REAL : A Network Simulator

Srinivasan Keshav
December 1988

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

Abstract

Performance analysis of computer networks is rapidly gaining importance as networks increase in size and geographical extent. A simulation approach is often useful. This report describes REAL, a computer network simulator, and presents results of some simulations to illustrate its analytical power. Details of implementation and a performance evaluation are also presented[1].

[1] This work was supported in part by Xerox Corporation, Palo Alto Research Center and in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089. The views and conclusions in this document are those of the author, and should not be interpreted as representing official policies, either express or implied, of any of the sponsoring agencies or corporations.

1. Introduction and Motivation

Performance analysis of computer networks is rapidly gaining importance as networks increase in size and geographical extent. The size of the networks and the inherent complexity of network protocols complicate this analysis. Analysis techniques such as queueing models have difficulty modeling dynamic behavior such as retransmission timeouts and congestion. In these cases, simulation offers a better method of studying computer networks, since one can simulate the details of actual protocols. Further, traces of execution under different environments provide insights into the dynamics of protocols that queueing analysis does not easily provide. Of course, there is considerable effort involved in building a simulator and substantial machine cost in running it, but this approach still is the most attractive.

The immediate motivation for building REAL (REalistic And Large) was to compare the 'fair queueing' gateway algorithm [Nag87] [DKS88] with first-come-first-served scheduling and with competing proposals from Digital Equipment Corporation (DEC)[RCJ87,RaJ87]. Results from the study are described in a companion paper [DKS88]. REAL was built using the NEST simulation toolkit from Columbia University [BDS87]. While the simulator was built specifically for this study, simple modifications can greatly extend its power and generality.

This report presents a description of REAL and the results of some simulations to illustrate its analytical power. The remainder of the report is organized as follows : Section 2 presents an outline of the simulator from a user's point of view. Section 3 describes details of the implementation. Section 4 sketches the transport protocols implemented. Section 5 presents some gateway scheduling algorithms. Section 6 describes results of some simulations using REAL, and Section 7 presents a performance evaluation of the simulator. Section 8 describes a design for extending REAL to parallel distributed simulation. Section 9 is an evaluation of this work, and Section 10 describes future work.

2. Outline of the Simulator

This section describes a user's view of the REAL simulator's two parts - a simulation server, and a display client (Figure 1). The *server* carries out simulations, and connects through a Berkeley UNIX socket to the client. The *client* maintains a 'simulation window' which reflects the current state of the simulation. A user sets up and controls the simulation through the display client. The client passes simulation parameters and control messages to the server; the server passes status information and results to the client.

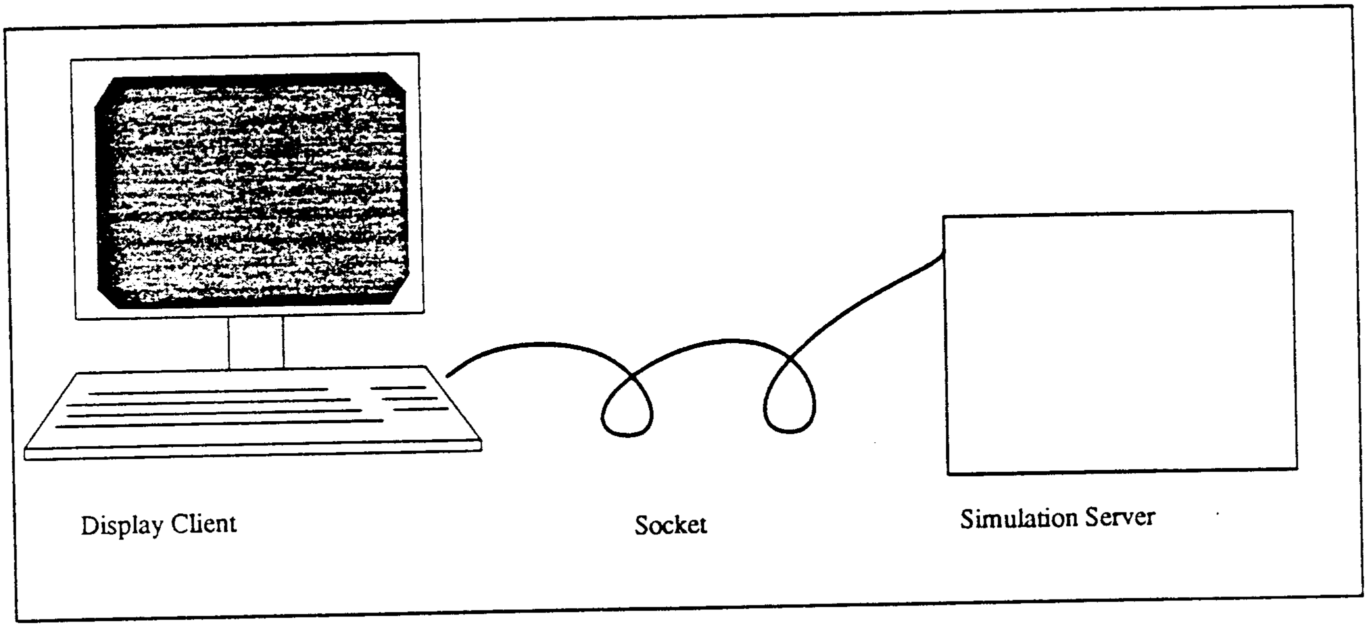


Figure 1 *REAL is partitioned into a display client and a simulation server*

Such a partition has two advantages. A simulation can be run on a large compute server, and simulations running in parallel on several machines can be controlled from a single screenful of windows. We have taken advantage of both features - by running large simulations on a VAX 8600, and by running up to six simulations in parallel on a SUN cluster.

2.1. Network Model

REAL simulates a packet switched, store and forward network similar to existing wide area networks such as the Xerox corporate net and the DARPA Internet. The network layer is datagram oriented, and packets can be lost or delivered out of sequence. The transport layer, which is modeled on the Transmission Control Protocol (TCP) [Tan81], provides reliable, sequenced packets using standard techniques of flow control, timeout estimation and packet retransmission. The network itself consists of a set of *sources* that execute a transport protocol, *gateways* that route packets and schedule them on their outgoing lines, and *sinks* that absorb packets sent to them, returning an acknowledgment for each packet received. Our analysis assumes that the bottleneck in a network is always the limited bandwidth of transmission lines, hence all sources, gateways and sinks are infinitely fast. Note that REAL sources attempt to model many of the details of flow control in the transport layer in contrast to some previous work, for instance [FrM84], where sources are abstracted as Poisson processes. We now discuss each component of the network (sources, gateways and sinks) briefly.

Sources are classified according to the workload they present to the simulator and the version of the transport layer protocol they employ (Figure 2). We chose to model three types of workloads : FTP, telnet and ill behaved. An FTP workload models large file transfers : whenever flow control permits, a source with an FTP workload (FTP source) immediately sends a maximal sized packet. A telnet source sends minimal sized packets with an exponential interpacket spacing, modeling the workload of a telnet (remote login) connection. An ill behaved source continuously sends maximally sized packets, limited only by the network's bandwidth.

