command, we have been successful only for the P path, which had the highest standard deviation of all paths in the typical user delays test. We believe that we were successful primarily because the load on *pokey* was lower during testing in this case. This reaffirms our previous conclusion that the variance of our results is caused primarily by load fluctuations at the destination host. The minimum, mean, and standard deviation of the results shown in Figure 4-2 can be found in Appendix C in tabular form.

We next describe two problems that originally occurred during execution of this test. Each problem caused drastic call processing performance degradations in our original results. The first problem involves the Radian network controller software that executes on a DEC PDP-11 microcomputer attached to each Datakit node. The Radian software contains an option to output varying levels of diagnostic information on the network console during call processing activity. The diagnostic information is generated on the network controller and output to the network console by the Radian software. The console is connected to the controller through the Datakit node with an RS-232 connection at 1200 baud. The Radian controller software places diagnostic messages destined for the console in an internal buffer. The contents of the buffer are emptied to the console at the rate of 1200 baud.

If a large volume of data is to be transmitted to the console, then the controller must wait until the buffer at least partially empties before it can write additional diagnostic messages to the buffer. This process significantly slows call processing since the Radian software cannot resume call processing duties until each diagnostic message is entirely written to the buffer. We discovered that this Radian option had been set on the Oakland Datakit console, which corrupted our original results for all virtual-circuit paths that contained that node.

The second problem we encountered was caused by electromagnetic interference on a number of unshielded RS-232 lines that connect terminals to the Evans Datakit node. The Radian network controller monitors the RS-232 leads on each terminal interface line. Voltage fluctuations on this lead can be mistakenly interpreted by Radian as various types of call processing messages. In this case, the interference on these lines loaded the Radian processor so that overall call processing performance through the node was severely degraded.

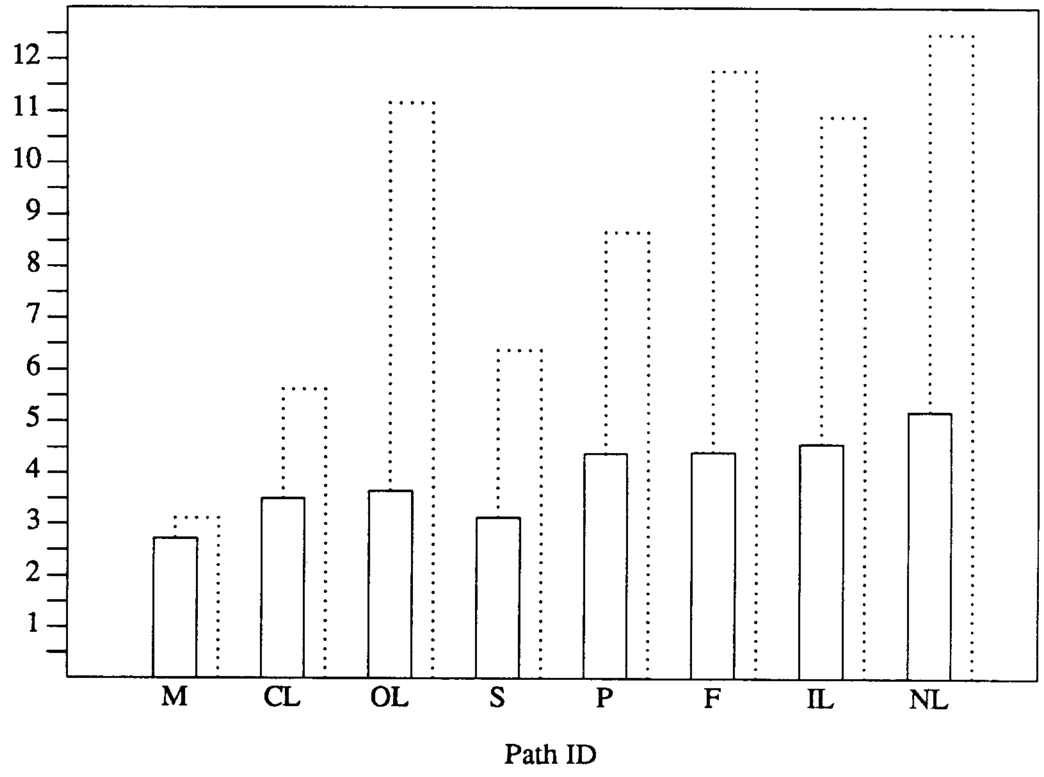
The measurements made while the console output problem was present are shown in dotted lines in Figure 4-3. For comparison, the results obtained after all network problems were corrected are shown in the foreground in solid lines. Results obtained while the terminal interface problem was present are of the same magnitude.

5. Measuring Radian Call-Processing Delays

5.1. Methodology

This test was created to measure Radian call processing delays directly by using timestamping code within the Radian network controller software, which runs on a DEC microcomputer attached to each Datakit node. We hoped to accomplish two goals: (1) to determine if Radian

FIGURE 4-3. MINIMUM REMOTE EXECUTION TIME (CONSOLE OUTPUT CASE)



call processing delay during call set up and takedown is constant across all XUNET nodes, and (2) to arrive at an estimate of this delay.

To implement the timestamping, we performed a minor Radian software modification. This modification simply consisted of increasing the resolution of a set of timestamps that were already a part of Radian. To accomplish our goals, we decided it was not necessary to insert additional timestamping code beyond that which was already included. The resolution was increased from 1 second to 1/60th of a second (which we round up to 0.02 seconds in our results).

We next present a brief description of the specific timestamp locations within the Radian software. This description is intended for those readers already acquainted with Radian operation. Those readers desiring more information about Radian are referred to Appendix A for a description of the theory and operation. In addition, for a more complete chronology of the Radian operations (including timestamping) that occur during a virtual circuit session, which includes call establishment, connect time, and call takedown, see Appendix B.

During the course of a Datakit virtual circuit session, each node in the virtual-circuit path issues three timestamps. Two of these timestamps occur during the call setup phase, and the final occurs during call takedown. The first call setup timestamp marks the beginning of the setup procedure at a particular node. It is issued by the Radian process on that node that first receives a call setup request message from a communicating device attached to the node. We refer to this as the "c" (lower-case c) timestamp.

The second timestamp occurs when the switch memory on a particular node is written to establish the call through that node. The event that triggers the write to switch memory depends on whether the node is connected to a subsequent node in the virtual-circuit path or whether it is connected directly to the destination host. In the former case, the switch memory is written at the same time a Radian process on that node forwards the call setup information to the next node in the virtual-circuit path. In the latter case, the switch memory is written when a Radian process transmits a connection request message to the server process on the destination host. In either case, we refer to this as the "C" (capital C) timestamp.

The final timestamp occurs during call takedown. One of the two parties communicating over the virtual circuit signals the node to which it is connected that it wishes to teardown the connection. In the case of a remote execution call, the destination host normally transmits a call teardown request signal after command execution is completed. A Radian control message indicating the call takedown is then passed back to the source host through each node in the path. When a Radian process in each node first receives this message, which enables it to erase the switch memory entry for that circuit, it issues a "d" timestamp.

To generate these timestamps for our experiment, we executed the same shell script used in the previous experiment. The UNIX command *echo* was again used in all test cases. A single execution of this script on *vangogh* yielded fifty trials for each of eight different virtual-circuit paths. However, despite the wide variety of virtual-circuit path lengths available for testing, the scope of our results was limited by the fact that the Radian software modifications were performed on only three of the XUNET nodes. Logistical constraints permitted us to modify only the Evans, Cory, and Illinois XUNET nodes.

5.2. Results

The results of the Evans node Radian timestamping are shown below in Figure 5-1 for the direct (non-looped) virtual-circuit path cases. The path ID's can be referenced in Table 4-1. The node count for each path is shown in parentheses next to each path ID. All Radian call processing delays are referenced to the initial "c" call request timestamp.

Figure 5-1 shows that, with the exception of the M path, the mean call set up time through the Evans node is constant at 0.30 seconds for every direct virtual-circuit path through that node (full results including minimum and standard deviation values are found in Appendix C). The disconnect times (d), however, are not the same, even for those paths having identical node counts (see paths S, P, F in Figure 5-1).

We conjecture two possible causes for this disparity in disconnect times. (1) the remote host command processing time is not constant over all destination hosts, and/or (2) the call processing time is not constant among all nodes in XUNET, implying that the delay to establish a complete virtual circuit (the sum of the delays incurred at each node in the path) is dependent on the particular path as well as the node count of the path. Similar Radian measurements taken on the Illinois and Cory nodes support the first theory. The mean time to set up a call (call request "c" to call connect "C") through the Illinois and Cory nodes was also measured to be 0.30 seconds, an identical delay to that encountered on the Evans node (except for the results obtained using the M path, which we will explain shortly using the timestamps taken for the looped virtual circuit paths).

