Delay and Throughput Measurements of the XUNET Datakit Network

Thomas VandeWater

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Datakit, a virtual-circuit switch developed at AT&T Bell Laboratories, is intended to provide efficient data communication over both local-area and wide-area configurations. In the wide-area case, its switching engine and the queuing disciplines it employs at the trunk interfaces try to balance the opposite requirements of low-delay interactive traffic and high-throughput data-transfer traffic. In order to experiment with distributed applications over Datakit technology, AT&T Bell Laboratories has installed an Experimental University Network (XUNET) that connects computers of the Berkeley Computer Science Division to machines at other universities and to machines at Bell Laboratories locations in New Jersey through 1.5 Mb/s T1 lines and Datakit nodes.

We measured the delay and throughput performance of XUNET. The window size and the small memory available in each host interface were the primary factors affecting the throughput. The queueing delays at each host affected the delays for small messages. We isolated and characterized potential bottlenecks in the network, and other sources of delay, to enable the network's designers to improve its performance. We provided a mathematical model for the delay and throughput characteristics of long-distance Datakit communication. The parameters of the model included protocol window size, source and destination distance, and line and interface speeds.

November 15, 1988

This research was partially supported by the Defense Advanced Research Projects Agency (DARPA) of DOD under Contract N00039-84-C-0089 Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command. 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 of the U.S. Government.

CONTENTS

1. INTRODUCTION 1
   1.1 Experiment Environment 1

2. DELAY MEASUREMENTS 2
   2.1 Software 2
   2.2 Packet sizes 3
   2.3 Determination of repetition count 4
   2.4 Expected results 4
   2.5 Analysis of results 6
   2.6 Delay formula 9
   2.7 Verification of results using different software 10

3. LOADED NETWORK MEASUREMENTS 12
   3.1 Software 12
   3.2 Methodology 12
   3.3 Expected results 13
   3.4 Analysis of results 13
   3.5 Measured throughput for a single process 15
   3.6 Measured throughputs for multiple processes 18
   3.7 Verification of results using different software 20

4. CONCLUSION 23

5. ACKNOWLEDGEMENTS 23

1. INTRODUCTION

Datakit, a virtual-circuit switch developed by AT&T Bell Laboratories, is intended to provide efficient data communication over both local-area and wide-area configurations. In the wide-area case, the switching engine and the queueing disciplines it employs at the trunk interface try to balance between the conflicting requirements of interactive traffic and data-transfer traffic. To experiment with distributed applications over Datakit technology, AT&T Bell Laboratories has installed an Experimental University Network (XUNET) (described in the next section), that connects computers of the Berkeley Computer Science Division to machines at other universities and Bell Laboratories locations in New Jersey through 1.5 Mb/s T1 lines and Datakit nodes. This report describes the results of delay and throughput performance tests performed on this network during the summer of 1988.

The tests consist of user level measurements of the delay and throughput for an unloaded network, the delay of a loaded network, and the network's performance when a broad range of processes generating traffic are run simultaneously. The goal of our tests is to provide a mathematical model for the delay and throughput characteristics of long-distance Datakit communication, to be used by the network's designers to improve its performance. It will be of interest to locate the potential bottlenecks, and other sources of delay, in the network. We anticipate that the window size and the small memory available in each host machine are the primary factors affecting the throughput. This report deals with timing measurements after a circuit is set up, that is, after the connection is made between the source and the destination. For a timing analysis of call set up and performance characterization of remote execution calls, see the report by Ron Arbo[ARBO88].

1.1 Experiment Environment

The Experimental University Network (XUNET) currently connects Bell Labs at Murray Hill, N.J. with the University of Illinois at Urbana-Champaign, the University of Wisconsin at Madison and the University of California at Berkeley. Each university has a small local area network with a Datakit switch at its hub. A wide area network, composed of Datakit switches at Oakland and Chicago connected by a 1.5 Mb/s trunk, forms the backbone of XUNET. The universities are connected to the backbone by 1.5 Mb/s trunks, as depicted in Figure 1[XUNE88].

Machines at all four locations were used to obtain the results presented below. At Berkeley, two machines were used, namely, Vangogh, a VAX 8600, and Monet, a VAX 750. The Wisconsin host, Pokey, and the Bell Labs machine, Fishonaplatter, are also VAX 750's. The Illinois host we began our experiments on was a VAX 750, but, during the course of our testing it was changed to a VAX 780. In the remainder of this report, UIUC will refer to the VAX 750 at Illinois and UIUC* will refer to the VAX 780.

Devices connected to Datakit communicate through interface boards connected to a bus. For example, each T1 trunk is terminated by one of these interface boards. The following is a brief description of the data path through a Datakit node. A VAX host communicates over Datakit through a KMC interface board. The KMC handles all Datakit URP protocol processing [FRAS83], and communicates with the CPU using a very simple protocol. The URP protocol also negotiates the window size between the source and the destination; in our experiments, URP set the window size at 1024 bytes. The CPU notifies the KMC that it has data to send, at that point a DMA transfer is initiated to the KMC's local on-board memory. Next, all protocol processing is performed independent of the CPU's control, and the data is transmitted. When data arrives from a T1 line to a Datakit interface, it is queued until that interface can gain access to the backplane. Once the data can access the backplane, it transmits the data onto the bus to the appropriate destination interface within the same Datakit node. For a more detailed description of Datakit, see [FRAS83].

Configuration of XUNET

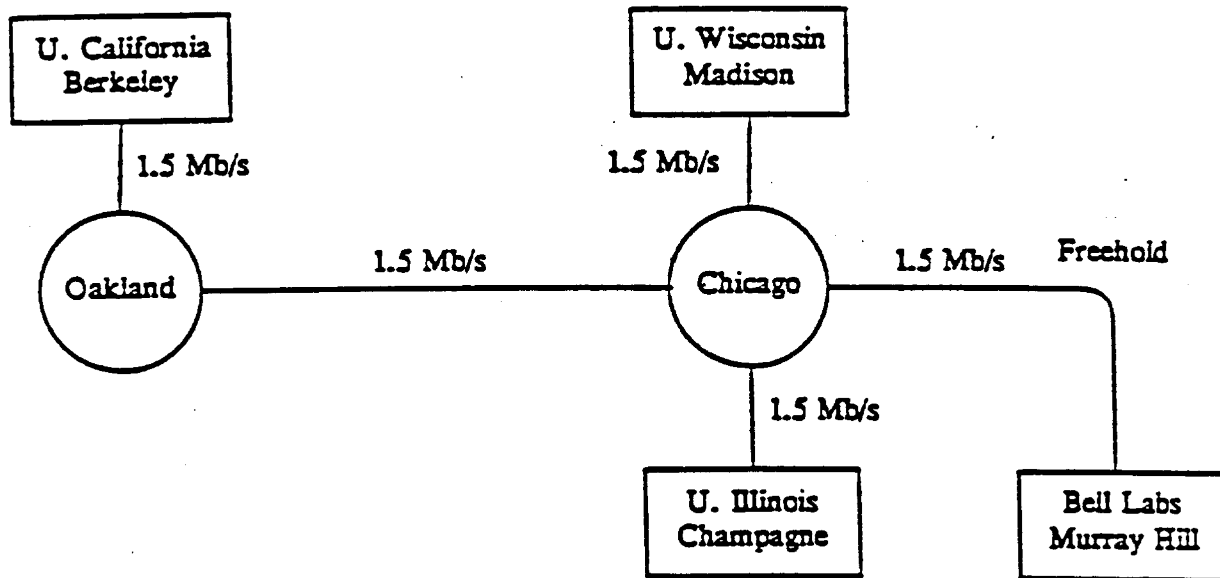

Figure 1[1]

2. DELAY MEASUREMENTS

This chapter describes the first of several tests designed to determine the performance of the XUNET Datakit network. The goal of this experiment is to formulate a mathematical model of packet delays at low load. The evolution of the experiment is followed, from its design to the results, and finally a discussion of those results.

2.1 Software

The software used to conduct this experiment consists of two programs, a packet generating program and a packet receiving program. The packet receiving program, dkbenchs (Datakit benchmark server), simply reads the packets from the network and writes them back to their originator. It continues until it reads an 'end of transmission' (EOT) in the first byte of an incoming packet. The behavior of the

1. Courtesy of XUNET planning meeting, Chicago, Illinois, January 1988.

packet receiving program can be described in pseudo Pascal code as follows:

```
repeat
        read packet from network
        write packet back to sender
until packet[1] = EOT
```

The packet generating program, dkbench (Datakit benchmark), works as follows: it sets up a connection to the specified destination and sends a predetermined number of packets of constant and given size. The program reads the clock, and immediately writes the packet to the destination. It reads the clock again after it has read the same packet from the network. The difference between the two clock readings is the round trip time (RTT). The behavior of our packet generating program can be described in pseudo Pascal code as follows:

```
N = number of repetitions
count = 0
while count < N do
begin
        read the clock
        write 'packet' to destination
        read 'packet' from network
        read the clock again
        round trip time = second clock reading - first clock reading
        count = count + 1
end
```

Note that only one packet is outstanding on the network at any time. Thus, there is no chance of reading the clock, transmitting a packet to the destination, and incorrectly taking the second reading of the clock for a different packet. Additionally, having only one packet on the network reduces the chances of local bus or network contention due to the traffic generated by the experiment to zero.

In addition, each program forked a process on each machine to monitor the source and destination load while RTT measurements were underway. The load averages on both the source and destination hosts were recorded every five seconds to ensure that the host loads did not change drastically during the experiment. If a significant change in the load average of the local host or the remote host was observed for a particular test run using our software, it was omitted and the test repeated. Thus, the results to be presented were collected with minimal variations in the load averages.

