# Performance Analysis of Queueing Algorithms on Datakit T1 Trunk Lines†

Michael J. Hawley

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

ABSTRACT

Datakit® VCS is a networking system which integrates local and wide area networks, and attempts to provide good interactive response in the presence of bulk transfer traffic. The queueing algorithms used at the trunk interface are the chief means of providing good interactive response. This report compares the performance of fifo queueing and per-channel queueing by implementing these disciplines on a programmable trunk board and measuring the throughput and interactive delay. These measurements show that (1) fifo queueing delays messages up to ten times longer than per-channel queueing when the trunk is heavily loaded, and that (2) per-channel queuing does not adversely affect trunk throughput or delay when the trunk is not loaded.

† This work was sponsored by AT&T Bell Laboratories.

® Datakit is a Registered Trademark of AT&T Bell Laboratories.

## 1. Introduction

The Datakit virtual circuit switch developed by AT&T Bell Laboratories is intended to provide efficient data communication over local-area, metropolitan-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 the conflicting requirements of interactive traffic and bulk-transfer traffic. For example, 56 Kbps trunks do per-channel queuing at the sending end, which allows short messages to be sent over the trunk quicker than large messages. In the case of T1 trunks (1.544 Mb/s) however, it was impractical to implement per-channel queueing until the recent advent of high speed RISC processors. In the spring of 1989, a special Datakit T1 trunk board was developed by Bell Laboratories which incorporates an AMD 29000 RISC processor [Byrne 89]. It was hoped that this processor would be fast enough for per-channel queueing at T1 speeds. This report describes the attempts to program this board to do per-channel queueing and gives the results of message delay and throughput performance tests performed on this board during the summer of 1989 at Bell Laboratories' Murray Hill location.

The performance tests involve user-level measurements of the throughput and message delay of a T1 trunk which is programmed to do either fifo or round robin queueing. The goal of the tests is to verify that performing per-channel queueing on this trunk board does not adversely affect the delay and throughput of an unloaded trunk, and to verify that per-channel queueing gives lower delays to short messages than fifo queueing. It is of interest to know how interactive delay increases with increasing load on the trunk for each of the queueing disciplines. This information will help Datakit designers decide whether or not the additional hardware required to implement per-channel queueing is justified. For example, if both disciplines perform identically when the trunk utilization is 98%, one would question the value of per-channel queueing.

### 1.1. Datakit

The test bed for all the experiments performed is Datakit. A detailed description of Datakit may be found in [Fraser 83]; a brief description follows.

A Datakit node consists of a cabinet that houses a shared backplane through which all attached devices communicate. Devices connect to the backplane via interface modules that plug into the cabinet and perform buffering and backplane contention procedures. Typical devices are host computers, terminals, and trunks to other Datakit nodes. Each node also contains a network controller which handles call processing. Each node performs switching and multiplexing of user data carried on virtual circuits.

The transport protocol Datakit uses is the Universal Receiver Protocol (URP) [Fraser 87]. URP has several modes, but the only one considered here is block mode with window flow control and retransmission on error. A URP window is specified as a number of bytes. The bytes are divided into at most 7 units called *blocks* (the default is 3), and a block is the unit of error detection and retransmission. A block may contain user data or URP control characters. URP uses 'Go-Back-N' error control, so any transmission errors are costly. URP does provide negative acknowledgements (NAKs) so that retransmissions may occur upon detection of an error. However, if the NAK is lost (or never sent) the timeout is (typically) one second. Each URP block has a four byte trailer which contains an end-of-block control character (BOT), two bytes containing the length of the block, and a sequence number control character (SEQ(i)).

In the experiments performed, three types of host interfaces were used: CURE boards, KMCs and AVDKs (A VMEbus Datakit interface). CUREs and KMCs are both DMA engines, the difference being that CURE boards have much more on-board memory and a faster processor, and KMCs perform URP protocol processing. The AVDK board is a DR11-style VMEbus Datakit interface which does not use DMA. The host simply reads from an input fifo and writes to an output fifo.

### 1.2. Queueing Disciplines

The queueing disciplines under consideration are first-in first-out (fifo) and per-channel queueing (or round robin queueing); both are pictured in figure 1. In fifo queueing, data is simply sent out on the outgoing line in the order it was received. In per-channel queueing, data is queued up in a separate fifo

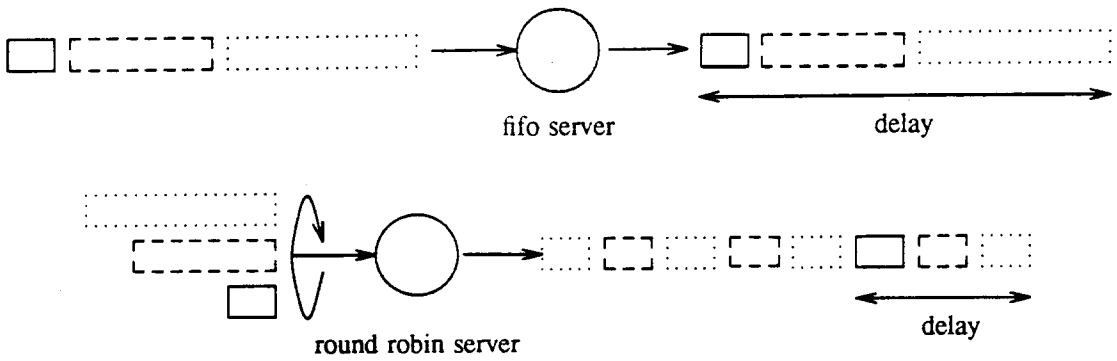


Figure 1 - Queueing Disciplines

for each virtual circuit, and a chunk (hereafter called a *quantum*) of data is taken from each fifo in turn and sent out on the outgoing line. In this way, if a small amount of data arrives on a channel, this data will be serviced before a long burst would be completely serviced. Thus, round robin queueing passively gives priority to short messages, so interactive delay is minimized. The only parameter in round robin queueing is the size of the quantum, which should be selected after weighing the conflicting requirements of throughput and delay. A smaller quantum will give shorter interactive delays, but the higher overhead (because the trunk framing is fixed regardless of the amount of user data in the trunk frame) will decrease the throughput.

### 1.3. Hardware Environment

The queueing disciplines mentioned above are implemented on a daughter board that plugs into a DSX1 trunk interface board [Byrne 89] that, in turn, plugs into a Datakit backplane. The daughter board consists of 1 Megabyte (MB) of fast SRAM, an AMD 29000 processor running at 16 MHz (where every instruction except loads from and stores to memory takes 1 cycle), a 4K envelope (E) (an envelope is the Datakit unit of transmission and consists of 8 data bits plus a bit to indicate whether the data is control or user data, plus a parity bit) input fifo, and a 1 KE output fifo.

Figure 2 shows how two DSX1 boards are connected. Note that data may come from the backplane at 8 Mb/s, but data can only be sent out on the T1 line at about 1.5 Mb/s, so queues will build up on the board. The DSX1 board may be used without its daughter board, in which case the data enters a 32 KE hardware fifo. Alternately, a daughter board may be plugged in, in which case the data is put into a small fifo on the daughter board. The 29000 reads data from the input fifo, buffers it in memory (according to the queueing discipline being practiced) and writes the data to the output fifo. The daughter board also has provisions for downloading programs into its memory and for connecting to a terminal.

The T1 trunk on which the measurements are made is connected between two Datakit nodes (of course), and has a physical length of about 3 feet. Because of this short length, the propagation delay along the trunk is negligible.

### 1.4. Software Environment

Programs for the 29000 are written in C. These are compiled and linked [AMD 89] on a Sun 3 and downloaded over the Datakit network into the daughter board's memory. The 29000 runs in Supervisor mode with all interrupts and memory management options turned off. There is no multiprogramming. The daughter board has no knowledge of the virtual circuits passing through it unless there is data for that virtual circuit stored in the board's memory; the daughter board does not participate in the call set-up. Thus, a channel is *active* only when it has data in the daughter board's memory.

Three trunk configurations are possible: (1) DSX1 board with no daughter board (referred to as *hfifo* in the graphs), (2) DSX1 board with daughter board running a fifo program (*fifo*), and (3) DSX1

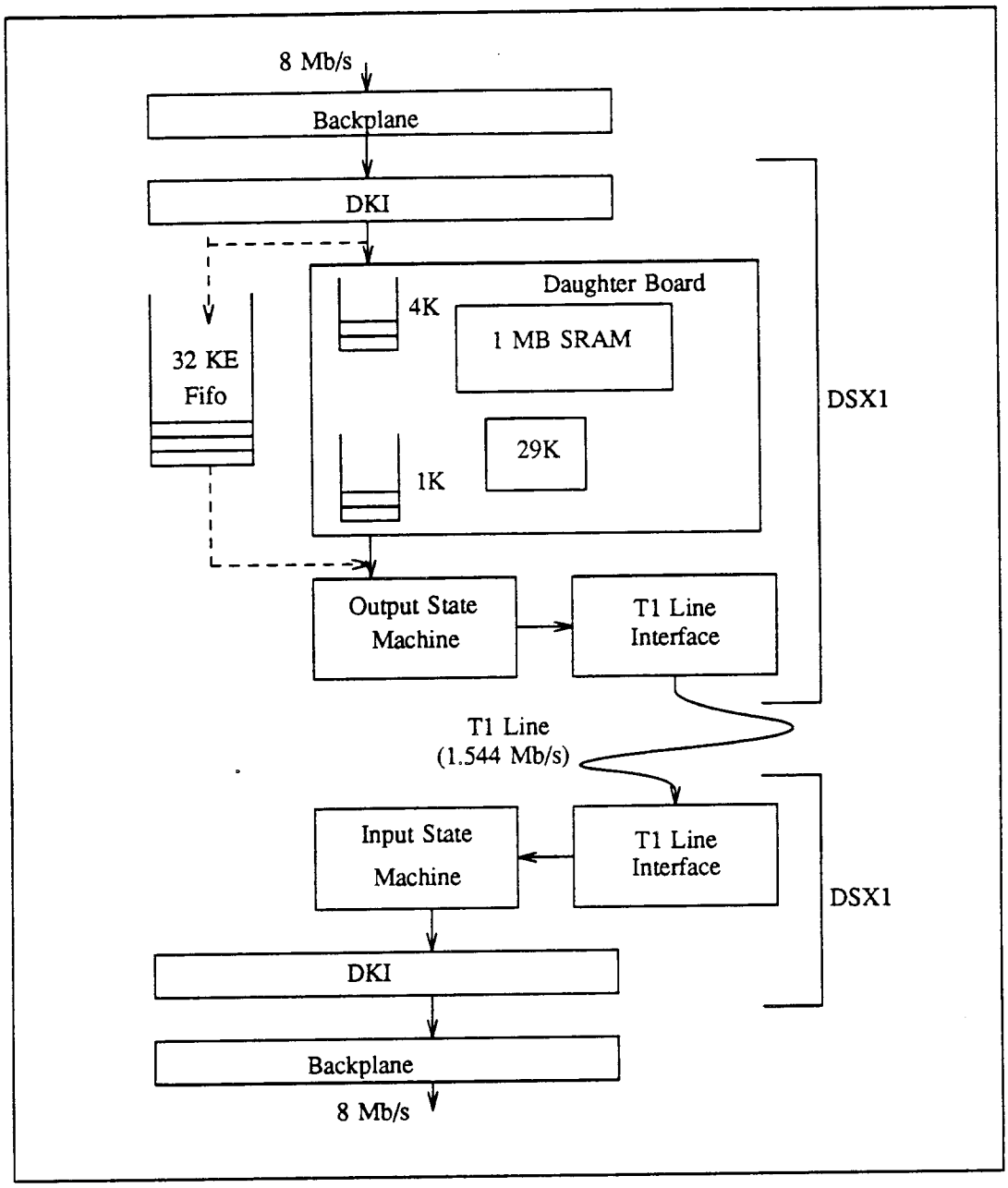


Figure 2 - DSX1 Board

This figure is a block diagram of the DSX1 hardware. Each DSX1 module has a Datakit interface (DKI) which sends data received from the backplane to the daughter board. If no daughter board is present, the data is queued up in a 32 KE hardware fifo. The daughter board has a 29000 processor and one megabyte of static RAM. The 29000 reads data from the 4 KE input fifo, buffers the data in memory (according to the queueing discipline desired), and writes the data to the 1 KE output fifo. In either case, the data is now passed to a state machine which assembles the data into LAPD [Byrne 89] frames. These frames are then passed to the T1 Line Interface, and sent out on the T1 line. At the other end of the T1 trunk, the data is passed from the T1 Interface to an Input State Machine which converts from the LAPD format to the backplane format. The data is then passed to the backplane. Both ends transmit and receive, of course, but only one direction is shown for clarity.

board with daughter board running a round robin program (*rrobin*). All measurements described in this report have been performed on each of the above configurations.

Some details on the round robin queueing program are in order here, since we want to do some 'back-of-the-envelope' calculations to determine if it is indeed possible to perform round robin queueing

at the T1 rate. First of all, due to requisite checking of hardware fifo status registers and various pointers, it takes a minimum of 18 29000 processor cycles to read data from the daughter board's input fifo or write it to the output fifo. When the additional overhead of managing the data structures is included, it takes (on average) 22.6 cycles to read a byte from the input fifo and store it into memory and 19.7 cycles to move a byte from memory to the output fifo (these figures assume a 64 E quantum, which will be justified later). (These numbers were calculated by counting the instructions in the assembly code.) Translated, this means it does not, in fact, add much overhead to do the round robin queueing. These numbers also show we can read data into the daughter board's memory at 710 KE/s, which is very close to the backplane speed of 800 KE/s; the input fifo on the daughter board should only overflow if a 35.6 KE burst comes in at the full 800 KE/s — this is extremely unlikely in practice.

The quantum for the round robin program was chosen to be 64 E. While testing other quanta may be interesting, time did not permit it. This value has minimal effect on the throughput of the trunk, yet keeps the service waiting time small. Tables 1 and 2 show the maximum throughput and in-memory queueing delays for various quanta (for an explanation of how these numbers were calculated, see section 2). The total delay a message experiences traversing the board, however, will be greater than the value shown in Table 2 because of queueing delay in the input and output fifos of the daughter board.

### 1.5. Network Environment

The computing facilities at Murray Hill are excellent and provide a good environment for network testing. The portion of the network which was used is shown in figure 3, where the squares represent Datakit nodes and the ovals represent hosts. Node 4 is a Hyperkit — next-generation Datakit with a 125 Mb/s backplane. The experimental T1 trunk stretches between Node 3 and Node 6, and all other trunks shown are 8 Mb/s fibers. The nodes are physically close together — no trunk line is longer than a few tens of feet. Hosts Crab, Coma, and Pipe are Vax 8550's, and Tempel is a MIPS R-2000. Note that Pipe has two Datakit interfaces. Network utilization is very low ( < 5% ) [Marshall 89], so there was little interference with our tests. Also, since the rest of the network is composed of 8 Mb/s fibers and backplanes, the T1 trunk is clearly the bottleneck.

## 2. Expected Results

### 2.1. Maximum Throughput

What is the maximum throughput of a T1 trunk? To answer this question, the framing on the T1 trunk must be examined, and we consider the round robin queueing case first. First off, one bit out of every 193 is used for synchronization. Next, because inverted HDLC is used on the T1 line, two one byte flags and two bytes of CRC are added to each frame of data. Also included in the frame are two other bytes of control and protocol discrimination, and the two byte channel number. A frame may only contain data from one virtual circuit. A frame will hold at most 256 bytes of data, and URP control characters require a stuff byte (since only 8 bit bytes are sent over the wire, the 9th bit which discriminates data/control requires a full byte for control characters — the absence of this stuff byte indicates data characters). Thus, in each frame we have 8 bytes of overhead.

A raw T1 line can transmit 1.544 Mb/s * (192/193) / (8 b/B) = 192 KB/s. Given a 64 byte quantum, we can get at most 192 * (64/(64+8)) = 170.7 KB/s through a T1 trunk. Now, to this we must add the URP overhead. If the URP block size is some factor of the quantum (true in our case), and the URP block is full, then the URP trailer will require its own frame. This frame will be 14 bytes long: 8 bytes of frame overhead + 4 bytes of URP trailer + 2 bytes stuffed in front of the BOT and SEQ(i). For a 256 B URP block size, the throughput is 192 KB/s * (256/(256+14+4*8)) = 162.7 KB/s. Similar calculations will yield the rest of the values in Table 1.