The results of the Evans node Radian timestamping for the looped virtual circuit path cases are shown below in Figure 5-2. The time stamp labels c, C, and d are subscripted with an "i" or an "o" to denote whether each stamp refers to the outgoing segment of the virtual circuit (before it reaches the looping node) or the incoming segment. In all cases, *vangogh* acted as the source host, and *monet* was the destination. Both of these hosts are attached directly to the Evans Datakit node. As before, the path ID's are referenced in Table 4-1, and the node count for each path is shown in parentheses next to the path ID.

The results of Figures 5-1 and 5-2 reveal that all calls through the Evans node can be classified into one of two categories based on the mean delay to perform the call set up. The first category

FIGURE 5-1. MEAN RADIAN CALL PROCESSING DELAYS
DIRECT VIRTUAL CIRCUIT PATHS

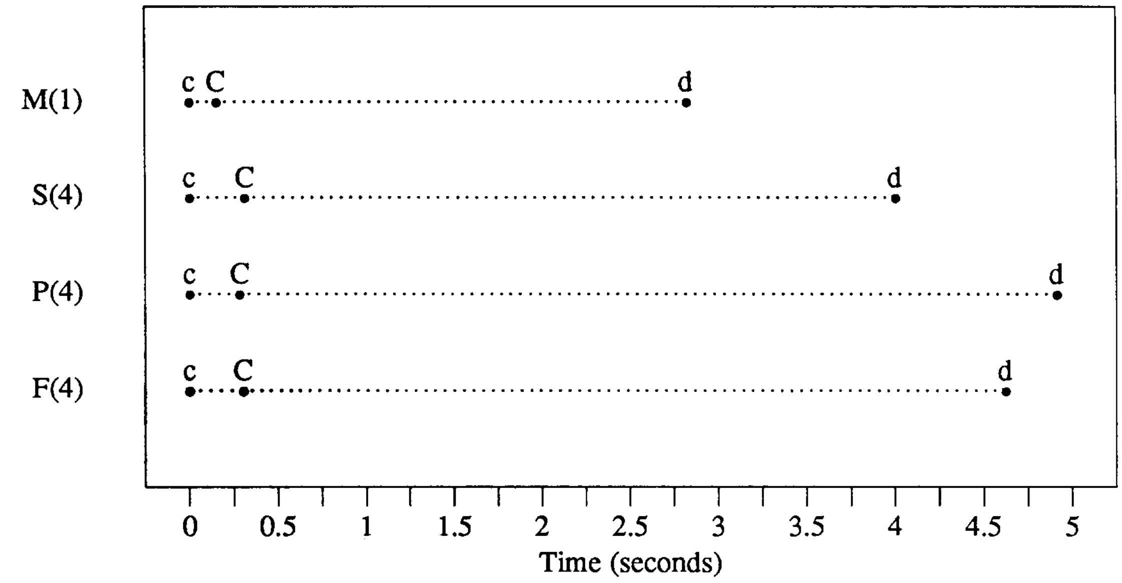
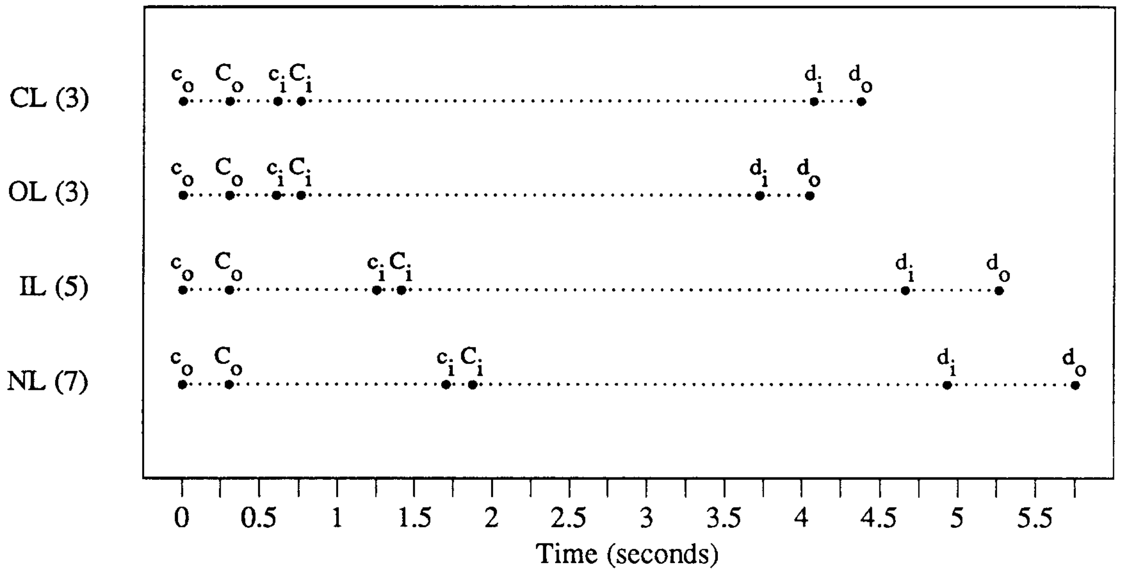


FIGURE 5-2. MEAN RADIAN CALL PROCESSING DELAYS
LOOPED VIRTUAL CIRCUIT PATHS



includes all calls established directly to a destination host from the Evans node (the *direct-connected* case). We have measured a mean call set up delay of 0.16 seconds for a call of this type. Examples are the incoming calls on the looped virtual-circuit paths, and the direct call through the virtual circuit path M. The second category includes those calls that are established to subsequent nodes in the virtual-circuit path (the *forwarding* case). We have measured a mean set up delay of 0.30 seconds for calls of this type. Examples are the outgoing calls for the looped virtual-circuit paths, and calls through each of the direct paths except M.

Analysis of the Radian call processing procedures for the two cases reveals two reasons for this apparent delay disparity. First, the call set up procedure in the two cases is inherently different.

Second, as we shall see in the call set up description that follows, the "C" timestamp for the *direct-connected* case occurs before the call set up is complete.

During call set up in both of the above cases, the "c" timestamp occurs directly after the call setup request message is transmitted via interprocess communication (IPC) from the source to the destination line process within the Radian controller on the Evans node. The destination line process in both cases then performs a few simple operations including window-size negotiation for the link. From this point, the Radian procedure differs depending on which of the above cases applies.

In the first case, in which the node is directly connected to a destination host, the destination Radian line process, which is assigned to a logical channel on the host interface, sends a call request message via IPC to the Radian server line process assigned to the destination host. This is a low-delay procedure since the message is being transmitted between two line processes that are both executing in the same microcomputer (which is acting as the controller on the Evans node). Immediately after the message is transmitted, the destination line process generates the "C" timestamp, although the call set up is not complete. In order to complete the call, the server line process must transmit the call request to the destination host. When the message arrives, a routine executing within the host performs channel and memory allocation and otherwise completes the connection to the destination host.

In the second case, in which the node is connected to the next node in the virtual-circuit path, the destination line-process, which is assigned to a logical data channel on a multiplexed trunk, performs a more complex and time-costly procedure before the "C" timestamp is issued. The destination line process first sends a "start conversation" request to the partner data channel line process at the remote end of the trunk. After the conversation request reaches the remote line process, it sends a dialtone message back over the trunk to the original destination line process. This message signifies that the remote process is now listening and ready to receive messages.

Only at this time does the destination line process transmit the dialstring information to the remote node and generate the "C" timestamp. We see that, since some communication occurs between nodes during this procedure, additional propagation delay is incurred in this case, relative to the first case. This delay can be significant in the wide-area context. In addition, extra line processes are involved in this procedure, possibly further increasing the overall delay.

It is important to note that, although we now understand why the timestamps indicate a delay disparity, the results are inconclusive about whether the mean call set up delay is different in the two cases. The reason is that the timestamps for the *direct-connected* case do not measure the delay incurred during the entire call setup procedure.

Referring again to the results shown in Figure 5-2, we see that mean call setup delays are constant for all XUNET nodes in the *forwarding* configuration. In addition, the variance of these delays is small - less than 23% of the mean in all test cases. We can now estimate the delays for the other XUNET nodes simply by analyzing the Radian timestamp results for the looped virtual-circuit paths through the Evans node.

For example, we see from Figure 5-2 that the mean delay between C_o and c_i for the IL path case is 0.94 seconds. We know that call setup occurs through three nodes during this period - first the Oakland node, then the Chicago node (which contains the loop), and finally back to the Oakland node. Therefore, the average call setup delay for these three nodes in this case is 0.94/3 or 0.31 seconds per node, which is very consistent with the mean call setup delay of 0.30 seconds measured for the Evans node in the forwarding configuration. Repeating the same calculations for the NL path case yields an average call setup delay of 0.28 seconds per node, also consistent with the results for the other looping paths.