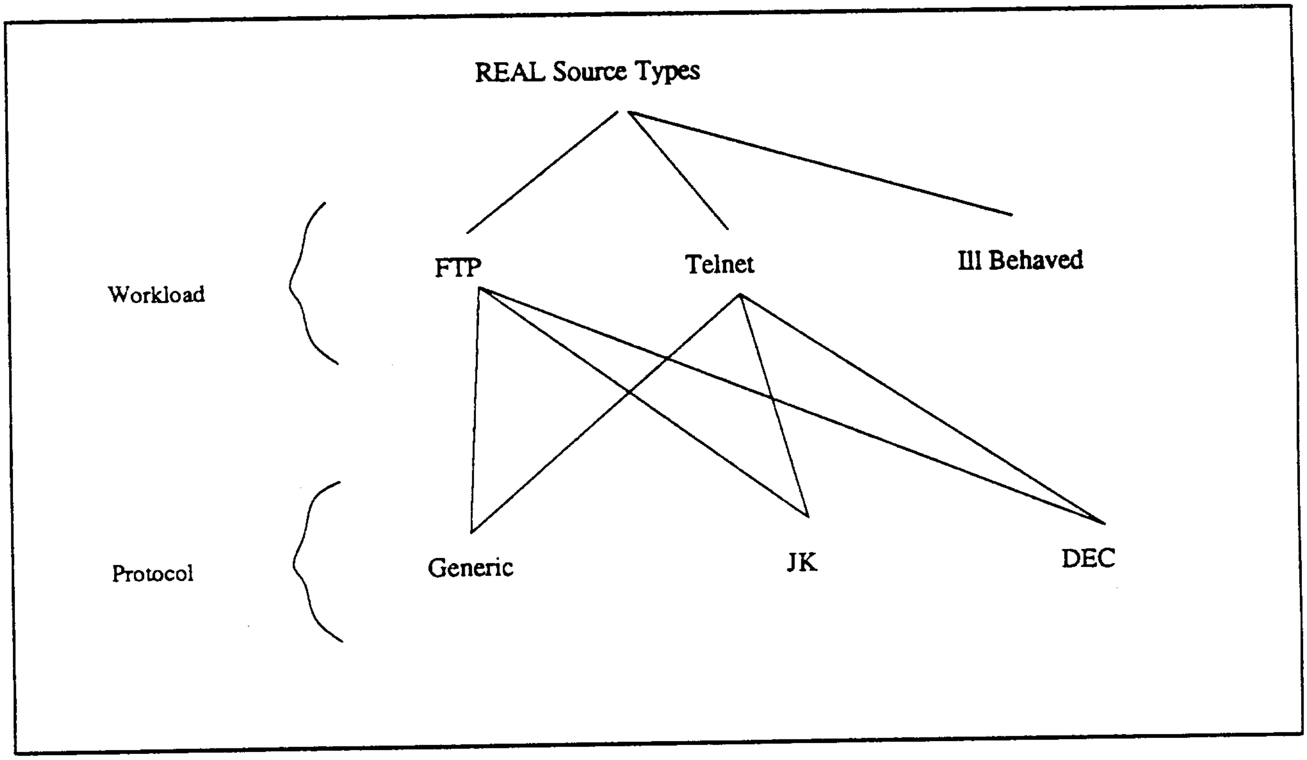


Figure 2 *Types of REAL sources*

We further divide these classes on the basis of their version of TCP. There are three kinds of FTP and telnet sources : generic, DEC and JK. Generic sources use 'standard' TCP with flow control, but do not take special measures to avoid network congestion (Section 4.1). JK sources add many of Jacobson and Karels' congestion avoidance ideas [Jac88] (Section 4.2), and DEC sources add the DEC binary congestion avoidance scheme [RCJ87,RaJ87] (Section 4.3) to the generic transport protocol.

Gateways (or routers)[2], perform two logical functions : routing and scheduling. At startup, a straightforward shortest path computation does static routing. Since we were particularly interested in scheduling policies, REAL implements several different scheduling algorithms, to be described later.

Sinks discard packets sent to them, sending an acknowledgment for each packet received. The acknowledgment header contains the number of the last in-sequence packet the sink received.

2.2. User Interface

The display client runs on a Sun 3 or Sun 4 workstation under `suntools`, and presents a mouse based interface to a user. The network is represented as a graph, and a display manager maintains the current status of the network in the simulation window. A number of control panels allow simulation parameters (such as the speeds of the communication lines) to be set interactively by mouse clicks, or by typing a value.

Network graphs are drawn interactively using the mouse, and the result is sent to the simulator. A node in the graph represents a source, gateway or sink, as specified by its 'node function', which is set for each node by clicking on the appropriate menu. As an example, to draw the graph shown in Figure 3, the mouse is used to create the six nodes (by clicks at the appropriate locations) and the edges between the nodes. Additional menus prompt for the protocol to be run at each node and the bandwidth/delay characteristics of each line. The simulation's parameters are set from the appropriate panel. The entire graph and associated simulation parameters can be sent to the server by clicking on a `send` selection or saved to a file and reloaded later.

The display client allows for graphical monitoring of a selected node's variables (such as the current window size) as the simulation proceeds. The nodes to be monitored are selected by clicking on their icon. For example, one might monitor the window size for the protocol running on node 1. This is analogous to attaching an oscilloscope to a signal line in a hardware circuit, and as useful.

REAL provides for automatic report generation. A library of routines for table generation collects relevant statistics (such as packets sent and received by each node), tabulates them, and prints them out at predefined intervals.

It is interesting to note that entire simulations can be set up from the display using only the mouse and menus. This visual programming style, analogous to that proposed in the Alternate Reality Kit [Smi87], makes the simulator fun to use. The availability of a graphical display to monitor the simulation in real time is useful, and is an example of scientific visualization.

3. Implementation Details

REAL is built upon the NEST (NEtwork Simulation Testbed) package from Columbia University [BDS87]. We now present an outline of NEST, enumerating and describing the additions we made.

NEST provides (a) a display client and (b) a simulation library. The display client manages a simulation window, where the simulated network and associated parameters are initially input, and subsequently updated as the simulation proceeds. The library includes a threads package, primitives for inter-thread communication, and a non blocking socket library for communication between the simulation server and the display client.

The network is specified by the set of nodes and their interconnecting edges. Each network node is assigned a C function, the 'node function'. After the set of nodes, their node functions and their interconnections are specified, the NEST primitive `simulate()` runs each node's function in parallel in a single common address space. REAL is thus written as a set of node functions, one for each kind of source,

[2] Disregarding differences in functionality, we use these terms interchangeably in this report.

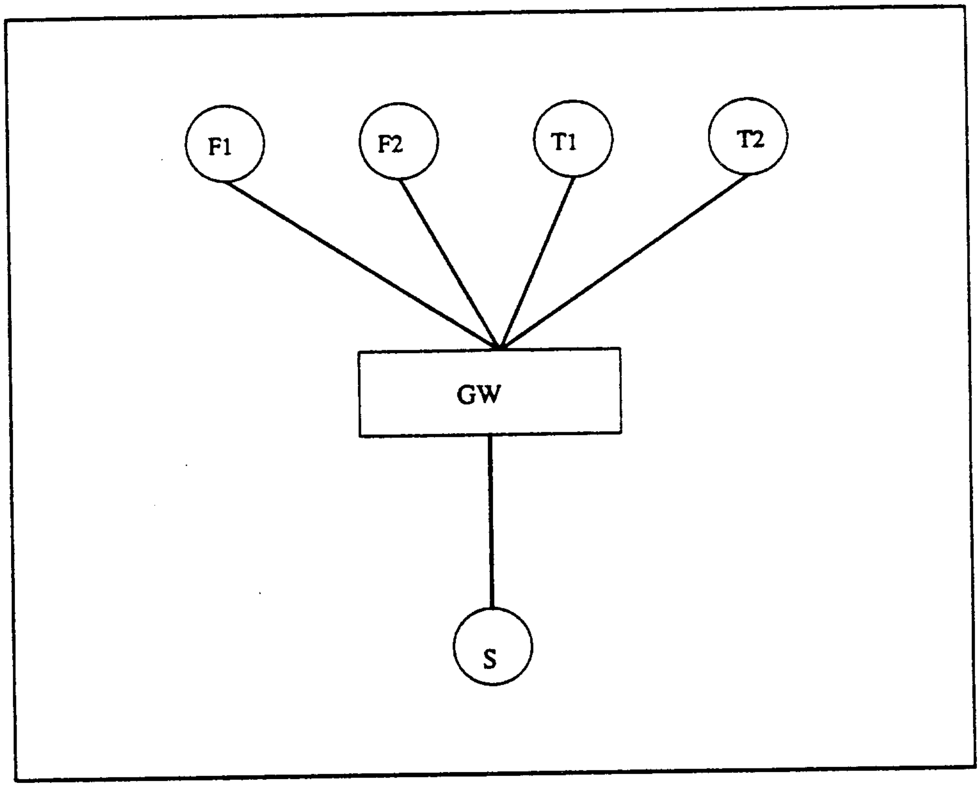


Figure 3 *A typical network*

gateway and sink. Writing such node functions is difficult. Not only do we have to solve standard synchronization problems, but also, since threads are preemptable, all node functions have to be made reentrant.