In practice it is possible to exceed this maximum throughput. The framing is done by a finite state machine implemented in hardware, and if more data arrives for a given channel before a frame has been transmitted, the new data will be added to the frame (until the frame is full — 256 B). This may happen when only one channel is active on the board; the channel will only get to output one quantum at a time, but those packets will end up next to each other in the output fifo and may be combined.

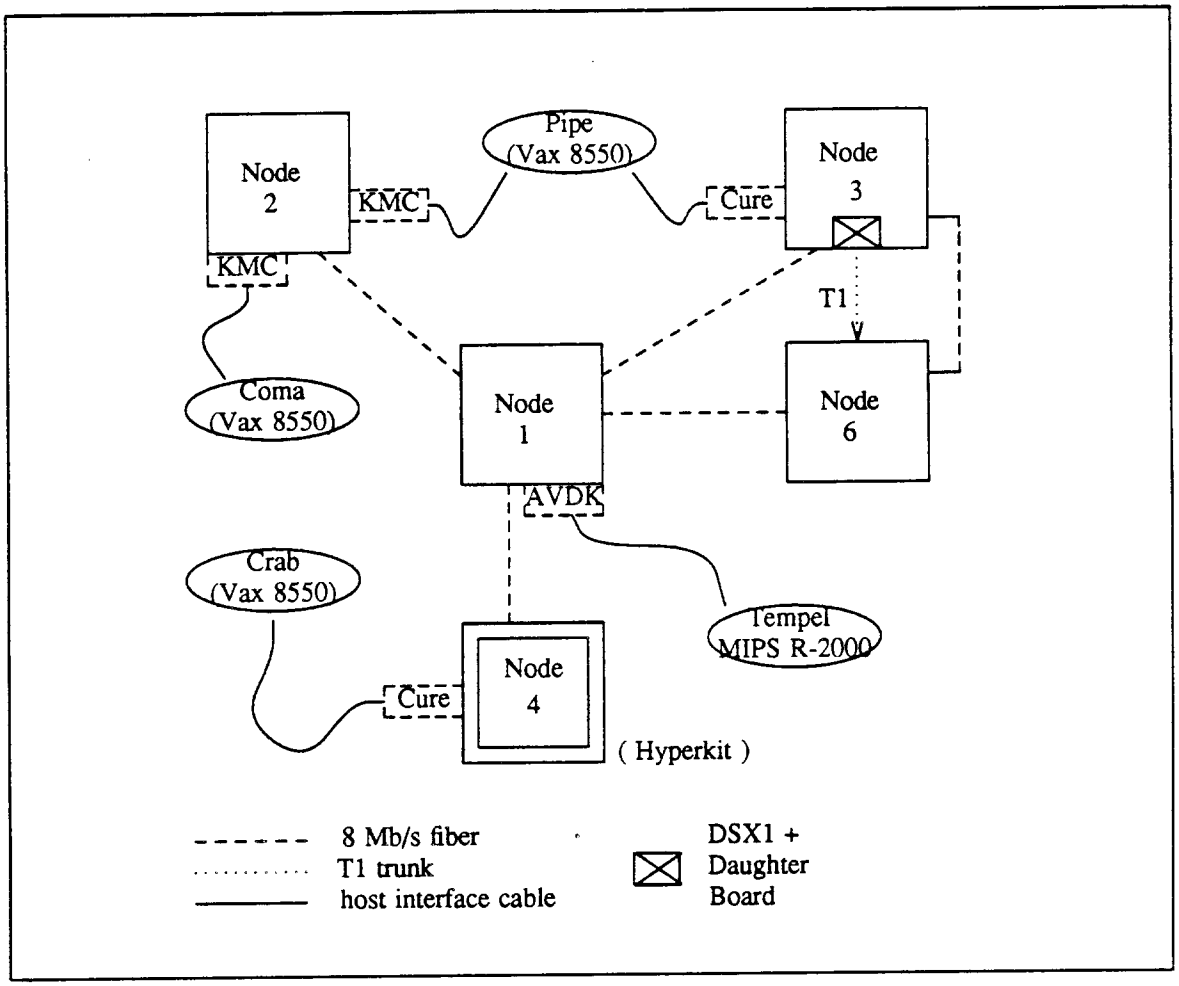


Figure 3 - Portion of Murray Hill Datakit Network Used for Tests

It is difficult to obtain an exact maximum throughput for the hardware and software fifos (*hfifo* and *fifo*) because we cannot determine what the effective quantum will be. The packets are sent to the output state machine as they are received off the backplane: with two bytes of channel number and 16 bytes of data. Most likely, several of these packets will be combined by the output state machine, but it is impossible to determine how many *a priori*. We expect, however, that the throughput should be about the same for fifo queueing as for round robin. In some sense, whether the fifos provide the maximum theoretical throughput is unimportant; the comparison of fifo and round robin throughputs is the most interesting information.

| **Theoretical Round Robin Throughput (KB/s)** | | | | |
|---|---|---|---|---|
| Quantum (bytes) | URP Block Size (bytes) | | | |
| | 256 | 512 | 1024 | ∞ |
| 16 | 123.4 | 125.7 | 126.8 | 128.7 |
| 32 | 147.2 | 150.3 | 151.9 | 154.4 |
| 64 | 162.8 | 166.6 | 168.6 | 170.7 |
| 128 | 171.9 | 176.2 | 178.4 | 180.7 |
| 256 | 176.8 | 181.4 | 183.7 | 186.2 |
| ∞ | 182.8 | 186.9 | 189.4 | 192.0 |

Table 1 - Maximum User Data Throughput for Various Quantums and URP Block Sizes

| Theoretical Round Robin Delay (μs) | | | | |
|---|---|---|---|---|
| Quantum (bytes) | Number of Active Channels | | | |
| | 2 | 4 | 8 | 16 |
| 16 | 39 | 79 | 158 | 315 |
| 32 | 79 | 158 | 315 | 630 |
| 64 | 158 | 315 | 630 | 1260 |
| 128 | 315 | 630 | 1260 | 2521 |
| 256 | 630 | 1260 | 2521 | 5043 |

Table 2 - In-Memory Queueing Delay for Various Quantums and Active Channels

### 2.2. Maximum Interactive Delay

We now consider the maximum delays a short message (defined as a message small enough to be transmitted in one contiguous burst) should encounter on a loaded trunk. First, for round robin queueing, the expected queueing delays are listed in Table 2 as a function of the number of active channels on the daughter board. These values were calculated as follows: it takes 19.7 cycles per byte to transfer data from the daughter board's memory to the output fifo. If there are 2 active channels, the message will have to wait at most 2 * 64 B * 19.7 cycles/B * 62.5 ns/cycles = 158 microseconds (μs) before it is placed in the output fifo. In practice, however, when the trunk is loaded, there will be additional delays due to queueing in the daughter board's input and output fifos. The 4K input fifo can be emptied in 5.6 ms (4 KE/(710 KE/s)) and the 1K output fifo can be emptied in 5.7 ms (1 KE/(176 KE/s)) (since channel numbers are also put into the output fifo and one envelope of user data requires eight bits on the T1 line, the rate of emptying is 193 KB/s * 192/193 * (64+2)/(64+8) * 1 E/1 B = 176 KE/s). Since the input fifo is read until it becomes empty, a good first degree estimate of the maximum round robin interactive delay (for a message of size one quantum or smaller) should be the sum of these two numbers, 11.3 ms.

The maximum interactive delay on a loaded trunk using the 32 KE hardware fifo can be calculated as follows. If we assume the fifo empties to the T1 line at a rate of 176 KE/s (we assume it is about the same as the round robin case), we find the maximum delay (for one packet) is 182 ms (32 KE/(176 KE/s)), assuming the packet is transmitted without errors. In practice the delay should be less than this because this case corresponds to the fifo almost overflowing. A similar calculation for the software fifo yields a maximum delay of 2.8 s (500 KE/(176 KE/s)). This number is not very significant because the software fifo will never fill up in practice. Further, a 2.8 second delay would trigger timeouts, so this fifo is far too big to be a 'sane' fifo.

### 2.3. Hardware Fifo vs. Software Fifo vs. Round Robin

The software and hardware fifos should perform almost identically; the only difference is the amount of buffering available: 32 KE for the hardware fifo and about 500 KE for the software fifo (10 bits require two bytes of storage, and there is 1 MB of memory on the board). Provided no fifo overflows occur (in practice they do not), the performance should be the same in all respects.

The throughputs achieved by each of the disciplines should be equal. The fifos may have higher throughput because the processing delay is shorter (no overhead of the data structures). However, depending on how packets are combined in the output state machine, round robin may perform better because it sends bigger chunks to the output state machine. Which factor will have greater influence cannot be determined *a priori*.

When the trunk is unloaded the interactive delays should be the same, because the round robin program will simply act as a fifo — there will never be more than one active channel. We hope per-channel queueing does not increase the delay for the unloaded case due to the overhead of managing the data structures.

The interactive delay results should only diverge when the load on the trunk is high enough to start building queues on the trunk board. This requires a burst data rate greater than T1 speed. As the queues within the trunk boards build, the interactive delay should rapidly increase for fifo queueing and remain almost constant for round robin queueing (round robin queueing delay only remains approximately constant because we have an upper bound on the number of active channels).

## 3. Methodology

This section will present the general methodology common to all the measurements described in this report. The detailed methodologies for each test are presented in the results section.

All testing was performed by shell scripts (which may be found in Appendix B) on the host Coma (see network configuration in figure 3). The Unix®/Datakit *rx* command was used to start data generating processes on either Crab or Pipe. Bulk traffic used to measure throughput and provide background traffic was generated on Crab by a program called *mpuke* (multiple puke) and transmitted by the CURE to Tempel via the T1 trunk under test (the exact path of the connection was: CURE!Crab -> Node 4 -> Node 1 -> Node 3 -> T1 -> Node 6 -> Node 1 -> AVDK!Tempel). A process on Tempel called *meat* (multiple eat) sank the data as fast as it could. Interactive traffic (i.e. traffic composed of short messages) was generated on Pipe by a program called *ipuke* (interactive puke), transmitted by Pipe's KMC on Node 2, and looped through the network back to its KMC (the exact path: KMC!Pipe -> Node 2 -> Node 1 -> Node 3 -> T1 -> Node 6 -> Node 1 -> Node 2 -> KMC!Pipe). *Ieat* sank the data.

Originally, interactive traffic was going to be sent and received by Pipe's CURE on Node 3 so the connection could be looped directly back from Node 6, thus avoiding the traversal of the rest of the network. However, the standard deviation of the delay measurements was much larger for this set-up than for going from the KMC back to itself. The possibility that the CURE was interfering with itself was considered, and a test was run than sent data from Pipe's CURE to its KMC. This configuration also produced very high standard deviations, so the *ipukes* were routed from the KMC back to the KMC.

Since the rest of the network (backplanes and trunks) runs at 8 Mb/s, the T1 line is the clear bottleneck, despite the bulk traffic's double traversal of Node 1 (Node 1's backplane is less than 50% utilized in this case, so little contention occurs). In any case, delays caused by contention on Node 1's backplane will affect all the queueing strategies equally, and we are most interested in the difference between the strategies.

The following sections will describe *mpuke* and *ipuke* in more detail.

### 3.1. *Mpuke*

*Mpuke* first establishes a Datakit connection with the remote *meat* process, and sends a zero length message to synchronize the two ends. *Meat* reads the zero length message and sends a zero length message back. Next, *mpuke* sends *meat* the number of *iterations* it will perform, and *meat* sends back a zero length acknowledgement. *Mpuke* then does a system call to set the URP window size for the conversation (note that the sender may set the window size independently of the receiver). Now the test is ready to begin. *Mpuke* reads the time from a device called */dev/fineclock* which gives 1 µs resolution (a read of the clock takes approximately 79 µs on a VAX 8550), and begins to transmit 1 MB of data as fast as it can. When it has sent all the data, it sends a zero length message to *meat*, and *meat* replies with the number of characters read. *Mpuke* then reads the clock again, and computes the throughput by: throughput = bytes sent / (time_end − time_start). The time to read the clock is insignificant compared to the time to send the data, so it is ignored. The 1 MB of data is sent *iterations* times, and the throughput is calculated each time. Very little data is sent from *meat* to *mpuke*, so there is very little delay when *meat* sends the length to *mpuke*.

### 3.2. *Ipuke*

*Ipuke* is similar to *mpuke* in that it sets up a connection to *ieat*, synchronizes the two ends and sends the *iterations*. The major difference is that *ipuke* measures the delay of a single message. The

® Unix is a trademark of AT&T Bell Laboratories.

sequence of events that occur in *ipuke* after the connection is established is as follows: *ipuke* reads */dev/fineclock*, sends a message of some specified size, and immediately reads the length from the remote *ieat*. When the read() returns, the clock is again read, and the round trip time is calculated. The *ieat* process reads a message from the network, and writes the length of the message back. This is repeated *iterations* number of times.

### 3.3. Miscellaneous

The tests were all run during unsocial hours of the evening when the test programs were usually the only active processes on the machines used. During the course of the tests the load on the hosts is monitored, and if the load varies substantially, the tests are performed again. The transmitter status is also monitored using the Unix/Datakit *dkstat* command. Dkstat lists how many bytes were transmitted and received by the host interface in the last 10 seconds and tells how many retransmissions occurred. Retransmissions occur whenever data is lost, such as when buffer overruns occur. If the number of retransmissions is excessive or if the interface is being heavily used by another process, the test is repeated.

## 4. Results

This section presents the results of three experiments performed using each of the queueing disciplines described above. The experiments measure the throughput achieved on the T1 trunk for each queueing discipline, the time to traverse the unloaded trunk for variously-sized messages, and the time those message sizes took to traverse a loaded trunk. Each subsection will cover one of the tests, and will describe the specific methodology used, the results obtained, and the conclusions that were drawn.

### 4.1. Throughput

The throughput each queueing discipline achieved was measured by doing *mpukes* from Crab's CURE to Tempel.

#### 4.1.1. Methodology

The variables studied in this test are the number of simultaneous *mpuke/meat* pairs and the URP window size. Either one or four *mpukes* are remotely started on Crab from Coma via a shell script (see Appendix B for shell scripts used in the tests). URP block sizes of 256, 512, and 1024 are considered (256 is the default), with either 3 or 7 outstanding blocks (3 is the default, 7 is the maximum). Each *mpuke* measures the throughput of transmitting 1 MB of data ten times, and the first and last measurements are discarded. The resulting data has a very low standard deviation, so the 8 data points are sufficient to obtain accurate results. The shell script records the load of the hosts involved every five seconds using the Unix *load* command (which lists the average number of jobs in the run queue for the past one, five and ten minutes) and records the network interface status every ten seconds using *dkstat*. If the load varies significantly during a test, that test is repeated.

#### 4.1.2. Results

Figure 4 shows the throughput of one conversation, and figure 5 shows the throughput of 4 simultaneous conversations. Both graphs have the same scale for comparison purposes. It is clear that four simultaneous conversations achieve a higher aggregate throughput that one conversation. The reason for this is that when only one conversation is going, some amount of time is spent waiting for acknowledgements, so the bottleneck resource in the path (whatever it may be) is idle. Another factor is pipelining of processes: since the host handles the URP protocol processing, and since the CURE uses DMA, data from one conversation could be transmitted while data from another conversation is getting URP processed. Several processes may be pipelined to achieve some simultaneity so the throughput out of the host interface is higher.