Similarly, using the delay measured between d_i and d_o for each looping path, we can determine the average call teardown delay per node. However, based on the placement of these timestamps within the Radian code, we see that the calculations will be slightly different in this case. The d_i timestamp occurs directly after the destination line process in the Evans node receives a call takedown message from the destination host. After the timestamp is recorded, the line process begins the takedown process, which includes a switch memory modification, and the notification of the subsequent node in the virtual-circuit path. Therefore, the call teardown time for the Evans node is included in the observed delay between d_i and d_o.

Again using the IL path as an example, we see that the observed mean delay between d_i and d_o is 0.60 seconds. This includes the delay to teardown the call through the Evans, Oakland, Chicago, and Oakland (again) nodes, in that order. Since this measurement includes the teardown time for four nodes, the average delay is 0.60/4 or 0.15 seconds per node. Similar calculations on the other looped paths yield average disconnect times of 0.16 seconds for the OL and NL paths, and 0.15 seconds for the CL path.

We make a final observation about the results shown in Figure 5-2. It appears that the network-related delays that we have measured have, in general, very small variance, and are also very small in magnitude compared to the delays encountered at the remote host. We find that execution time of the command *echo* at the remote host (which is roughly equal to the delay seen between C_i and d_i) ranges from 2.9 to 3.3 seconds depending on the remote host, while the network delays within each node are an order of magnitude smaller. These large host-related delays, which often also have large variance, have the potential to make the study of network characteristics very difficult at the user level. In this test, we have taken one approach to avoid this problem by measuring some of the network delays at the network level. In the next experiment, we take a different approach by attempting to factor the host-related delays out of our user-level measurements, so that the network delays can be studied independently.

6. Factoring Remote Host Delays from User-Level Remote Execution Delay

6.1. Methodology

This test was designed to measure the various delays encountered at the destination host during remote executions. We characterize the sources of these delays, and make some observations about how they are affected by various host configurations and utilization levels. In addition, we factor the delays encountered at the remote host from the overall remote execution user delay, thereby enabling us to analyze the network delays independently.

There are two primary sources of delay at the destination host during a remote execution: (1) processing time for the UNIX server process, which we describe shortly (we call this "server time"), and (2) processing time to run the UNIX command(s) that were specified for remote execution (we call this "command time"). The command time can be further factored into two components, the time to create the shell on which the command(s) will execute, and the time to actually perform command execution.

Depending on the particular UNIX command, command execution may actually be performed by a separate process that is created by the shell. In this case, command execution time would include the delay to create this additional process. For our testing, we have chosen to remotely execute the UNIX command *echo* in all cases, which is a shell facility. Therefore, command execution is performed by the shell process itself; no additional processes are necessary.

Our approach was to measure these host-related delays using timestamping code within the UNIX server process that handles Datakit call requests on XUNET hosts. We next describe the timestamped server operation at the destination host. In all cases, the timestamps are implemented within the server code using the UNIX *gettimeofday* system call, which provides 1 ms accuracy.

All Datakit call requests to a particular host are received on one logical channel, known as the "server channel". In the idle state, the server process listens on the server channel for incoming call requests. Each request specifies a separate channel, known as a data channel, on which the call will eventually be established if it passes the security procedure.

When the server process receives a call request, it issues the first of three timestamps. It then performs various security checks on the call request based on access control information stored on the destination host. If the call request passes, the server process returns a "proceed" message to the source host over the data channel and initiates creation of the shell on which the specified commands will execute. At this point, the server process issues the second timestamp. Aside from further timestamping, the duties of the server process are now complete for this particular call. The server processing delay is calculated by subtracting the value of the first timestamp from the second. The result is recorded in a UNIX server log file.

The server process resumes listening on the Datakit server channel for subsequent call requests. After shell creation is completed, the shell executes the *echo* command. Upon completion, the shell process sends a signal to the server process before exiting. This triggers the third timestamp within the server process, from which the command execution time can be calculated by subtracting the value of the second timestamp from the value of the third.

To generate these timestamps for the purposes of our experiment, we executed the same shell script on *vangogh* as in the previous two experiments. A single execution of this script yielded fifty remote executions of the UNIX command *echo* for each of the eight virtual-circuit paths.

The results of each repetition consist of three delay measurements. In addition to the server and command delays measured using the server process timestamps, we record the overall user delay as well, by using timestamps within the *rx* code at the source host, as was done in the Typical User Delays Test.

We processed these raw results by matching the overall user delay of each repetition with the server and command delays for the same repetition. Since the server and command delays are the primary contributors to the remote host delay, we subtract the sum of these two delays from the user delay, theoretically leaving the delay caused only by network factors.

6.2. Results

We begin with a characterization of the mean delays encountered at the destination host during a remote execution. These delays are shown for various destination host configurations in Figure 6-1, and for various host utilization levels in Figure 6-2. We see that, in both cases, the command delay, which is shown in dotted lines, is much larger than the server delay, shown in solid lines.

Command time is composed of two parts - shell creation time and command execution time. We believe that when the UNIX command to be executed is *echo*, the delay incurred when creating the shell is much larger than the actual command execution time.

Rough measurements on a VAX 11/750, using the UNIX *time* measurement facility, support this theory. Delay measurements for local execution of the command *echo* using a previously created shell process yield delays on the order of 0.02 seconds. In the case when shell creation time is included in the measurement, average delay increases to roughly 2.0 seconds.

In Figure 6-1, we see the effects of destination host configuration on command and server delays. Each of the four destination hosts shown is a VAX 11/750, except for the destination host *shark*, which is a higher performance VAX 11/780. Partly because it is a faster machine, the command and server delays are smaller than for the other destination hosts. Mean command delay using *shark* is lower than command delay for *monet*, the host yielding the next lowest average delay, by 0.422 seconds, or a 21% margin. Although this is a significant margin, we cannot attribute it solely to the differences in host configuration. We assume that differences in host utilization

FIGURE 6-1. MEAN REMOTE HOST DELAY (EFFECTS OF HOST CONFIGURATION)

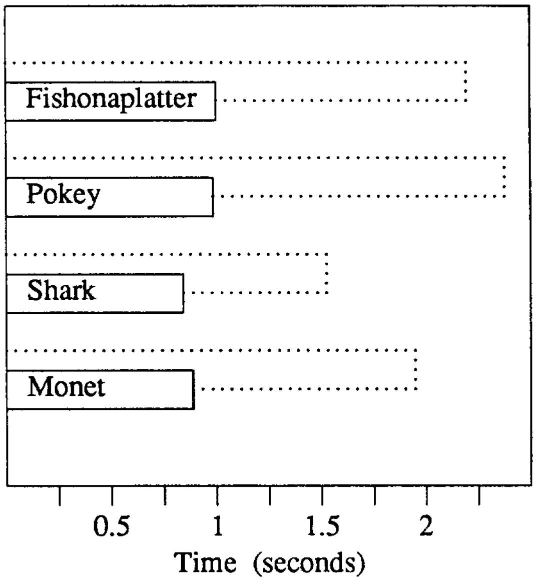
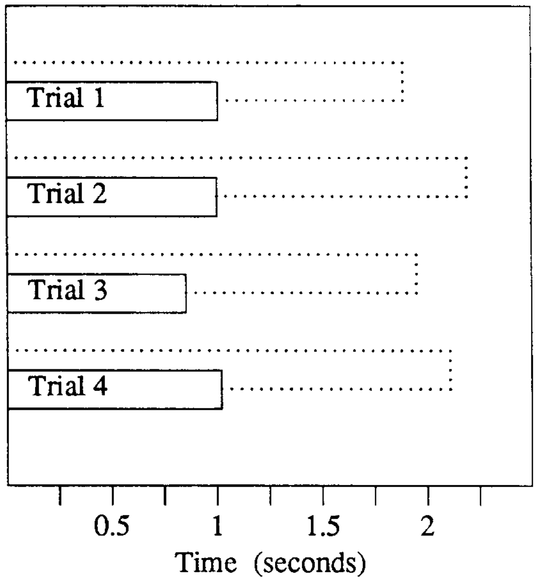


FIGURE 6-2. MEAN REMOTE HOST DELAY (EFFECTS OF HOST LOAD)



levels during testing also played a role, as evidenced by the differences in command delay among the other hosts, even though they are all VAX 11/750's.

Variations in destination host utilization level can also have a significant impact on the observed command and server delays. We see the effects of fluctuating destination host loads in Figure 6-2. Each of the four trials shown was performed using *monet* as the destination host. The only difference between the various trials is the load level on *monet*. The mean command execution delay varies by as much as 0.301 seconds, or 14%, among these paths.

These disparities in mean delay, caused by host load fluctuations and differing destination host configurations, emphasize the importance of measuring host-related delays independently during remote execution testing. The host-related delays can then be factored from the user delay, leaving consistent network delay results.