Communication between the server and the display occurs over a non blocking socket. The network, described formally in a 'graph language' structure, is converted to a bit stream and exchanged between the display and the server. We wanted to leave the graph language unchanged to maintain compatibility with other NEST simulators, and so added an extra socket between the display and the server to carry REAL-specific simulation parameters. We implemented the socket using the nonblocking socket library. An extra panel was added to the Simulation window to hold REAL parameters.

We now describe some libraries that we added.

- The Tables Library creates tables of statistics, adds entries to them and prints them neatly in a report.
- A large Buffer Management Library supports a variety of buffer allocation schemes at the routers. Buffer data structures and associated procedures were strictly encapsulated to assure correctness of buffer management.
- As the simulation progresses, values of selected variables can be written onto a file. A separate process reads these values and creates graphs (using the UNIX `graph` utility) that are then displayed on a Tektronix emulation window (`tektool`).
- Finally, a number of tracing options were built into REAL. These allow simultaneous tracing of one of many network functions, such as routing, buffer management and source protocols. This proved to be a valuable debugging tool, as well as a way of checking a protocol implementation's correctness.

4. Transport Protocols

This section describes details of the implementation of REAL's transport protocol options. Some familiarity with windowing mechanisms and TCP is assumed [Tan81]. Recall (Figure 2) that three types of TCP transport are implemented : generic, JK (Jacobson-Karels), and DEC. A source implementing a protocol version X is called an X source for convenience.

4.1. Generic Transport

Generic sources model features common to many of today's transport protocols such as TCP and XNS. They represent the minimal common functionality necessary for a reliable, connection-oriented transport layer built on an unreliable, connectionless network layer. Generic sources implement sliding window flow control and retransmission timeout.

4.1.1. Sliding windows

A *window* is the largest number of unacknowledged packets that a source can have outstanding. Once the window limit is reached, a source waits for new acknowledgments before sending more packets. Since only sinks send acknowledgments, they can dictate the rate at which they receive packets, i.e., the flow rate.

4.1.2. Timeouts

When a packet is dispatched at time t, a timer is set for an interval I. If the source does not receive an acknowledgment by $t+I$, it retransmits the packet. I is usually set to be bR, where R is the exponentially smoothed round trip time[3] and b is typically 2.

Generic sources behave badly in the presence of congestion since they do not have sophisticated congestion control mechanisms. For example, consider a generic source that has a window size of 5, which loses a packet with sequence number 10 at a gateway due to congestion. Window flow control allows packets with sequence numbers 10-14 to be outstanding. Assume that the sink receives packets 11-14. However, these packets are out of sequence and the sink does not acknowledge them. When packet 10 times out, the source retransmits it and the sink acknowledges packets 10-14. At this point, the source sends a burst of 5 packets, which further aggravates the congested gateway. Thus, the packet loss due to congestion triggers a burst of packets from a generic source which sustains the condition.

4.2. JK Sources

Jacobson and Karels (JK) have made several modifications to the basic TCP algorithm to improve their behavior under congestion. These include 'slow start', window shutdown on a lost packet, refinements in timeout estimation, and fast retransmit in case of duplicate acknowledgments [Jac88]. We describe some of these briefly below.

In the generic TCP implementation, the implementor fixes the window size once and for all. In contrast, JK propose that the window size be dynamically adjusted to permit congestion control. Initially a source sets its window size to 1. As it receives acknowledgments, this is slowly increased ('slow start') to the maximum allowed window size, until a timeout occurs, signalling a packet loss due to congestion. At this point, the window is shut down to one, and the process resumes. Thus, congestion control is implemented by *window shutdown*. The actual window size increase algorithm is somewhat more complicated - details can be found in [Jac88].

Another interesting feature in JK protocols is 'fast retransmit'. A TCP receiver replies to packets with an acknowledgment that contains the sequence number of the last in-sequence packet it has seen so far. When a packet is lost, subsequent acknowledgments contain the sequence number of the packet before the lost packet. Thus, if a JK source receives (window-size -1) acknowledgments with the same sequence number, it detects a lost packet and retransmits the packet even before it times out.

[3] If the current estimate is R, and the actual round trip time for a packet is T, the new estimate is $kR + (1-k)\ T$, where k is a constant in $[0,1]$.

JK sources implement these improvements, as well some others suggested by Karn [KaP87]. We would like to repeat that JK sources are not exact TCP implementations - they are generic sources to which improvements suggested by Jacobson/Karels and Karn are added. They model details of TCP that are relevant to congestion control and ignore other details such as keep alive messages, out-of-band messages etc.

4.3. DEC Sources

Jain and Ramakrishnan at DEC have described a congestion control scheme based on a single bit in the header of every (data and acknowledgment) packet, the 'decbit'. A gateway sets the decbit on a data packet when it detects incipient congestion. This is echoed back by the destination to the source that sent that packet to the gateway. Sources decrease their window size when they receive a packet with decbit set. Such sources, implemented according to the algorithms presented in two technical reports from DEC [RCJ87, RaJ87], are called 'DEC sources'.

Window decrease and increase are decided at a decision point, and at every such point, the window size is either decreased or increased by one. If the previous window size was wp, and the current window size is w, then a decision point is reached when $wp + w$ acknowledgments have arrived since the last decision point. The source counts the number of packets with set decbits for the last w packets. If more than 50% of the bits have been set, the window size is decreased by one, else it is increased by one. This acts like a low pass filter on the incoming bits, and stabilizes the system. The references above discuss other aspects of the algorithm in detail.

5. Gateway Scheduling Algorithms

REAL implements several gateway scheduling algorithms. Of these, the interesting ones are first come first served (FCFS), fair queueing (FQ), and the DEC congestion avoidance algorithm.

5.1. FCFS Gateway

In an FCFS gateway both the buffer allocation and the bandwidth management scheme are FCFS. The gateway places packets in buffers in the order they are received, and drops packets that arrive when there is no more buffer space. The buffers thus form a FIFO queue, and, whenever an output line becomes idle, the next packet in the queue is sent.

5.2. DEC Gateway

A DEC gateway schedules packets FCFS, and, when it detects incipient congestion, it sets the decbits of some packets. Congestion is detected by monitoring the average queue length of an output line's transmission queue. When the average queue length is less than 1, no bits are set. When it is more than 2, bits are set on all outgoing packets. Else, the gateway sets bits on packets from host-host pairs that ask for more than their fair share of throughput. This implicitly introduces fairness into the bandwidth allocation scheme.

5.3. Fair Queueing Gateway

A fair queueing gateway implements the fair queueing scheduling policy (FQ). This policy is based on an idea presented by Nagle [Nag87], and is best described by a metaphor, the bit-by-bit-round-robin (BR) scheme (Figure 4). In this scheme, the gateway selects one bit each from each source destination pair having a packet in the gateway and sends it. BR is the fairest scheduling policy possible, since every active pair gets exactly as much bandwidth as any other. Furthermore, each host-host pair is protected from the others, in that no host can delay the servicing of any other host. Note that the unit that obtains fair service is a source-destination pair, as opposed to a source or a destination alone.