2.2 Packet sizes

The nine data packet sizes we used for the tests were: 1, 2, 16, 17, 112, 113, 1023, 1024 and 1025 bytes. This range of packet sizes was chosen to represent the type of traffic that may be present on the network, and also to exercise the buffer management schemes provided by the protocol implementation. The smallest sizes of 1 and 2 bytes represent character-at-a-time user process transactions, and indicate the minimum message transaction delay. The network's response time, that is, the time to send a short message and receive the reply, is measured with small packet sizes. The packet size of 16 bytes is chosen because the back plane of a Datakit node breaks the byte streams into short packets of fixed length, each packet containing up to 16 bytes of user data [FRAS83]. A seventeen byte packet requires two accesses to the back plane, and we may experience a slightly higher delay. The 112 byte size represents individual lines of text, such as program listings or documentation, as might be output by some

user process. Also, 112 bytes is the maximum user data packet size that can fit into one 128 byte 'mbuf' (in mbufs, 8 bytes are used for link pointers, 4 bytes for the data offset, 2 bytes for the size, and 2 bytes for the type) [CABR84]. The 113 byte packet represents 2 mbufs and we intuitively expect a slightly higher RTT with packets of this size.

The 1024 byte packet represents a single mbuf page. In UNIX† 4.3 BSD a logical page of data consists of 1024 bytes. Data not page-sized is normally stored in chains of the smaller size buffers. Thus, a series of 'copy' instructions from user space into the system's small mbufs is required. The one kilobyte buffers are passed by the network protocol software to the network driver by simply augmenting a reference count, whereas the chains of the smaller buffers must be assembled into one contiguous buffer before being given to the network driver. Therefore, if the data is page sized, a copy operation can be avoided [CABR84]. Also, 1024 bytes is our anticipated URP window size. The packet size of 1023 bytes is selected because a chain of the smaller mbufs is required to transmit such a packet. The packet size of 1025 bytes requires two URP windows.

In summary, packet sizes were chosen to correspond to those of the kernel's internal buffers and also to those used by the Datakit for communication.

2.3 Determination of repetition count

Each XUNET test consisted of sending a fixed size packet a predetermined number of times. The 'Repetition count' is the number of times each packet is to be sent [CABR84]. For our tests, we chose a repetition count of 350.

We want to find a repetition count large enough to yield accurate results, yet also small enough such that the experiments can be conducted within a reasonable amount of time. UIUC was arbitrarily selected to be the location to determine a suitable repetition count. Dkbench ran from Vangogh to UIUC for various packet sizes with repetition counts of 50, 200, 350, and 500. The results of these experiments are displayed in Figure 2. Figure 2 shows that the standard deviation of the round trip times is roughly the same except the case in which the repetition count is 200. We did not want to choose a repetition count which would yield results with a large standard deviation for the round trip times. Therefore, we did not select a repetition count of 200 or below for the XUNET tests. We selected the repetition count of 350 because the standard deviation among the results was consistently low. Finally, the amount of time necessary to conduct the tests with this repetition count was still felt to be reasonable.

2.4 Expected results

The user process network latency is defined as the minimum cost to complete a 1 byte network transmission from a source to a particular destination. Thus, minimum latency is represented by the minimum time required to successfully send a single byte of data, i.e., the propagation delay. The minimum user process network latency to Fishonaplatter, given that the maximum speed at which information can propagate is 2/3 the speed of light and assuming the distance to this machine is 3,000 miles, is :

† UNIX is a trademark of AT&T Bell Laboratories.

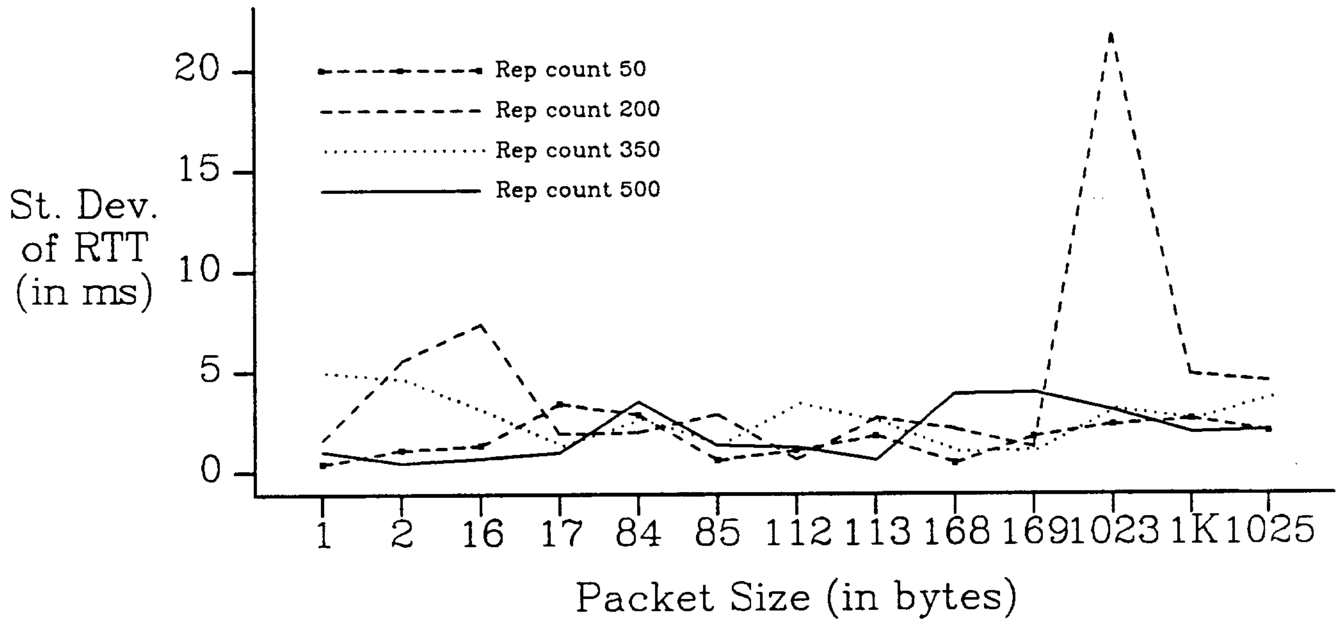

Figure 2

$$3000\ miles\ x\ 1609.344\ \frac{meters}{mile}\ x \left[\frac{1}{(2/3)3\ x\ 10^8 \frac{m}{s}} \right] = 24.15\ ms. \qquad [1]$$

Thus, neglecting each machine's processing time, we expect the response time for a one byte packet, i.e. the time to travel from Vangogh to Fishonaplatter and back, to be approximately 48 ms. If similar calculations are made for UIUC (approximating the round trip distance from Berkeley to Illinois at 3,900 miles), we obtain a response time of 32 ms. Finally, for the local case (Vangogh to Monet), the propagation delay is minimal and can be assumed to be zero. The nonzero values in the local case are due to processing time and bus contention. The propagation delay in the local case has negligible effect on our results.

We would also like to predict the round trip time of packets utilizing the expected full window of 1 KB. The transmission time is the time required for the host machine to place the packet onto the wire. It is dependent upon the transmission rate of the wire and also the size of the packet. XUNET is composed of T1 lines utilizing a 1.544 Mb/s transmission rate; however, at present, constraints limit this rate to about 7/8 of the bandwidth, i.e., 1.344 Mb/s [FRAS83]. At this rate, the transmission time for a 1024 byte packet is:

$$\frac{1024\ bytes\ x\ 8\ \frac{bits}{byte}}{1.344\ \frac{Mb}{s}} = 6.1\ ms \qquad [2]$$

The round trip time can be approximated as:

$$2 \times \text{transmission time} + \text{response time} = \text{round trip time} \qquad [3]$$

This equation is arrived at by the following reasoning: the sender places the packet onto the wire (i.e., the transmission time), the packet travels to the receiver, and the receiver sends the packet back to the local host. Therefore, the absolute minimum round trip time for a 1024 byte packet traveling from Vangogh to Fishonaplatter and back is: 2 x 6.1 + 48.3 = 60.5 ms. Similar calculations for the same packet size to UIUC yield a round trip time of 44.2 ms. Again, we would like to reiterate that these back-of-the-envelope calculations only correspond to network delay, that is, we are not including the execution time on either the local or the remote machines. However, performing these quick calculations enables us better to interpret the accuracy of our results.

2.5 Analysis of results

The goal of these tests was to develop an equation for the round trip time of various packet sizes sent from Vangogh to remote locations. We are interested in the minimum values gathered from each test suite. The minimum values give us the best characterization of an idle network with lightly loaded source and destination machines. In addition, the statistics of minimum values normally show smaller deviations than the means.

2.5.1 Small packet sizes performed well

The initial results for the response time measurements correspond extremely well to our back-of-the-envelope calculations. For example, the response times of a 1 byte message to Fishonaplatter are approximately 54 ms, only 6 ms above our lower bound. UIUC also displays promising results for a 1 byte message. The minimum round trip time to UIUC is 43 ms in comparison to our theoretical value of 32 ms. Finally, Monet, in the local case, has a 1 byte minimum round trip time (response time) of 5 ms.

However, as we increase our packet size the results begin to deviate very drastically from the theoretical results. At Fishonaplatter, for example, the minimum time to send and receive the echo back of a 112 byte packet is 156 ms, and, for a packet size of 1024 bytes, the minimum round trip time is measured at 1266 ms. (Once again, the theoretical time for a 1024 byte packet is 60 ms, a 2000% difference.) We do not think our software is responsible for the large values, primarily because it appears to function properly for the small packet sizes of 1, 2, 16, and 17 bytes. Thus, we need to find the causes of these discrepancies.