A last observation based on Figures 6-1 and 6-2 is that the server process does not experience the same degree of variation in processing time as the command process. The mean *server* processing time varies by only 0.164 seconds, or 16% among all destination host configurations and load levels tested, while the mean command delay varies by as much as 0.853 seconds, or 36%. We believe this is so because the command process, which includes shell creation, is a more CPU intensive process than the server process.

Figure 6-3 shows the portion of remote execution delay that is caused by network factors, for each virtual-circuit path. We calculated these delays by subtracting the remote host delay results from the user delay, as previously discussed. We see from Figure 6-3 that the network delays for the virtual-circuit paths having identical node counts are essentially constant. In addition, the total network delay increases as a roughly linear function of node count. To calculate the average call processing delay per node for each virtual-circuit path, we divide the network delay measured for each path by the number of nodes in that path to obtain an average delay per node. The results are shown in Figure 6-4. The delay values are connected only to identify trends.

We see that these results are consistent with the network level call processing delays that we measured using timestamping code within the Radian software (the delays measured this way are represented with a dotted line in Figure 6-4). Specifically, the average call processing delay per

FIGURE 6-3. MEAN NETWORK DELAYS (SECONDS)

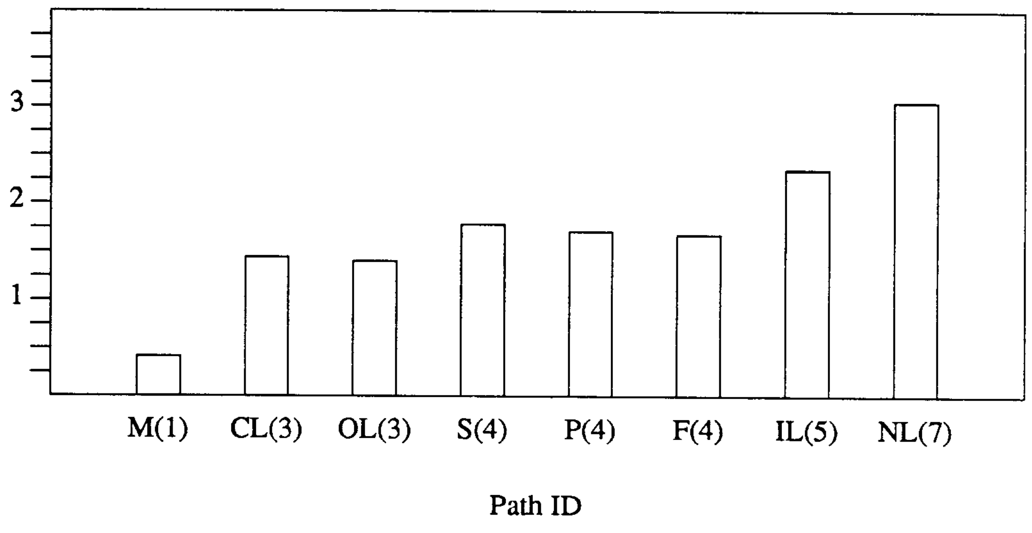
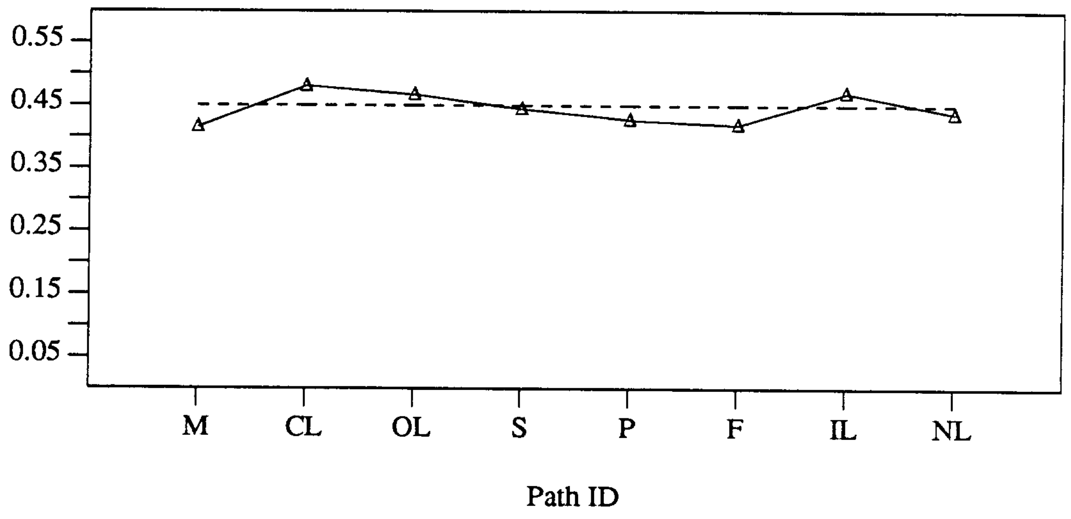


FIGURE 6-4. AVERAGE NETWORK DELAY PER NODE (SECONDS)



node is about 0.45 seconds, and it is independent of the particular virtual circuit path. We believe that the small deviations between the two delay results are caused by measurement errors as well as load fluctuations on the destination host, which can make the measurement of command execution time inaccurate.

7. Conclusion

The user delay for remote executions across a wide area Datakit network is composed of host and network-related factors. In mathematical terms:

$$T_{rexec} = T_{net} + T_{host}$$

where T_{rexec} represents the remote execution user delay. We have determined that the host related delay factors are often much larger in magnitude, and more difficult to quantify than the network factors. Fluctuating host loads during testing, and differences in destination host configuration cause inconsistencies in command execution time among different virtual-circuit paths. These inconsistencies can mask the true delay characteristics of the network. One way to avoid this problem is to measure the delays incurred at the remote host independently while also measuring the overall user delay. Using this method, the remote host delays can be factored from the user delay, leaving consistent network delay results.

Delays incurred at the destination host during remote execution are caused by UNIX server processing time, and execution time to run the specified command(s).

$$T_{host} = T_{server} + T_{cmd}$$

For low-execution delay commands (such as *echo*), command time, T_{cmd}, is composed primarily of shell creation delay, which can take up to two seconds on a VAX 11/750. Mean server delay on a VAX 11/750 takes roughly one second, depending on host utilization at the time. We found server delay to be less sensitive to changes in host load and configuration than the command delay.

The results of our experiments show that the network-related delay, T_{net}, is a linear function of the node count in the virtual circuit path:

$$T_{net} = (T_{cp})N$$

where N is the number of nodes in the virtual circuit path, and T_{cp} represents the average call processing delay per node. Although we do not differentiate in this formula between nodes in the *forwarding* and *directly-connected* configurations, further investigation is necessary to fully characterize the effects of host load fluctuations on the call-processing delay for *directly-connected* nodes.

The call processing delay, T_{cp}, can be further factored into delays incurred during call set up, and those incurred during call teardown:

$$T_{cp} = T_{setup} + T_{teardown}$$

The results of the Radian call processing experiment show that mean call processing time during virtual circuit setup (T_{setup}) is about 0.30 seconds per node, and mean call processing time during call teardown ($T_{teardown}$) is about 0.15 seconds per node, for a total mean call processing delay of 0.45 seconds per node during the course of a single virtual circuit session. Combining the above equations into a single formula that describes remote executions over XUNET, we get:

$$T_{rexec} = (T_{setup} + T_{takedown})N + T_{server} + T_{cmd}$$

This formula has a number of implications to the design of distributed applications over Datakit networks. We briefly mention one of these implications. Because the network call-processing delay is linearly dependent on the node count in the virtual circuit path, a restriction on the maximum network diameter may be necessary for Datakit networks that are supporting distributed applications that require a given level of call-processing performance. We might envision the scenario in which applications designers must predict the expected maximum diameter of a given Datakit network in order to calculate the minimum call-processing performance that can be expected from that network in the future. This implication adds another complication to the design of distributed applications over Datakit.

Based on our formula, many people may argue that because a delay penalty is incurred during call processing in virtual-circuit networks, datagram networks are more suitable for distributed applications. We do not support this claim because the overall effect of the performance penalty on distributed applications can be minimized by proper design of the application. Applications Designers should make tradeoffs between the delays incurred by setting up and taking down virtual circuits as necessary, and leaving them open, despite the fact that they consume network bandwidth. If these tradeoffs are handled properly, distributed applications can enjoy the benefits of virtual circuit technology, while minimizing the performance penalties.

Besides deriving this formula, and the average values of its variables, we have learned a number of other valuable lessons during the course of this work, many of which may apply to the wider environment of network testing in general. We found that it is difficult to accurately measure network call processing performance within a distributed system by utilizing only user-level measurements. This is especially true if the extraneous load on the hosts used for testing cannot be absolutely controlled while testing is taking place. The use of faster, more powerful hosts may improve the accuracy of the results, but it appears that, to obtain the most consistent network call processing results, measurements at the network level are necessary.