However, BR is not practical since there are significant overheads involved in switching between packets. The fair queueing policy [DKS88] approximates BR as follows : it maintains a *round number*, R, the round BR would be at any moment in time. When a packet arrives, the gateway assigns it a *finishing*

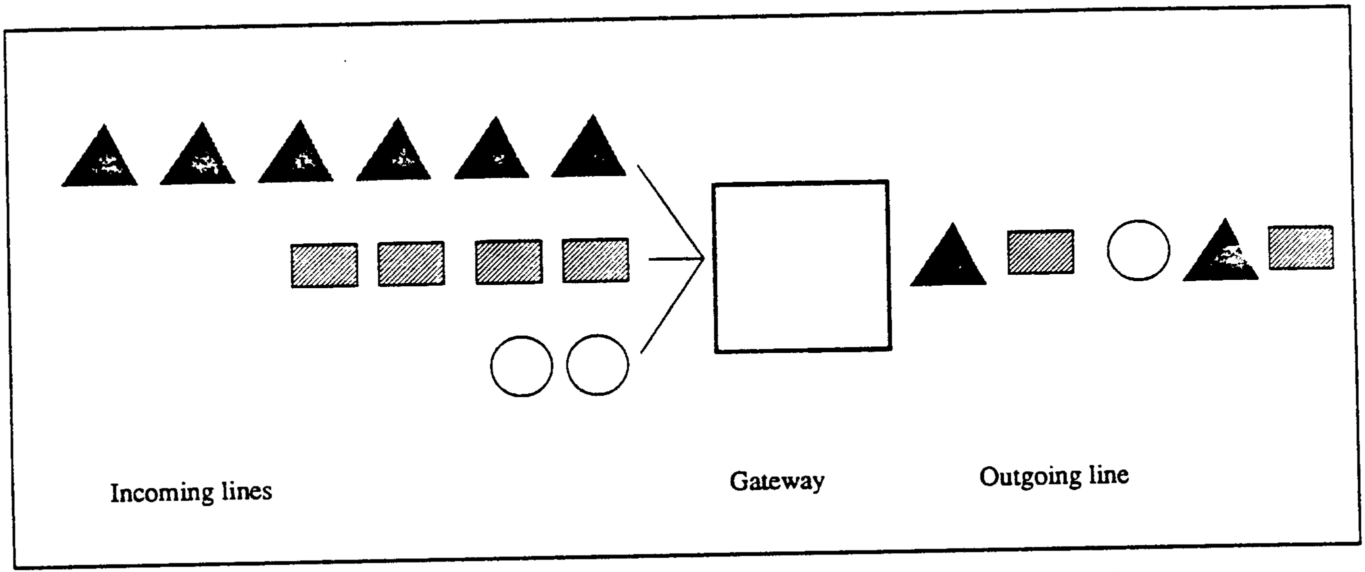


Figure 4 *The bit by bit round robin (BR) scheme. Each shape represents one bit.*

number that is the current round number plus the packet size[4]. FQ schedules the packet with the least finishing number first. However, small packets with a smaller finishing number may arrive after a FQ gateway began transmission of a larger packet with a larger finishing number. This violation does cause some unfairness in the scheduling, but this is bounded by the largest allowed packet size.

In addition to implementing FQ, a FQ gateway also allocates buffer space fairly : whenever there is not enough buffer space for a newly arrived packet, the gateway drops packets from the source-destination pair that has the most packets in the gateway.

A variant of the FQ algorithm (FQ with bit setting) sets bits on packets from source destination pairs that have filled more than a third of their fair share of buffers. This allows the sources to decrease their window size, as in the DEC algorithm, reducing the mean queueing delay that their packets experience.

6. Simulation Results

This section presents some of the results obtained from the simulator. The results illustrate the detailed study of protocols that is possible using REAL. Further comparisons are presented in a companion paper [DKS88].

The results are described in the context of a set of benchmark networks. Each benchmark tests an aspect of gateway and source protocols that we think are important in real networks. Further, parameter values have been chosen to reflect those of the Arpanet. For example, long haul line speeds are 56kbps. Six protocol pairs were tested on each benchmark : these are presented in Table 1. We now discuss the intent of each benchmark, and the results obtained for that benchmark.

6.1. Single Underloaded Gateway

This benchmark is intended to test the system in unstressed conditions (Scenario 1). Protocol/scheduling policy pairs (henceforward called protocol pairs) that behave badly in this situation are clearly unacceptable. As it turns out, all the protocol pairs examined behave well in this benchmark. It is interesting to note the delay incurred by telnet packets in fair queueing (FQ) gateways . Since telnet packets are small, their finishing numbers are only slightly above the current round number, and are automatically given preference over larger FTP packets. Thus, they processed and sent with very low delays, which

[4] Actually, MAX (R, finishing number of the last packet from this host-host pair) + packet size.

Scenario 1

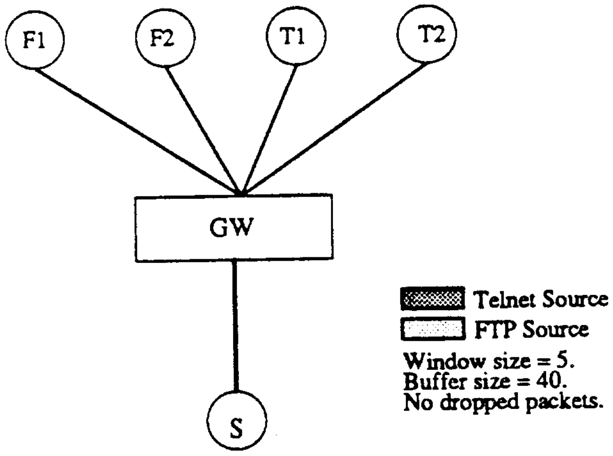
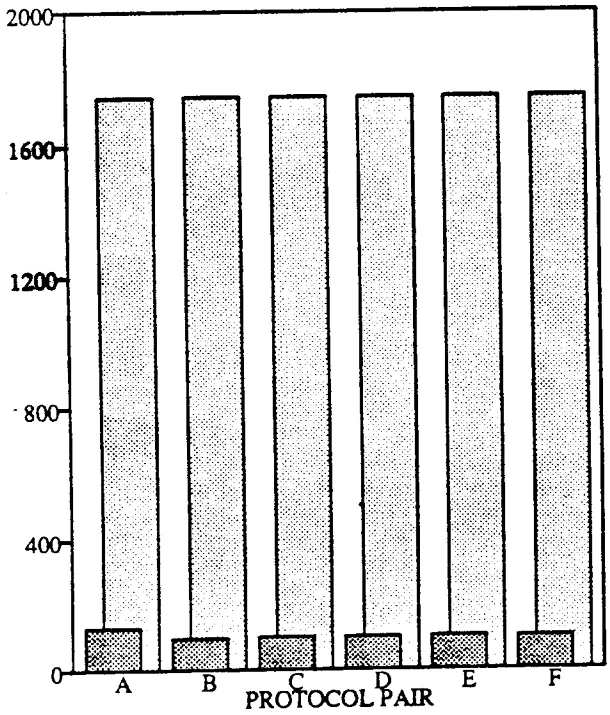
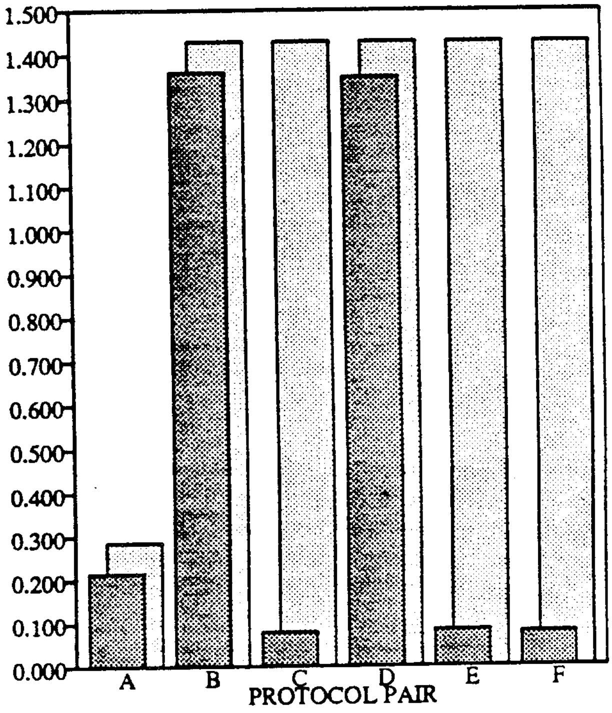