For one conversation, the plain DSX1 board with the hardware fifo performs about the same as when the daughter board is running the round robin scheduler, but the software fifo lags behind. This lag is probably due to the way the software fifo transfers the data: priority is always given to filling the

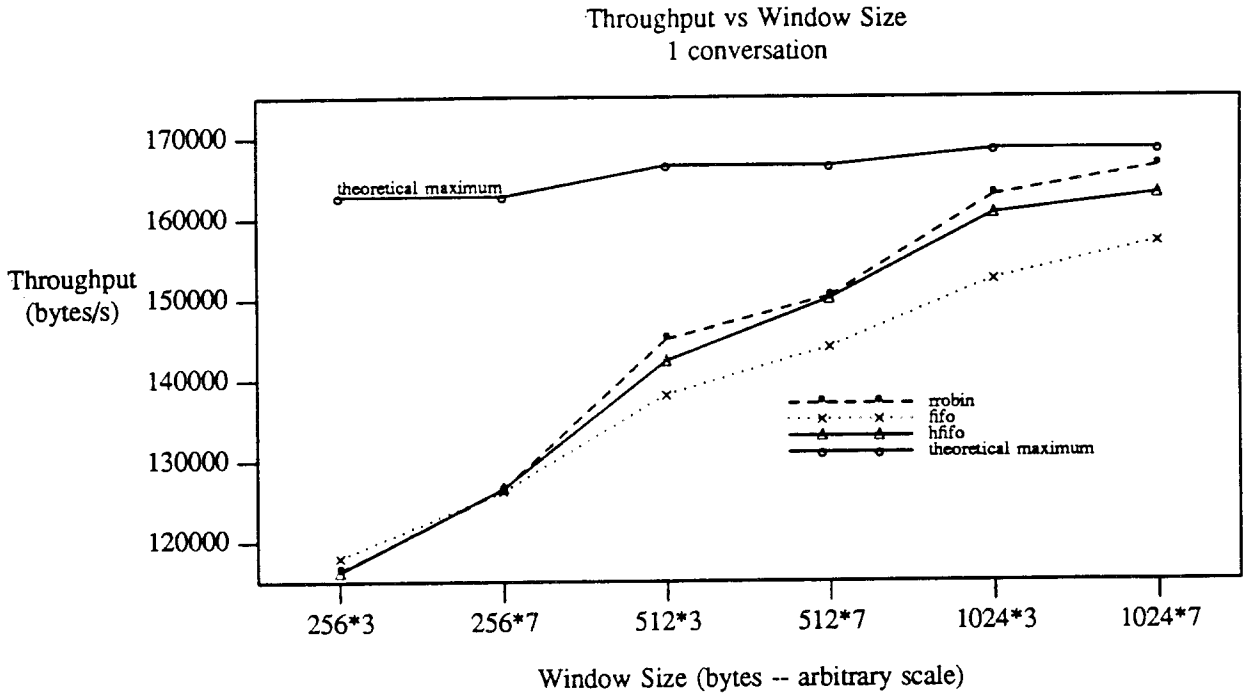


Figure 4 - T1 Trunk Throughput for a Single *Mpuke/Meat* Pair
As a Function of Queueing Discipline

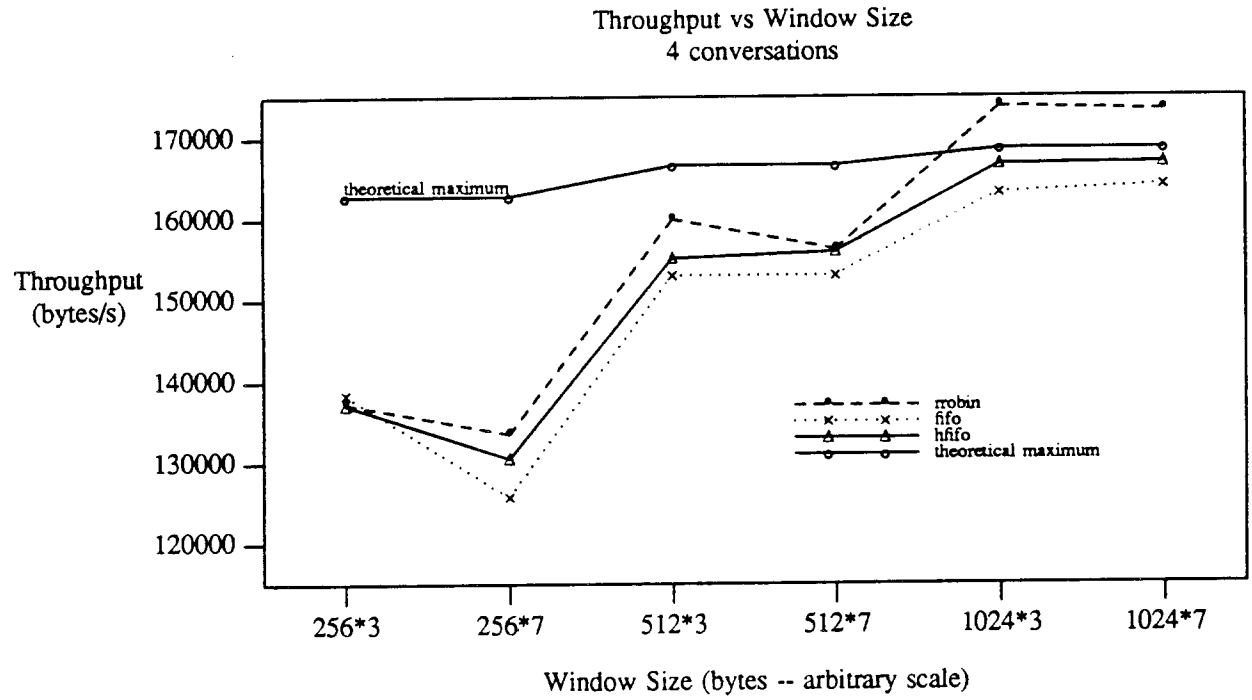


Figure 5 - T1 Trunk Throughput for Four *Mpuke/Meat* Pairs
As a Function of Queueing Discipline

output fifo, so unless the output fifo fills up, data is simply copied from the input fifo to the output fifo and no buffering takes place. This impacts the amount of data the output state machine may combine in one LAPD frame: a frame may contain very little data because the frame is shipped out before more data for that channel arrives at the state machine. This framing algorithm also may explain why the round robin scheduler achieves higher throughput than the fifos and higher throughput than the theoretical maximum for four simultaneous conversations (see figure 5). Each chunk the round robin scheduler passes to the output state machine contains 64 bytes of data (and two bytes of channel number), so at least 64 B will be put into each LAPD frame. The fifos only pass 16 bytes of a data per-channel number, so smaller frames may be transmitted. Intuitively, it seems more likely that two or four 64 B frames would be combined than eight or sixteen 16 B frames, so the round robin scheduler most likely gets bigger LAPD frames out on the T1 line than the fifos. Round robin achieves higher throughput because larger LAPD frames are sent, thus lowering the overhead.

In figure 5, the curves are fairly flat across each block size as the number of outstanding blocks increases from three to seven. This indicates that the throughput primarily depends on the URP block size and not the URP window size. In fact, when seven blocks are outstanding, the throughput decreases noticeably for a block size of 256. The cause of this is unknown. The reason the number of packets outstanding does not affect the throughput is easier to explain: given the short physical lengths of the trunks, three blocks is enough data to fill the pipe (very little buffering occurs along the path because of the low utilization of the fiber trunks and backplanes). Increasing the number of blocks would tend to increase the throughput if the propagation delays were longer.

#### 4.1.3. Discussion

The conclusions we draw from this experiment are that (1) round robin gives a little better throughput, and (2) the throughput in this network environment depends mostly on the URP block size. Thus, we find the overhead of round robin queueing does not adversely affect throughput as we may have expected; in fact, round robin achieves higher throughput because larger chunks of data are transmitted over the T1 trunk.

### 4.2. Interactive Delay — Unloaded Trunk

The interactive delay of the three configurations is measured by running an *ipuke* on Pipe which transmits and receives data on its KMC.

#### 4.2.1. Methodology

The issue under investigation in this experiment is how the delay through the unloaded trunk is affected by the size of the message traversing the trunk. Fourteen message sizes are considered in the tests: 1, 2, 12, 13, 60, 61, 64, 65, 256, 257, 511, 512, 768, and 769 bytes. The 1 and 2 byte messages correspond to keyboard input. A 12 byte message with a 4 byte URP trailer will require one 16 byte backplane packet; a 13 byte message will require two backplane packets. Similarly, a 60 byte message with a URP trailer will pass through the round robin program in one 64 B quantum; a 61 B message requires two quanta. In the Version 9 Unix kernel running on Pipe and Crab, system writes are put into *stream blocks* of size 64 B or 1024 B. A 64 byte system write requires one stream block, and 65 byte write requires two. A 511 B write is put into 8 64 B stream blocks which are chained together. A 512 B write is put into a 1024 B stream block. The default URP window size is 768 B, so a message of size 769 B requires two URP windows.

The number of individual measurements to use to compute an average is a key decision in any experiment. The number must be large enough to yield accurate results, but low enough to allow the tests to be conducted in a reasonable amount of time. To determine this number, messages of the above sizes were sent though the round robin configuration 100, 200, 300, and 500 times. The standard deviations of these tests are shown in figure 6. This figure is somewhat disturbing because 500 iterations shows the highest standard deviation. The cause of this is unclear, but even for 500 iterations the mean to standard deviation ratios are fairly high. Three hundred iterations was chosen because the standard

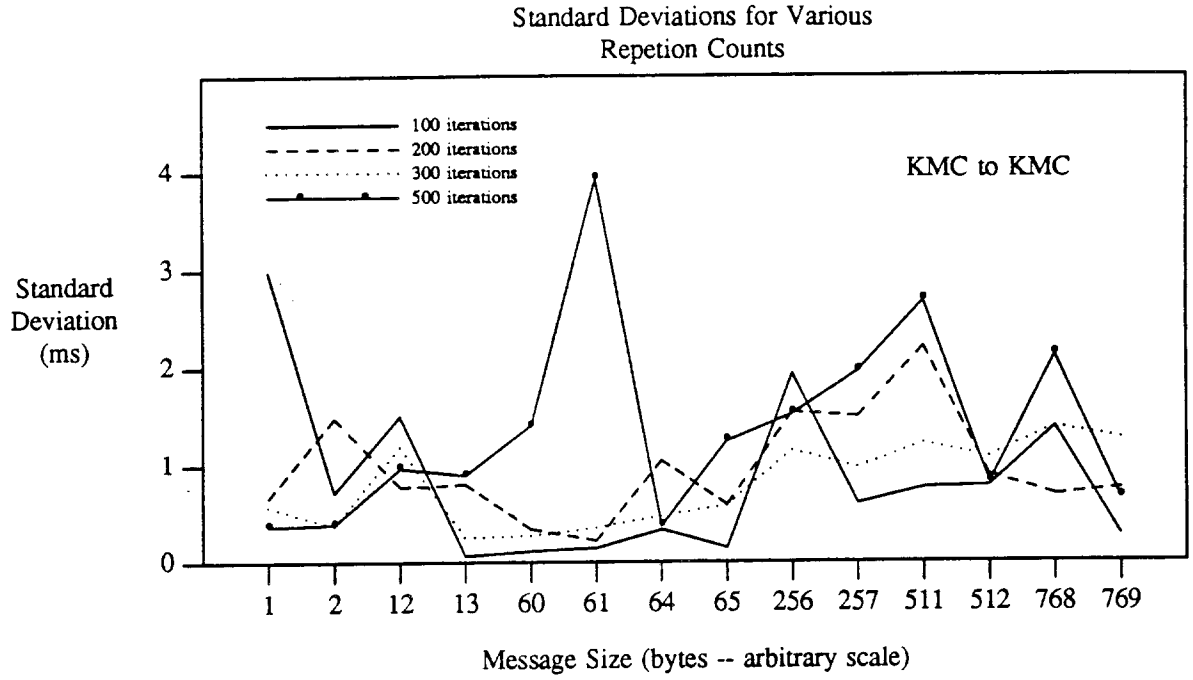


Figure 6 - Standard Deviation of Delay for Various Message Sizes and Various Repetition Counts — KMC to KMC

deviation was consistently low, and the time required to run the tests was reasonably short. The mean to standard deviation ratios from the actual measurements are shown in figure 10.

As mentioned earlier, sending data from Pipe's CURE to Pipe's KMC was considered. This idea was dismissed because of the results in figure 7. The standard deviation of the delays was extremely high (compared to the mean) for messages between two and 256 bytes in length. The exact cause of this is unknown; however, the Datakit console on Node 3 reported parity errors received by the host interface. Why the deviations decrease for message lengths of 256 and up is unknown.

We are primarily interested in the minimum values for the delay, because these values best characterize the delay experienced by a message in an idle network with unloaded source and destination machines and because minimum values have smaller standard deviations. A clear peak is evident at the low end of the delay scale in all the data collected, such as the data pictured in figure 8. Given the experimental error involved here, the average of this peak should be sufficiently close to the minimum that it can be used as the minimum. Due to the large number of data files, some mechanized method of finding the peak is desired. The following algorithm was applied to six of the data sets and was found to find the average of the peak.

First, the mean (x1) and standard deviation (s1) of all the data points (except the first ten and last ten measurements which are eliminated because of possible start-up and ending transients) are found using an *awk* [Aho 87] script. Then the mean (x2) and deviation (s2) are again found for the data, but all points greater than x1 + s1 are eliminated (in practice, points are never below x1 - s1, so this case is not considered). This is repeated a second time, but all points above x2 + s2 are eliminated, and we end up with x3 and s3. This final mean, x3 is used as the minimum. The mean to deviation ratio x3/s3 that results is much greater than 10 (see figure 10). Appendix A contains a summary of the actual results of the measurements, including the number of points eliminated, the final threshold (x2 + s2) used, and the mean to deviation ratio (x3/s3). In figure 8, the threshold used and the calculated average are shown; the algorithm worked very well for this case.

Intuition is the main justification of this algorithm. It should converge to the average of the numbers around the peak, and in practice it did for the six test cases. Further, there are two main rea-

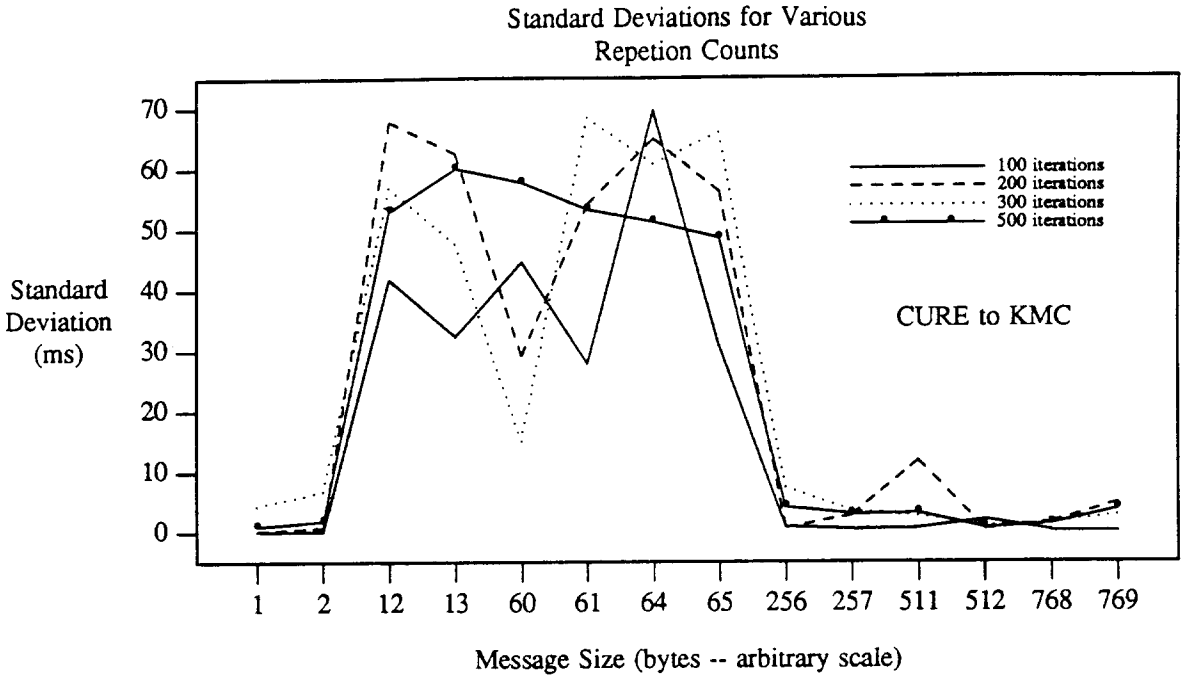


Figure 7 - Standard Deviation of Delay for Various Message Sizes and Various Repetition Counts — CURE to KMC