The logistical problems of performing call processing tests on a wide-area network without access to centralized administration have also become painfully apparent during the course of our testing. First, we were unable to install the Radian software modifications on all XUNET nodes simply because we did not have the authority to perform the necessary rebooting on the majority of the nodes. We were forced to settle for installation of the modified Radian software on only three of the seven XUNET nodes.

Second, the call processing performance degradation caused by the output of diagnostic information on some XUNET consoles would have been impossible to diagnose without at least indirect access to all of the XUNET consoles. These two logistical difficulties limited the scope of our test results, and caused valuable time to be wasted.

The final lesson learned during the course of this project is the value of automating network tests. Before we corrected the console I/O problem, many test results had already been collected. Because this problem corrupted the results of our original tests, we were forced to re-execute many of the tests after the problem was remedied. Automating the raw data collection and data processing procedures saved us a tremendous amount of time during the re-execution phase of testing.

8. Acknowledgements

A number of people have helped make this project a reality. First, I thank Sandy Fraser of AT&T Bell Laboratories, who has been instrumental in establishing XUNET, to the benefit of graduate students across the country. I also thank my advisor, Domenico Ferrari for providing encouragement, support, and the freedom to pursue my own Datakit research interests.

Riccardo Gusella deserves tremendous thanks for providing the benefit of his guidance, advice and support throughout the course of this project. Only a fraction of this work would have been accomplished without his help. I also thank Caryl Carr for her help regarding XUNET administrative and technical concerns. In addition, Bill Marshall and Dave Presotto answered many of my questions regarding Datakit and Radian operation. Finally, I thank Tom VandeWater, a fellow graduate student doing research on XUNET, for his help and support.

REFERENCES

1. A. V. Aho, B. W. Kernighan and P. J. Weinberger, *The AWK Programming Language*, Addison Wesley, Reading, MA, 1987.
2. J. L. Bentley and B. W. Kernighan, GRAP–A Language for Typesetting Graphs, *Communications of the ACM 29*, 8 (Aug. 1986), 782-792.
3. S. J. Boies, User Behavior on an Interactive Computer System, *IBM Systems Journal 13* (Jan. 1974), 2-18.
4. *UNIX User's Manual - Reference Guide*, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, Apr. 1986.
5. A. G. Fraser, Towards a Universal Data Transport System, *IEEE Journal on Selected Areas in Communications SAC-1*, 5 (Nov. 1983), 803-816.
6. A. G. Fraser and W. T. Marshall, Data Transport in a Byte Stream Network, Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ, Apr. 28, 1987.
7. S. J. Hanson, R. E. Kraut and J. M. Farber, Interface Design and Multivariate Analysis of UNIX Command Use, *ACM Transactions on Office Information Systems 2*, 1 (Mar. 1984), 42-57.
8. R. K. Hanson, A Characterization of the Use of the UNIX C Shell, Report No. UCB/CSD 86/274, Computer Science Division, University of California, Berkeley, Dec. 1985.
9. W. T. Marshall, private communication, Aug. 1988.
10. L. E. McMahon, TDK Principles of Operation, Computing Science Technical Report, AT&T Bell Laboratories, Murray Hill, NJ , (in preparation).
11. L. E. McMahon and W. T. Marshall, XUNET Programmer's Manual, Computing Science Technical Report, AT&T Bell Laboratories, Murray Hill, NJ , (in preparation).
12. *Network File System Protocol*, SUN Microsystems, Protocol Specifications, 1984.

Appendix A - The Radian Network Controller

Radian (Research And Development Interactive Network) is a special purpose operating system that runs on the Datakit network controller, a DEC PDP-11/23. Radian functionality can be classified into two broad categories: call processing, which includes virtual circuit setup and takedown for all devices attached to the network, and operations, administration, and maintenance (OA&M) chores, which include call logging, maintaining network configuration information, and monitoring network hardware failures. We concentrate on Radian call processing functions.

The Radian discussion is organized as follows: in Section 1, we discuss the Radian design philosophy. In section 2, we discuss the relevant aspects of the Radian implementation. For those readers desiring more detail than is provided in these two sections, Appendix B contains a highly detailed chronology of the Radian operations that occur during a single virtual circuit session.

1. The Radian Design Philosophy

Many of the Radian design goals follow naturally from the goals of the Datakit network as a whole. We first examine some important Datakit design goals, and then discuss how they have influenced the subsequent Radian design.

One of the primary Datakit design goals is that it be highly modular in nature [5]. There are two aspects to this modularity requirement. First, the network should be modular on a short term, day-to-day basis. Reconfiguring or expanding the network to accommodate additional users should ideally be fast, convenient and, in most cases, inexpensive. Experience has shown that, if networks are to efficiently evolve to meet the ever-changing demands placed on them, this feature is essential.

The second aspect of Datakit modularity is modularity from a long term perspective. As new devices requiring data communications services are introduced to the marketplace, the Datakit network should ideally be able to accommodate those new devices without mandating a complete redesign of the network (with all of the associated hassles and expense). By meeting this goal, the Datakit technology will maintain interoperability with the latest data communications market offerings.

To satisfy this two-pronged modularity requirement, Radian utilizes a process per device software architecture. Each communicating device attached to the network is assigned its own distinct Radian process in the network controller with which the device communicates to accomplish call processing functions. The flexibility of the Radian architecture lies in the fact that various device attachment configurations can be accommodated simply by the creation or removal of the appropriate Radian processes. In addition, when new device types are introduced in the data communications market, new Radian process types can be developed to enable Datakit networks to easily accommodate these new devices.

We should clarify the concept of a communicating device. The single most important characteristic of a communicating device is that it utilizes a single virtual circuit data stream for its communication needs. Examples include: a terminal attached to Datakit through a single RS-232 interface, or a host process utilizing a single logical data channel in a multiplexed host interface. Subsequently, we group all types of communicating devices together under the term *terminal*. We next describe the Radian implementation that enables these terminals to communicate using the Datakit hardware.

2. The Radian Implementation

The Radian operating system is composed of a set of independent, but communicating processes that control the state of the Datakit switch to effect call processing. Radian processes receive call processing requests and other supervisory information from terminals attached to the network. Each terminal is assigned its own separate Radian process with which it communicates this information. Each of these Radian processes can in turn communicate with other Radian processes to determine whether it is appropriate to change the state of the Datakit switch.

The Radian processes to which terminals communicate are called *line processes.* A single line process instance is created for each attached terminal. Different terminal types communicate with different line process types and multiple instances of a single terminal type require multiple instances of the appropriate line process type [10]. The goal is to ensure that each terminal has its own distinct Radian process of the appropriate type with which it is able to communicate supervisory information.

2.1. Line Processes

A terminal communicates with its line process by using a private protocol that may not be known to other Radian line process types. Because the protocol is private, different terminal types can use protocols that are tailored to their specific needs. The job of each line process is to translate the tailored protocol that it uses when communicating with its terminal into a universally known protocol through which it is able to communicate with all other Radian line processes.

Through its private protocol, a terminal communicates signalling and supervisory information to its assigned line process. Based on this information, the line process initiates conversations with other line processes to decide what action to take as a result of receiving this information. The line processes may decide that the state of the switch needs to be changed, and, if so, the change is implemented in the switch memory.

During virtual circuit set up procedures, two line processes are involved in the conversation, one associated with the terminal requesting the call set up (the source), and one associated with the terminal to which the call will connect (the destination). The protocol used between line processes in this situation is called the Virtual Line Protocol.

2.1.1. The Virtual Line Protocol

The Virtual Line Protocol (VLP) is the means by which line processes can communicate with one another. It is universally known among all line processes, and acts as the "glue" by which the line processes of different terminal types are able to communicate. The VLP is implemented by use of system calls which move message data (often only a few bytes) from the user data area of a source line process to the user data area of a destination line process.

One interesting feature of VLP is the use of conversation identification numbers, called *cparams*. A conversation identification number is included with every VLP message. It is used to identify a particular conversation with another line process. The originator of a VLP conversation chooses the number, by picking one which will uniquely identify that particular VLP conversation. Any subsequent messages during that conversation will carry the same cparam number as a message argument. This method assures that line processes will avoid confusing leftover messages from previous conversations with messages associated with a conversation that has just started [10].

2.1.2. UNIX CSC Line Processes

VLP conversations are often started by a line process after it receives a call processing request from its terminal. There are a variety of ways for terminals to communicate these requests to the appropriate Radian line processes within the network controller. Terminals attached to Datakit through RS-232 lines need only very simple supervisory messages to communicate their call requests to Radian. These messages usually can be transmitted simply by asserting the appropriate RS-232 leads. On the other hand, UNIX hosts connected to Datakit use a much wider, more complex set of messages to communicate call processing requests, as well as perform other functions.