2.5.2 The 84 byte window problem

We have determined after running the tests at very specific packet sizes, that multiples of 84 bytes cause large increases in our round trip times. Figure 3 displays the results of this experiment depicted on a log-log graph. Why is there an increase in the round trip time at 84 byte multiples? The answer lies within the KMC. The KMC is programmed to use either small or large buffers, the small buffer holding 84 bytes of user data, and the larger one 756 bytes of data. The driver in each host dictates to the KMC which buffer to use. The current implementation uses the 84 byte buffer, thus we are dealing with an 84 byte window! For example, if we want to transmit 1 KB of data, the KMC transmits 84 bytes, copies them into a buffer, and transmits another 84 bytes *only* after receiving an acknowledgement for the first 84 bytes.

To further clarify what is happening within the KMC, there can be 3 outstanding blocks of data in a window. The KMC currently incorporates blocks of 32 bytes with 4 bytes of overhead per block, yielding 28 bytes of data per block. Thus, 84 data bytes is our

current window. We alerted the various Datakit locations about the 84 byte window problem; in Section 2.5.3 we present the improved results using the larger 756 byte buffers.

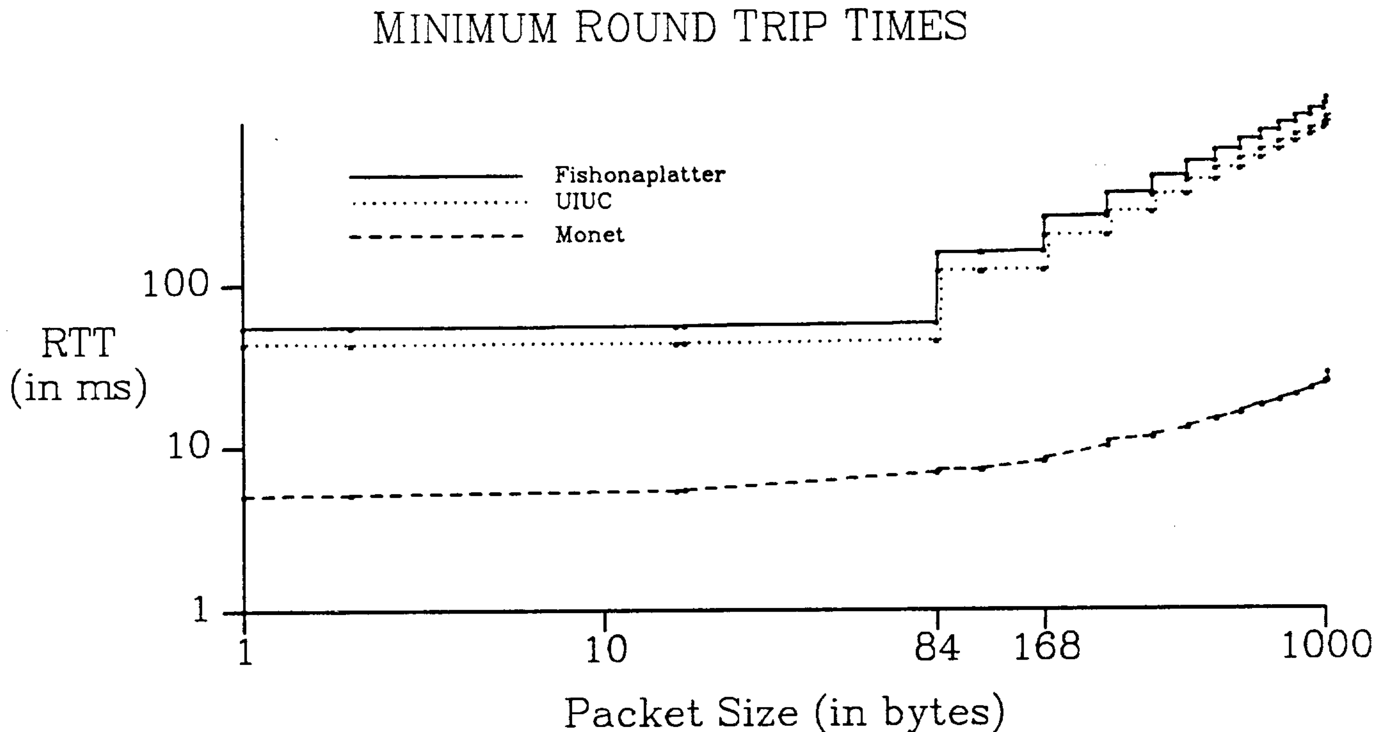

Figure 3

Fishonaplatter requires a minimum round trip time of 57 ms for 84 bytes, yet 155 ms for an 85 byte packet, an increase of approximately 100 ms or two round trip times. The reason there is such a large increase for only one extra byte is due to a combination of the underlying protocols and the experimental software. We now follow the events occurring in the cases of 84 and 85 byte packets. In an 84 byte packet, the entire packet fills the window in one transmission. The remote host, after receiving the 84 bytes, sends the 84 bytes back to the sender, and the underlying protocol piggybacks an acknowledgement on the same packet. Thus, the transaction requires one round trip time.

Next, we examine the case of 85 bytes by displaying what happens in each round trip time.

First Round Trip Time

First, the sender transmits 84 bytes to the receiver. The underlying protocol at the receiver sends an acknowledgement for the first 84 bytes which in turn opens up the window for the sender to transmit the remainder of the message. The program at the remote host is still waiting to receive the entire packet. *The remote program does not write back the packet to the sender until the entire packet is received at the remote host.*

Second Round Trip Time

The sender transmits the final byte and the destination host (e.g.,

Fishonaplatter) has now received the entire packet and begins to write the packet back to Vangogh, starting with the first 84 bytes.

Third Round Trip Time

Vangogh acknowledges receiving the first 84 bytes, thereby opening the window for Fishonaplatter and enabling it to transmit the final byte.

Hence, the entire message transaction requires about three round trip times. Note that the same increase occurs from 168 bytes to 169 bytes, and multiples of 84 bytes thereafter. Also, UIUC exhibits the same increases, but Monet does not, because the propagation time is negligible and the small window does not affect the results.

2.5.3 Improved results with 756 byte window

Figure 4 displays the results of the same tests repeated after some of the hosts switched to a 756 byte window. Vangogh, Fishonaplatter, and Monet made the necessary driver changes, enabling the KMC at each location to use the larger buffer of 756 data bytes. On the date the tests were reconducted, UIUC did not have their changes instituted in the driver code to utilize the larger window. Each of the three outstanding blocks of data was 256 bytes instead of 32 bytes. A 4 byte trailer was again incorporated into each block, yielding 252 bytes of user data per block. Note in Figure 4 the drastic improvement in the round trip times to Fishonaplatter. The jumps are now at 756 byte multiples of two round trip times, as we anticipated. A comparison of Figure 4 with Figure 3 shows that UIUC's results are better because the window from Vangogh to UIUC is 756 bytes even though the window from UIUC to Vangogh remains at 84 bytes. There are still jumps in the delay times at 84 byte multiples, but the increases are only by one round trip time and not two round trip times as in Figure 3. The sender now transmits an entire 85 byte packet in one transmission, whereas in the previous case the local host required two transmissions to write a packet of size 85 bytes to the destination.

The minimum round trip times to Monet demonstrate marginal improvement; i.e., the 1 KB packet, which previously had a minimum round trip time of 24.7 ms, now achieves a minimum of 22.7 ms. This marginal improvement accentuates the notion that the small window in the local case has practically no effect on the results.

2.5.4 Impact of packet sizes

We now discuss the impact, if any, packet sizes had on the experiment's results. We focus this section on the results obtained in the local case, as we do not want the effects of propagation to overshadow the impact of the various packet sizes. The Monet curves in Figures 3 and/or 4 depict the results discussed below.

For the cases of 1 and 2 bytes the results are indifferent. The times with these values give us a representation of an idle network's response time. Although we expect a slight increase in the minimum RTT from 16 to 17 bytes, none is clearly evident. Thus, a packet that requires two, versus one, access to the Datakit backplane does not appear to be an additional source of delay. There is no consistent increase in the RTT from 112 to 113 bytes packets: therefore using one mbuf as opposed to two mbufs does not create additional delay. Both of these conclusions (16 to 17 bytes and 112 to 113 bytes) are important because, as the Datakit hardware and the kernel are redesigned to reduce delays, knowing that the backplane and the mbufs are not prominent sources of delay can be very beneficial.