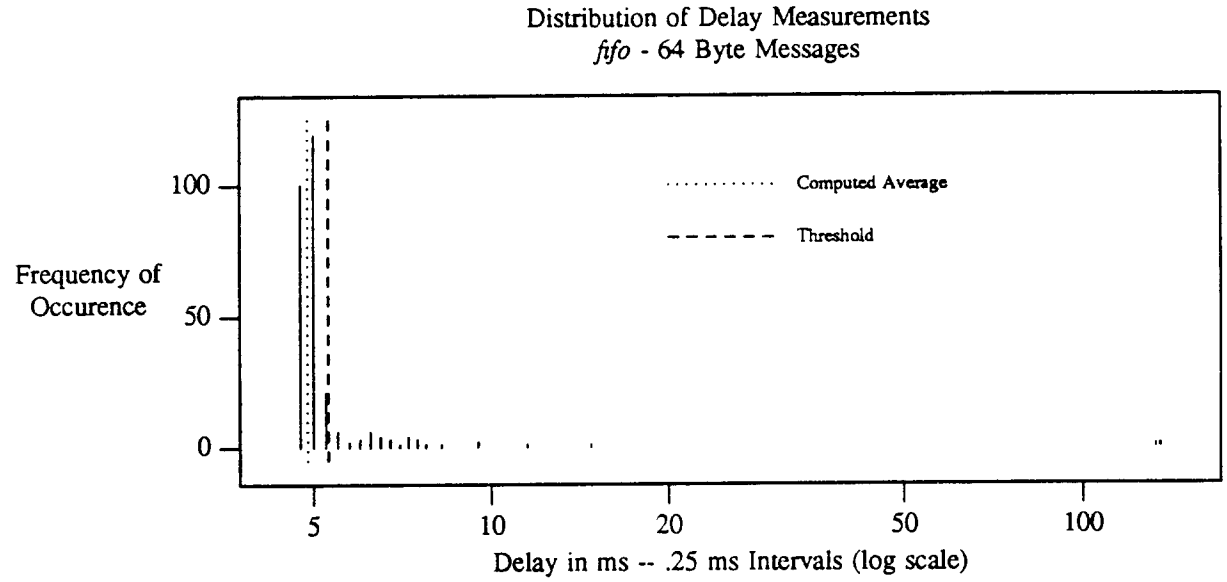


Figure 8 - Sample Distribution of Message Delay Times
(*fifo* — 64 B Messages)

sons why we measure the delay of an unloaded trunk. First, we want to verify that per-channel queueing does not increase the message delay compared to fifo queueing. If we treat the data for both queueing disciplines the same, then this purpose will be fulfilled. Second, we want to gather data to compare with the case of the trunk being loaded. As will be seen when the loaded trunk measurements are presented, the message delay for the fifo case increases 10 to 25 fold over the unloaded case. Thus, if this algo-

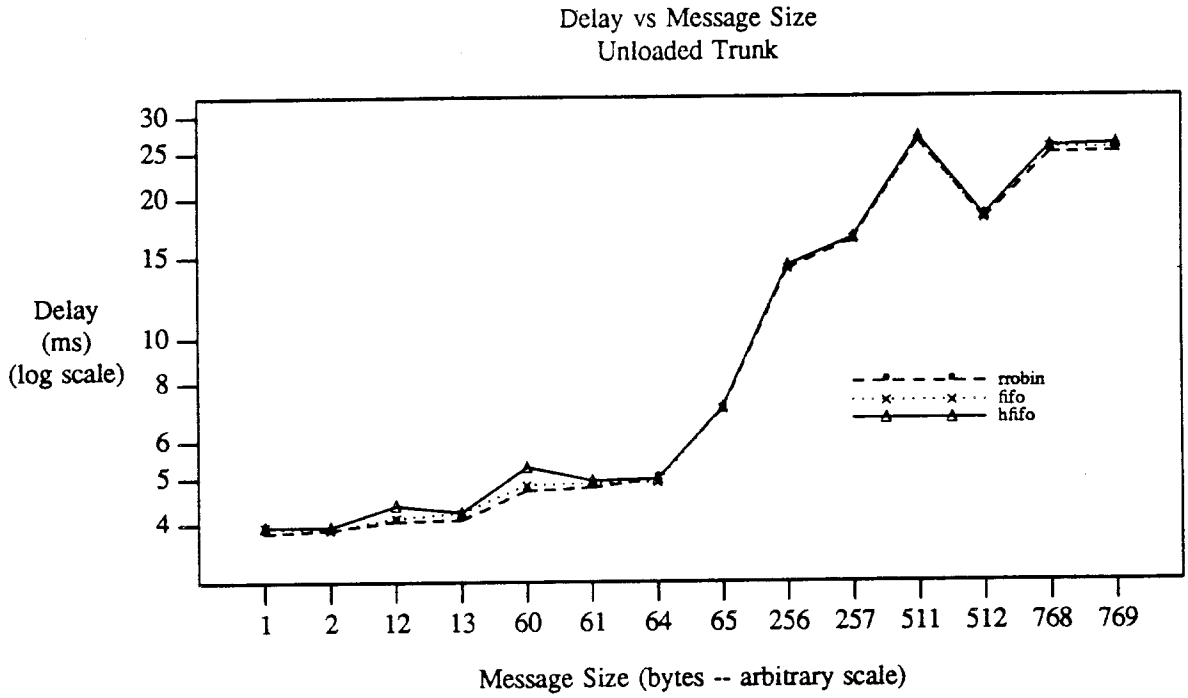


Figure 9 - Message Delay on an Unloaded T1 Trunk

rithm gets within a few percent of the true minimum, this is sufficient resolution to compare it to the loaded trunk case.

Originally we considered having several simultaneous *ipuke/ieat* pairs such as we had for the throughput tests, but the question must be asked, what would this tell us? Not much. Recall that a channel is only considered *active* when it has data in the SRAM on the board. Since it takes a 1 B message about 4 ms to traverse from *ipuke* to *ieat* and back, and since the message is only on the daughter board for about 50 µs, there will never be more than the one active channel on the board (particularly since all the packets are coming from one transmitter and thus are serialized), and no queueing will occur. Thus, the 'simultaneous' conversations would not be simultaneous, and no interesting information would be obtained.

### 4.2.2. Results

Figure 9 shows the results of these measurements, and figure 10 shows the mean to standard deviation ratios for each of the disciplines. We see from the large mean to deviation ratios that this data is not very spread out, especially considering the minimum-finding algorithm mentioned above only eliminated on the order of 40 points out of the possible 280. This is what we expected — if the load on the network and hosts is low, there should be very little variation. It is clear from figure 9 that all of the queueing disciplines performed almost identically. This, too, is expected; the round robin scheduler is just a glorified fifo when only one conversation is active.

We now see the factors that affect the delay of a message. First off, a message that requires two backplane packets takes no longer than a message that only requires one — there is no difference in delay from 12 bytes to 13 bytes. It also takes no longer for the round robin program to process two quanta (61 B) of data instead of one (60 B). We do, however, see a fairly significant 2 ms jump from a 64 byte message to a 65 byte message (from 5 ms to 7 ms). Does it take 2 ms to allocate another stream block? No. This is an artifact of the way the host and the KMC interact. The host only passes one stream block at a time to the KMC, and passing each requires one interrupt. The KMC performs the URP protocol processing, which in this case means adding the URP trailer to the message. In the case

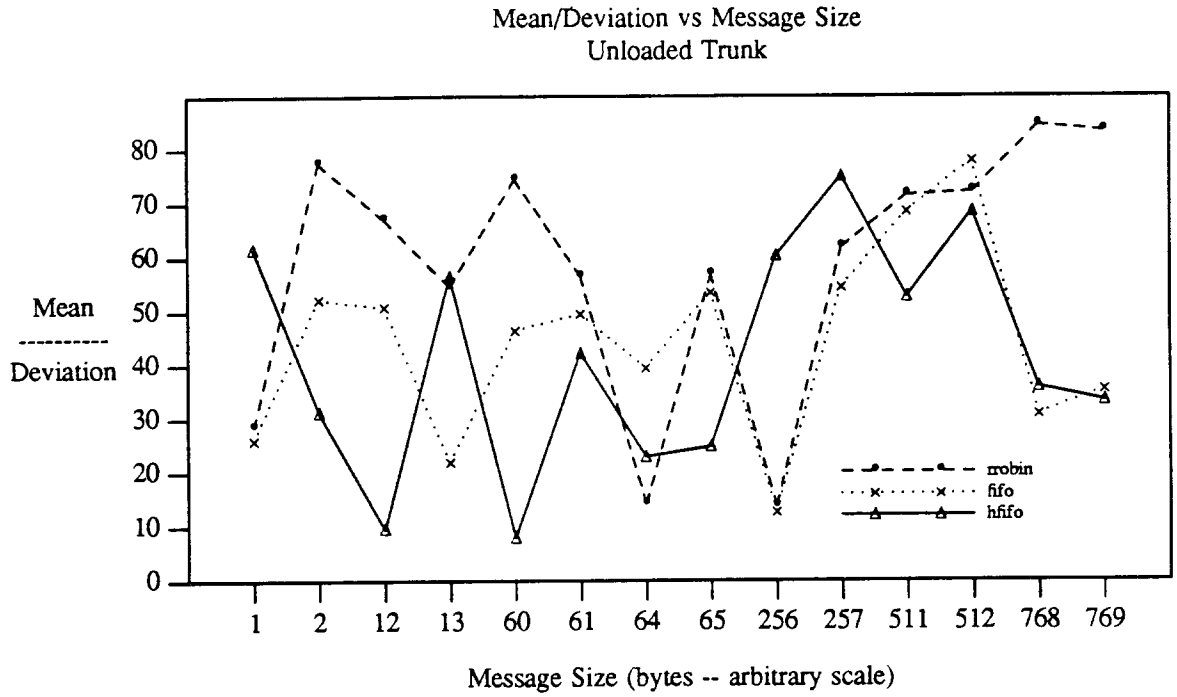


Figure 10 - Mean to Standard Deviation Ratio of Message Delays

of a 64 B message, the host passes the KMC one stream block, and the KMC sends out one burst containing the data and the URP trailer. When a 65 B message is to be transferred, the host passes the KMC the first 64 B in a stream block. The KMC sends these bytes out in one burst, and tells the host it is ready for more data. The host then passes the KMC a second stream block containing the last data byte, and the KMC tacks on a URP trailer and sends this second burst out. Two milliseconds is a long time to get the next stream block for transmission, and the appropriate persons at Bell Laboratories were notified about this.

We see another 2 ms increase between a 256 B message to a 257 B message which is an artifact of this same interaction (a 256 B message requires 4 stream blocks, and a 257 B message requires 5). The 256 and 257 B messages were supposed to show how multiple URP blocks affected the latency, but since there is no increase between 768 B and 769 B (where 3 and 4 URP blocks are required, respectively), we assume the extra 2 ms is due to the extra stream block required. Note that messages of size 768 B and 769 B are placed in 1 KB stream blocks. The regularity of this 2 ms phenomena leads us to the following postulate: it takes 5 ms to send out one stream block, and 2 ms per additional stream block, plus the transmit time over the T1 line. The 5 ms term accounts for the delays experienced in the network plus the allocation of the first stream block. According to this postulate, we expect a 256 B message to experience a delay of 5 ms + 4 stream blocks * 2 ms/stream block + 256 B/ 176 KB/s = 14.5 ms. The actual delay is 14 ms, which is very close to our prediction. Similarly, a 257 B message takes 16 ms, where 16.5 ms is predicted. The results are also close for a 511 B message which takes 26.4 ms in practice, and 23.9 ms in theory. There is a marked decrease in delay from 511 B to 512 B (26.4 ms to 18 ms) because a 1 KB stream block is allocated for the 512 B message.

### 4.2.3. Discussion

Two conclusions result from this experiment: (1) The primary factor affecting the delay of a message over the unloaded trunk is the number of stream blocks which must be allocated for that message, and (2) All the queueing disciplines perform identically.

### 4.3. Interactive Delay — Loaded Trunk

Now we examine the delays experienced by variously-sized messages with background traffic present on the T1 trunk. Crab generated the background traffic with four simultaneous *mpukes* to Tempel, and Pipe sent *ipuke* traffic from its KMC back to its KMC.

#### 4.3.1. Methodology

The tests performed in this experiment are identical to the ones performed in the unloaded trunk case. The same message sizes and number of iterations are used. The only difference between these tests and those above is that here bulk background traffic is also sent across the T1 trunk. The background traffic is generated by four simultaneous *mpukes* from Crab to Tempel. The variable under investigation is the effect of the URP window size of the background traffic on the interactive delay (and, of course, the queueing discipline used). Note that varying the URP window effectively regulates the trunk utilization. Ideally, we would like to be able to set the trunk utilization to some specific value, particularly since the disciplines do not all have the same throughput for a given window size. There is no way to accomplish this, however, and setting the URP window is the best we can do.

The URP window sizes considered are 256 * 3, 512 * 6, 1024 * 3, and 1024 * 7 bytes. These windows were chosen to give a good range of queue build-up. As mentioned, 256 * 3 is the default, but it gives sufficiently small throughput that we do not expect to see much divergence among the disciplines. The 512 * 6 and 1024 * 3 windows are the same size in bytes, and should provide some insight as to whether the block size is the critical parameter affecting the delay or if delay depends more on the number of outstanding blocks allowed on the other virtual circuits. The 1024 * 7 window is the largest URP window allowed in practice; specifying a large block size will simply result in getting a 1024 B block.

The shell script that ran these tests first started the Crab to Tempel *mpukes* with one of the given window sizes, and then waited until each of the *mpukes* had sent 1 MB of data across the trunk before starting the *ipuke* process. This ensured that the tests would not be affected by start-up transients.

In this testing, we are most concerned with the average delay experienced by a message in traversing the network. This again raises the issue of eliminating outliers from the data, and again an automated method is desired due to the large amount of data acquired. An algorithm somewhat similar to that used to find the minimums is used. First, the mean and deviation (x1 and s1, respectively) of all the data points for a given message size are found (again, we eliminate the first and last ten points to remove startup and ending transients). Then the mean and deviation are again found (x2 and s2) but points greater that x1 + 8 * s1 are eliminated. In the actual application of this algorithm, at most three points out of the 280 collected were eliminated. The small number of points eliminated indicates that spread of the random variable values is smaller than the threshold of x1 + 8 * s1, and thus, this method is justified. Intuitively, if the load on the hosts involved and on the network is low, which indeed was the case, the delays should be fairly constant.

#### 4.3.2. Results

##### 4.3.2.1. Hardware Fifo (*hfifo*)

Figure 11 shows how the delay for various message sizes increased as the background traffic's window size increased. The most obvious feature of the graph is the significant increase in the delay as the background traffic increases. For the 1024 * 7 background window, the delay increased forty-fold for small packets and fifty-fold for large packets over the unloaded trunk case. The shape of each of the curves is similar when plotted on a log y scale, indicating that there is a multiplicative increase from window to window.

The relative positions of the curves show what factors affect the message delay. Both the URP block size and the number of outstanding blocks affect the delay, but the block size has a greater effect. Though the window sizes 512 * 6 and 1024 * 3 are the same, the effect of the larger block size dominates, and the delay for the 1024 * 3 window is about two times greater (for the larger message sizes).

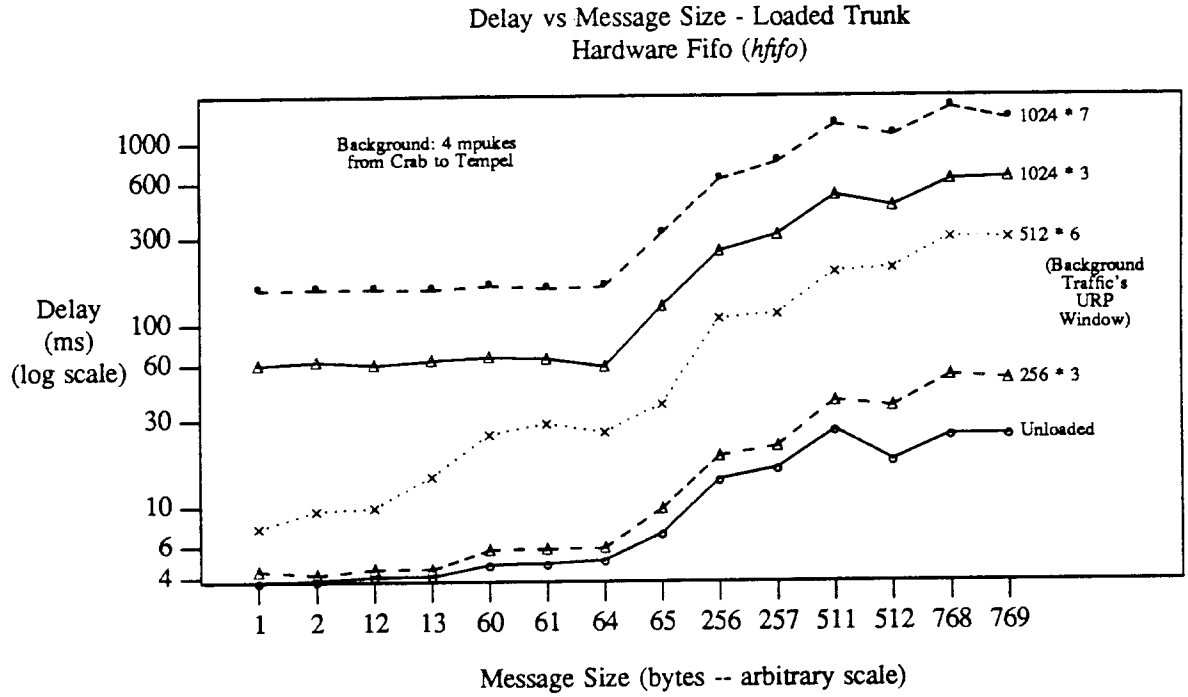


Figure 11 - Message Delay on a Loaded Trunk for Various URP Windows
Hardware Fifo Configuration

The number of outstanding blocks also plays a factor though, because the delay more than doubles (it increases by a factor of about 7/3) when the number of outstanding blocks increases from 3 to 7. The window size of 256 * 3 does not significantly increase the delay with respect to the unloaded trunk case, which is expected because the T1 trunk is not nearly fully utilized by this window.