A separate logical channel in each multiplexed host interface that connects a UNIX host to Radian is reserved for supervisory messages of this type. This channel is known as the Common Supervisory Channel (CSC); supervisory messages pertaining to any logical data channel on that particular interface are carried on this channel. To distinguish supervisory messages that pertain to different logical data channels on the same interface, each message carried on the CSC contains an argument which specifies the data channel to which it pertains.

The reason a separate channel on each host interface is dedicated to the transmission of these supervisory messages is to avoid the collision of user and supervisory data on the same channel. Suppose a virtual circuit has been previously established on one of the logical data channels in a multiplexed host interface. Suppose now that the host wishes to close that circuit. The host cannot communicate that request to Radian over the data channel carrying the virtual circuit, because it may be confused with the user data being carried on that channel. There must be some way for Radian to distinguish between user and supervisory data. The Radian designers have chosen to accomplish this by transmitting supervisory messages on a separate logical channel, the CSC.

The Radian line process associated with the CSC is called the CSC line process. This process forwards supervisory messages from the data channels on that interface to the UNIX host over the supervisory channel. It also receives supervisory messages from the host on the CSC channel, and forwards them to the appropriate data channel[3].

Besides forwarding supervisory information, the UNIX CSC processes perform another important function. Each CSC process monitors the status of the host to which it is attached by receiving "keep-alive" messages from the host at periodic intervals (usually every 5 seconds). Every 30 seconds, the UNIX CSC process in the controller checks if it has received any "keep-alives" during the last thirty seconds. If not, the CSC process tears down any active virtual circuits on that module (since they connect to a dead host, and may be consuming valuable bandwidth elsewhere in the network) [10]. Next, the CSC process sets a network alarm indicating that the host is down. The alarm is reported on the network console, allowing the Datakit administrator to easily keep track of the status of all UNIX hosts on the network.

Both UNIX CSC and data channel line processes use URP (the Universal Receiver Protocol) Grade of Service 3[4] for exchanging supervisory messages with an attached UNIX host[5] [11]. This provides an added measure of reliability to the call set up process since messages that have been corrupted in transit are detected and discarded. This is in contrast with the trunk CSC and trunk

[3] The forwarding of these messages from CSC to data channel line process is private and does not fall under the domain of the VLP protocol previously discussed.

[4] URP Grade of Service 3 (GOS #3) provides error detection without retransmission and without flow control [6]. Flow control is not necessary in this case since messages are of short, fixed length. Retransmission is not necessary since a timeout mechanism is provided at the user level.

[5] It will become evident shortly that some limited supervisory information is transmitted over the data channel in addition to that transmitted on the CSC channel.

data channel line processes which do not implement URP. These processes will be discussed next.

2.1.3. Trunk CSC Line Processes

During call set up, VLP messages are exchanged between the line processes associated with the source and destination terminals. Consider the situation in which the source and destination terminals are not attached to the same Datakit node. It is necessary to define a mechanism so that source and destination line processes can transparently communicate VLP messages between adjacent nodes. As in the UNIX CSC channel case, it is important to be able to transmit these messages without contention from user data on a virtual circuit.

The adopted solution is to use a CSC channel on every multiplexed trunk that connects adjacent Datakit nodes. This channel carries VLP messages originating at any data channel line process on either end of that trunk. At each end of the CSC channel is a trunk CSC line process which communicates with the partner CSC line process located on the same channel at the remote end. If a data channel line process wishes to communicate supervisory information to the data channel line process at the remote end, it sends the information in the form of a private message to the CSC process at the local end of the trunk. The CSC process forwards the information to the CSC process at the remote end of the trunk. From there, the message is again forwarded to the appropriate data channel line process (the channel number is specified in the message received by the remote CSC process, a method similar to that used in the UNIX CSC case).

Besides forwarding supervisory information between nodes, each trunk CSC process also monitors the status of the node at the remote end of its trunk. It does so by exchanging keep alive messages periodically with the remote trunk CSC process. If a trunk CSC does not receive three consecutive keep alives, it sets a network alarm so that the Datakit network administrator can monitor the status of the network (this feature is similar to the keep alive messages transmitted on the UNIX CSC)[11].

2.2. Operations, Administration, and Maintenance Processes

The job of the OA&M processes is wide ranging and varied. Administrative processes are used for network console I/O, alarm reporting, and record keeping chores in addition to many others. However, we are primarily concerned with only one function of these processes. That is the job of maintaining network configuration information. This information ensures that the correct type of Radian line process is created to communicate with each connected and active terminal on the network.

The configuration information is stored internally in a network configuration table which is updated when configuration information is entered by the Datakit administrator at the network console. Within certain constraints, the addition of a new device to the network can be accomplished simply by entering a few commands at the network console, and physically attaching the device to the network.

2.3. Switch Model

Each Radian line process exchanges supervisory information with its associated terminal and with other line processes to decide whether it is appropriate to change the state of the Datakit switch to effect virtual circuit set up or teardown. An abstract model of this switch exists at the Radian user level, which is implemented by system level routines. All of the necessary switch state changes can be implemented on this abstract model through use of system calls.

Each terminal has its own separate switch model, which specifies connections pertaining only to that particular terminal. The model also only applies to non-supervisory data [10]. Supervisory data from each terminal is permanently routed to its associated line process.

Each terminal's switch model consists of three poles. A pole corresponds to a place where data can both originate and terminate. The state of the switch model is determined by the connection pattern among these three poles. The poles are listed below along with a brief description of each.

Terminal:	The communicating device connected to that channel.
Process:	The line process for that channel.
Switch:	A meeting point. Separate channels specifying the same meeting point will be logically connected.

The switching element is always in one of four possible states. The connection pattern among the three data termination points is shown below in Figure A-1 for each of the four possible states.

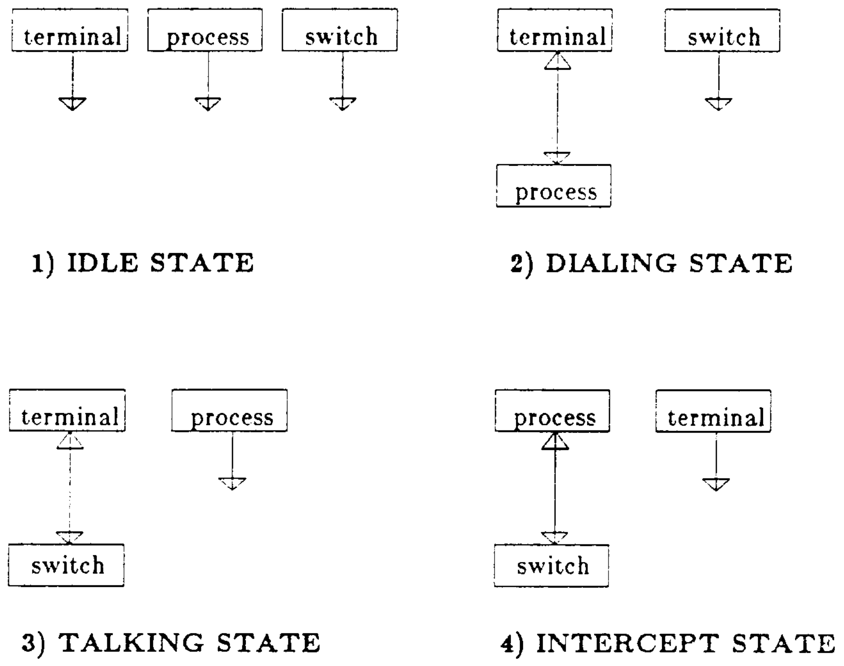


FIGURE A-1. THE FOUR SWITCH MODEL STATES

When a channel is in the **idle** state, non-supervisory data originating from all three sources is discarded. The switch is kept in this state when the data channel is not being used to carry a virtual circuit. During a virtual circuit set up, the switch is put in the **dialing** state, so that the associated terminal can communicate dialstring information to its line process directly over the data channel. After the virtual circuit is established, the switch model is put in the **talking** state, and the line process is effectively removed from the path through which the user data travels. The data travels directly from the terminal to the switch meeting point (and then to a destination terminal or line process, if another channel specifies the same meeting point, and sets its switch model to the talking state). In this configuration, the switch is often said to be in the "cut-through" state because the line process has been cut out of the user data path[6]. The final switch model state is known as the **intercept** state. It is included here for completeness; we do not encounter this state in our virtual circuit session example. It is used when a line process wishes to communicate directly with a terminal other than its own.

[6] Because the line process is removed from the user data path in this state, a Datakit network controller can theoretically be rebooted without any effect on existing virtual circuits through the node.