Table 1 : Legends for Histograms

Legend	Transport Protocol	Gateway Policy
A	DEC	DEC
B	Generic	FCFS
C	Generic	FQ
D	JK	FCFS
E	JK	FQ
F	DEC	FQ + bit setting


Throughputs for Typical Sources
Underloaded Network

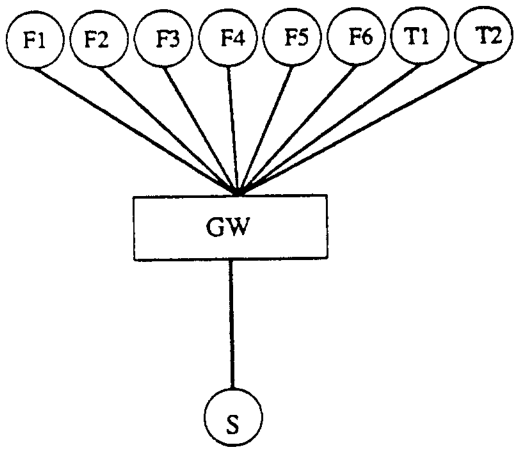

Round Trip Times for Typical Sources
Underloaded Network

is desirable.

6.2. Single Overloaded Gateway

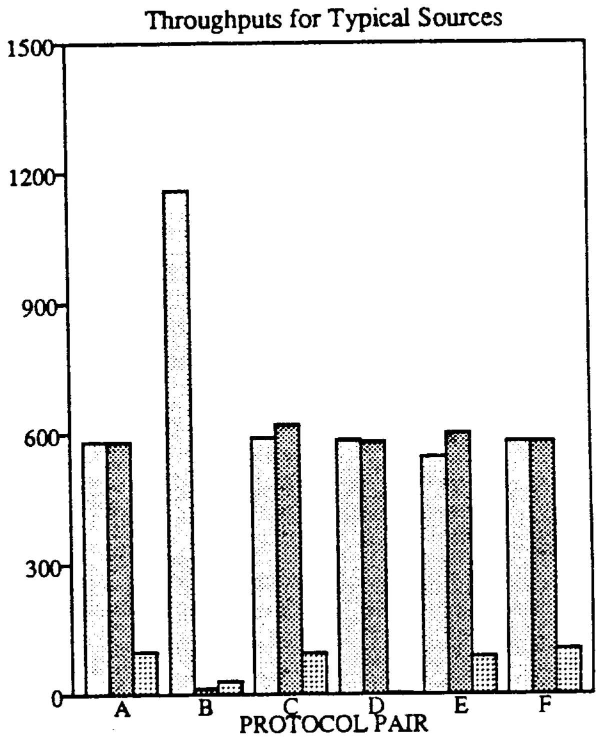
This benchmark tests a gateway under severe overload (Scenario 2). Six FTP sources, each with a maximum window size of 5, send packets to a gateway with only 15 buffers. Congestion can arise when a source is unable to keep a full window's worth of packets in each intermediate router, as in this benchmark.

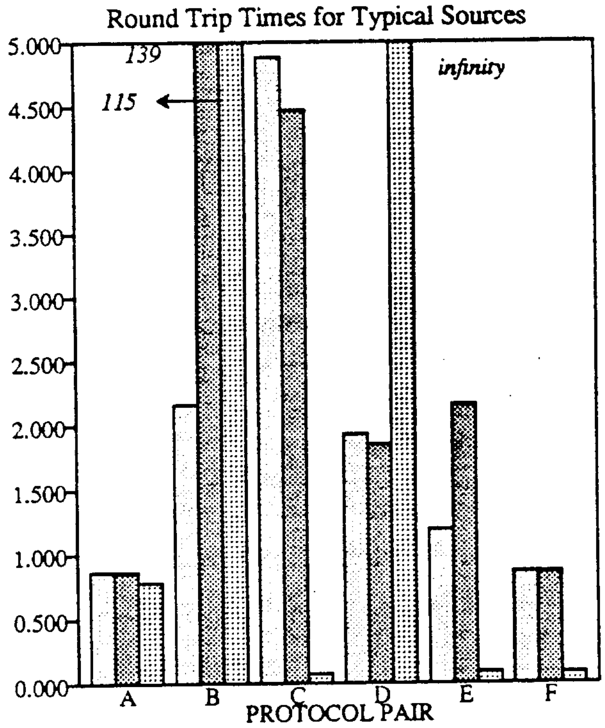
Scenario 2

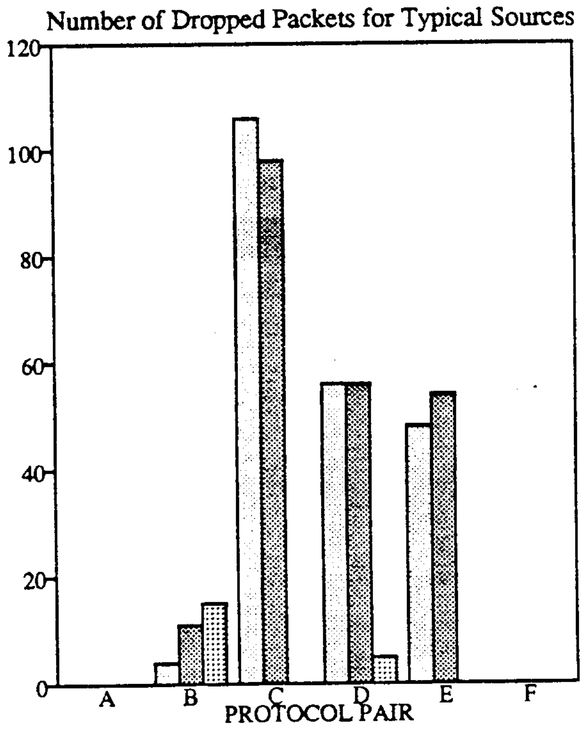


□ Winning FTP Source
▩ Losing FTP Source
▦ Telnet Source

Window size = 5.
Buffer size = 15.







If a protocol pair behaves badly under congestion, then it will do badly here.

In this scenario, packets from each source have one hop on a slow line. The transmission delay on this line is long enough for a source to receive an acknowledgment for the previous packet and send a new packet to the gateway. If each source keeps its window size at two, there will always be a packet waiting to be sent on the slow line, and an increase in the window size will not help. However, if a source expands its window size to the maximum size or does not adjust its window size, there will not be enough buffers. The gateway will be forced to drop packets either from the offending source, or from some other source. This could lead to spurious retransmissions, some hosts being shut out of the network, or to other undesirable behavior.