The host interface artifacts also appear in the graph, particularly for the two largest windows. The delay is constant for all messages between 1 and 64 B, but then doubles for a message 65 B long; messages 64 B or smaller are encapsulated in one burst, but a message of 65 B is transmitted in two bursts. The delay is dominated by the queueing delay on the trunk board, so two packets will take twice as long as one. We notice similar behavior for 256 and 257 B messages. The delay of a 256 B message is about four times the delay of a 64 B message, and the delay of a 257 B message is five times the delay of a 64 B message, just as theory would predict. Notice, too, that there is no perceptible difference between sending a 768 B message and sending a 769 B message, so extra URP blocks do not add significant delay. The 512 * 6 window shows some interesting behavior; the delay curve starts out low, then increases for larger message sizes to about half the delay of the 1024 * 3 window. Most likely the background traffic was not quite enough to saturate the T1 line, but the addition of the *ipuke* traffic brought the total throughput closer to saturation, so the queues grew larger.

We now consider some 'back of the envelope' calculations for messages which traversed the trunk in one packet (i.e. 1 B to 64 B messages). With the 1024 * 7 window, a single packet took 157 ms to complete its loop from Pipe to Pipe. If we assume the packet spent the whole time waiting in the T1 board's fifo (we could subtract the 5 ms delay a packet experiences in the unloaded case, but it would not significantly affect the calculation), using the fifo drain time calculated in section 2.2, we find the packet was queued behind (157 ms/182 ms) * 32 KB = 27.6 KB. This is almost exactly the sum of the background window sizes, 28 KB (= 1024 * 7 * 4). Thus, a packet must wait behind a full window of each of the background conversations. Clearly the T1 trunk is fully saturated. A similar calculation for the 1024 * 3 background window (which causes a delay of 60 ms) yields 10.5 KB. This is slightly less than the 12 KB aggregate background window, indicating that the trunk is not quite fully saturated.

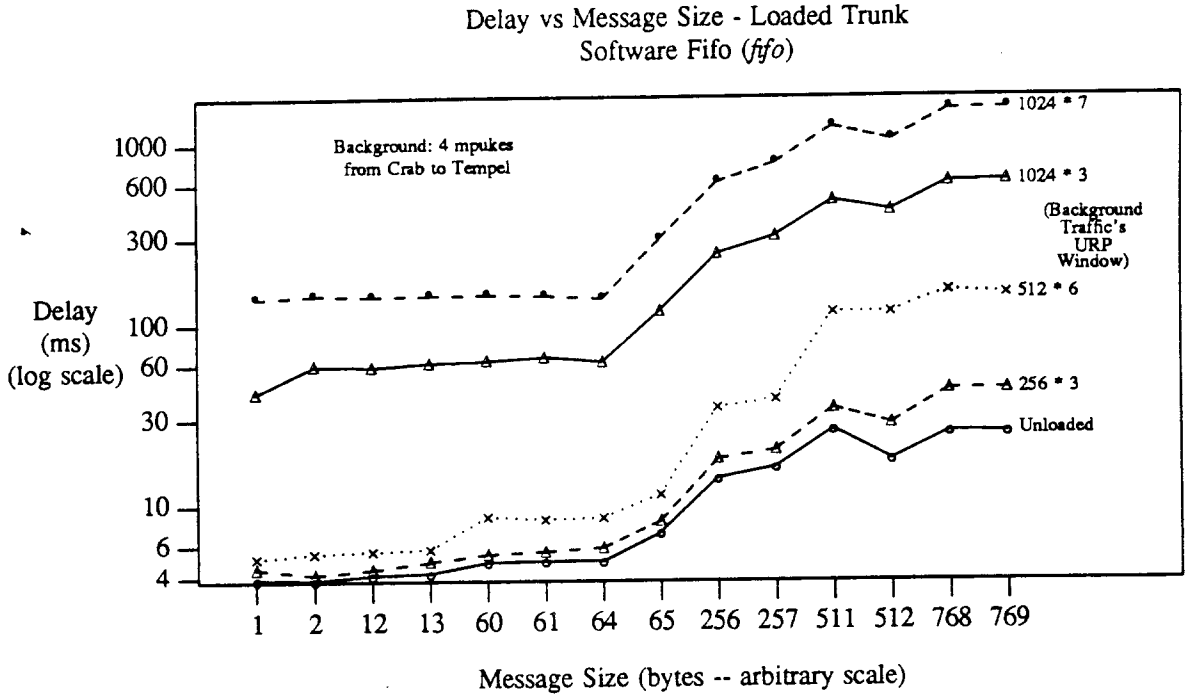


Figure 12 - Message Delay on a Loaded Trunk for Various URP Windows
Software Fifo Configuration

This calculation for a 64 B message with a 512 * 6 background shows 3.6 KB are in the fifo ahead of the message (this time the 5 ms delay for the unloaded case was subtracted).

In short, the results obtained for the hardware fifo configuration are exactly as expected: the delay does not significantly increase for a 256 * 3 window, but does steadily increase with the larger window sizes.

#### 4.3.2.2. Software Fifo (*fifo*)

Now the results for the software fifo are considered. The graph in figure 12 looks rather familiar — it is almost the same as the graph for the hardware fifo. We find the same host interface artifacts, and the same relationships hold between the individual curves (figures 14 through 17 show the side by side comparisons of the queueing strategies for each window size). Because of the similarities, these results will not be discussed in depth; only the causes of any differences will be discussed.

The major difference between the two fifo configurations is the shape of the 512 * 6 window curve. As shown in figure 15, the software fifo provides shorter delays for this window size. The delay curve for the 1024 * 7 window is also slightly lower than the hardware fifo's curve (157 ms vs. 145 ms for one packet messages). The reason for these differences is most likely found in figure 5 — the throughput of the software fifo is lower than the throughput of the hardware fifo. This will particularly make a difference for the 512 * 6 window, where the larger messages in the hardware fifo tests are apparently just enough to build up the queue. This difference explained, the round robin results are considered next.

#### 4.3.2.3. Round Robin (*rrobin*)

The first thing to notice about figure 13, which contains the results for the round robin queueing discipline, is the scale for the delay: it is 1/10 the scale of the fifo graphs. This graph is also 'bumpier,' which is primarily a result of the smaller scale used, but is also affected by the small mean to standard deviation ratios for these data points (these ratios are typically around 1; see Appendix A for a complete

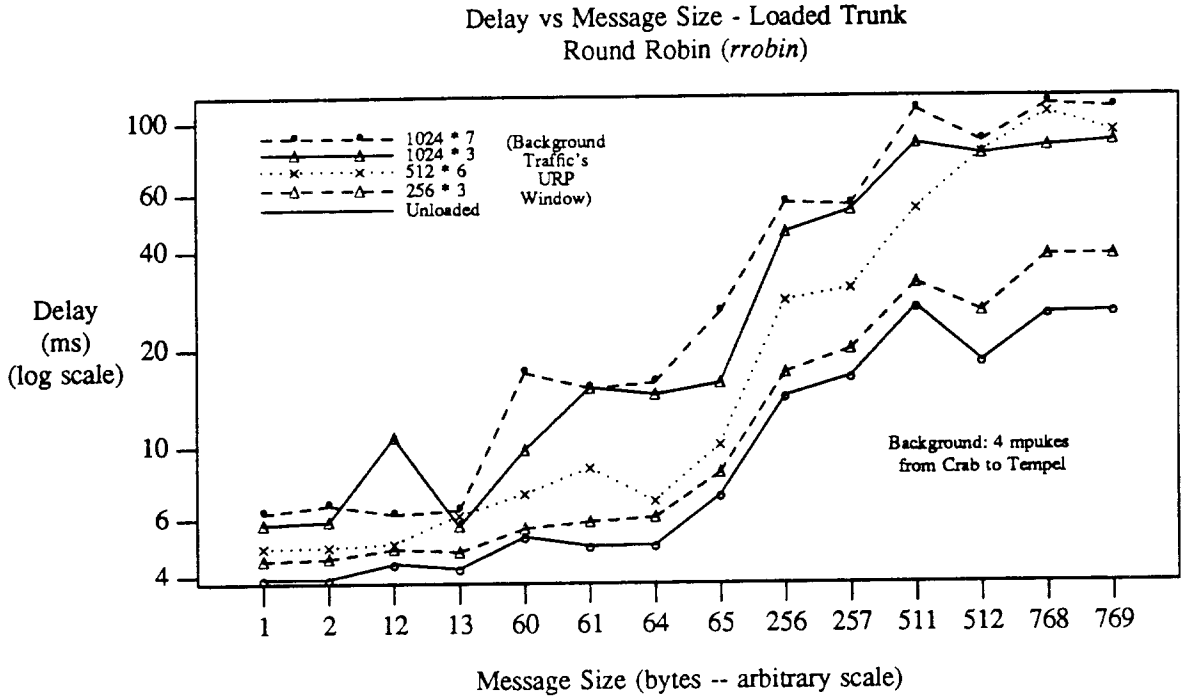


Figure 13 - Message Delay on a Loaded Trunk for Various URP Windows
Round Robin Configuration

listing of the results, including the mean to deviation ratios). The standard deviations are smaller than those of the fifo results, but, due to the smaller mean delay times, the mean to deviation ratio is small. However, the data does show the type of curves expected; the delay for very short messages (1 B to 13 B) only increases by at most 2.8 ms (for a 2 B message: 3.9 ms unloaded to 6.6 ms for a 1024 * 7 window (this excludes the odd spike at 12 B)) over the unloaded trunk case, and the delay for larger messages also does not increase as much as for the fifo cases. Some increase in delay for round robin queueing is expected due to queueing delays at the input and output fifos to the daughter board.

The delays measured for round robin queueing are lower than the calculated maximum delay of 11.3 ms (see section 2.2) (except for the 60 B packet with a background window of 1024 * 7, which could be due to the load on Pipe), which is expected since the input fifo to the daughter board should not fill up (data can be read from the input fifo at much greater than T1 rates). This is the reason for doing per-channel queueing: short messages experience very little delay in the presence of bulk background traffic. Unfortunately, several anomalies occurred during the testing, which calls the validity of the data into question. A discussion of these anomalies follows.

The first problem with the tests was the load on Pipe. At various points during the tests, the load (which is the average number of jobs in the run queue over the last minute) increased to as high as 3 or 4, though it remained below 1 for the majority of the testing. The load on Pipe is the probable cause for the spike in delay at 12 B for the 1024 * 3 window. Some of the tests were attempted again, but the load on Pipe continued to fluctuate, and the re-trials were postponed.

Another, more critical, problem also hampered the testing. Due to some anomaly, when a large quantity of background traffic is being sent over the trunk, such as the four *mpukes* used in this test, and more traffic from another conversation is added, such as the *ipuke*, some of the data is lost somewhere, and URP retransmissions occur. This means that the user level throughput is cut about in half. This would seem to lower the utilization of the T1 trunk, but this is not necessarily the case. When packets arrive at the receiver, their length is compared to the length in the URP trailer, and if the values are different, a REJ(i) control character is sent to the transmitter, where i is the sequence number of the

rejected block. Since URP is a 'Go-Back-N' protocol with respect to retransmissions, subsequent blocks are also rejected. If the channel number is corrupted, the receiver cannot send a REJ(i) and transmission would be halted until a timeout occurs, but this should be rare. Since the return direction is unloaded, the REJ(i) should reach the transmitter quickly, and the transmitter will re-send the corrupted block. If the REJ(i) arrives quickly enough, the transmitter will be able to continuously send data, so, although the user level throughput will decrease, the utilization of the trunk should remain high. It is difficult to verify this in practice however, because it requires either examining the bits on the T1 line or looking at the internals of the transmitter, neither of which is particularly convenient. It should be noted that no retransmissions occurred for messages sized 1 B to 13 B.

Where is this data lost? Typically, data losses indicate a buffer overrun somewhere along the path of the connection. This was investigated by starting up four *mpukes* and an *ipuke* and monitoring the interface status with *dkstat*. When retransmissions were regularly occurring, the Datakit consoles of each node were examined, and the status of each module along the path of the virtual circuits was examined. No overflows were occurring. The next place considered was the round robin program. The program was thoroughly inspected by two persons, and it seemed to be in order. A bug affecting something else was found in the C compiler for the 29000, so this was the next place to be investigated. The assembly code was examined, and it, too, seemed to be in order. The final place this bug could be is in the hardware on the DSX1 board or the daughter board. This is the most likely suspect for this anomaly, since the hardware is fairly new and has not been tested at high speeds. We suspect a glitch in the output fifo's connection to the hardware state machine, though we have been unable to verify this. Several weeks were spent trying to track this bug down (including the ordering of a new compiler from AMD), but to no avail. Time pressures mandated completing the project before the source of the bug was discovered.

The obvious solution to the host loading problems was to run the tests again. Unfortunately, when the tests were run again a day later, the background traffic was being retransmitted even when no *ipukes* were going. The cause of this is uncertain, particularly since the same program was running on the daughter board, and nothing in the network had changed. The round robin program was then recompiled, but this time with the optimizer turned on (we did not use the optimizer previously because we did not trust the compiler). For unknown reasons (except, perhaps a timing bug in the hardware), this cleared up the errors that had been occurring, but introduced new problems. Now, when the *ipuke* sent its data, a large portion (up to 60%) of the data was received with errors, but no REJ(i) was generated, indicating that URP trailers were being lost. The resulting data points were dominated by values between 2 and 3 seconds, indicating that timeouts had occurred. Though the original data showed retransmissions for the background traffic, no retransmissions occurred for the *ipukes*. These tests also suffered from the URP retransmissions of the background *mpukes*, so the data was discarded and the original data was used in spite of the fluctuating load on Pipe.

The problems with interfering traffic will have to be solved before this board can be productively used in the round robin configuration, but the data collected does give a clear indication that round robin queueing will significantly lower the delays experienced by short message traffic. While the host load played a factor in these results, the overall picture is clear enough — round robin queueing significantly decreases message delay in the presence of bulk traffic.

