A Model of TCP/IP Suitable for Parallel Simulation of Large Internets

Marc Abrams

TR 92-10

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

April 10, 1992

A Model of TCP/IP Suitable for Parallel Simulation of Large Internets

(Model Version 0)

TR 92-10

10 March 1992

Marc Abrams

Computer Science Department
Virginia Tech
Blacksburg, VA 24061-0106
abrams@cs.vt.edu

Abstract

Our objective is to investigate the effects of scale on the Internet. The current Internet connects on the order of 10^5 hosts. What unanticipated network dynamics will arise as the Internet grows to a million or more nodes? This report describes a model of TCP/IP based on extended finite state machines intended to permit simulation on this scale. The model yields the throughput and delay averaged over all connections of a workload consisting of client and server hosts interconnected through a subnet of gateways. A variety of parameters characterize the workload, protocol software speed, subnet, and internet topology. The model presented embodies a number of conjectures on what aspects of a protocol should be omitted to permit simulation of large numbers of nodes, and conjectures on what protocol aspects should be included to give qualitative, rather than quantitative, insight into scaling the Internet.

Contents

A Model of TCP/IP Suitable for Parallel Simulation of Large Internets

1. Introduction

One vision of the role that computers will eventually play in society is analogous to the role of a telephone. Just as a telephone is useless unless it is plugged into a telephone network, so will a computer be of limited utility unless plugged into a computer communication network. And, just as any telephone can call any other telephone in the world, so one day will any computer be able to access any other computer in the world. NSFnet, MILNET, and many corporate networks are rapidly reaching this vision.

The key technology required to interconnect computers worldwide is *internetworking*. We will henceforth refer to a personal computer, workstation, mainframe, or supercomputer that you or I use as a *host* computer. Today there exist thousands of local area networks around the world that each directly connect dozens of hosts. An *internet* is a combination of hardware and software that interconnects individual local area networks into what appears to be a single, unified system through which hosts in different parts of the world may communicate. Technically, internetworking is achieved by *gateway* computers. A gateway passes data from one local area network to another local area network.

An internet composed of hosts, local area networks, gateways, and connections between gateways has some similarity to highway systems. Just as traffic engineers must design and enhance highways to accommodate the existing volume of cars and trucks, so must internetwork designers provide sufficient capacity to carry existing volumes of data. Drivers experience traffic jams and congestion in major cities, through mountain passes, during peak periods, or when accidents occur. Similarly, data flowing over the internetwork may slow to a snail's pace when its destination is a mainframe, its only route is through a heavily used cross-country fiber cable, during peak periods, or when a gateway crashes, respectively. Access to roads is controlled by a variety of rules and conventions, such as traffic lights, stop signs, and right-of-way rules. Similarly, hosts exchange data over the internet using the rules and conventions of a

communication protocol. Most experience that exists in internet design is based on the protocol suite called *TCP/IP* (transmission control protocol/internet protocol).

A problem of immense commercial important is to identify what protocols best manage the data pathways of an internet. Often details of the algorithms and methods used to implement a protocol and the parameters used with the protocol (analogous, for example, to the timings of a traffic light) dramatically affect the data capacity of the internet. Therefore network designers need to model the internet to evaluate proposed protocol algorithms and parameter settings before they are installed and unleashed on the world. The need to model is growing more critical because the rate at which data can be transmitted over a network soon will increase a hundred-fold. Increased data communication rates will permit new uses of geographically distributed computers. For example, video, voice, and data could be transmitted over a common network. Desktop workstations could display graphic images from scientific computations on remote supercomputers.

Modeling internets presents a paradox: internets today connect tens to hundreds of thousands of hosts, yet experiments are practical with at most dozens of hosts and gateways. Furthermore, no satisfactory analytic models exist. This leaves simulation as the only practical method. However, the speed and memory size of any single computer -- even a supercomputer -- limits the number of hosts and gateways that may be simulated. This paper addresses the critical question of whether an arbitrarily large internet model could be simulated, given a large enough parallel processor. The question is worth exploring because characteristics that have led to good parallel simulation performance in the past -- loose coupling among model components and *look-ahead* (model components can predict some future actions before receiving enabling events) -- appear to exist in internet models.

The following two sections give some background useful to understanding the TCP/IP model. Readers without knowledge of how TCP/IP works should read the communication network primer in Section 2. For readers with a background in performance modeling, Section 3 describes why the internet cannot be modeled with product form queueing networks. Protocol experts may wish to proceed directly to

Section 4, giving the problem description which the model solves, Section 5, giving the simplifying assumptions, Section 6, giving the workload that we model, Section 7, overviewing the TCP/IP model itself, Section 8, giving the project status, and Section 9, describing enhancements to the TCP/IP model that are necessary to make it nearly identical to the specification. Appendix A contains the TCP/IP model, specified as a set of extended finite state machines.