Although a packet size of 1023 bytes requires a chain of the smaller mbufs to transmit (as opposed to a packet size of 1024 bytes which fills a single mbuf page), no increase in the RTT is evident between the two packet sizes. However, a consistent increase in RTTs from 1024 to 1025 bytes is observed. A packet of 1024 bytes fits into a single URP

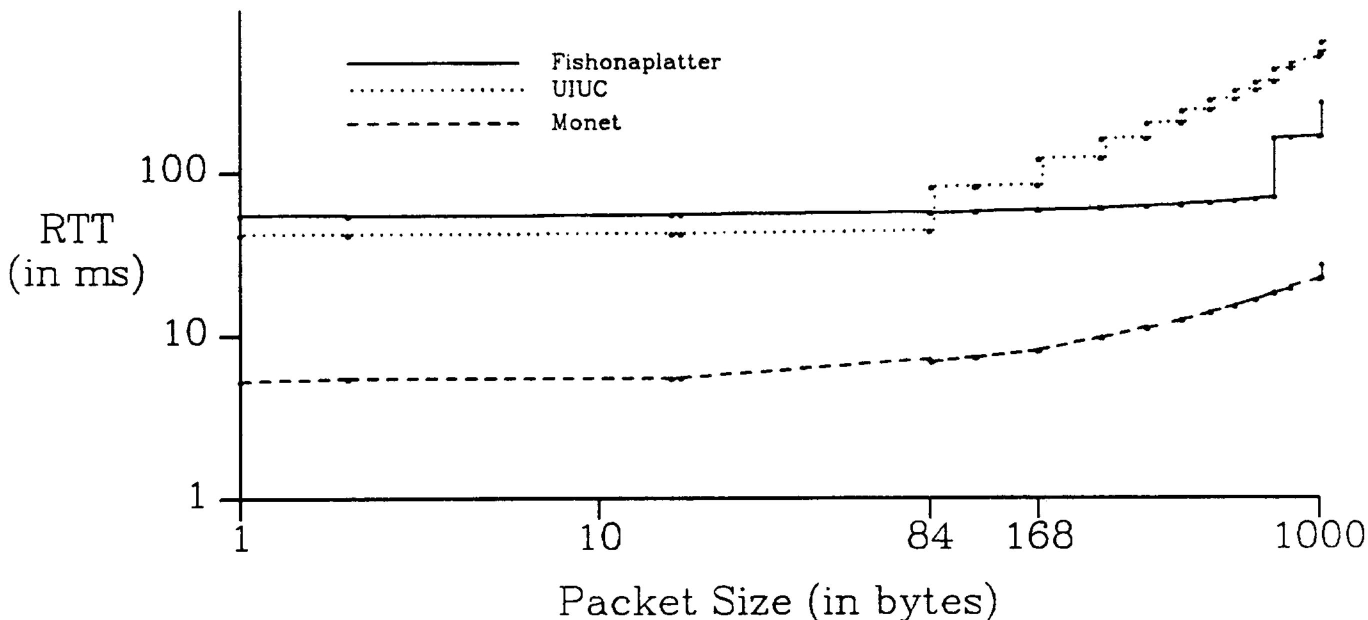

Figure 4

window, whereas a packet of 1025 bytes requires two windows and causes substantially higher round trip times. The additional delay is consistently on the order of two round trip times for packet sizes of 1024 bytes and 1025 bytes.

2.6 Delay formula

The goal of this chapter is to formulate a mathematical model of round trip time valid for a lightly loaded network like XUNET. The primary factor the round trip time is dependent on is the window size. Through examining the initial results using the 84 byte window, the round trip time can be characterized by:

$$\frac{PacketSize}{windowSize} \ x \ response\ time \ x \ 2 + response\ time = round\ trip\ time\ (ms) \qquad [4]$$

Our formula for small windows can be interpreted as follows: if the packet fits within one window, the round trip time is simply the response time. However, if the packet is greater than one window, the round trip time our formula generates is the response time plus the number of additional windows necessary to completely send the message times two response times for each of these additional windows. This formula holds for wide-area configurations involving a significant propagation delay between the sender and the receiver.

Our initial formula for the round trip time does not include any processing time, it is dependent solely upon the response time and the packet size. This simple formula yields the round trip time with an accuracy of 6%, it is consistently 6% above the true value. All values within the same window size (i.e. 1-84 bytes and 85-168 bytes) will yield the same RTT. We would like to reiterate this formula is to send a packet to a destination and back, thus, it would not hold a one way transfer applications (i.e., file transfer).

The local case yields a different formula for round trip time. For Monet, our formula clearly does not hold as increases are not evident at 84 bytes by multiples of two round trip times. The index is primarily dependent upon the processing time of each packet. The Vangogh/Monet tests show a processing time of approximately 0.02 ms/byte. Thus, the local case round trip time formula is:

$$PacketSize \; x \; 0.02 \frac{ms}{byte} + response \; time = round \; trip \; time \; (ms) \qquad [5]$$

The 0.02 ms/byte processing time includes all the Datakit bus contention times, two transmission times, and also the small propagation time, which we consider negligible.

Equation 4 does not hold for the tests involving the improved window of 756 bytes because the window is no longer the predominant factor in determining the round trip time. Processing time is also a factor. In the previous case, the small window overshadowed most of the effects of bus contention and transmission time. However, with a larger window, these delays become more evident. Equation 5 holds, however, holds up to one window for Fishonaplatter. For example, using Equation 5 the round trip time for a 756 byte packet is:

$$756 \; bytes \; x \; 0.02 \frac{ms}{byte} + 54.64 \; ms = 69.76 \; ms$$

The actual time measured was 69.97 ms. Taking into effect the transmission time and all other delays, the round trip time formula for packet sizes of one window (756 bytes) or less to Fishonaplatter is:

$$response \; time + \frac{2 \; x \; packetSize \; x \; 8}{1.344 \; \frac{Mb}{s}} + packetSize \; x \; 0.008 \; \frac{ms}{byte} = RTT \; (ms) \qquad [6]$$

The final parameter in the equation states there is a 0.008 ms/byte additional delay factor. This delay is to be attributed to CPU execution time, queueing delays, and bus contentions. None of the 0.008 ms/byte should be attributed to the network because we have taken the response time to be our measured response time for a 1 byte packet, and not the theoretical minimum. To further reduce this 0.008 ms/byte delay factor, it is due to both the local and remote machines processing the data; thus, each host experiences a local additional delay of only 0.004 ms/byte, or 4 microseconds per byte of additional delay.

2.7 Verification of results using different software

In this section, we verify that the software used in the experiments described above functions properly. We do so by presenting the results of a similar experiment performed with software designed by Gary Murakami of the University of Illinois. The software, dkxload, is a highly modified version of software from AT&T for benchmarking Datakit networks and Datakit host interfaces. Processes again set up full-duplex connections. The local process sends/writes data to the remote side as fast as the protocol's configuration and systems permit [MURA88]. Slight modifications were made to that software to enable the tests to be performed from UC Berkeley.

In this section we describe the "Round Trip Time" (RTT) test, although the software does perform various types of tests. The overall design of the RTT test is similar to our dkbench software. The test measures the round trip time for various packet sizes. The local sending process sends a packet and then waits for the echo from the remote

process. Unlike our software, which for each packet size we sent 350 packets, dkxload sends packets of each size for three minutes and records the total number of packets transmitted. Thus, the values we present are the mean RTTs as opposed to the minimum times presented in Section 2.5. We expect the means to be only slightly higher than the minimum round trip times because of the large number of packets that are sent with each iteration. For example, the dkxload tests to Fishonaplatter send between 600 and 3,300 packets depending on the size of the packet. Obviously, smaller packets are transmitted in larger numbers in a three minute time period than larger ones.

2.7.1 Analysis of results

Table I

"RTT" test to Fishonaplatter using dkxload				
Packet size(bytes)	*Packets per sec*	*Kbytes per sec*	*Mean RTT (ms)*	*Dkbench results (ms)*
1	18.07	0.018	55.33	54.64
2	18.07	0.036	55.32	54.29
16	17.94	0.287	55.73	55.14
17	17.88	0.304	55.94	55.13
84	17.51	1.470	57.12	56.88
85	17.49	1.487	57.18	56.52
112	17.21	1.927	58.12	57.34
113	17.28	1.952	57.88	57.14
168	16.88	2.836	59.23	58.41
169	16.95	2.865	58.98	58.60
756	13.21	9.987	75.69	69.97
757	5.76	4.361	173.6	160.52
1024	5.79	5.928	172.7	166.35
1025	3.68	3.780	271.5	267.36

The results for Fishonaplatter are presented in Table I. The final column in the table lists the minimum values obtained from the dkbench software. As we anticipated, because dkxload presents the mean values, the dkxload results are consistently slightly higher than the dkbench results. We conclude from this table that our software functions properly, or, both tools are faulty in similar ways. With dkxload, as with dkbench, we have an increase in the RTT of approximately two minimum round trip times from 756 to 757 bytes because of the windowing effect.

In addition to verifying our software, the dkxload test also offers two additional pieces of information: the number of packets sent per second and the number of kilobytes transmitted per second. Referring once again to Table I, we see that the number of packets sent per second declines as the packet size is increased. This is an obvious conclusion. As the packet size increases, the time necessary to transmit the packet also increases, as does the number of accesses to the Datakit backplane, and more mbufs are required to hold the data. Thus, the number of packets sent per second decreases. Notice the drastic effect the windowing has on the number of packets transmitted per second. A full window of 756 bytes transmits 13.21 packets/sec, yet, with a packet size of 757 the network only transmits 5.76 packets/sec for this round trip time test in which the packet is being echoed back to the sending process.

The number of kilobytes transmitted per second peaks at a full window of 756 bytes. From one byte to the window of 756 bytes, the number of kilobytes transmitted per

second increases. However, again notice the drastic effect the windowing has on these results, in which there is a large propagation time between the source and destination. The effect due to windowing causes the number of bytes transmitted per second to drop from approximately 10 Kb/s at a full window to under 4.4 Kb/s for a packet size one byte greater than the window size. The significance of the values for the number of kilobytes transmitted per second is discussed in the next chapter. However, note that the number of kilobytes transmitted per second (i.e., the throughput) of a lightly loaded network from Vangogh to Fishonaplatter with echoing peaks at 9.987 Kb/s.

3. LOADED NETWORK MEASUREMENTS

In this chapter we measure several performance parameters of the Datakit network, specifically the mean RTTs with background traffic and the throughput. The goal of this experiment is to determine the effectiveness of a loaded Datakit network, and the potential bottlenecks in the network.