#### 4.3.2.4. fifo vs. none vs. rrobin

Figures 14 to 17 present a side by side comparison of the queueing disciplines for each background window, and the delay of the unloaded case is included for reference. For a 256 * 3 background window (figure 14), all of the queueing disciplines show a small increase in delay, though the increase for round robin queueing is smallest. We see a greater separation between the disciplines for a 512 * 6 window (figure 15), though the software fifo exhibits close to the same performance as round robin; the reason for this was previously discussed: the software fifo had a lower throughput, so the trunk was not as heavily utilized. In figure 16, the disciplines are widely separated, and both fifo programs exhibit almost identical performance. The round robin delay is about 1/5 to 1/10 the fifo delay, so we see its advantages. Figure 17 is very similar to figure 16, except the fifo delays have more than doubled, and

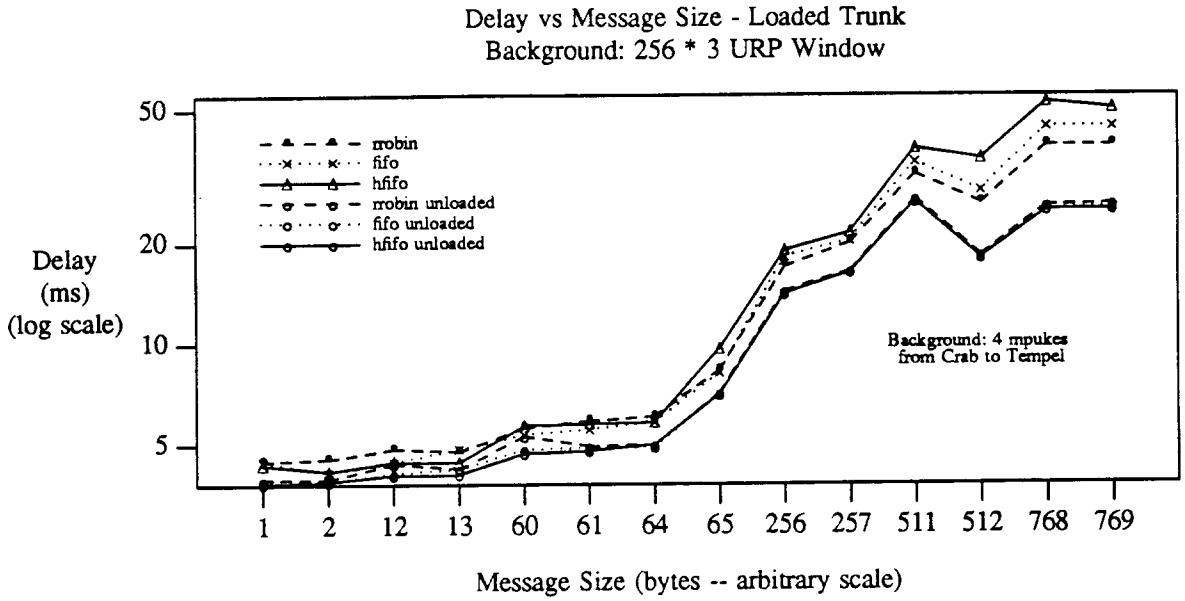


Figure 14 - Comparison of Message Delay for Fifo and Round Robin Queueing
Background URP Window: 256 * 3

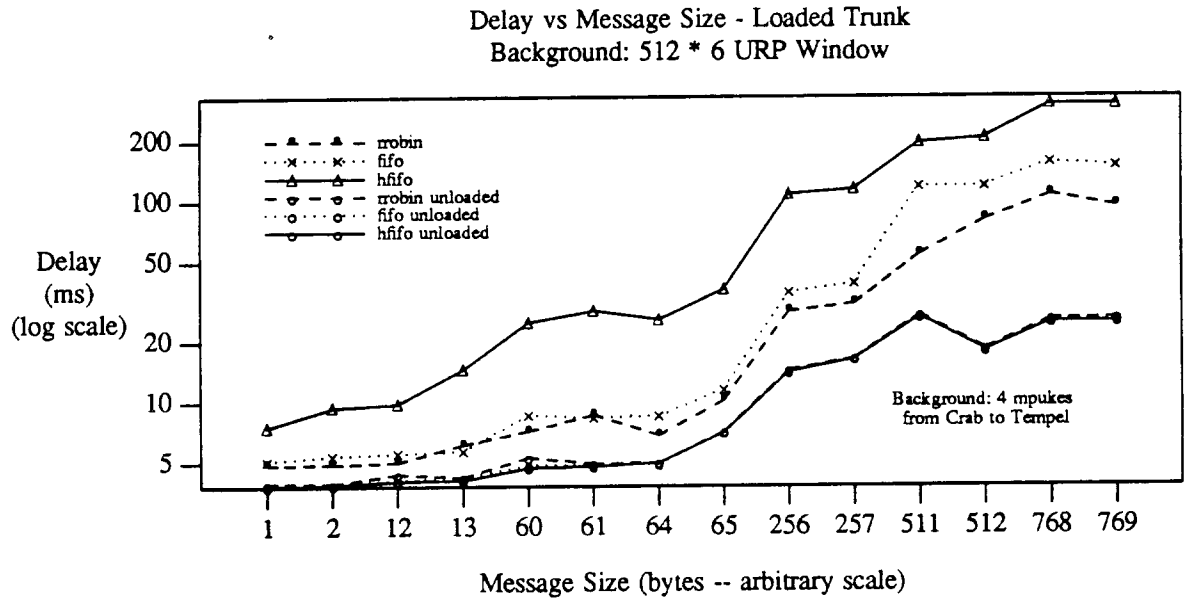


Figure 15 - Comparison of Message Delay for Fifo and Round Robin Queueing
Background URP Window: 512 * 6

the round robin delay has increased just slightly. Here round robin provides delays 1/10 to 1/50 of the fifo delays.

### 4.3.3. Discussion

It is clear that round robin queueing gives significantly lower delays across a broad range of message sizes. The delay will be lower for any message that is shorter in length than the background traffic, so in these experiments even 768 B messages are short (relative to 1 MB). This is particularly

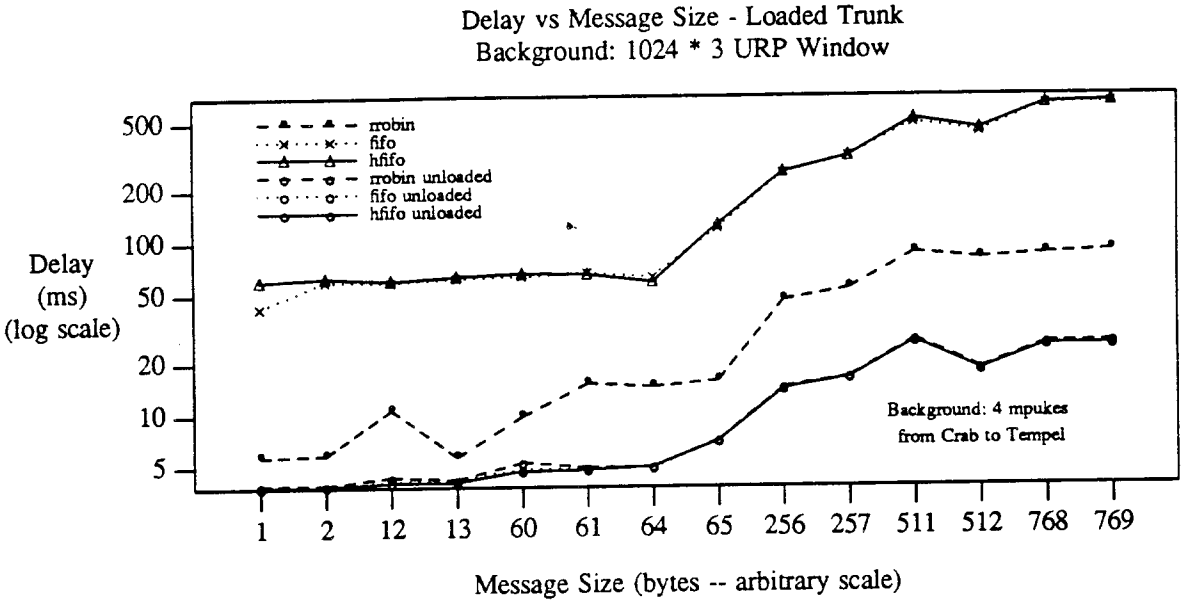


Figure 16 - Comparison of Message Delay for Fifo and Round Robin Queueing
Background URP Window: 1024 * 3

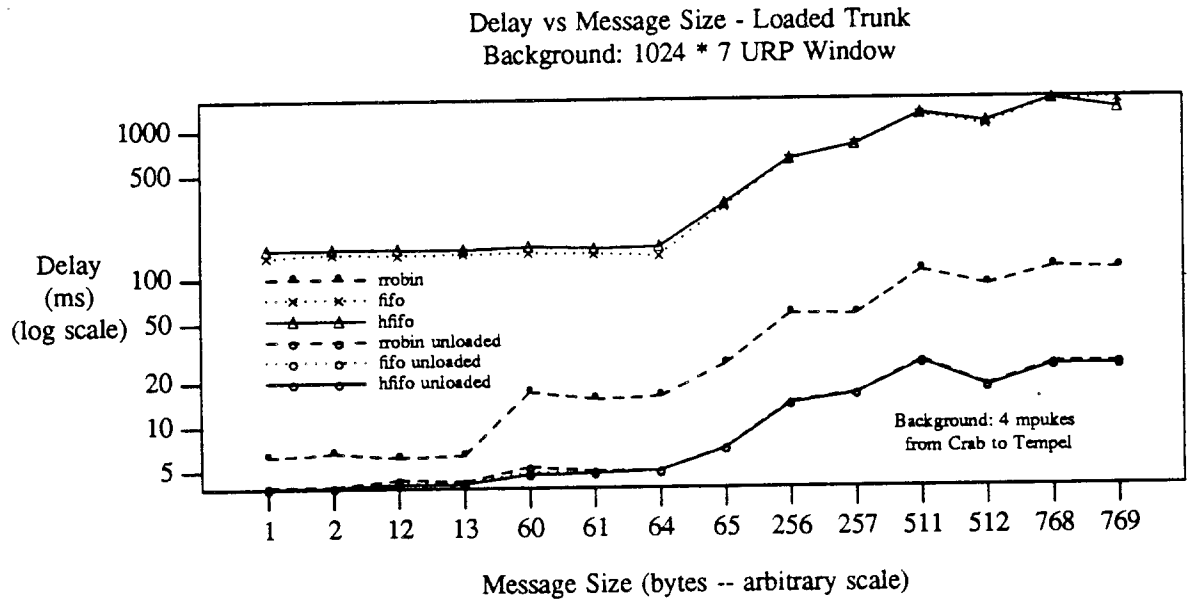


Figure 17 - Comparison of Message Delay for Fifo and Round Robin Queueing
Background URP Window: 1024 * 7

significant in a wide-area environment, where large windows and large block sizes are needed to attain throughput close to the line rate. In a wide-area network fifos are simply a bad idea.

## 5. Summary

The main conclusion of this work is that round robin queueing does indeed provide lower delays for short messages, particularly if large windows sizes are used. The message delay of a fifo network will increase linearly with respect to the aggregate window of the background traffic if the network is

close to saturation, whereas the message delay of a round robin network will remain approximately constant (assuming a bounded number of virtual circuits). Also, this project has shown that it is practical to implement this queueing discipline on a programmable trunk board. The 29000 processor is fast enough to do per-channel queueing at T1 speeds, though some bugs need to be worked out of this implementation. The performance improvement provided by round robin queueing is certainly noticeable, and will be more noticeable in a wide area network where large windows are needed to achieve high throughputs.

Many more tests could be done on this board in the future. For example, other queueing disciplines could be implemented such as the two lists of queues algorithm described in [Fraser 83]. This board provides an excellent environment for the study of queueing disciplines. Another area open for investigation is testing in the context of a wide area network, such as XUNET (a wide area Datakit network which connects U.C. Berkeley, the University of Illinois, the University of Wisconsin at Madison, and Bell Laboratories at Murray Hill). The work of [Arbo 88] and [Vand 88] could be repeated with these trunk boards in place. The effects of the propagation delays of a wide area network on message delay and throughput should be studied. The major accomplishment of this project was actually getting this board working, however.

Many things were learned in the course of trying to program new hardware using a buggy compiler package (for a log of the various challenges along the path, see Appendix C). First and foremost, whenever trying to program new hardware, always make sure an adequate debugging environment is available. The tools available to debug this board were oscilloscopes, logic analyzers, and print statements in the program. Unfortunately, none of these is particularly helpful when debugging a program that should be running at T1 speeds, for example, print statements take eons to complete compared with the speed of the program, and so they disturb what they are trying to measure. The logic analyzer did come in handy to find a timing glitch caused by a bad PAL on the daughter board. And the oscilloscope helped spot the three 32 MHz crystals that failed during the course of the summer. But these tools are not useful when trying to debug a C program running on the processor. The board was not designed to be compatible with AMD's debugger tools, so no options were available. All this was complicated by quirks in the compiler and the linker, where things would fail silently (e.g. C structures were not initialized properly, and one of the linker directives would set all the pointers in the load module to zero, silently). A great deal was learned from these experiences.

One of the key early decisions made was to completely automate the data collection and data analysis through the use of shell scripts and *awk* scripts. Even the production of this report was automated by a Makefile, so if any data changes, the graphs and tables will be automatically updated. Automation is highly recommended for any type of testing, because some tests will invariably have to be repeated (such as the round robin delay measurements through the loaded trunk).

## 6. Acknowledgements

This project would not have been possible without the help of many people. I thank my research advisor, Domenico Ferrari, for allowing me and encouraging me to pursue this project. Most importantly, I must thank Alan Kaplan of AT&T Bell Laboratories for agreeing to take me on as a summer student, without him, this project would not have been possible. He was invaluable in helping to identify the source of problems and taught me a great deal about hardware. I thank Caryl Carr for arranging things so I could come here for the summer; Norman Wilson for writing the */dev/fineclock* driver so I could use a precise clock and for explaining the details of the KMC interface to me so I could identify the various quirks in the delay data; Ed Sitar and Dennis Ritchie for installing the CURE board on Pipe and getting it working; Jim McKie for tolerating my abuse of Tempel; Bill Marshall who answered questions about Datakit and set up the Datakit nodes so I could run my tests; the people who were kind enough to read through this report and give me feedback; and the many other people of Center 1127 at Murray Hill who answered my numerous questions. Finally, I thank my management at Indian Hill for allowing me the freedom to pursue this project, and of course, my parents, who gave me the desire and the freedom to excel, and who celebrate their 33 rd wedding anniversary as I write this.

## 7. References

[Aho 87]
A.V. Aho, B.W. Kernighan and P.J. Weinberger, *The AWK Programming Language*, Addison Wesley, Reading, MA, 1987

[AMD 89]
Advanced Micro Devices, 29K Cross Development Software User's Manual, Beta Release 2.0, Aug. 1989

[Arbo 88]
R.S.Arbo, A Performance Study of Remote Executions in a Wide-Area Datakit Network, Masters Report, U.C. Berkeley, Fall 1988

[Bentley 86]
J.L. Bentley and B.W. Kernighan, GRAP — A Language for Typesetting Graphs, *Communications of the ACM 29,8*, (Aug. 1986), 782-792

[Byrne 89]
R.J. Byrne, Feature Description of the Exploratory BLX Datakit T1 Module, Internal Memorandum, AT&T Bell Laboratories, Liberty Corner, NJ, Jan. 3, 1989

[Fraser 83]
A.G. Fraser, Towards a Universal Data Transport System, *IEEE Journal on Selected Areas in Communications SAC-1,5* (Nov. 1983), 803-816

[Fraser 87]
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

[Marshall 89]
W.T. Marshall, private communication, July 1989

[Vand 88]
T. VandeWater, Delay and Throughput Measurements of the XUNET Datakit Network, Masters Report, U.C. Berkeley, Fall 1988