2. A Communications Network Primer

(Readers familiar with internetworking and TCP/IP should skip this section.) To understand the internet model presented later, a few principles of data communication are presented next. Tanenbaum's classic introduction contains further information [TANE], and Comer gives a good introduction to internetworking with TCP/IP [COME].

To pass data from a *source* host to a *destination* host on a different local area network, the data first passes from source host to a gateway; the gateway forwards the data to another gateway, and this continues until the data reaches a gateway attached to the local area network of the destination host. To transmit data from Washington, D.C. to Tokyo may require the data to be received and forwarded several times as it passes through a sequence of gateways.

An internet can be described as a graph whose vertices each correspond to a *node*. A node may function only as a host, only as a gateway, or both as a host and as a gateway. An edge exists between two vertices in the graph if each node can directly send data to the other node over a physical channel. Every subset of graph nodes that are directly connected forms a *network*. Each node that connects to two or more networks must be a gateway.

Consider the sequence of events that occurs to allow host A to communicate with another host B. Host A must first establish a *connection* with host B. (In our simulation model, each connection that host A has established is identified by a *local connection number*, which is a small integer starting at 1.) This is analogous to dialing a telephone number and waiting for an answer. Then hosts A and B may send each

other one or more messages over the connection, which is analogous to a telephone conversation. Finally, the two hosts release the connection, which is analogous to hanging up the telephone. For simplicity, our simulation model will assume that all connections have been made prior to the start of simulation time, and are released after completion of the simulation. Therefore we need only consider the actual transfer of messages. The end points of a connection must be hosts.

Each node is assigned one or more *IP addresses*. A connection, in this paper, is identified by the IP addresses of each of its end points. AN IP address can be partitioned into two fields: *netid* and a *nodeid*. (See Comer for more details [COME, pp. 62-63].) The *netid* specifies the name of a network. The *nodeid* specifies the name of a host or gateway attached to a particular *netid*. We will denote an IP address as *netid.nodeid*. Therefore IP addresses 1.0 and 1.1 denote two nodes on the same network.

Each copy of software implementing a protocol that runs on a computer is referred to as an *entity*. Each node that can function as a host runs one TCP protocol entity. TCP views the internet as a single unified, but unreliable network. Its job is to insure reliable delivery of messages. TCP is layered on top of IP. Each node runs one IP protocol entity. An IP entity is *local to* a TCP entity if the two entities run on the same host. IP is aware of the physical composition of the internet as gateways connecting networks. IP also knows the size of data unit that may be transferred across each individual network of the internet.

When a host sends a message to another host, the message may be arbitrarily long. The TCP entity running on the sending host partitions the message into a sequence of *segments*. TCP passes each segment in sequence to the IP entity. IP decides how large a unit of data to send through the internet; each unit of data is called a *datagram*. In general, IP knows the maximum size of data, in bytes, that each network connecting a pair of gateways can carry; this quantity is called the *maximum transfer unit* (MTU) for that network.

For simplicity, we assume that all networks comprising the internet have an identical MTU, and that all segments containing data sent by a TCP entity have identical length, called the *maximum segment*

size (MSS) which is equal to MTU minus the IP header size in bytes. For simplicity we assume the message length is always an integral multiple of MSS.

When IP receives a segment from TPC, IP puts the segment into one IP datagram (analogous to putting a letter in an envelope). At this point the datagram must be delivered to the IP entity of the destination host. Each IP entity will compare its netid to the netid of the destination IP address. If the two match, then IP entity and the destination are connected to a common network, and the IP entity directly delivers the datagram to the destination. Otherwise the IP entity selects a gateway on a common network to which the datagram is forwarded. (This case corresponds to the datagram making an intermediate hop in its path from a source host to a destination host.) The selection algorithm first uses an *IP routing table*, if present. The table contains a set of entries, each corresponding to a different network in the internet. An IP entity holding a datagram searches the table for an entry corresponding to the destination netid of the datagram. If an entry is found, then the entry contains the IP address of a gateway to which the datagram is directly delivered. Otherwise, the datagram is delivered to a default gateway address.

Eventually the datagram is routed to the IP entity running on the destination host, which passes the segment contained in the datagram to the TCP entity running on the destination host. The destination host TCP now informs the sending host TCP that the segment was received by sending an acknowledgement for the segment to the source host. This acknowledgement is carried in an IP datagram and is routed through gateways just as any other segment is routed. (The process of sending a segment and then waiting for an acknowledgement is analogous to sending a friend a letter and then expecting the friend to send a letter back to you acknowledging receipt of the original letter.)