The DEC protocol pair does not exhibit any of these problems (case A). The DEC/(FQ with bitsetting) scheme performs equally well, and gives a lower delay to packets from telnet sources (case F). However, the other protocol pairs do exhibit problems due to congestion; we examine these below.

The Generic/FCFS pair has no congestion control and the system reaches a stable state with some sources sending packets as fast as they can (winners), while others do not get any packets through (losers)(case B). This behavior, which we call 'segregation', was first observed by Sturgis at Xerox PARC [Stu88]. Packets from winning gateways arrive shortly after the gateway has processed one of their packets and has released a buffer space. So, they are rarely dropped. On the other hand, when packets from losing sources arrive at the gateway, usually all the buffers are full. Thus, packets from losers are preferentially dropped.

When FQ is used at the gateway along with generic sources, there is a slight improvement (case C). This is because a source is protected from other sources, and will not lose packets due to a burst of packets from another source. Nevertheless, a source is able to get only two and a half buffers in the gateway on average, whereas the window size is five. Hence, bursts of three or more packets from a source will inundate the gateway, leading to dropped packets, retransmission and segregation.

With JK/FCFS all the FTP sources get a fair share of the bandwidth (case D), but the telnet sources get no throughput. The reason is that JK FTP sources keep increasing their window size till the gateway buffers are full. Thus, packets from the intermittent telnet sources get dropped. In general, we feel that the JK/FCFS pair will suffer from such problems, since the intermittent (telnet) sources are never protected from the established (FTP) sources, and are vulnerable to starvation.

The gateway drops packets with both JK/FCFS and JK/FQ pairs (cases D and E). This is because of the steady attempt by JK sources to increase their window size. The increase proceeds till a packet is dropped, which triggers a window size shutdown to one, after which the cycle repeats.

6.3. Ill behaved Hosts

An ill behaved source sends packets to the gateway as fast as it can, limited only by network bandwidth. Such a source is intended to model a failure mode of the system, when a host floods a gateway and ignores the congestion that it thus causes. We examine the behavior of each protocol pair in the presence of such a source.

In general, FQ gateways protect well behaved sources from the ill behaved one. This is expected, and demonstrates the 'firewall' nature of fair queueing. An interesting question is why the ill behaved source does not even get its fair share of the output bandwidth. The reason is a policy decision to increment a source's finishing number even if the packet that caused this increase is dropped. Thus, if a source chooses to ignore the danger signal of dropped packets, it will cause further damage to itself.

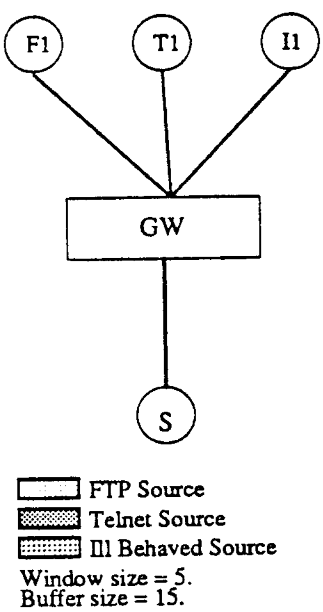
6.4. Summary of Results

We note that congestion control or avoidance requires coordination between the source transport protocol and the gateway scheduling policy. The simulations isolate some points of failure of current TCP implementations and FCFS gateways. REAL can be modified to analyze modifications to these protocols or alternate protocols.

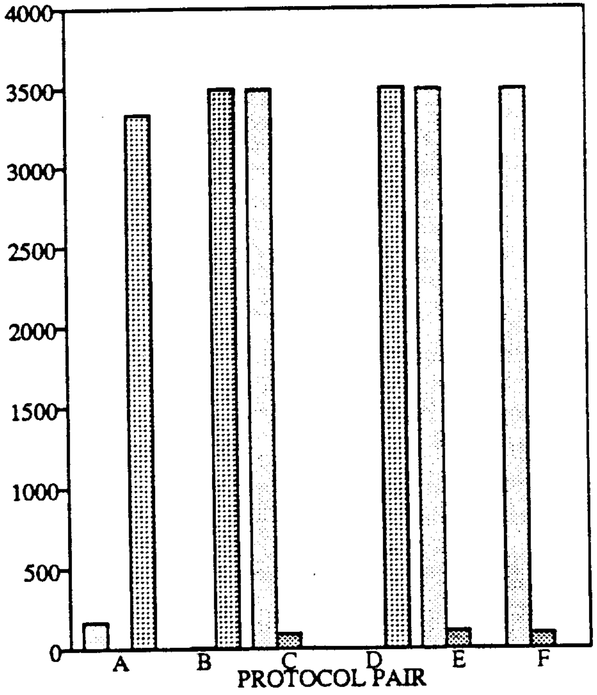
Fair queueing gateways seem to offer several advantages over existing FCFS gateways. These advantages are described in greater detail in [DKS88].

Simulations of small networks, such as those described above, do not adequately model the behavior of networks with many hundreds of gateways and sources. For example, these simulations ignore adaptive routing routing. However, it is slow and cumbersome to simulate networks with more than about 100

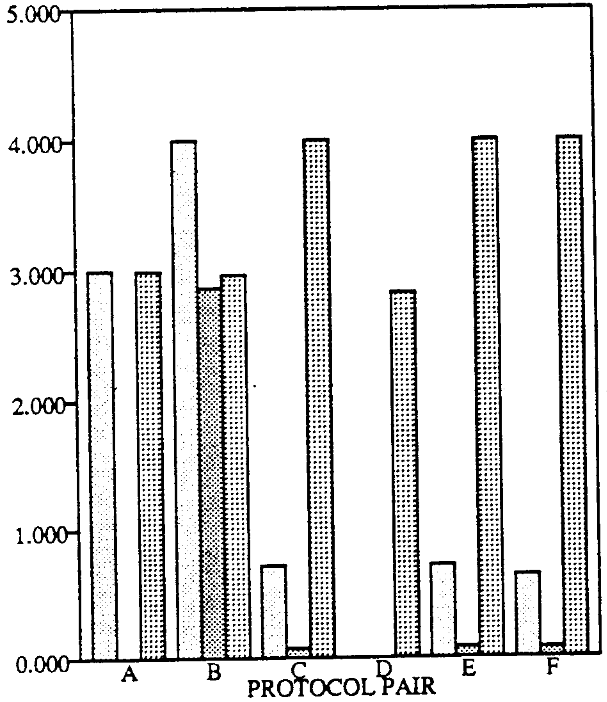
Scenario 3



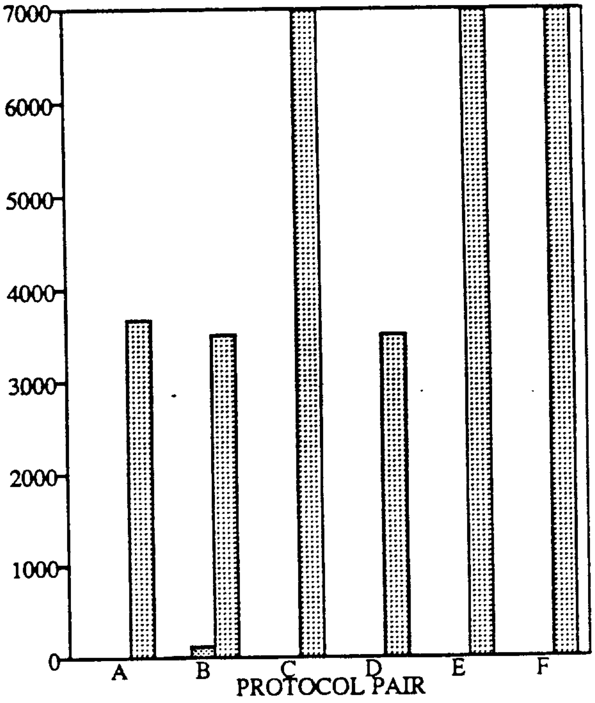
Throughputs for Typical Sources



Round Trip Times for Typical Sources



Number of Dropped Packets for Typical Sources



nodes on REAL. This motivated us to distribute the simulator on a cluster of computers, allowing us to simulate much larger networks (Section 8).