**Appendix A: Results of Measurements**

- Throughput Tests
- Standard Deviation Tests
- Message Delay — Unloaded (minimums)
- Message Delay — Unloaded (averages)
- Message Delay — Loaded

| Throughput Statistics | | | | | |
|---|---|---|---|---|---|
| Block Size (B) | Out-Standing Blocks | # Conv. | Ave. Tput Per Conv. (KB/s) | Std. Dev. (Kb/s) | Total Tput (Kb/s) |
| Hardware Fifo | | | | | |
| 256 | 3 | 1 | 116271 | 819 | 116271 |
| 256 | 7 | 1 | 126594 | 1068 | 126594 |
| 512 | 3 | 1 | 142305 | 903 | 142305 |
| 512 | 7 | 1 | 149981 | 1129 | 149981 |
| 1024 | 3 | 1 | 160609 | 1444 | 160609 |
| 1024 | 7 | 1 | 162997 | 846 | 162997 |
| 256 | 3 | 4 | 34274 | 3900 | 137099 |
| 256 | 7 | 4 | 32629 | 930 | 130519 |
| 512 | 3 | 4 | 38779 | 640 | 155117 |
| 512 | 7 | 4 | 38979 | 1659 | 155918 |
| 1024 | 3 | 4 | 41658 | 90 | 166635 |
| 1024 | 7 | 4 | 41701 | 155 | 166806 |
| Software Fifo | | | | | |
| 256 | 3 | 1 | 117957 | 607 | 117957 |
| 256 | 7 | 1 | 126178 | 711 | 126178 |
| 512 | 3 | 1 | 138083 | 566 | 138083 |
| 512 | 7 | 1 | 143939 | 1236 | 143939 |
| 1024 | 3 | 1 | 152393 | 2923 | 152393 |
| 1024 | 7 | 1 | 157050 | 1289 | 157050 |
| 256 | 3 | 4 | 34589 | 1630 | 138357 |
| 256 | 7 | 4 | 31434 | 1946 | 125736 |
| 512 | 3 | 4 | 38259 | 628 | 153036 |
| 512 | 7 | 4 | 38258 | 1290 | 153035 |
| 1024 | 3 | 4 | 40782 | 310 | 163131 |
| 1024 | 7 | 4 | 41008 | 672 | 164034 |
| Round Robin | | | | | |
| 256 | 3 | 1 | 116375 | 440 | 116375 |
| 256 | 7 | 1 | 126454 | 7417 | 126454 |
| 512 | 3 | 1 | 144972 | 1156 | 144972 |
| 512 | 7 | 1 | 150255 | 1285 | 150255 |
| 1024 | 3 | 1 | 162848 | 1515 | 162848 |
| 1024 | 7 | 1 | 166398 | 1334 | 166398 |
| 256 | 3 | 4 | 34304 | 1218 | 137217 |
| 256 | 7 | 4 | 33382 | 576 | 133528 |
| 512 | 3 | 4 | 39966 | 635 | 159867 |
| 512 | 7 | 4 | 39052 | 1862 | 156209 |
| 1024 | 3 | 4 | 43438 | 139 | 173755 |
| 1024 | 7 | 4 | 43316 | 2067 | 173264 |

| Standard Deviation Test Statistics | | | | | | |
|---|---|---|---|---|---|---|
| Message size (B) | Min. delay (ms) | Std. Dev. (ms) | Mean/ Dev. | Good Points | Bad Points | Thres-hold (ms) |
| 100 Iterations | | | | | | |
| 1 | 4.493 | 2.985 | 1.505 | 79 | 1 | 104.877 |
| 2 | 3.982 | 0.717 | 5.555 | 79 | 1 | 109.507 |
| 12 | 4.686 | 1.506 | 3.112 | 80 | 0 | 16.734 |
| 13 | 4.024 | 0.065 | 61.604 | 79 | 1 | 9.954 |
| 60 | 4.807 | 0.111 | 43.287 | 80 | 0 | 5.695 |
| 61 | 4.823 | 0.143 | 33.717 | 80 | 0 | 5.967 |
| 64 | 4.922 | 0.331 | 14.887 | 79 | 1 | 11.940 |
| 65 | 7.017 | 0.141 | 49.812 | 79 | 1 | 12.012 |
| 256 | 14.410 | 1.937 | 7.440 | 79 | 1 | 115.791 |
| 257 | 16.599 | 0.600 | 27.653 | 80 | 0 | 21.399 |
| 511 | 27.066 | 0.763 | 35.488 | 79 | 1 | 33.170 |
| 512 | 18.399 | 0.781 | 23.571 | 80 | 0 | 24.647 |
| 768 | 25.297 | 1.392 | 18.175 | 79 | 1 | 127.247 |
| 769 | 25.163 | 0.274 | 91.845 | 79 | 1 | 32.022 |
| 200 Iterations | | | | | | |
| 1 | 3.999 | 0.661 | 6.047 | 179 | 1 | 15.687 |
| 2 | 4.070 | 1.485 | 2.741 | 179 | 1 | 72.227 |
| 12 | 4.315 | 0.775 | 5.567 | 179 | 1 | 16.241 |
| 13 | 4.413 | 0.803 | 5.494 | 180 | 0 | 10.837 |
| 60 | 4.969 | 0.337 | 14.724 | 179 | 1 | 10.267 |
| 61 | 4.956 | 0.218 | 22.698 | 179 | 1 | 8.707 |
| 64 | 5.232 | 1.046 | 5.001 | 180 | 0 | 13.600 |
| 65 | 7.145 | 0.587 | 12.181 | 179 | 1 | 20.751 |
| 256 | 14.510 | 1.543 | 9.402 | 179 | 1 | 83.676 |
| 257 | 16.561 | 1.499 | 11.051 | 179 | 1 | 89.136 |
| 511 | 26.893 | 2.216 | 12.137 | 178 | 2 | 92.342 |
| 512 | 18.297 | 0.894 | 20.470 | 179 | 1 | 83.998 |
| 768 | 25.233 | 0.685 | 36.827 | 178 | 2 | 91.268 |
| 769 | 25.358 | 0.752 | 33.712 | 179 | 1 | 82.121 |
| 300 Iterations | | | | | | |
| 1 | 4.026 | 0.567 | 7.095 | 278 | 2 | 9.212 |
| 2 | 3.857 | 0.373 | 10.340 | 278 | 2 | 10.657 |
| 12 | 4.134 | 1.190 | 3.474 | 279 | 1 | 59.040 |
| 13 | 4.181 | 0.253 | 16.548 | 278 | 2 | 9.690 |
| 60 | 4.822 | 0.267 | 18.090 | 279 | 1 | 9.332 |
| 61 | 4.935 | 0.351 | 14.067 | 278 | 2 | 10.383 |
| 64 | 5.087 | 0.470 | 10.825 | 278 | 2 | 12.368 |
| 65 | 7.183 | 0.578 | 12.432 | 278 | 2 | 15.781 |
| 256 | 14.345 | 1.148 | 12.500 | 279 | 1 | 71.766 |
| 257 | 16.362 | 0.971 | 16.854 | 279 | 1 | 71.793 |
| 511 | 26.742 | 1.219 | 21.939 | 279 | 1 | 80.753 |
| 512 | 18.395 | 1.080 | 17.040 | 279 | 1 | 75.171 |
| 768 | 25.707 | 1.391 | 18.482 | 279 | 1 | 82.634 |
| 769 | 25.545 | 1.269 | 20.137 | 278 | 2 | 97.751 |
| 500 Iterations | | | | | | |
| 1 | 3.936 | 0.366 | 10.741 | 477 | 3 | 9.357 |
| 2 | 3.836 | 0.392 | 9.789 | 479 | 1 | 64.162 |
| 12 | 4.141 | 0.968 | 4.278 | 477 | 3 | 13.357 |
| 13 | 4.383 | 0.895 | 4.896 | 479 | 1 | 63.412 |
| 60 | 5.047 | 1.406 | 3.588 | 478 | 2 | 64.467 |
| 61 | 5.260 | 3.932 | 1.338 | 478 | 2 | 84.252 |
| 64 | 4.961 | 0.370 | 13.416 | 477 | 3 | 10.640 |
| 65 | 7.028 | 1.246 | 5.639 | 479 | 1 | 67.873 |
| 256 | 14.420 | 1.525 | 9.456 | 477 | 3 | 78.259 |
| 257 | 16.797 | 1.963 | 8.559 | 477 | 3 | 88.049 |
| 511 | 27.282 | 2.685 | 10.160 | 476 | 4 | 105.718 |
| 512 | 18.254 | 0.814 | 22.416 | 478 | 2 | 77.251 |
| 768 | 25.489 | 2.130 | 11.967 | 475 | 5 | 86.391 |
| 769 | 25.295 | 0.650 | 38.891 | 477 | 3 | 103.471 |

| Minimum Delay Statistics | | | | | | |
|---|---|---|---|---|---|---|
| Message size (B) | Min. delay (ms) | Std. Dev. (ms) | Mean/ Dev. | Good Points | Bad Points | Thres-hold (ms) |
| Hardware Fifo | | | | | | |
| 1 | 3.824 | 0.134 | 28.555 | 253 | 27 | 4.568 |
| 2 | 3.878 | 0.050 | 77.407 | 237 | 43 | 4.077 |
| 12 | 4.047 | 0.060 | 66.951 | 248 | 32 | 4.181 |
| 13 | 4.071 | 0.075 | 54.519 | 267 | 13 | 4.568 |
| 60 | 4.706 | 0.063 | 74.453 | 219 | 61 | 4.859 |
| 61 | 4.777 | 0.085 | 56.437 | 226 | 54 | 4.920 |
| 64 | 4.988 | 0.352 | 14.187 | 248 | 32 | 6.401 |
| 65 | 7.028 | 0.124 | 56.834 | 250 | 30 | 7.341 |
| 256 | 14.047 | 1.025 | 13.707 | 272 | 8 | 15.286 |
| 257 | 16.262 | 0.263 | 61.778 | 256 | 24 | 17.073 |
| 511 | 6.451 | 0.370 | 71.523 | 249 | 31 | 27.996 |
| 512 | 17.932 | 0.249 | 72.143 | 262 | 18 | 18.991 |
| 768 | 24.842 | 0.294 | 84.523 | 265 | 15 | 26.195 |
| 769 | 24.918 | 0.299 | 83.420 | 264 | 16 | 26.281 |
| Software Fifo | | | | | | |
| 1 | 3.933 | 0.151 | 26.046 | 241 | 39 | 4.733 |
| 2 | 3.873 | 0.074 | 52.167 | 243 | 37 | 4.188 |
| 12 | 4.108 | 0.081 | 50.722 | 229 | 51 | 4.448 |
| 13 | 4.213 | 0.192 | 21.910 | 242 | 38 | 5.261 |
| 60 | 4.832 | 0.104 | 46.348 | 233 | 47 | 5.075 |
| 61 | 4.877 | 0.099 | 49.485 | 224 | 56 | 5.159 |
| 64 | 4.898 | 0.124 | 39.402 | 252 | 28 | 5.263 |
| 65 | 7.043 | 0.132 | 3.389 | 267 | 13 | 7.887 |
| 256 | 14.103 | 1.112 | 12.682 | 251 | 29 | 15.821 |
| 257 | 16.287 | 0.300 | 54.337 | 253 | 27 | 17.529 |
| 511 | 26.395 | 0.386 | 68.469 | 252 | 28 | 27.966 |
| 512 | 17.954 | 0.230 | 77.912 | 239 | 41 | 19.077 |
| 768 | 25.586 | 0.832 | 30.770 | 245 | 35 | 28.216 |
| 769 | 25.329 | 0.717 | 35.321 | 243 | 37 | 27.862 |
| Round Robin | | | | | | |
| 1 | 3.939 | 0.064 | 61.496 | 257 | 23 | 4.121 |
| 2 | 3.936 | 0.126 | 31.210 | 241 | 39 | 4.496 |
| 12 | 4.373 | 0.456 | 9.590 | 246 | 34 | 5.915 |
| 13 | 4.227 | 0.075 | 56.430 | 231 | 49 | 4.594 |
| 60 | 5.285 | 0.666 | 7.934 | 242 | 38 | 7.190 |
| 61 | 4.937 | 0.117 | 42.092 | 232 | 48 | 5.263 |
| 64 | 4.978 | 0.217 | 22.959 | 252 | 28 | 6.193 |
| 65 | 7.094 | 0.285 | 24.861 | 244 | 36 | 8.336 |
| 256 | 14.316 | 0.238 | 60.221 | 227 | 53 | 15.072 |
| 257 | 16.435 | 0.219 | 75.077 | 232 | 48 | 17.009 |
| 511 | 26.985 | 0.513 | 52.617 | 244 | 36 | 28.831 |
| 512 | 18.268 | 0.267 | 68.506 | 234 | 46 | 19.147 |
| 768 | 25.704 | 0.717 | 35.859 | 245 | 35 | 28.160 |
| 769 | 26.001 | 0.782 | 33.237 | 247 | 33 | 28.302 |

| Average Delay Statistics | | | | | | |
|---|---|---|---|---|---|---|
| Message size (B) | Ave. delay (ms) | Std. Dev. (ms) | Mean/ Dev. | Good Points | Bad Points | Thres-hold (ms) |
| Hardware Fifo | | | | | | |
| 1 | 4.069 | 0.870 | 4.677 | 279 | 1 | 27.620 |
| 2 | 4.163 | 0.969 | 4.296 | 279 | 1 | 13.287 |
| 12 | 4.151 | 0.449 | 9.241 | 279 | 1 | 10.114 |
| 13 | 4.243 | 1.331 | 3.188 | 279 | 1 | 65.362 |
| 60 | 4.773 | 0.218 | 21.898 | 278 | 2 | 9.953 |
| 61 | 4.855 | 0.280 | 17.357 | 279 | 1 | 9.735 |
| 64 | 5.702 | 3.127 | 1.823 | 278 | 2 | 73.394 |
| 65 | 7.306 | 1.255 | 5.820 | 279 | 1 | 20.870 |
| 256 | 14.162 | 1.320 | 10.725 | 279 | 1 | 73.580 |
| 257 | 16.745 | 3.203 | 5.228 | 278 | 2 | 81.491 |
| 511 | 27.713 | 6.840 | 4.052 | 275 | 5 | 108.300 |
| 512 | 18.318 | 2.882 | 6.355 | 277 | 3 | 88.750 |
| 768 | 25.250 | 3.761 | 6.713 | 278 | 2 | 110.624 |
| 769 | 25.272 | 2.302 | 10.980 | 277 | 3 | 112.875 |
| Software Fifo | | | | | | |
| 1 | 4.479 | 2.437 | 1.838 | 279 | 1 | 34.795 |
| 2 | 4.118 | 10.783 | 5.258 | 279 | 1 | 14.645 |
| 12 | 4.424 | 10.934 | 4.739 | 279 | 1 | 14.191 |
| 13 | 4.637 | 11.594 | 2.908 | 279 | 1 | 68.792 |
| 60 | 5.093 | 10.799 | 6.372 | 279 | 1 | 12.667 |
| 61 | 5.205 | 0.796 | 6.540 | 279 | 1 | 12.877 |
| 64 | 5.140 | 1.197 | 4.295 | 278 | 2 | 20.214 |
| 65 | 7.301 | 1.655 | 4.412 | 278 | 2 | 65.870 |
| 256 | 14.627 | 2.915 | 5.018 | 279 | 1 | 74.025 |
| 257 | 16.820 | 3.055 | 5.506 | 278 | 2 | 84.441 |
| 511 | 26.758 | 1.537 | 17.412 | 278 | 2 | 108.582 |
| 512 | 18.472 | 1.764 | 10.474 | 279 | 1 | 76.216 |
| 768 | 26.320 | 2.631 | 10.005 | 278 | 2 | 129.136 |
| 769 | 27.085 | 8.604 | 3.148 | 276 | 4 | 124.479 |
| Round Robin | | | | | | |
| 1 | 3.996 | 0.331 | 12.072 | 279 | 1 | 19.735 |
| 2 | 4.178 | 0.769 | 5.431 | 279 | 1 | 19.393 |
| 12 | 4.916 | 2.185 | 2.250 | 279 | 1 | 75.529 |
| 13 | 4.557 | 0.974 | 4.678 | 279 | 1 | 15.607 |
| 60 | 6.015 | 2.966 | 2.028 | 279 | 1 | 79.451 |
| 61 | 5.213 | 0.856 | 6.088 | 279 | 1 | 13.869 |
| 64 | 5.222 | 0.971 | 5.378 | 278 | 2 | 93.238 |
| 65 | 7.803 | 3.192 | 2.445 | 279 | 1 | 90.338 |
| 256 | 14.815 | 1.574 | 9.412 | 277 | 3 | 30.768 |
| 257 | 16.869 | 1.360 | 12.403 | 278 | 2 | 29.752 |
| 511 | 27.550 | 2.088 | 13.194 | 277 | 3 | 112.740 |
| 512 | 18.648 | 1.149 | 16.230 | 279 | 1 | 38.644 |
| 768 | 28.098 | 11.066 | 2.539 | 277 | 3 | 133.232 |
| 769 | 26.653 | 3.205 | 8.317 | 277 | 3 | 129.111 |