Two details must be added to complete this picture. First, TCP does not simply send a segment and then wait for an acknowledgement before sending the next segment. (If TCP did this, then the rate at which TCP sends segments will be limited by the round trip delay time. For satellite transmissions, this rate would be only one segment each half second.) Instead, TCP optimistically sends W segments before waiting for an acknowledgement; W is called the *receiver advertised window size*. For simplicity we

assume that sending and receiving TCP entities use the same fixed window size of W for all connections during simulation. (Normally, TCP dynamically adjusts the window size during the lifetime of a connection.)

The second detail that must be added arises in recalling that TCP presumes the internet may sometimes loose or corrupt datagrams. For example, a gateway may be swamped by datagrams from all over the world that should be routed to the same gateway, have no memory in which to store the datagrams, and will simply discard some datagrams. Therefore, TCP maintains one timer for each unacknowledged segment over any connection. Whenever a data segment is sent over local connection number i, the timer is set to expire after time TO_i. When TCP receives an acknowledgement for a segment, it disables the corresponding timer. If a timer expires (implying TCP did not receive an acknowledgement for corresponding segment), TCP retransmits the segment.

The performance of TCP is related to the value chosen for TO_i. If the value is too small, TCP will be impatient and will flood the internet with multiple copies of segments. If the value is too large, the time required to transmit a message when losses occur will increase. Therefore the value of TO_i should vary with the level of congestion present in the internet. The *round trip delay* of a segment is the time required for the segment to transit from the source host to the destination host plus the time expected for an acknowledgement to transit from the destination host back to the source, provided that the segment and its acknowledgement are not lost. Ideally, the value of TO_i should be a little longer than the round trip delay that the next segment to be sent will experience if it and its acknowledgement are not lost. The value of TO_i, for all i, is a constant TO in the TCP/IP model of the Appendix.

3. Why Not Use a Queueing Network Model of TCP/IP?

Figure 1 depicts a high level model of the Internet, and illustrates Host A sending a message to Host B. When Host A sends a datagram containing part of the message, the datagram enters the network and traverses a sequence of gateways until it reaches Host B. Host B then returns the customer, which is now an acknowledgement datagram, through the Internet to Host A.

Six additions to this basic picture violate product-form assumptions for queueing networks:

1. When Host A sends a datagram, a copy is made of the datagram before the datagram enters the Internet.. The copy is enqueued in a *retransmission queue.*

2. A datagram sent by Host A can only enter the Internet if the window is open. The window may be imagined as a pipe extending though the internet; the pipe can hold at most W datagrams. (The W datagrams excludes acknowledgements and retransmitted datagrams.) Host A blocks until the window is open. (Product form models with just this addition can be formulated, by using a closed queueing network with W customers. See Schwartz for more details [SCHW, Section 5.2].)

3. The Internet is a network of queues, each with finite buffers. If a datagram arrives at a server and no buffer is available, the datagram is deleted.

4. Each datagram that enters the Internet may follow a different path through the network, according to a certain routing algorithm. The network has feedback and cycles.

5. When Host B returns an acknowledgement to Host A, the acknowledgement will cause the datagram it acknowledges, if any, in the retransmission queue to be deleted.

6. The service time of a datagram in the retransmission queue is the value of TCP's retransmission timer at the time the datagram begins service. The timer value is a function of the round trip delay of all datagrams that were acknowledged and were not retransmitted.

4. Problem Description

Given the parameters listed in Table 1, and assuming that the parameters specifying time are given in units of seconds, write a specification of a simulation model to estimate:

- the throughput, in segments per second, and round trip delay, in seconds, seen by the application layer of messages transmitted over each transport connection that exists between any two hosts during the simulation.

- the total number of segments retransmitted by all TCP entities, and

- the total number of datagrams dropped at any IP entity.

5. Simplifying Assumptions

The following assumptions are made. Assumptions 1 to 4 were explained in Section 2.

1. The graph describing the network interconnecting nodes being modeled is fixed during simulation.

2. A node may function exclusively as a gateway, exclusively as a host, or as both a host and as a gateway. One TCP entity runs on each host. One IP entity runs on each node.

3. All networks comprising the internet have identical MTU's. Therefore IP never fragments a datagram. All segments containing data have identical length (denoted MSS), which is equal to MTU minus the length of an IP header in bytes. The length of a message passed to TCP is an integral multiple of MSS minus the TCP header size in bytes.

4. All sending and receiving TCP entities use a fixed window size of W.

5. Whenever a receiving TCP entity receives a data segment, the entity immediately constructs and passes to IP an acknowledgement segment for the data segment. Acknowledgement segments are never piggybacked onto data segments. (Therefore every datagram contains either a data segment or an acknowledgement but not both.)