3.1 Software

In addition to dkbench and dkbenchs, we use two other programs designed to load the network. One of the programs again resides on the local host, and the other is stationed on the remote host. The local program, trafgen (traffic generator), works as follows:

```
virtual circuit connection established
read the clock
        repeat
                write 'packet' to destination
        until 1 MB sent
read the clock again
transfer time = second clock reading - first clock reading
```

The program writes 1 MB to the destination as fast as it can. The user program is never reading from the network (i.e., the remote program does not echo back the 1 MB of data), and all error corrections and retransmissions are left to the underlying protocol. The time we obtain to transfer the data does not include call set up time or the call take down time, because the virtual circuit connection is established prior to reading the clock. The clock is read immediately before the first packet of the 1 MB is sent to the destination and the clock is read again directly after the last packet is transmitted to the remote machine.

In determining an appropriate packet size to send to the destination, preliminary trials were conducted with packet sizes of 1024, 2048, and 4096 bytes. The results were indifferent to changes in the packet size: in all cases approximately the same time was required to transfer the 1 MB of data. We conjecture the reason for this is that all the packet sizes are multiples of the URP window size of 1024 bytes. Therefore, we decided to use a packet size of 1024 bytes.

The remote program, trafgens (traffic generator server), simply reads the packets from the network and discards them as fast as possible. It continues until it reads an EOT in the first byte of a packet. Once it receives the EOT, the program exits.

3.2 Methodology

As in the previous experiments, the source host was Vangogh; for the remote host, we primarily used Fishonaplatter. We measured mean round trip times with background

traffic and throughput at the maximum distance of 3,000 miles, to determine whether the results would be in accordance with the theoretical ones.

Each test consisted of measuring the mean round trip times using dkbench, while a variable number of trafgen processes are running simultaneously in the background. We accomplish this through the use of shell scripts. The scripts start up between one and fifteen trafgen programs and then sleep for approximately five seconds. Next, dkbench transmits 100 packets with sizes of either 1 byte or 756 bytes to the same destination. One byte is chosen to represent character-at-a-time transactions, and 756 bytes to fill a window, emulating a user transferring large amounts of data. The repetition count of 100 is picked so that the dkbench program can start and finish while the trafgen programs are running. We reduced the repetition count from 350 to 100 because we were interested in the mean values and not in the minimal ones. To calculate a good estimate of the mean, we omit the few outliers in each trial.

For each trial, the minimum, mean, and standard deviation of the round trip times from the packets generated by dkbench are recorded, and for trafgen we record the start times, end times, and the total number of bytes transferred per second. From the trafgen program we calculate the total background traffic being transferred while the dkbench program is running. Although not all the trafgen programs start and end at exactly the same time, we consider these discrepancies to be negligible because the differences among the start times of the various trafgen processes and those among their end times are insignificant in comparison to their total running time. Finally, as with the experiments in Chapter 2, all tests were run at unsocial hours during the night to minimize the effects of unwanted network traffic and of host loads on the local and remote machines.

3.3 Expected results

Intuitively, we expect the minimum values for the packets generated by dkbench to be independent of the background traffic and the mean values to increase as the amount of background traffic generated increases. We anticipate the minimum times to be the same because the case always exists in which the packet is unaffected by the background traffic and receives service immediately. Whereas in the previous chapter we were interested in the minimum values for a non-loaded network, in this chapter we are interested in the average RTTs caused by a loaded network. Clearly, as the number of processes transferring background traffic increases, we conjecture that our mean RTTs to also increase.

3.4 Analysis of results

Figures 5 and 6 display the results from the first test in our experiment. For this test, dkbench and all trafgen software were on Vangogh and dkbenchs and trafgens programs resided on Fishonaplatter. These figures were obtained by having dkbench send 100 packets of 756 bytes each. Figure 5 displays the mean and minimum RTTs as functions of the number of background trafgen processes running simultaneously with the dkbench generated packets. As we anticipated, the minimum values did not fluctuate at all throughout this test.

The mean RTTs, however, show significant increases over the minimum values especially when seven or more trafgen processes are running simultaneously in the background. With six and below simultaneous background processes running while the dkbench measurements are being taken, the mean and the minimum correspond very closely to one another. Why do the increases start at approximately seven background processes? To help us answer this, Figure 6 displays the results from the same test as functions of the total background traffic being created by the trafgen processes. Up to background traffic of about 60,000 bytes/s, the mean delays correspond very closely to the minimum values obtained.

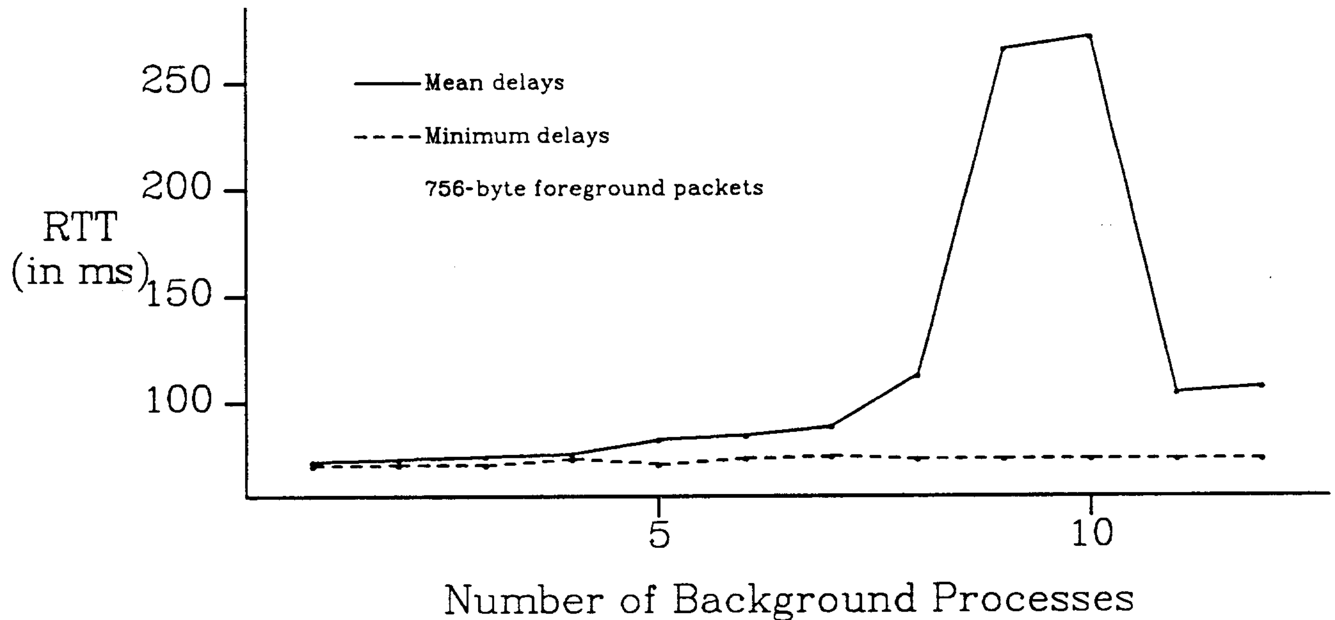

Figure 5

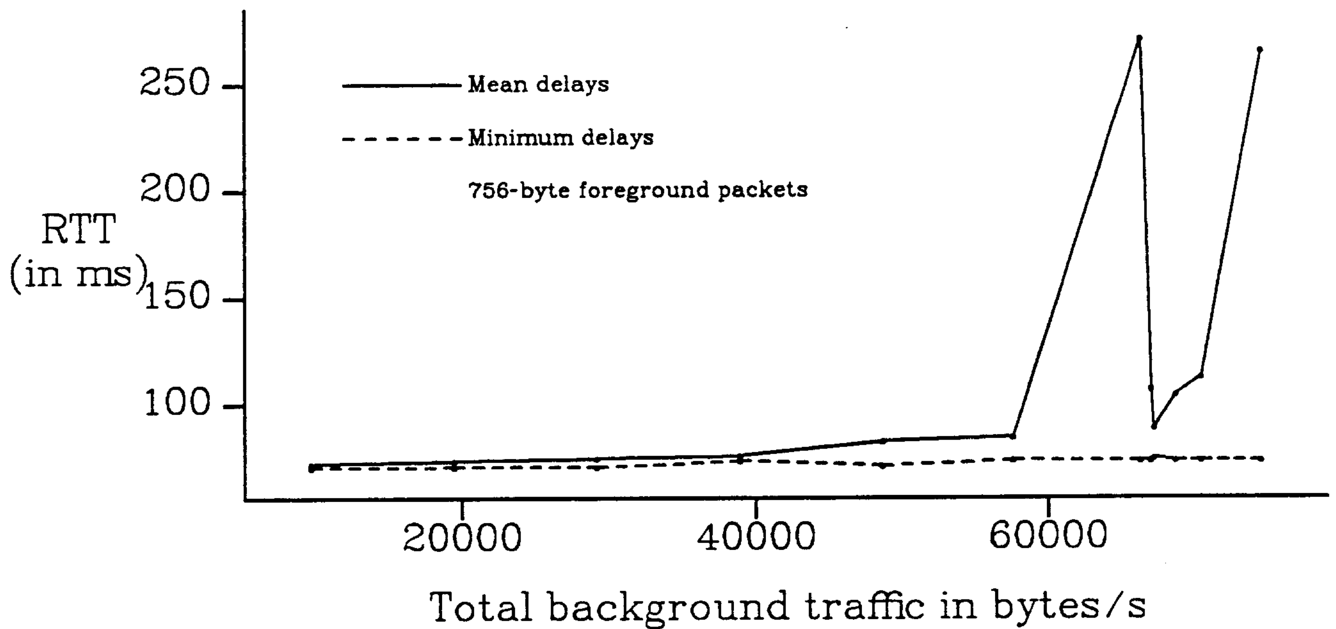

Figure 6

Is the line speed of the T1 lines causing increases in the mean round trip times? Is the network congested?
The answer is clearly no. The effective speed of the T1 line is 1.344 Mb/s, or 168 kB/s. We achieve a background throughput of less than half the potential throughput of the

network. Thus, the bottleneck is clearly not the speed of the network. The reasons for the increases in the mean round trip delays must therefore lie within the Datakit hardware.