| Message Delay Statistics Loaded Trunk — Hardware Fifo | | | | | | |
|---|---|---|---|---|---|---|
| Message size (B) | Ave. delay (ms) | Std. Dev. (ms) | Mean/ Dev. | Good Points | Bad Points | Thres-hold (ms) |
| 256 * 3 window | | | | | | |
| 1 | 4.349 | 0.482 | 9.022 | 279 | 1 | 16.926 |
| 2 | 4.162 | 0.201 | 20.734 | 280 | 0 | 5.770 |
| 12 | 4.443 | 0.532 | 8.351 | 278 | 2 | 15.849 |
| 13 | 4.427 | 0.167 | 26.443 | 279 | 1 | 11.994 |
| 60 | 5.685 | 0.481 | 11.809 | 278 | 2 | 12.479 |
| 61 | 5.745 | 0.489 | 11.758 | 278 | 2 | 13.308 |
| 64 | 5.815 | 0.581 | 10.005 | 279 | 1 | 12.313 |
| 65 | 9.632 | 7.487 | 1.287 | 279 | 1 | 179.590 |
| 256 | 18.914 | 1.344 | 14.072 | 279 | 1 | 76.276 |
| 257 | 21.399 | 1.264 | 16.934 | 279 | 1 | 77.642 |
| 511 | 38.054 | 12.100 | 3.145 | 279 | 1 | 197.515 |
| 512 | 35.417 | 10.200 | 3.472 | 279 | 1 | 185.504 |
| 768 | 52.250 | 12.443 | 4.199 | 279 | 1 | 228.691 |
| 769 | 49.843 | 3.525 | 14.139 | 278 | 2 | 121.154 |
| 512 * 6 window | | | | | | |
| 1 | 7.497 | 1.861 | 4.028 | 280 | 0 | 22.385 |
| 2 | 9.393 | 2.260 | 4.156 | 279 | 1 | 73.183 |
| 12 | 9.801 | 2.449 | 4.002 | 280 | 0 | 29.393 |
| 13 | 14.481 | 4.583 | 3.160 | 279 | 1 | 84.548 |
| 60 | 24.852 | 14.067 | 1.767 | 279 | 1 | 227.705 |
| 61 | 28.436 | 4.943 | 5.752 | 278 | 2 | 112.037 |
| 64 | 25.628 | 4.938 | 5.190 | 278 | 2 | 107.970 |
| 65 | 36.322 | 8.919 | 4.072 | 278 | 2 | 143.390 |
| 256 | 107.498 | 18.451 | 5.826 | 279 | 1 | 309.634 |
| 257 | 113.952 | 20.144 | 5.657 | 279 | 1 | 356.157 |
| 511 | 194.882 | 22.932 | 8.498 | 278 | 2 | 451.377 |
| 512 | 205.314 | 31.743 | 6.468 | 279 | 1 | 502.758 |
| 768 | 300.861 | 24.369 | 12.346 | 280 | 0 | 495.813 |
| 769 | 298.790 | 42.085 | 7.100 | 280 | 0 | 635.470 |
| 1024 * 3 window | | | | | | |
| 1 | 59.966 | 11.479 | 5.224 | 279 | 1 | 217.266 |
| 2 | 62.744 | 9.892 | 6.343 | 279 | 1 | 189.825 |
| 12 | 60.167 | 13.114 | 4.588 | 279 | 1 | 231.266 |
| 13 | 63.572 | 14.418 | 4.409 | 279 | 1 | 245.399 |
| 60 | 66.191 | 8.436 | 7.846 | 278 | 2 | 166.512 |
| 61 | 65.118 | 12.906 | 5.045 | 279 | 1 | 199.362 |
| 64 | 58.533 | 19.169 | 3.053 | 280 | 0 | 211.885 |
| 65 | 125.872 | 17.553 | 7.171 | 279 | 1 | 414.052 |
| 256 | 252.655 | 38.371 | 6.585 | 280 | 0 | 559.623 |
| 257 | 312.089 | 57.637 | 5.415 | 280 | 0 | 773.185 |
| 511 | 512.694 | 50.286 | 10.196 | 279 | 1 | 1061.680 |
| 512 | 446.279 | 32.431 | 13.761 | 280 | 0 | 705.727 |
| 768 | 622.695 | 75.913 | 8.203 | 280 | 0 | 1230.000 |
| 769 | 638.457 | 40.967 | 15.585 | 280 | 0 | 966.193 |
| 1024 * 7 window | | | | | | |
| 1 | 156.212 | 16.496 | 9.470 | 279 | 1 | 312.611 |
| 2 | 157.751 | 15.663 | 10.072 | 279 | 1 | 302.455 |
| 12 | 157.414 | 21.650 | 7.271 | 279 | 1 | 370.308 |
| 13 | 156.079 | 11.831 | 13.193 | 279 | 1 | 316.948 |
| 60 | 163.457 | 24.918 | 6.560 | 279 | 1 | 401.420 |
| 61 | 158.555 | 19.280 | 8.224 | 278 | 2 | 444.633 |
| 64 | 162.299 | 20.816 | 7.797 | 279 | 1 | 373.648 |
| 65 | 314.509 | 22.817 | 13.784 | 280 | 0 | 497.045 |
| 256 | 623.582 | 42.607 | 14.636 | 280 | 0 | 964.438 |
| 257 | 777.427 | 62.411 | 12.457 | 280 | 0 | 1276.720 |
| 511 | 1250.773 | 74.479 | 16.794 | 280 | 0 | 1846.600 |
| 512 | 1091.651 | 64.147 | 17.018 | 279 | 1 | 1714.530 |
| 768 | 1545.262 | 100.279 | 15.410 | 280 | 0 | 2347.490 |
| 769 | 1331.216 | 507.930 | 2.621 | 280 | 0 | 5394.660 |

| Message Delay Statistics Loaded Trunk — Software Fifo | | | | | | |
|---|---|---|---|---|---|---|
| Message size (B) | Ave. delay (ms) | Std. Dev. (ms) | Mean/ Dev. | Good Points | Bad Points | Thres-hold (ms) |
| 256 * 3 window | | | | | | |
| 1 | 4.453 | 1.026 | 4.341 | 279 | 1 | 62.480 |
| 2 | 4.125 | 0.259 | 15.929 | 279 | 1 | 8.166 |
| 12 | 4.417 | 0.654 | 6.756 | 279 | 1 | 225.651 |
| 13 | 4.842 | 3.207 | 1.510 | 279 | 1 | 70.595 |
| 60 | 5.340 | 0.313 | 17.053 | 277 | 3 | 12.085 |
| 61 | 5.534 | 1.177 | 4.702 | 280 | 0 | 14.950 |
| 64 | 5.797 | 2.375 | 2.441 | 278 | 2 | 50.646 |
| 65 | 8.143 | 0.691 | 11.787 | 279 | 1 | 14.900 |
| 256 | 18.071 | 7.215 | 2.505 | 279 | 1 | 171.922 |
| 257 | 20.275 | 1.554 | 13.048 | 279 | 1 | 83.971 |
| 511 | 34.638 | 10.120 | 3.423 | 279 | 1 | 226.111 |
| 512 | 28.456 | 9.149 | 3.110 | 279 | 1 | 197.894 |
| 768 | 43.973 | 5.626 | 7.816 | 277 | 3 | 147.066 |
| 769 | 44.024 | 7.639 | 5.763 | 278 | 2 | 137.955 |
| 512 * 6 window | | | | | | |
| 1 | 5.103 | 1.700 | 3.001 | 279 | 1 | 165.030 |
| 2 | 5.408 | 1.713 | 3.157 | 279 | 1 | 160.000 |
| 12 | 5.544 | 1.800 | 3.080 | 280 | 0 | 19.944 |
| 13 | 5.693 | 1.559 | 3.651 | 279 | 1 | 56.087 |
| 60 | 8.572 | 2.516 | 3.407 | 278 | 2 | 88.596 |
| 61 | 8.307 | 2.085 | 3.983 | 280 | 0 | 24.987 |
| 64 | 8.484 | 2.501 | 3.392 | 279 | 1 | 71.391 |
| 65 | 11.457 | 2.766 | 4.142 | 280 | 0 | 33.585 |
| 256 | 34.898 | 16.843 | 2.072 | 279 | 1 | 235.553 |
| 257 | 38.886 | 17.180 | 2.263 | 280 | 0 | 176.326 |
| 511 | 117.612 | 76.187 | 1.544 | 280 | 0 | 727.108 |
| 512 | 117.859 | 117.426 | 1.004 | 280 | 0 | 1057.270 |
| 768 | 154.379 | 81.059 | 1.905 | 280 | 0 | 802.851 |
| 769 | 148.371 | 61.118 | 2.428 | 280 | 0 | 637.315 |
| 1024 * 3 window | | | | | | |
| 1 | 41.988 | 27.023 | 1.554 | 280 | 0 | 258.172 |
| 2 | 59.472 | 14.642 | 4.062 | 279 | 1 | 240.214 |
| 12 | 58.701 | 14.826 | 3.959 | 280 | 0 | 177.309 |
| 13 | 61.859 | 14.613 | 4.233 | 279 | 1 | 231.499 |
| 60 | 63.444 | 13.568 | 4.676 | 279 | 1 | 187.326 |
| 61 | 66.500 | 18.086 | 3.677 | 280 | 0 | 211.188 |
| 64 | 62.610 | 7.986 | 7.840 | 278 | 2 | 153.504 |
| 65 | 119.591 | 21.391 | 5.591 | 278 | 2 | 375.124 |
| 256 | 248.759 | 28.296 | 8.791 | 280 | 0 | 475.127 |
| 257 | 311.391 | 28.317 | 10.996 | 280 | 0 | 537.927 |
| 511 | 486.928 | 47.337 | 10.286 | 280 | 0 | 865.624 |
| 512 | 428.383 | 65.823 | 6.508 | 279 | 1 | 1065.860 |
| 768 | 613.303 | 80.261 | 7.641 | 280 | 0 | 1255.390 |
| 769 | 622.797 | 76.809 | 8.108 | 280 | 0 | 1237.270 |
| 1024 * 7 window | | | | | | |
| 1 | 140.012 | 25.074 | 5.584 | 279 | 1 | 582.775 |
| 2 | 145.950 | 16.171 | 9.025 | 279 | 1 | 347.440 |
| 12 | 143.626 | 20.655 | 6.954 | 279 | 1 | 332.310 |
| 13 | 145.822 | 15.015 | 9.712 | 279 | 1 | 290.111 |
| 60 | 146.740 | 20.735 | 7.077 | 279 | 1 | 351.202 |
| 61 | 144.760 | 19.598 | 7.386 | 279 | 1 | 332.015 |
| 64 | 140.064 | 24.296 | 5.765 | 279 | 1 | 373.178 |
| 65 | 299.009 | 22.940 | 13.034 | 279 | 1 | 529.062 |
| 256 | 613.843 | 64.485 | 9.519 | 280 | 0 | 1129.720 |
| 257 | 779.148 | 85.109 | 9.155 | 279 | 1 | 1569.340 |
| 511 | 1225.683 | 69.110 | 17.735 | 280 | 0 | 1778.560 |
| 512 | 1039.660 | 108.732 | 9.562 | 280 | 0 | 1909.520 |
| 768 | 1522.284 | 85.787 | 17.745 | 280 | 0 | 2208.580 |
| 769 | 1533.324 | 81.782 | 18.749 | 280 | 0 | 2187.580 |

| Message Delay Statistics Loaded Trunk — Round Robin | | | | | | |
|---|---|---|---|---|---|---|
| Message size (B) | Ave. delay (ms) | Std. Dev. (ms) | Mean/ Dev. | Good Points | Bad Points | Thres- hold (ms) |
| 256 * 3 window | | | | | | |
| 1 | 4.460 | 0.527 | 8.464 | 279 | 1 | 10.347 |
| 2 | 4.527 | 0.399 | 11.332 | 278 | 2 | 12.271 |
| 12 | 4.845 | 0.450 | 10.756 | 279 | 1 | 9.053 |
| 13 | 4.753 | 0.486 | 9.789 | 279 | 1 | 23.191 |
| 60 | 5.598 | 0.762 | 7.346 | 279 | 1 | 25.410 |
| 61 | 5.869 | 2.396 | 2.449 | 279 | 1 | 52.182 |
| 64 | 6.050 | 2.738 | 2.210 | 279 | 1 | 67.743 |
| 65 | 8.299 | 0.815 | 10.186 | 279 | 1 | 26.054 |
| 256 | 16.896 | 1.998 | 8.456 | 279 | 1 | 77.039 |
| 257 | 19.911 | 2.841 | 7.009 | 279 | 1 | 83.923 |
| 511 | 31.866 | 0.866 | 36.778 | 278 | 2 | 105.884 |
| 512 | 26.096 | 2.369 | 11.017 | 278 | 2 | 104.744 |
| 768 | 38.774 | 11.873 | 3.266 | 279 | 1 | 227.408 |
| 769 | 38.695 | 9.749 | 3.969 | 279 | 1 | 215.514 |
| 512 * 6 window | | | | | | |
| 1 | 4.872 | 1.432 | 3.401 | 278 | 2 | 82.553 |
| 2 | 4.900 | 1.389 | 3.527 | 279 | 1 | 67.179 |
| 12 | 5.024 | 0.590 | 8.515 | 279 | 1 | 11.610 |
| 13 | 6.141 | 9.818 | 0.626 | 279 | 1 | 279.276 |
| 60 | 7.153 | 3.008 | 2.378 | 279 | 1 | 73.501 |
| 61 | 8.596 | 18.086 | 0.475 | 279 | 1 | 435.934 |
| 64 | 6.797 | 1.408 | 4.829 | 279 | 1 | 73.147 |
| 65 | 10.128 | 3.248 | 3.118 | 279 | 1 | 70.992 |
| 256 | 28.110 | 24.561 | 1.145 | 278 | 2 | 437.484 |
| 257 | 30.713 | 30.994 | 0.991 | 278 | 2 | 526.443 |
| 511 | 54.003 | 42.092 | 1.283 | 279 | 1 | 483.370 |
| 512 | 80.110 | 92.460 | 0.866 | 280 | 0 | 819.790 |
| 768 | 106.568 | 65.627 | 1.624 | 279 | 1 | 711.223 |
| 769 | 93.328 | 63.689 | 1.465 | 279 | 1 | 690.395 |
| 1024 * 3 window | | | | | | |
| 1 | 5.764 | 4.032 | 1.430 | 278 | 2 | 90.783 |
| 2 | 5.877 | 3.482 | 1.688 | 279 | 1 | 120.301 |
| 12 | 10.721 | 18.960 | 0.565 | 279 | 1 | 192.390 |
| 13 | 5.697 | 0.639 | 8.910 | 279 | 1 | 11.696 |
| 60 | 9.792 | 5.281 | 1.854 | 279 | 1 | 253.656 |
| 61 | 15.172 | 21.447 | 0.707 | 278 | 2 | 497.653 |
| 64 | 14.469 | 20.302 | 0.713 | 279 | 1 | 366.096 |
| 65 | 15.713 | 14.606 | 1.076 | 278 | 2 | 208.078 |
| 256 | 45.782 | 71.709 | 0.638 | 279 | 1 | 712.728 |
| 257 | 53.475 | 37.089 | 1.442 | 279 | 1 | 406.849 |
| 511 | 85.837 | 78.257 | 1.097 | 280 | 0 | 711.893 |
| 512 | 79.186 | 62.767 | 1.262 | 279 | 1 | 708.997 |
| 768 | 83.989 | 65.702 | 1.278 | 280 | 0 | 609.605 |
| 769 | 86.945 | 62.143 | 1.399 | 279 | 1 | 795.806 |
| 1024 * 7 window | | | | | | |
| 1 | 6.245 | 4.553 | 1.372 | 279 | 1 | 122.094 |
| 2 | 6.611 | 5.995 | 1.103 | 279 | 1 | 270.950 |
| 12 | 6.194 | 0.865 | 7.159 | 278 | 2 | 18.815 |
| 13 | 6.386 | 1.039 | 6.148 | 279 | 1 | 15.940 |
| 60 | 16.877 | 16.455 | 1.026 | 278 | 2 | 314.765 |
| 61 | 15.110 | 9.382 | 1.611 | 277 | 3 | 219.403 |
| 64 | 15.748 | 11.180 | 1.409 | 278 | 2 | 185.444 |
| 65 | 25.912 | 19.482 | 1.330 | 279 | 1 | 420.136 |
| 256 | 56.228 | 45.120 | 1.246 | 279 | 1 | 582.299 |
| 257 | 55.453 | 26.024 | 2.131 | 278 | 2 | 417.105 |
| 511 | 108.747 | 68.923 | 1.578 | 280 | 0 | 660.131 |
| 512 | 86.675 | 85.911 | 1.009 | 280 | 0 | 773.963 |
| 768 | 112.884 | 59.239 | 1.906 | 279 | 1 | 703.895 |
| 769 | 109.820 | 57.618 | 1.906 | 279 | 1 | 663.275 |