Appendix B: A Chronology of Radian Operations during a Virtual-Circuit Session

The following is a step by step description of Radian procedures that occur during a remote execution between a source and destination host that are located on separate Datakit nodes connected by a multiplexed trunk. We assume in this description that all interprocess messages arrive at their intended destinations without error.

In Radian, different message formats are used when communicating between different classes of line processes. In keeping within this framework, each message in the following description is prefixed with a single letter which indicates the format of the message [11]. Each message is referenced by one of these single letter codes, followed by a single word message name chosen to convey the purpose of the message. The various single letter codes are shown below, along with a brief description of each.

E_:	A message exchanged between a terminal attached to the network and its associated line process.
P_:	A private message exchanged between processes which work together to support a single network interface. For example, a supervisory message between a CSC process and a data channel process on the same trunk. These messages are not known to other types of line processes.
V_:	A VLP message, which is known system-wide. These messages provide the glue by which all line processes communicate.

[1] A user level process on the source host wishes to make a call on the Datakit network. Using the Common Supervisory Channel, the source host sends an URP call request message, E_DIAL, to the Radian line process assigned to that channel (we will call this process *scsc*: see Figure B-1 for explanations of all the line process abbreviations). The call request specifies which channel number the source host wishes to call out on as well as suggested window sizes[7] for the source and destination hosts.

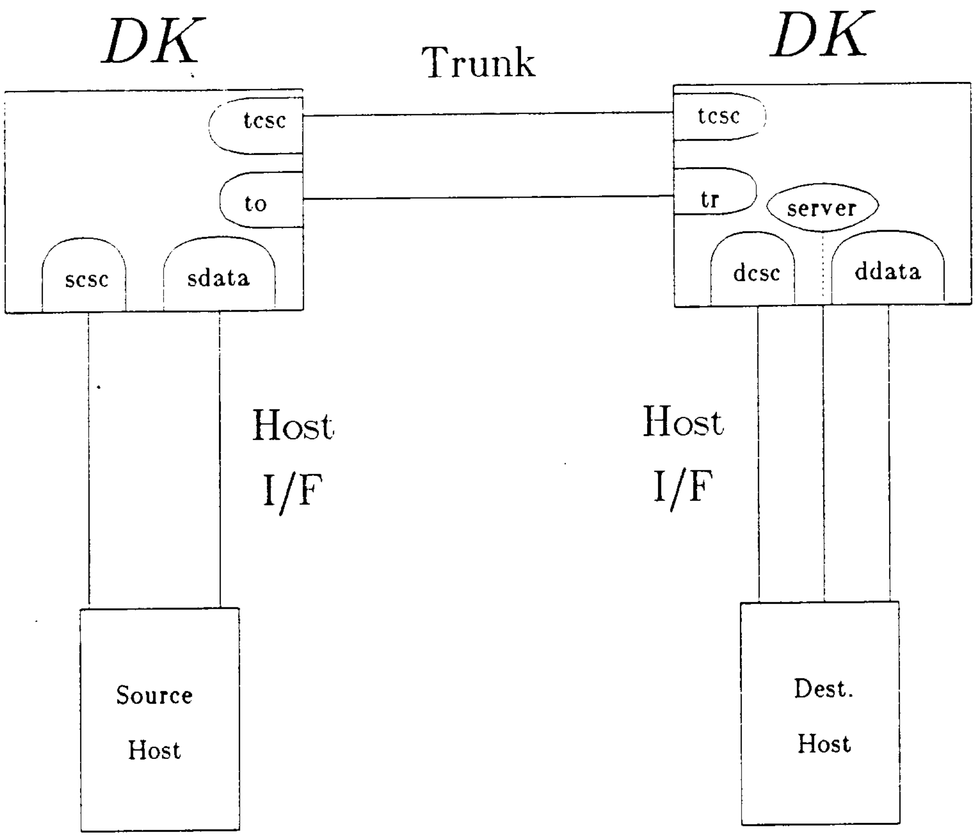
[2] Scsc translates the channel number into an internal IPC address for the line process assigned to that channel (call this process *sdata*). Using this internal address, scsc issues a P_OFFHOOK message to sdata to alert it that the host wishes to establish a call. The message contains window size suggestions for the call.

[3] Sdata receives the P_OFFHOOK message and immediately sets its channel to busy, so that no other line process will attempt to connect a call through that channel. The switch is set to the Dialing state, and an URP initialization request is sent to the source host to establish an URP conversation [5]. Since the switch is in the Dialing state, sdata will communicate directly with the source host over this URP data channel.

While waiting for the host's response to the URP initialization request, sdata retrieves the value of the maximum window size for its module type from an internal database. The source host window size is set to the minimum of the value suggested by the source host in the P_OFFHOOK message, and the value retrieved from the database.

[4] The URP conversation is initialized, and sdata receives dialstring information from the source host. The ascii dialstring is translated into an IPC destination line process address on that particular node, and sdata issues a VLP V_CALL message to that line process (call the process torig). After this message is successfully delivered, a timestamp is generated which

[7] Windowing is the means by which flow control is implemented in the URP transport protocol.



scsc:	the source host UNIX CSC line process
sdata:	the source host data channel line process
tocsc:	the trunk originator CSC line process
trcsc:	the trunk receiver CSC line process
torig:	the trunk originator data channel line process
trecv:	the trunk receiver data channel line process
dcsc:	the destination host UNIX CSC line process
ddata:	the destination host data channel line process
server:	the destination host server line process

FIGURE B-1. LINE PROCESS ARRANGEMENT

indicates that a connection request has been received by this node.

Sdata now sets the switch to the Talking state. The destination is specified to be the channel associated with the torig process. However, the switch memory is not written at this time. Only one half of the virtual circuit is established. In reference to the switch model discussed in Section A.2.3, we can say that, at this point, the terminal has been connected to a "meeting point". Another line process must also connect its terminal to the same meeting point in order to complete a connection between two channels. So, before the memory can be written, the torig process must set its switch model to the Talking state as well, while specifying the channel associated with the sdata process as the destination [9].

[5] The torig process receives the V_CALL request complete with with the full dialstring. It immediately sets its channel busy and performs window size reduction similar to that performed by sdata earlier in the call set up. The V_CALL message is forwarded to the local trunk CSC process (call it tocsc) in the form of a private P_OPEN message which also includes the source channel number and windowsize values. The switch model is set to the Dialing state and to waits indefinitely for a response to the P_OPEN that it just issued. Since the switch model is in the Dialing state, the line process can listen on the trunk data channel for the response.

[6] The tocsc process receives the P_OPEN message from the to process. It immediately forwards the message to the partner trunk CSC process at the remote end of the CSC channel (call it trcsc), this time in the form of an E_INCALL message.

[7] Trcsc process receives the E_INCALL message from the tocsc. The destination data channel number is specified as part of the message. The trcsc forwards the message to the line process associated with this channel (call it trecv) in the form of a P_INCALL message which includes window size information.

[8] The trecv process receives the P_INCALL message and sets its channel to busy. It performs the window size reduction, and sets the switch to the Dialing state. An "O" is sent over the data channel to the torig process that originally sent the P_OPEN and is currently waiting for a response. This "O" acts as a dialtone to notify torig that the connection has been successfully established between the two ends of the trunk and that the trecv process is waiting for the dialstring. In case the first "O" is not correctly received, trecv will timeout in five seconds and send another "O" until it receives the dialstring from torig.

[9] When the torig process receives the "O", it sends the ascii dialstring over the data channel. The switch model is then set to the Talking state with the destination specified to be the sdata process. At this time, both the source and destination line processes (sdata and torig, respectively) have set their switch models to the Talking state, and each has specified the other as the destination. (Recall that the sdata process set the switch to the Talking state after issuing the VLP V_CALL message to torig.) The switch memory is written at this time to complete the virtual circuit through this node. A timestamp is generated at this point indicating that a connection has been established through the node.

[10] The trecv process receives the ascii dialstring, and translates it into a destination line process IPC address on the same node. It then issues a VLP V_CALL message (containing the dialstring) to the destination line process (in this case, ddata). A call request timestamp is recorded within this node's log after the V_CALL message is issued.

[11] The ddata process receives the V_CALL request, sets its channel to busy, and performs the window size negotiation for the link. The dialstring is translated into a destination line process address (in this case, call it the server process). Ddata then sends a P_INCALL message to the server process, notifying it that it wishes to set up a call with the host. The P_INCALL message contains the dialstring and the channel number on which the call to the host should be established. The ddata process then sets the switch to the Talking state, specifying trecv as the destination. Since trecv already has set its switch to the Talking state, and has specified ddata as the destination, the switch memory is written and a timestamp of the connection is generated at this time.

[12] The server process receives the P_INCALL message and forwards it to the destination host in the form of an E_INCALL message containing: the channel number of ddata, the lower 16 bits of the controller clock at the time the message is sent, the window size information,

and the dialstring.