6. Each TCP entity has infinite buffers.

7. Each IP entity has two buffer pools. One, of finite size, is shared by all datagrams containing data segments (but no acknowledgements, as stated in assumption 5) received from another IP entity; the number of buffers is the same for all IP entities. The second pool has infinite size and is used for datagrams containing acknowledgements (but no data segments, as stated in assumption 5) and, if the IP entity is running on a host, all datagrams containing data segments that arrive from the TCP entity. A gateway discards incoming datagrams if and only if it has no free buffers. (This has two implications: acknowledgements are never lost, and the TCP and IP entities on the same host can exchange segments at an arbitrarily high rate without loss.)

8. Datagrams are only lost for the reasons explained in assumption 7. Furthermore, data is never corrupted during transit.

9. IP generates no source quench messages. (Therefore only TCP, and not IP, takes any action to avoid or recover from network congestion.)

10. The routing tables used by IP are static. (Therefore all segments sent between two hosts will follow the same path through the internet.)

11. For simplicity, TCP's multiplicative decrease congestion avoidance and additive increase recovery algorithms and the sender congestion window are not modeled.

12. A sending TCP entity does not regularly transmit to the receiving TCP when the window is zero [TCP spec, p. 42].

13. The retransmission timeout value for all connections is fixed and equal to constant *TO*.

14. The PSH control bit is set in the TCP header of the last segment sent for each message.

15. All datagrams are treated the same by an IP entity. Therefore datagrams inbound from the network do not have priority over outbound datagrams.

16. Each connection is identified by the IP addresses of its two end points. TCP ports are not modeled. Hence each host can have multiple connections, but can only provide one service to all connections.

17. TCP/IP data transfer is modeled. TCP connection establishment and release is not modeled. Therefore the initial condition of the model is that all connections required by the workload are established.

6. TCP/IP Model

Table 1 contains parameters required by the TCP/IP model. Each of the N nodes being modeled is assigned a unique *physical address* in the range 0,1,...,N.

Network topology: Routing matrix *ROUTE[p,i]* and *IP[p]* implicitly (and redundantly) specify the graph of node interconnection. A node with physical address p is connected to network n if and only if *ROUTE[p,n]* is NULL and n is the netid of some address in set *IP[p]*.

Model description: The time delays represented in the model are discussed first, followed by a description of the three layers comprising the model.

Time delays: Figure 2 illustrates the model parameters that represent delays. The notation x:y is used to denote an IP address with netid x and nodeid y. The diagram illustrates two hosts and two gateways. The two hosts run TCP and an application. The two gateways have multiple IP addresses. Three networks are illustrated: numbers 10, 40, and 80. The propagation delays over the three networks are represented by random variables *PROP[10]*, *PROP[40]*, and *PROP[80]*, respectively. Queueing delays due to contention by different nodes on the same network for access to a shared communication medium is not modeled and is assumed to be zero. The networks could be local, metropolitan, or wide area networks; only the propagation delays would differ.

The total time required for the protocol to execute is lumped into a single random variable for each host and gateway. Lumping the times into a single parameter simplifies the model, and may represent sufficient detail in a model of millions of nodes. Random variable P_H denotes the time required to process

one datagram and the corresponding segment at a host. Random variable P_G denotes the time required by a gateway to process one datagram at any gateway. Table 1 precisely defines the activities represented by the two random variables.

Finally, the time required by the application program is dependent on the workload. The random variables shown, T and V, correspond to the workload discussed in section 7. They represent, respectively, the time for a client to think between generating requests and for the server to service requests.

IP layer: The IP layer is modeled as an open network of G/G/1 queues with two customer classes representing datagrams containing data and datagrams containing acknowledgements. The routing of the queueing network model is specified through routing matrix *ROUTE*. The sources of the queueing network each correspond to a TCP entity. The queueing discipline at each queue is FCFS with no distinction made for the class of an arriving datagram (e.g., data versus acknowledgement). The queue for each IP entity at a host (respectively, gateway) can hold an unbounded number of datagrams containing acknowledgements but at most B_H (respectively, B_G) datagrams containing data. A datagram containing data that arrives at an IP entity whose buffers are filled is dropped.

TCP layer: The TCP layer adds three features to the queueing network model that necessitate a solution by simulation.

- When a segment containing data has been successfully delivered from a sending TCP entity to a receiving TCP entity, corresponding to a datagram traversing a path through the queueing network, the receiving TCP entity immediately sends an acknowledgement datagram back through the queueing network to the sending TCP entity.