7. Performance Evaluation

This section discusses the simulator's performance. The time to perform a simulation depends on a number of factors; among them, the protocols being simulated, the network configuration, the machine on which the simulation is carried out, and the machine's load. For simplicity, we decided to fix all parameters except for the number of sources, and only studied the dependency of simulation time on the number of sources in the simulated network.

The standard simulation we chose was, quite arbitrarily, the network shown in Figure 3. All sources executed the JK protocol with an FTP workload, and the gateway did FCFS scheduling. We wanted to study scaling when the network was underloaded, so we adjusted the buffer size to ensure there was no congestion. Thus, when there were n sources in the simulation, and the maximum allowed window size was 5, we allocated 5n buffers to the gateway.

The bottleneck resource in the system was the bandwidth of the line connecting the gateway to the sink. Simulations ran for 500 seconds of simulated time, so in each simulation the same number of packets were transmitted on the bottleneck : the bandwidth of the bottleneck times the simulation time. In this set of simulations, the bottleneck line speed was 7 packets/second, so 3500 packets were transmitted in each simulation.

The simulations were run on a Vax 8600 when the load average ranged from 1.0 to 1.1. The only other user process on the machine was a low priority simulator that had been running for several days and presented a relatively constant load to the system. Thus, we believe that the results are free from anomalies arising from fluctuations in the machine workload.

Figure 5 presents the results of the evaluation. The number of sources in the simulation is reported on the horizontal axis and the simulation time (in minutes) on the vertical axis.

Observe that the simulation time varies roughly linearly with the number of sources. Also, the simulation scales well, with about seven more seconds of real time needed for each additional source.

We hypothesized that the simulation time is also directly proportional to the total number of packets transmitted and tested this by running the simulations for 500 and 1000 simulated seconds. If the hypothesis were true, the simulation time should nearly double. This is confirmed by Figure 5. Thus, we conclude that the simulation time depends approximately linearly on the number of sources in the simulation and on the total number of packets transmitted in the simulation. Of course, the result is derived from the single case that we tested, but it is hoped to be an indication of the general nature of things.

8. Extension to Parallel Distributed Simulation

In this section, we describe a distributed version of REAL that is able to simulate reasonably large networks.

The key to distributed simulation lies in the monitor facility provided by NEST. NEST supports the idea of simulation pass, which is an interval of simulated time. At the end of each pass, control flows to a user specified monitor function that can access the simulation's address space. The monitor executes in zero simulated time.

The distributed version of REAL (DisREAL) binds together a number of REAL instances. It implements a variant of barrier synchronization, where the monitor of each REAL instance blocks till all the other monitors have completed their slice of the simulation. At this point, all messages that cross computer boundaries are routed to the proper REAL instance, and the next pass is started. Note that synchronization is done at the end of every pass. If the synchronization overhead is high, then choosing a small pass time is potentially very expensive.

A master-slave configuration makes it easy to practically implement barrier synchronization. Each REAL instance runs as a slave and executes a pass. At the end of each pass it sends the DisREAL master a message indicating completion. When the master has received all the completion messages, it routes cross-computer messages and sends each slave a 'proceed' message. However, there is an important detail here.

Consider a cross-computer message M that is sent from NEST instance N1 to instance N2. If M is sent from N1 at the beginning of a pass, and should be received by N2 before the end of the pass, then the barrier scheme does not work since M will be delivered by the master only at the beginning of the next

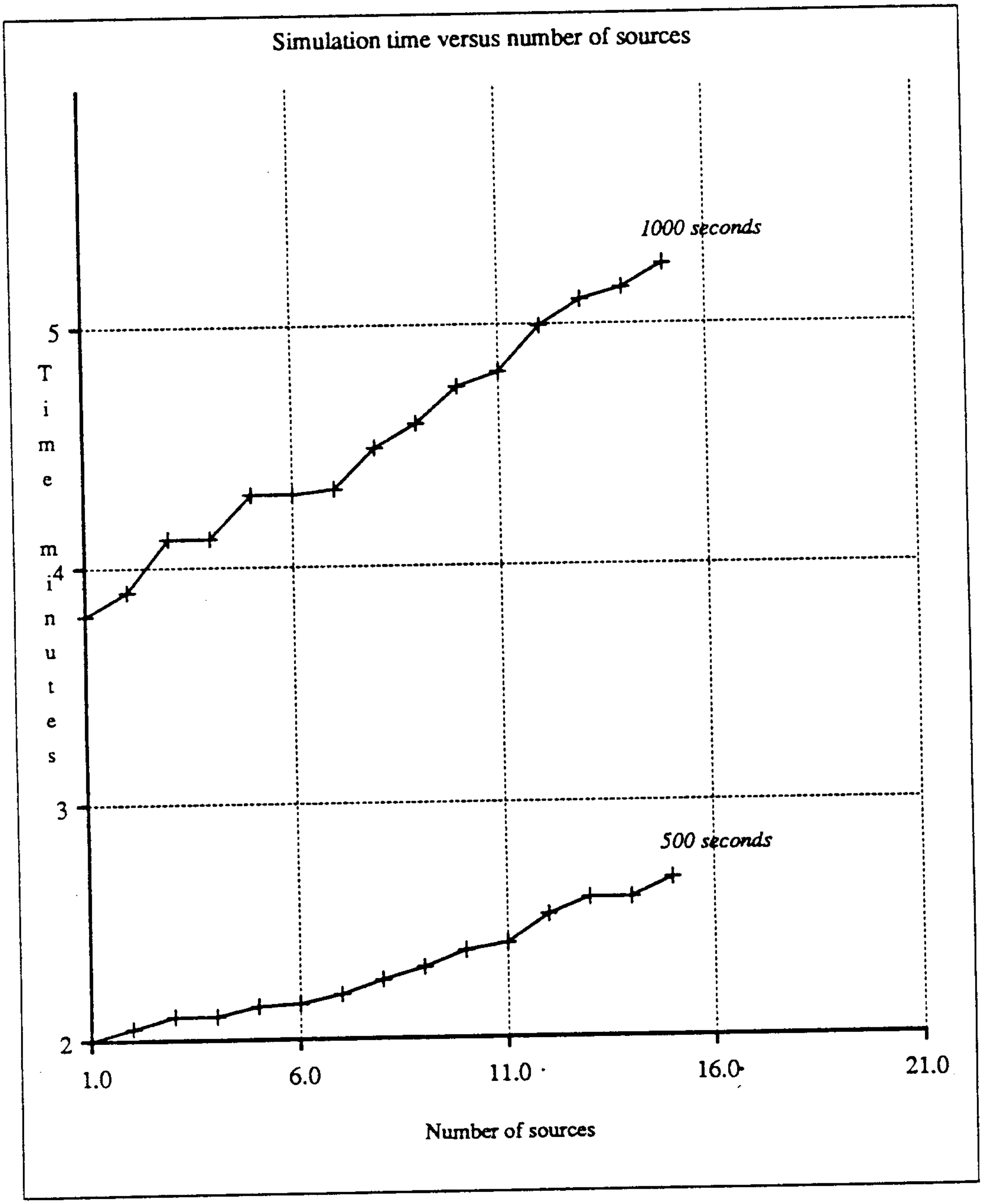


Figure 5 *Results of performance evaluation*

pass. There are two ways to get around this. In the Timewarp [Jef85] style of synchronization, N2 would proceed beyond the time it should have received M. When M is actually received, N2 will need to do a rollback to the previous state. This is costly and undesirable. Our solution is to impose the restriction that every cross-computer message must have a delay that is greater than the pass time. This guarantees that there will be no need for a rollback. This is illustrated in Figure 6. We see that, as long as the delay in the message is greater than the pass time, it is necessary that at least one barrier be crossed during the transit time of the message. This is sufficient to deliver the message on time.