[13] After the host processes the message, it sends an E_ACK message back to the server which contains: the channel number on which the virtual circuit has just been established (it should be the same as the channel number requested in the E_INCALL message), the lower sixteen bits of the controller clock that were received in the E_INCALL message, and a reason code. For a successful call, the reason code will be zero.

[14] The server process, upon receiving the E_ACK, forwards a P_ANSWER message to the ddata process. This ANSWER is an acknowledgement that the call setup was successful. The message is propagated back through all of the involved line processes in both nodes (over the CSC channels in the case of the trunk, and the host interfaces). Each data channel line process, upon receiving the ANSWER message, goes to a dormant state in which the process is Idle until it receives another supervisory or administrative message. Note that each line process is no longer in the user data path, because the switch is in the Talking or "cut-through" state for each process. **The call is now established.**

When the remote execution session is complete, the virtual circuit is gracefully closed by the following sequence of events:

[15] The dcsc receives an E_CLOSE request from the destination host[8]. The message is forwarded to ddata in the form of a P_ONHOOK message.

[16] Ddata receives the P_ONHOOK, and sets the switch to Idle state which generates a disconnect timestamp [9]. It then sends a V_HANGUP to trecv, and an P_CLOSE to dcsc (which acts as an acknowledgement of the P_ONHOOK message). Lastly, it sets its channel to available so that future virtual circuits can now use that channel.

[17] Dcsc receives the P_CLOSE from ddata, and writes an E_ISCLOSED acknowledgement to the destination host.

[18] Trdata receives the V_HANGUP and the switch is set to the Idle state. Also, a P_CLOSE message is sent to the trcsc process. Every 15 seconds another P_CLOSE is sent to the trcsc until an P_ONHOOK message arrives from trcsc confirming the virtual circuit takedown on that link. Only then is trecv ready to accept another call set up.

[19] Upon receiving the P_CLOSE, the trcsc sends a P_ONHOOK to trecv (which acts as an acknowledgement of reception of the P_CLOSE), and an E_CLOSE to tocsc.

[20] Upon receiving the E_CLOSE, the tocsc acknowledges it with an E_ISCLOSED and sends a P_ONHOOK to torig.

[21] Todata receives the P_ONHOOK, and sends V_HANGUP to sdata. The switch is then set to the Idle state which generates another disconnect timestamp - this time for the node on which to is executing.

[22] Sdata receives the V_HANGUP, the switch model is set to Idle state, and a P_HANGUP is sent to scsc every five seconds until it is acknowledged with a P_ONHOOK.

[23] Scsc receives the V_HANGUP, and sends an E_CLOSE to the source host. The E_CLOSE is acknowledged with an E_CLOSE request from the host, at which point the scsc sends the P_ONHOOK to sdata, allowing sdata to become ready for the next call.

[8] Normally, the destination host initiates call takedown after remote command execution is completed. However, if the user at the source host aborts remote command execution before completion, the source host initiates call takedown in a similar fashion.

[24] On receipt of the P_ONHOOK, sdata sends a P_CLOSE to scsc.

[25] Scsc receives a P_CLOSE acknowledgement from sdata at which point a E_ISCLOSED is sent to the source host. The call is now completely torndown. All line processes are in the same state as before the call was established.

Appendix C - Selected Data in Tabular Form

Table containing data used in Figures 3-1 and 3-2:

Command	Path ID							
	M		S		P		F	
	Mean	*SD*	*Mean*	*SD*	*Mean*	*SD*	*Mean*	*SD*
ls	4.099	0.391	5.271	0.393	9.201	2.503	6.395	0.414
cd	3.230	0.327	5.197	0.683	8.616	2.092	5.850	0.256
echo	3.180	0.261	5.157	0.487	7.806	0.749	5.759	0.247
cat	3.482	0.341	5.143	0.256	7.913	1.512	6.119	0.770
u	3.507	0.352	5.005	0.418	9.399	1.446	6.473	1.245
finger	5.223	0.270	6.307	0.738	11.933	3.821	7.059	0.434
grep	3.727	0.369	5.511	0.739	9.381	2.590	6.233	0.478
pwd	3.691	0.554	5.147	0.468	9.047	2.288	6.247	0.736
mv	3.908	0.570	5.244	0.397	9.583	2.763	6.963	1.582
cp	3.923	0.749	5.392	0.589	9.565	2.883	6.245	0.396
f	4.617	1.016	5.746	0.548	8.983	1.883	6.999	0.486
who	4.264	1.240	5.408	0.526	8.689	1.715	6.269	0.287
egrep	3.823	1.040	5.123	0.314	8.135	1.523	6.323	0.228
so	4.506	0.707	5.835	0.412	9.243	1.039	7.014	0.375
df	5.153	0.679	6.651	0.507	12.605	2.302	7.364	0.282
date	4.037	0.502	5.307	0.194	9.636	1.235	6.089	0.189
w	5.350	0.626	8.581	1.840	12.127	2.125	7.359	0.280
uptime	4.468	0.415	5.769	0.802	9.690	1.655	6.527	0.195
tail	3.620	0.481	5.303	0.483	8.170	1.531	6.217	0.337
ln	3.777	0.441	5.314	0.931	8.311	1.626	6.220	0.294
rm	3.626	0.280	5.265	0.462	7.592	1.344	6.183	0.274
fgrep	3.886	0.661	5.203	0.520	8.589	1.153	6.307	0.471
chmod	3.642	0.736	5.130	0.564	8.072	1.353	6.094	0.454
mkdir	3.824	0.703	5.233	0.579	8.350	1.595	6.490	0.498
rmdir	3.672	0.577	5.424	1.000	9.050	2.211	6.408	0.853
sleep	4.745	0.595	5.991	0.514	8.938	0.951	7.367	0.620
du	4.481	0.354	5.141	0.381	9.794	2.191	6.757	0.977
wc	3.763	0.393	5.232	0.717	8.108	1.358	6.105	0.213
head	3.541	0.252	5.190	0.548	7.920	1.448	6.510	0.258

Table containing data used in Figure 4-1:

Path ID	User Delay		
	Minimum	*Mean*	*SD*
M	2.820	3.252	0.456
CL	3.503	4.573	1.301
OL	3.648	4.203	1.068
S	3.133	4.147	0.629
P	4.376	5.073	0.487
F	4.410	4.808	0.622
IL	4.566	5.539	1.116
NL	5.194	5.953	0.854

Table containing data used in Figure 5-1:

Path	Radian Operation					
	$c \rightarrow C$			$C \rightarrow d$		
	Min	*Mean*	*SD*	*Min*	*Mean*	*SD*
M	.07	.16	.07	2.30	2.67	.26
S	.23	.31	.07	2.67	3.70	.82
P	.23	.28	.04	3.92	4.63	.69
F	.25	.30	.05	3.88	4.32	.67

Table containing data used in Figure 5-2:

Path	Radian Operation								
	$c_o \rightarrow C_o$			$C_o \rightarrow c_i$			$c_i \rightarrow C_i$		
	Min	*Mean*	*SD*	*Min*	*Mean*	*SD*	*Min*	*Mean*	*SD*
CL	.23	.30	.04	.23	.30	.07	.07	.15	.06
OL	.23	.30	.04	.18	.30	.07	.07	.16	.05
IL	.25	.30	.05	.77	.95	.17	.07	.16	.06
NL	.23	.30	.04	1.23	1.40	.17	.07	.17	.06

Path	Radian Operation					
	$C_i \rightarrow d_i$			$d_i \rightarrow d_o$		
	Min	*Mean*	*SD*	*Min*	*Mean*	*SD*
CL	2.28	3.31	1.32	.25	.30	.04
OL	2.42	2.96	1.22	.23	.31	.06
IL	2.41	3.25	.99	.50	.60	.16
NL	2.55	3.06	.56	.70	.82	.10

Table containing data used in Figures 6-1, 6-2, and 6-3:

Path ID	User Time			Server Time			Total Remote Time		
	Min	*Mean*	*SD*	*Min*	*Mean*	*SD*	*Min*	*Mean*	*SD*
M	2.726	3.103	.254	.734	.869	.115	2.229	2.560	.279
CL	3.503	4.573	1.301	.739	1.022	.379	2.230	3.131	1.260
OL	3.648	4.203	1.068	.738	.853	.239	2.361	2.801	1.041
S	3.133	4.147	.629	.776	.844	.289	1.608	2.370	.607
P	4.376	5.073	.487	.776	.986	.195	2.808	3.365	.475
F	4.410	4.808	.622	.835	1.002	.294	2.870	3.198	.599
IL	4.566	5.539	1.116	.740	1.000	.348	2.364	3.189	1.057
NL	5.194	5.953	.854	.838	1.008	.161	2.469	2.896	.510