- The number of datagrams containing data that are buffered in all IP entities on the path corresponding to *each* connection in the system is limited to the receiver advertised window size, W.

- Each TCP sending entity starts a retransmission time for each datagram containing data that it introduces into the queueing network. If an acknowledgement is not received for such a datagram before

the corresponding timer expires, the sending TCP entity retransmits the datagram. The retransmission timer value is given by model parameter *TO* (Table 1).

Model specification: The Appendix contains a specification of the model in the form of extended finite state machines with actions written in the C++ language.

7. Example Workload Model

Appendix B contains a sample workload based on the *client-server paradigm* that drives the internetwork model of TCP/IP in Appendix A. (One could use any workload with the TCP/IP model of Appendix A.)

Each host runs exactly one *client* or *server*. Clients may represent personal computers or diskless workstations of individual users, while servers may represent file servers, print servers, and supercomputers. Prior to the start of simulation, each client establishes a connection with each server. Any server can service any client. A server is *local* to a client if the server host is attached to the same network as the client host.

The following workload is considered. Client host *A* randomly chooses whether to direct its request to a local host or to a remote (non-local) host. If the request is local (respectively, remote), Client *A* then selects one of the local (respectively, remote) servers with equal likelihood. Let *B* denote the chosen server. Client A then sends a request message to server *B* (e.g., send file hello.c) and blocks awaiting reply. When server *B* receives a request message, it immediately sends a response message (e.g., sends file hello.c) which wakes up client *A*. (Therefore each server requires time zero to fill a request (i.e., read the needed data from disk).) The length, in multiples of MSS, of each request and response message is determined by random variable *L*. After a client receives a response message, it waits for a random think time before repeating the sequence described above.

Appendix B Model: Clients and servers determine the arrival process of datagrams to the queueing network. Clients send request messages which introduce datagrams into the network; servers acknowledge the

datagram, wait for service time V, and send a response message that introduces additional datagrams; and clients wait for think time T before selecting a server at random and repeating the cycle.

8. Project Status

Modeling internets with thousands, millions, or billions of nodes will require execution of a simulation on a parallel computer. Therefore we are studying implementation of the model on a variety of architectures using a variety of parallelization techniques. We are investigating two classes of architectures, MIMD and SIMD. We are investigating three parallelization techniques: optimistic and conservative discrete event simulation [FUJI] as well as a numerical solution method.

We are currently producing three Internet simulation models:

- Implement the model in Sim++ [LOMO], a simulation programming language that runs on Time Warp [JEFF] on a network of transputers.

- Implement the model using Nicol's conservative synchronous parallel simulation algorithm [NICO] on hypercubes (joint work with D. M. Nicol of William and Mary).

- Model a window flow control mechanism running on a network that looses datagrams by a set of recurrence relations that are evaluated in parallel [GREE] on a Connection Machine (joint work with J. Wang of Virginia Tech) [WANG].

9. Future Extensions

The TCP/IP model given in the Appendix excludes the following items from the TCP/IP specification [TCP]. The items dynamically modify several model parameters, which complicates verification of the model given here. Adding the first five modifications is straightforward; the last modifications require some deeper changes to the TCP entities in the Appendix.

1. Base the receiver window advertisement on the number of free buffers in the TCP receive entity, and implement finite buffers in the TCP receive entity. (Currently this window is a fixed constant, W, and a TCP receive entity can buffer an infinite number of segments from its IP entity.) Delay sending acknowledgements to prevent silly window syndrome.

2. Allow acknowledgements to be piggybacked on data segments being sent in reverse direction.

3. Implement the slow start algorithm, multiplicative decrease congestion avoidance algorithm, and sender congestion window.

4. Implement Jacobson's algorithm estimating round trip time and variance of segments [COME, pp. 190-192]. (Currently the retransmission timer for all nodes and all connections is fixed at TO.)

5. Model queueing delay due to contention by different nodes for access to a shared communication medium (i.e., a LAN).

6. Relax the assumption that each message is a multiple of MSS bytes in length. Therefore some segments containing data may have length smaller than MSS, which will require an algorithm to partition a message into segments.

7. Replace random variables P_G and P_H by a larger set of random variables that avoids scheduling events with zero delays and adds a more realistic model of protocol timings to the model. The new random variables should capture the different times required to send, resend, and receive a segment, to process an acknowledgement, and to send versus receive a datagram. The random variables should correspond to the data that one can expect to measure from a protocol, for example the average time required to send and receive a segment and corresponding datagram broken down by layer (i.e., user, TCP, IP, and network adaptor).

References

[COME] D. E. Comer (1991), *Internetworking with TCP/IP, Volume I*, Englewood Cliffs: Prentice-Hall.