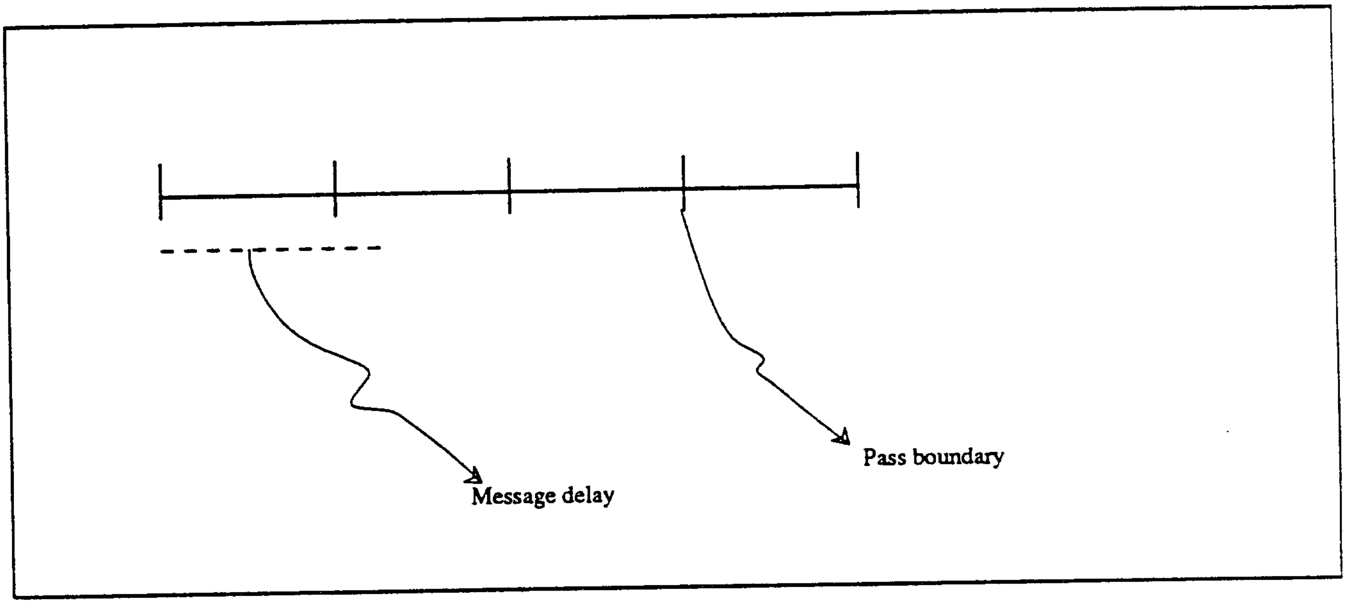


Figure 6 *If a message has a delay greater than the pass time, the barrier scheme guarantees on-time delivery*

Our scheme has two advantages. First, it is easy to implement, and, unlike the Timewarp scheme, it does not need rollback. Second, it models the structure of real networks. Currently, long haul networks have relatively fast LANs on the ends, with much slower (56kbps) intermediate trunks. Long haul line delay is an order of magnitude larger than LAN line delay. If each NEST instance contains all the nodes in a local area, the messages we require to have high delay will be the same as those sent on the (high delay) trunks. Thus, we can simplify the distributed simulator by exploiting the nature of the target network. We are currently implementing this scheme.

With such an implementation, networks of reasonably large size can be simulated. We estimate that, with 100 nodes per NEST instance, and on a network of 20 Suns, we should be able to accurately simulate networks of 2000 nodes in about the same time as it takes to simulate 100 nodes on a single Sun.

9. Evaluation

Our primary design goals were to simulate reasonably *large* networks *realistically*. In our opinion, we have had reasonable but limited success in meeting these goals. By simulating certain aspects of TCP protocols in detail[5] we have obtained insights that we would not otherwise have. Yet, we ignore other, perhaps equally important, details about packet buffering, fragmentation/reassembly, copying and checksumming. We are currently able to simulate networks with as many as 100 nodes. However, the proposed extension will allow us to simulate networks with a few thousands of nodes. on a local-area distributed system of moderate size.

There are several payoffs from the 'visual programming' approach. One is a simple and intuitive user interface. Further, simulation scenarios can be set up easily with the mouse based display client. Finally, one can graphically monitor the state of the simulation, making it easier to verify protocol correctness.

[5] Specifically details of flow control, timeout estimation and retransmission, and others relevant to congestion control.

10. Future Work

Our immediate goal is to complete the implementation of the distributed simulator outlined above. Once this is done, we will simulate much larger networks.

The choice of a benchmark to evaluate transport protocol behavior is an open problem. Such a benchmark must specify the system topology, line speeds, gateway buffer sizes and the system workload. Furthermore, a simulation of a transport protocol and a gateway scheduling policy on the benchmark should accurately predict protocol performance. We hope to create and justify a set of benchmarks in this spirit.

Finally, we will use the simulator to implement and test a variety of congestion control schemes.

11. Acknowledgments

This work was done at the Xerox Palo Alto Research Center under the guidance of Dr. Scott Shenker and Dr. Alan Demers. I gratefully acknowledge their generous advice and help. Thanks also to Christian LeCocq at Xerox PARC, Alex Dupuy at Columbia University and Peter Danzig at UC Berkeley for their support.

12. References

[BDS87] Bacon, D., Dupuy, A., Schwartz, J. and Yemini, Y., "Nest : A Network Simulation and Prototyping Tool", *USENIX 87*, 1987.

[DKS88] Demers, A., Keshav, S. and Shenker, S., "Analysis and Simulation of a Fair Queueing Algorithm", *Submitted for Sigmetrics 89*, September 1988.

[FrM84] Fraser, A. and Morgan, S., "Queueing and Framing Disciplines for a Mixture of Data Traffic Types", *AT&T Bell Laboratories Technical Journal 63*, 6 (1984), 1061-1087.

[Jac88] Jacobson, V., "Congestion Avoidance and Control", *ACM SigComm Proceedings* , August 1988, 314-329.

[Jef85] Jefferson, D. R., "Virtual Time", *ACM Trans. Prog. Lang and Systems 7*, 3 (July 1985), 404-425.

[KaP87] Karn, P. and Patridge, C., "Improving Round-Trip Time Estimates in Reliable Transport Protocols", *ACM SigComm Proceedings*, 1987, 2-7.

[Nag87] Nagle, J., "On Packet Switches with Infinite Storage", *IEEE Trans. on Communications 35* (1987), 435-438.

[RCJ87] Ramakrishnan, K. K., Chiu, D. and Jain, R., "Congestion avoidance in Computer Networks with a Connectionless Network Layer - Part IV - A Selective Binary Feedback Scheme for General Topologies", *Dec. Technical Report-510*, November 1987.

[RaJ87] Ramakrishnan, K. K. and Jain, R., "Congestion avoidance in Computer Networks with a Connectionless Network Layer - Part II - An Explicit Binary Feedback Scheme", *Dec. Technical Report-508*, April 1987.

[Smi87] Smith, R. B., "Experiences with the Alternate Reality Kit : An Example of Tension between Literalism and Magic", *CHI+GI 1987 Conference Proccedings*, 1987.

[Stu88] Sturgis, H., *Private communication*, 1988.

[Tan81] Tanenbaum, A. S., *Computer Networks*, Prentice Hall, Englewood Cliffs, NJ, 1981.