Are the increases in the mean RT times caused by congestion and queueing in the host interface?
To answer this question, we have repeated the test with dkbench on a different machine. Figures 7 and 8 display the results from the second test in our experiment. For this test, the server programs are once again on Fishonaplatter and all trafgen software is run from Vangogh; however, the dkbench software resides on Monet. These figures display the results obtained by sending 100 packets of 1 byte each. These results are similar to the results from the first test in several ways. The mean RTT does not vary significantly over the minimum for less than seven simultaneous background processes and a total background throughput of 60,000 bytes/s. Since both tests yielded similar results, we conclude that the local host interface is not causing the increases in the mean RTTs.

Is the bottleneck the trunk interface?
Our hypothesis is that the trunk interface is indeed the bottleneck due to the overflow and packet loss it causes beyond a certain threshold of load. Data is arriving from the Datakit backplane at a rate of 8 Mb/s to a FIFO queue connected to the trunk. Data waits on the queue until it can be sent over the trunk. The speed of the trunk is only 1.5 Mb/s. Thus, the FIFO is growing at a rate of 8 Mb/s - 1.5 Mb/s = 6.5 Mb/s. Also, data in the FIFO queue is stored in 16 byte chunks as it arrives from the broadcast bus. Consecutive chunks stored in the queue are not necessarily all from the same process. Since the trunk empties the FIFO queue, the traffic on the trunk is highly multiplexed, and large amounts of data from a single source have interleaved amongst them characters from other sources. In addition, this procedure burdens the remote host interface as it tries to demultiplex the incoming byte stream into separate buffers. Thus, at the remote host, overflow may occur which results in packet loss. A potential future experiment, beyond the scope of this paper, could be to determine whether the losses are caused more by the local host or the remote host.

This also explains the reason the network does not achieve a maximum throughput and maintain it. The total trafgen traffic peaks at about 8 processes and then begins to decline. This can be explained by an argument similar to the above: the greater the number of simultaneous processes, the higher the degree of multiplexing on the trunk, and the heavier the burden at the remote host interface. Hence, a reduction of total throughput is the outcome.

3.5 Measured throughput for a single process

This section describes an experiment designed to measure the throughput achievable by a single process. We first discuss the maximum achievable throughput based upon our minimum RTT measurements in Chapter 2. Next, we compare the maximum throughputs to the measured throughputs, and determine the effective window size as opposed to the actual window size. Finally, we measure the correction factor, the additional delay incorporated with each window sent, and conjecture about the sources of this delay.

3.5.1 Methodology

This experiment involves various sources and destinations. Each test consists of measuring the time to do a file transfer (i.e., a 'push' in Datakit) from the source to the destination. However, we want to factor out the time to do call set up and call take

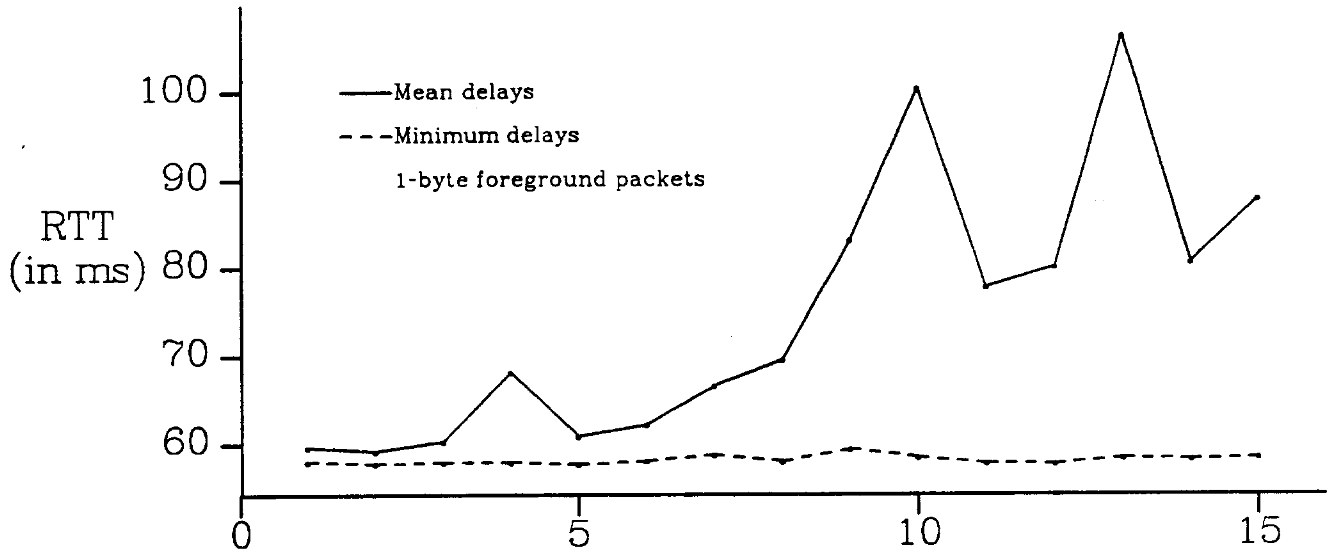

Figure 7

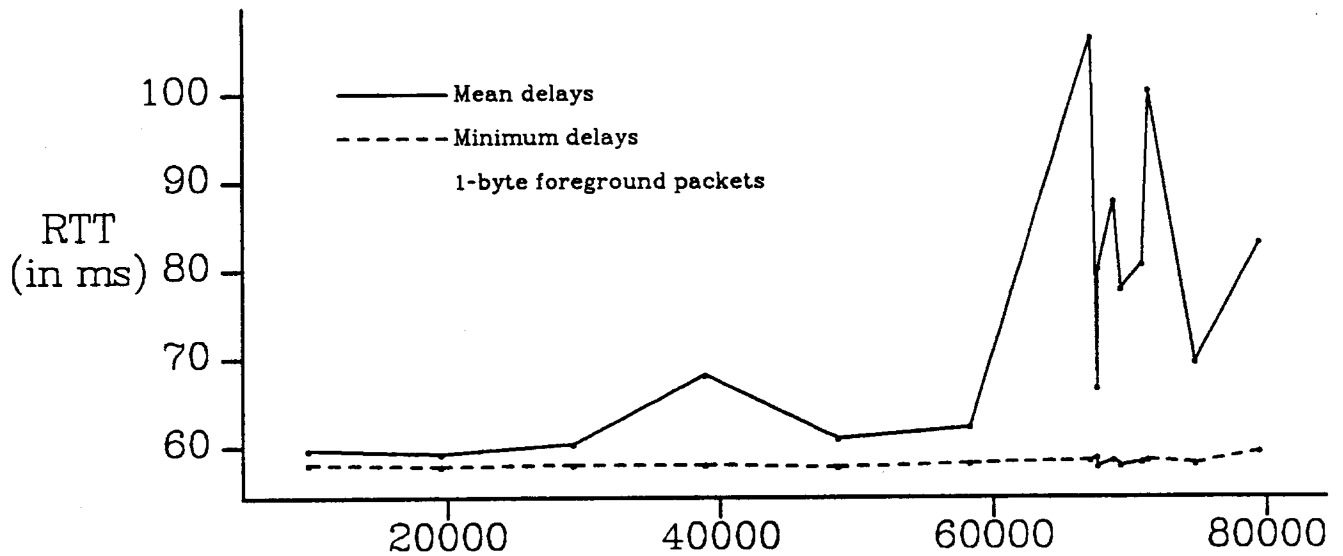

Figure 8

down to obtain the throughput of an established connection. In order to factor out this time, ten transmissions of a zero length file are made from the source to the destination, and the minimum time is recorded.

Second, a large (~300 KB) file if transferred ten times from the source to the destination, and the minimum values of the transfer times are recorded. Transferring

a large file gives us a good indication of the throughput (the number of bytes transmitted per second). All times are recorded with the UNIX *time* command. Although the accuracy of this command is only to the second, multiple repetitions of the tests were conducted and all the values obtained were very consistent. Also, as with all of our other experiments, all tests are run at unsocial hours of the night to minimize the effects of unwanted network traffic and host loads on the local and remote machines.

Once the minimum times to transfer the zero length file (called ZERO in this section) and the large file (called LARGE hereafter) are recorded, the measured throughput of the network is obtained by dividing the size of LARGE by the difference of the transfer times of LARGE and ZERO, or:

$$Measured\ throughput = \frac{SizeofLARGE}{(transfer\ time\ of\ LARGE) - (transfer\ time\ of\ ZERO)} \quad [7]$$

Next, the measured throughput is compared to the maximum throughput. The maximum throughput for a single process is:

$$Maximum\ throughput = \frac{WindowSize}{Minimum\ round\ trip\ time} \quad [8]$$

where the minimum round trip time is the one obtained in Chapter 2. This formula is arrived at by the following: the most amount of data a process can transmit is a full window, and before the process can transmit another window, it must wait for an acknowledgement. Thus, the maximum number of bytes transmitted per second (i.e., the throughput), is dependent upon the window size, and the time to receive the acknowledgement.