[FUJI] R. M. Fujimoto (1990), "Parallel Discrete Event Simulation," *CACM 33* (10), Oct., 30-53.

[GREE] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani (1990), "Unboundedly Parallel Simulations Via Recurrence Relations," *Proc. ACM SIGMETRICS*, Boulder, CO, May, 1-12.

[JEFF] D. R. Jefferson, "Virtual Time," *ACM Trans. on Programming Languages and Systems 7* (3), July 1985, 404-425.

[LOMO] G. Lomow and D. Baezner, "A Tutorial Introduction to Object-Oriented Simulation and SIM++," in *Proc. 1989 Winter Simulation Conference*, Wash. D.C., Dec., 140-146.

[NICO] D. M. Nicol (1990), *The Cost of Conservative Synchronization in Parallel Discrete Event Simulations*, Report 90-20, Inst. for Comp. App. in Sci. and Eng., NASA Langley Research Center, May.

[SCHW] M. Schwartz (1987), *Telecommunication Networks*, Addison-Wesley.

[TANE] A. S. Tanenbaum (1988), *Computer Networks*, 2nd edition, Englewood Cliffs: Prentice-Hall.

[TCP] (1981)*Transmission Control Protocol*, Information Sciences Institute, USC, RFC 793, Sept.

[WANG] J. J. Wang and M. Abrams (1992), *Approximate Time-Parallel Simulation of Queueing Systems with Losses*, Computer Science Dept., Virginia Tech, TR 92-08, April 1992.

Appendix A: Specification of TCP/IP Model

A.1. Overview

The TCP/IP model is specified using extended finite state machines (EFSM's) along with fragments of code in the C++ language. (Tanenbaum gives an introduction to specifying protocols using extended finite state machines [TANE, pp. 251-252].) An EFSM consists of a state transition diagram along with a set of variables. The EFSM maps the current state and variable values to a new state and variable values. An event is denoted as $E[p_0,p_1,...,p_n]$, where E is the event name and $p_0,p_1,...,p_n$ are parameters to the event. The function

long Schedule($E[p_0,p_1,...,p_n]$, t, e)

schedules event $E[p_0,p_1,...,p_n]$ to occur after t time units at the EFSM for e. The return value of Schedule(...) is an integer that uniquely identifies the scheduled event for possible use by function Cancel(...), described below. The value of e is one of "self," "my_app," "my_tcp_snd," "my_tcp_rcv," "my_ip," or an IP address. "Self" denotes the EFSM that executes the schedule statement. "My_app" denotes the application that runs on the same node as the EFSM that executes the schedule statement. "My_tcp_snd," "my_tcp_rcv," and "my_ip" denote, respectively, the tcp send entity, tcp receive entity, and ip entity on the same node as the EFSM that executes the schedule statement. Finally, if e is an IP address, then E is scheduled at the EFSM for the IP entity with IP address e.

The function

Cancel(long i)

does the following. If i corresponds to a scheduled but unexecuted event (i.e., i is the return value of a previous call to Schedule(...), and event i has not yet been executed), the effect is as though the call to Cancel(...) and the corresponding all to Schedule(...) were never made. Otherwise the call to Cancel(...) has no effect.

With respect to modeling time, the occupancy time of each state in an EFSM is zero. Delays in the model are introduced in exactly one manner, which is through parameter t of the call to Schedule described above. Generally parameter t is nonzero. We avoid scheduling events after zero delay to facilitate execution of the model by a parallel simulation algorithm, and zero delays reduce parallelism in these algorithms.

Each EFSM in this document is written as a table in which each possible event corresponds to a row and each possible state corresponds to a column. Each table entry is either blank or of consists of one or more entries of the form $P_i:A_i/S$, specifying that if predicate P_i holds, then perform action A_i and change the current state to S. Action ~ means do not perform any action; state ~ means remain in current state. A blank table entry represents an error condition.

The model consists of two layers: TCP and IP. The TCP layer is the most complex, and is specified as two EFSM's, representing one connection of the sending and one connection of the receiving TCP entities, in Sections A.4.2 and A.4.3, respectively. The IP layer is simpler than TCP, and is specified as a single EFSM representing both the sending and receiving entities in Section 5. The specification also states which events form the interfaces between the application/TCP and TCP/IP layers, in Sections A.3.1 and A.3.2, respectively.

Normally the TCP/IP model is driven by a workload forming an application layer on top of TCP. A sample workload is given in Appendix B.

The specification of each of the six components starts with declarations of necessary data types, events, functions, and variables. The specification uses the simulation model parameters of Table 1 without declaration. The model below closely follows and uses notation introduced in the TCP specification [TCP].