Finally, we compute the effective window based upon the measured throughput, and the correction factor incorporated with each window sent, according to the following formula:

$$Measured\ throughput = \frac{WindowSize}{RTT + correctionfactor} = \frac{Effective\ window}{RTT} \quad [9]$$

3.5.2 Analysis of results

The results for this experiment are displayed in Table II. Before we analyze the results we would like to step through one entry in the table and explain how the results were obtained. From UIUC* to Vangogh[2], the window size is 84 bytes. UIUC* still has not incorporated the necessary changes to have a 756 byte window. Other values necessary for this Table II entry are as follows:

---

2. Recall UIUC* denotes experiments conducted with a VAX 780 at the University of Illinois.

Table II

Maximum Throughput vs Measured Throughput for a Single Process					
Source / Destination	*Window size (bytes)*	*Effective window (bytes)*	*Maximum throughput (bytes/s)*	*Measured throughput (bytes/s)*	*Correction factor (ms)*
UIUC* >> Vangogh	84	82.68	2100	2066.9	0.6
Vangogh >> Fishonaplatter	756	518.76	13830	9490.7	25
Fishonaplatter >> Vangogh	756	520.53	13830	9523.2	24.7
Vangogh >> UIUC*	756	459.49	17784	10808.9	27.4

minimum time to send ZERO:	4 seconds
minimum time to send LARGE:	166 seconds
size of LARGE on UIUC*:	334848 bytes
minimum round trip time:	40 ms

From these values, the entries in our table can be easily calculated:

Theoretical maximum throughput =	84/0.040 = 2100 bytes/sec
Measured throughput =	334848/(166-4) = 2066.9 bytes/sec
Effective window =	2066.9 x 0.040 = 82.68 bytes
Correction factor per window =	84/2066.9 - 0.040 = 0.6 ms

It is clear that for a small window (84 bytes) the measured throughput and effective window are extremely close to their maximum values. This is due to the window size being the bottleneck in the network. For such a small window size, queueing delays and CPU processing times are negligible. Notice that the correction factor for the round trip time of a full window is only 0.6 ms; hence, we are actually dealing with a round trip time of 40.6 ms.

However, for the remainder of the cases we are dealing with a 756 byte window, and decreases in performance are evident. Tests from Berkeley to New Jersey yield an effective window in the order of 520 bytes, and tests to Illinois only about 460 bytes. Why is there such a drastic decrease in the network's ability to utilize the full window? First, note that the correction factors for both the cases of Illinois and New Jersey are approximately the same, i.e., around 25 ms. This leads us to believe that the delays are due to processing time in each host and are not due to the network. If the delays were due to the network, the correction factor for New Jersey would be larger than the correction factor to UIUC* because of the differences in the propagation delay. This delay could be the time required for the 'push' program to process the acknowledgement and transmit the next window. The other major cause of delay is explained in Section 3.7.2.

3.6 Measured throughputs for multiple processes

This section describes another simple experiment designed to measure the throughput due to multiple processes. For this experiment, we use only Fishonaplatter as the remote host. This is because the tests uncovered many bugs in the 4.3 Datakit driver, which often caused the machine to crash, and employees at Bell Labs of Murray Hill, N.J. were extremely tolerant and cooperative.

3.6.1 Methodology

The two programs we use with this test are trafgen and trafgens, described in Section 3.1. Each test consists of running a variable number of trafgen programs simultaneously. We accomplish this by using shell scripts. The script runs between one and 14 trafgen processes simultaneously. For each process running, the start time, end time, and throughput are recorded. From these results, we calculate the total amount of data being transferred. To reduce the possibility that the local host load would affect the results, half of the trafgen processes originated from Vangogh and the remainder were run on Monet. Once again, we would like to mention that not all the trafgen programs started and ended at exactly the same time, and thus the total throughput we measured has a margin of error. We consider this discrepancy to be negligible because the differences among the start times and among the end times were insignificant in comparison to the total running time.

3.6.2 Analysis of results

Table III

Throughputs for Multiple Processes		
Number of processes	*Overall throughput*	*Throughput per process*
1	9761.15	9761.15
2	19515.2	9757.62
3	29261.9	9753.97
4	38947.6	9736.9
5	48616.1	9723.21
6	58280.3	9713.38
7	67955.0	9707.86
8	71829.5	8978.69
9	72508.3	8056.48
10	63550.8	6355.08
11	58098.9	5281.72
12	56426.9	4702.24
13	54905.2	4223.48
14	50005.2	3571.8

Table III contains the results obtained from this test, and Figures 9 and 10 display graphs of the throughputs obtained for each process and of the total throughput versus the number of trafgen processes. First of all, note that the throughput obtained for a single process is in the order of 10,000 bytes/s. This parallels, as it should, the results from the previous section for the throughput of a single process. As the number of simultaneous trafgen processes increases, the throughput per process maintains approximately the same level around 10,000 bytes/s until we reach seven or more simultaneous trafgen processes. We conjecture again that the trunk interface is causing a bottleneck at that point.

As we further increase the number of simultaneous processes, the throughput per process continues to decline as depicted in Figure 9. Many channels talking simultaneously cause Datakit bus contention, but the drop in throughput per process may be caused by buffer overruns at the remote location. There is a limited buffer capacity on Fishonaplatter, which is receiving data from multiple channels as fast as our local machines can transmit it. Data may arrive which Fishonaplatter cannot accept because its buffers are full, thus, it must request a retransmission. These

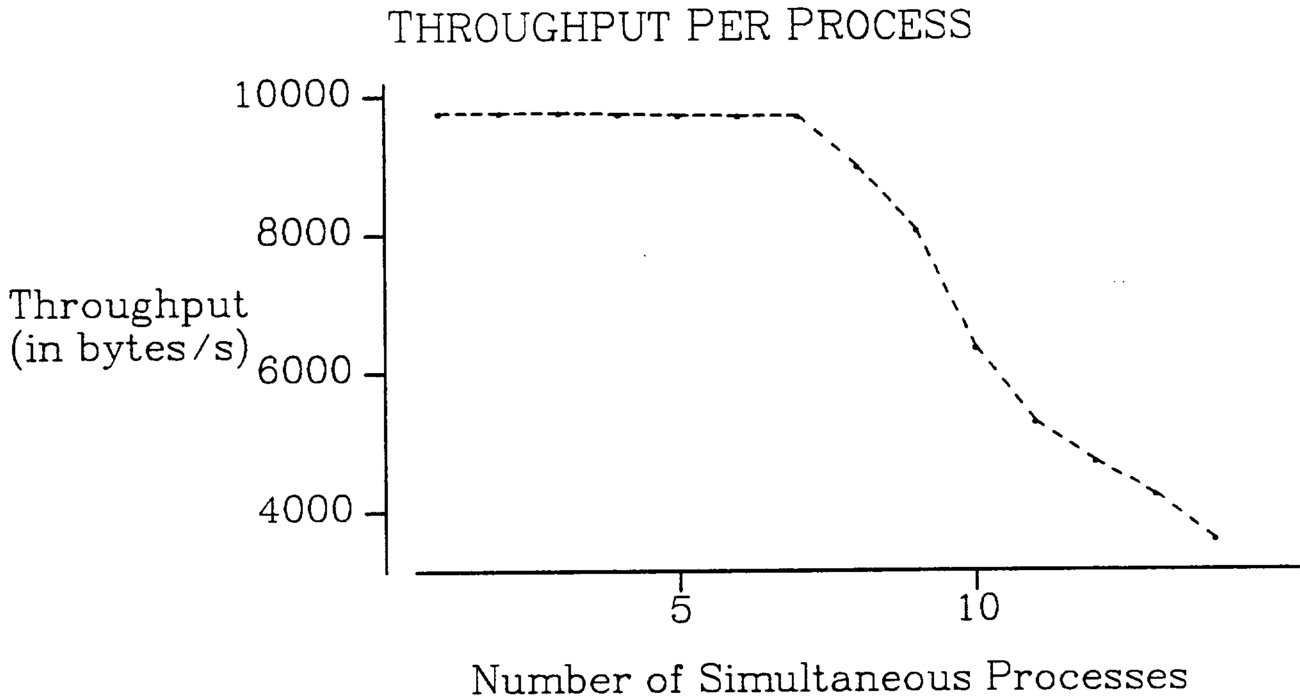

Figure 9

buffer overruns at the remote host interface could be severely reducing the throughput per process.

Figure 10 displays the total throughput versus the number of simultaneous processes generating load. The throughput linearly increases until we reach about 7 processes, then starts to decrease. As displayed in Table III, the maximum throughput is about 72 KB/s due to the T1 trunk interface. The maximum throughput is always limited by that of the lowest capacity element in our chain of buffers and interfaces. In this case it is due to congestion of the trunk interface board. However, why does the throughput decrease and not maintain a steady rate? This can be attributed to buffer overruns at the receiver. The higher the number of channels transmitting simultaneously, the greater the possibility that a buffer is overrun and a retransmission is required.

3.7 Verification of results using different software

This section verifies the results discussed in Section 3.5.2 by describing a similar experiment with the dkxload software designed by Gary Murakami. In the Chapter 2, we verified the dkbench results with the RTT test. In this section, we verify the single process throughput rates using the transfer test.

The transfer test measures single process transfer rate, i.e., the maximum data transfer rate that a single process application can expect. The transfer rate is iterated over a range of user write (and read) data sizes. The transfer test, as with the RTT test of dkxload, runs for three minutes for each packet size. The packet sizes we chose the same as those we used in Chapter 2.

3.7.1 Analysis of results

To gain confidence in the results presented above in Section 3.5.2, we ran the transfer dkxload test with Vangogh and Fishonaplatter. The results are presented in Table IV. For a packet size of 1024 bytes (the same packet size we used for trafgen), we once

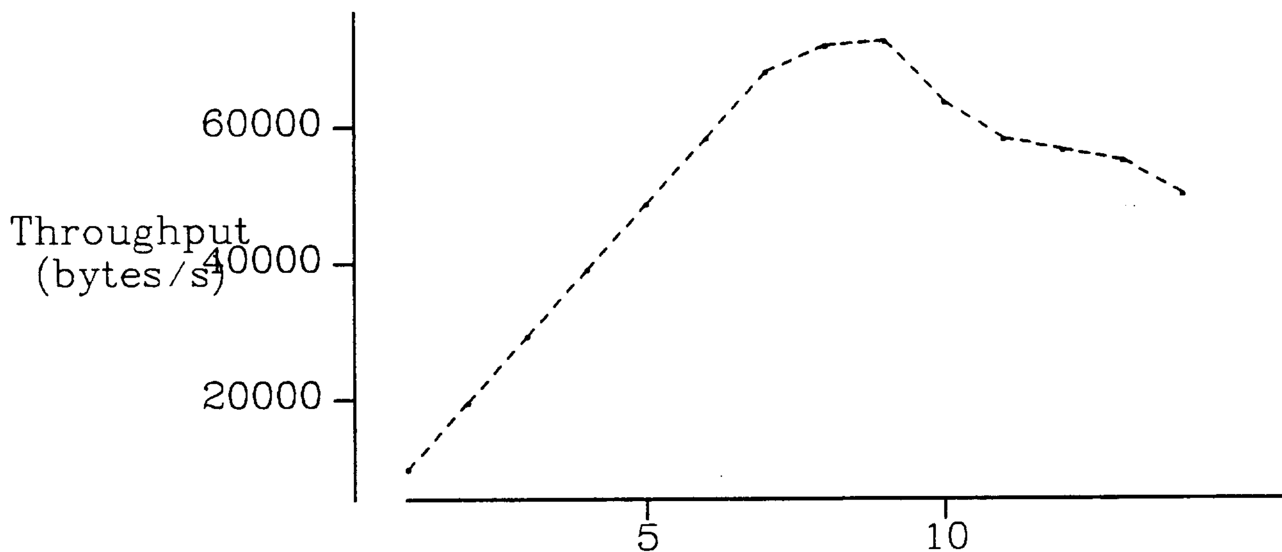

Figure 10

again achieve a throughput of approximately 9,700 bytes/s. This also corresponds closely to our file transfer throughput of 9,500 bytes/sec presented also in Section 3.5.2. Both trafgen and dkxload yield a slightly higher throughput in comparison to our file transfer throughput because their remote server programs drop the data, whereas for a file transfer the data is written to a file. The results from these packet sizes show that our software functions properly.

A closer look at the throughput generated uncovers two interesting problems with the network. First, the file transfer throughput and the maximum throughput for a single process discussed previously in this chapter were about 9,600 bytes/s, yet Table IV exhibits a throughput of nearly 13,000 bytes/s with a packet size of 756 bytes. Second, why is there a significant drop in the throughput from 1024 to 1025 bytes when we expect the throughput for these packet sizes to be roughly the same because they are both within two windows?

3.7.2 Limited throughput due to the KMC

The KMC is the cause of these strange new results. The KMC arbitrarily uses blocks that are 1/4 the URP window size of 1024 bytes. For an URPK window a block of data is in the following format:

<252 bytes of data><4 byte trailer>

Thus, for any window size, the KMC would be allowed to have up to 4 unacknowledged blocks outstanding; however, for an unknown reason, the KMC limits itself to three blocks outstanding, yielding the 756 byte window size. The KMC code and the kernel interface allow only one outstanding request per channel at a time in the KMC. The KMC does not tell the host it is ready for another transmission until all the blocks of the previous request have been acknowledged. The mbufs being fed to the KMC are 1024 bytes long, which means that the KMC transmits 756 bytes, waits for an ACK, transmits the final 268 bytes, and, after receiving an acknowledgement for the 268 bytes, accepts another mbuf of data. This is poor design, and is severely limiting our

Table IV

Transfer test to Fishonaplatter			
Message size(bytes)	*Throughput (Kbytes/s)*	*Mean transfer rate (ms)*	*Kilo packets/s*
1	0.0198	50.42	19.83
2	0.039	50.39	19.84
16	0.315	50.76	19.70
17	0.335	50.73	19.71
84	1.635	51.38	19.46
85	1.652	51.43	19.44
112	2.164	51.75	19.32
113	2.182	51.78	19.31
168	3.212	52.29	19.12
169	3.233	52.28	19.13
756	12.96	58.33	17.14
757	7.36	102.86	9.72
1024	9.68	105.76	9.46
1025	6.56	156.25	6.40

throughput. For transferring a large file, the data is sent to the KMC 1024 bytes at a time. Thus, the data is being transmitted from the sender to the receiver in the following byte sequence: 756, 268, (another mbuf) 756, 268, 756, 268, and so on. Hence, the typical usage of the network is not utilizing its full potential.

The packet size of 1025 bytes accentuates further the limitations of this implementation. First, a 1024 byte mbuf is sent to the KMC. The KMC transmits a full window, 756 bytes of data. After receiving an ack it sends the final 268 bytes of the mbuf to the destination. The KMC receives an ack from the remote program and another request arrives to the KMC containing the final byte. Thus, the KMC transmits the final byte and the transfer is complete. This explains the significant drop in throughput from a packet size of 1024 to 1025 bytes. Even though we have a 756 byte window, the KMC transmits a 1025 bytes packet in three transmissions of sizes 756 bytes, 268 bytes, and 1 byte. The problem is obvious.

Table IV displays a throughput of approximately 13,000 bytes/s with a packet size of 756 bytes. 756 bytes is the size to achieve maximum throughput. The reason is as follows: the program is transmitting multiple packets, each 756 bytes in length, or, each request to the KMC is 756 bytes. The KMC transmits the entire packet, waits for an ACK, and accepts another request. The KMC in this case is always transmitting a full window. The theoretical maximum throughput for a single process from Vangogh to Fishonaplatter is listed as 13,830 bytes/sec in Section 3.5.2, even though we only achieved a throughput of approximately 9,500 bytes/s. In this section we discovered that the network is capable of achieving a throughput of nearly 13,000 bytes/sec for a single process. The effective window computed from Vangogh to Fishonaplatter was 518 bytes with file transfer, yet, if the network utilizes its potential, it can achieve an effective window of 705 bytes. The KMC is severely limiting our throughput.

4. CONCLUSION

The design objectives of Datakit were to derive an adequate architecture for the combination of local and wide area networks to present a uniform appearance to the user, to be effective as a transport mechanism for a great variety of traffic patterns, and to be usable economically by a wide range of consumer products[FRAS83]. To test the effectiveness of Datakit in achieving these design objectives, our experiments measured the delay and throughput of the XUNET Datakit network. We have found that the network is utilizing a window much smaller than one that the network is capable of sustaining. We have found the delay is largely dependent upon the window size for large packets, and the propagation delay. The throughput per process is currently not achieving its full potential because of the KMC interface problem. Also, the throughput for multiple processes maintains a steady rate and then begins to decline due to congestion at the trunk interface.

In order to reduce delays and increase throughput for wide-area configurations, the window size should be expanded. In nearly all of our measurements involving a significant propagation delay, the window had a tremendous effect upon the results. Once the window is expanded and the network is capable of maintaining an acceptable throughput for many processes, the network will be much better suited to wide area applications. These tests were designed, and the methodologies laid out, with the intention they will be repeated at a later date once modifications are made to the network. Experiments of this type must be performed in order to test the effectiveness of any network. We hope that the tests will enable network designers to improve the overall performance of XUNET.

5. ACKNOWLEDGEMENTS

I would like to thank the following people for their outstanding contributions towards this report. First, Domenico Ferrari, for allowing me to undertake such an exciting and state of the art project. Caryl Carr and Dave Presotto of AT&T Bell Laboratories, for their patience and understanding throughout the summer, and their dedication to fix the Datakit driver for Fishonaplatter. Ron Arbo, a fellow graduate student also involved with Datakit evaluation, for his insight about the network and his inspiration to work harder and dwell deeper into my work. Finally, I would like to give incredible and uncountable thanks to Riccardo Gusella, for all his time, guidance, and support. The success of our experiments would have been severely limited without Riccardo's expertise.

REFERENCES

[ARBO88]
Arbo, Ronald, "A Performance Sudy of Remote Executions in a Wide-area Datakit Network", University of California at Berkeley, expected October 1988.

[CABR84]
Cabrera, Luis Felipe; Hunter, Edward; Karels, Mike; Mosher, David, "A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2 BSD" University of California at Berkeley, 1984.

[CHE81]
Che, H; Marshall, W.T, "Design and Performance Analysis of Computer Interfaces for a Datakit-Based Computer Network", AT&T Bell Laboratories, New Jersey, July 1981.

[FRAS83]
Fraser, Alexander F, "Towards a Universal Data Transport System", IEEE COMPCON, England, September 1983.

[XUNE88].
Material received from the XUNET planning conference held in Chicago, January 1988.