Type int must be represented by at least 32 bits.

Finally, the specification here assumes that there are no errors in the input data to the model. For example, the end point of each TCP connection always corresponds to a legal IP address of a host and there is a path between any two nodes in the network being modeled.

A.2. Declarations Common to Multiple EFSMs

The following definitions are used by at least two different EFSMs, and hence are global to all EFSMs.

A.2.1 Data Types

```
typedef unsigned int usint;

typedef usint Boolean;          //Zero is false, non-zero is true

typedef usint cn;               //Connection number used by a TCP entity

typedef usint length;           //Length of a message, in number of segments

enum type { UNKOWNT, DATA, ACK };
                                //Distinguishes segments with data only and acknowledgement
                                //only

typedef usint ssn;              //Segment sequence number

typedef usint phys_addr;        //Physical address (range 0..N-1)

struct ip_addr {                //32 bit internet (or IP) address
    usint NetId;
    usint NodeId;
    ip_addr();
    ip_addr(usint i, usint j);
};

class segment {                 //Segment passed to IP or received from IP
    type Type;                  //Type field of call to segment(...)
    cn DestCN;                  //Dest_cn field of call to segment(...)
    Boolean PSH;                //P field of call to segment(...)
    ssn Seq;                    //Seq field of call to segment(...)
public:
    segment();
    segment(type t, cn dest_cn, Boolean push, ssn seq);
                                //Create instance of class segment; used when t=DATA:
                                // dest_cn:  connection number at destination over which this
                                //           segment will arrive
                                // push:     set control bits of segment header to PSH (i.e., push)
                                //           if and only if push=TRUE
                                // seq:      set sequence field to seq
    segment(type t, cn dest_cn, ssn ack);
                                //Create instance of class segment; used when t=ACK:
                                // dest_cn:  same as above
                                // ack:      set sequence field to ack
};

typedef unsigned double time;  //Used to specify an interval of time

class statistic {               //Used to compute sample mean of a set of observations
public:
    void Observe(float x);      //Add x to list of observations
    float Mean();               //Return sample mean
};
```

A.2.2 Variables

```
time TNow;                  //Current time in system being modeled

statistic TPUT, RTD;
```

A.2.3 External Functions

```
time Sample(RV);            //Let RV denote a random variable. Function Sample returns a
                            //random variate from the distribution underlying RV.
```

A.2.4 Constants

```
int MaxInt = ...;           //Largest positive integer representable by data type int
```

A.2.5 Global Initialization

TNow = 0;
Create the following:
- N instances of the IP EFSM at physical addresses 0..N-1
- NT instances of the sending TCP EFSM at physical addresses 0..NT-1
- NT instances of the receiving TCP EFSM at physical addresses 0..NT-1

A.3. Interfaces

A.3.1 Application/TCP Interface

During initialization of the model, the application can initialize a TCP entity by calling:

```
Connect( phys_addr local_p, ip_addr remote_ip, cn local_cn, cn remote_cn );
                    //Initializes TCB[local_cn] in TCP at physical node local_p to have
                    //destination IP address remote_ip and destination connection
                    //number remote_cn
```

An application program schedules one of the two events listed for a particular TCP connection.

```
Send[cn i, length l]    //Application requests TCP to initiate transmission of message of
                        //length l over connection i. TCP will set PSH bit of last segment

Receive[cn i]           //Application signals TCP that it is prepared to accept next
                        //message from connection i
```

A TCP entity signals to an application completion of a message transmission or reception with:

```
Send_Done[cn i]         //After application schedules a Send[ml] event for connection
                        //number i and TCP sending entity has transmitted and received an
                        //acknowledgement for all segments of m, TCP send entity
                        //schedules SEND_DONE for application

Receive_Done[cn i]      //After an application schedules a Receive[] event for connection
                        //number i, TCP will return in m the next message arriving over
                        //connection i that has not been returned to the application layer.
                        //Note that message may arrive before or after the application
                        //schedules the Receive[] event that results in a //Receive_Done[i]
                        event.
```

Notes:

1. An application may not schedule two successive SEND events for the same connection without scheduling an intervening SEND_DONE event for the connection.
2. An application may not schedule two successive RECEIVE events for the same connection without scheduling an intervening RECEIVE_DONE event for the connection.

A.3.2 TCP/IP Interface

TCP passes segments to IP for transmission via:

```
IP_ToNet[ip_addr dest_IP, segment* seg]
                    //TCP entity requests its IP protocol entity to send segment seg (as
                    //a datagram) to IP address dest_IP
```

IP signals to TCP availability of an arriving segment with:

```
S_Data[cn i, segment* seg]   //Data segment arrives from TCP connection i
S_Ack[cn i, segment* seg]    //Acknowledgement segment arrives from TCP connection i
```

A.4. TCP EFSMs

Figure 2 illustrates the variables that must be maintained for each TCP connection.

During entire simulation, TCP protocol state is ESTABLISHED as defined by RFC 793. States SENDING, IDLE, RP and RP are sub-states of state ESTABLISHED.

The model specified here differs from the TCP specification in the following manner. In the specification, a segment header specifies source and destination port numbers, and a datagram header specifies source and destination IP addresses. When a TCP entity receives a segment, it uses the port and IP addresses to search for the corresponding TCB (transmission control block), which contains the connection state. To simplify the simulation, when the simulation is initialized, the a field of the TCB for each connection will be initialized to the connection number which the TCP entity at the other end of the connection uses for the connection. Whenever a TCP entity sends a segment, the segment will also contain the receiving entity's local connection number (See parameter dest_cn in function segment(...) and function DestCN() in class segment in Section A.2.1. Therefore the receiving TCP entity can use the local connection number as an index to the table of TCB's to retrieve the proper TCB without search. Avoiding search makes an implementation of the simulation model faster than a faithful representation of the TCP specification.

A.4.1 Definitions Common to Sending and Receiving Entities

Constants

```
usint MyPhysAddr;          //Physical address of node running this TCP entity
```

Data Types

```
enum snd_state { IDLE, SENDING };
enum rcv_state { RP, RP };

class sl {                 //List of segments; COULD BE IMPLEMENTED AS BIT MAP
public:
    sl();                  //Create instance of class sl with list initially empty
    void Insert(segment& s); //Insert copy of segment s into segment list, sorted by increasing
                           //value of the SEQ field of each segment
    segment* Head();       //Return a copy of head of list
    segment* RemoveHead(); //Return head of list and remove head from list
    Boolean IsEmpty();     //Returns true iff list is empty
};
```

//Because all data segments have identical size and each message is an integral multiple of the number of //bytes contained in a data segment, the definition of tcbinfo below *equivalently* assumes that each data //segment contains *one byte* of a message. Note that the model does not require values for MSS and MTU.

```
struct tcbinfo {                    //Transmission control block (one per connection); see Figure 2
    cn DEST_CN;                     //Connection number at destination to which this TCB corresponds
    ip_addr DEST_IP;                //IP address of destination TCP protocol entity

    struct TcbSndInfo {
        snd_state STATE;            //State of this connection in sending TCP entity
        long Timer[MaxInt];         //Timer[i] contains the return value for Schedule(...) of the timeout
                                    //event for segment sequence number i
        ssn FST;                    //Sequence number of first byte comprising message passed in last
                                    //SEND event for this connection
        ssn UNA;                    //Smallest sequence number that is unacknowledged
        ssn NXT;                    //Smallest unsent segment sequence number
        ssn LST;                    //Sequence number of last byte comprising message passed in last
                                    //SEND event for this connection
        TcbSndnfo()                 { STATE=IDLE; NXT=UNA=0; }
    } SND;
    struct TcbRcvInfo {
        rcv_state STATE;            //State of this connection in receiving TCP entity
        ssn NXT;                    //Smallest unreceived segment sequence number
        usint MsgsAvail;            //Number of messages received by application, but awaiting a
                                    //RECEIVE event from application to retrieve
        sl OOO;                     //Segments received out of order in receive window
        TcbRcvInfo()                { NXT=0; MsgsAvail=0; }
    } RCV;


}
```

Variables

```
tcbinfo TCB[Conn[MyPhysAddr]];
                                    //Transmission control blocks for all connections
```

Events

Scheduled by application

```
Send[cn,length]                     //Described in Section A.3.1
Receive[cn]
```

Scheduled by IP entity

```
S_Data[cn,segment]                  //Described in Section A.3.2
S_Ack[cn,segment]                   //Described in Section A.3.2
```

Scheduled by TCP entity for IP

```
IP_ToNet[ip_addr, segment]          //Described in Section A.3.2
```

Scheduled by TCP entity for application

```
Send_Done[cn]                       //Described in Section A.3.2
Receive_Done[cn]                    //Described in Section A.3.2
```

Scheduled by TCP entity for itself

```
ReTO[cn i, ssn s, segment seg]//Retransmission timer expires for segment seg, which is
                                    //sequence number s of connection i
IntSEvent[cn i]                     //An internal event for sender state machine for connection i
IntREvent[cn i]                     //An internal event for receive state machine for connection i
```

Constants

```
ip_addr MyIPAddr;          //IP address of this TCP sending or receiving entity
```

Function

```
Connect( phys_addr local_p, ip_addr remote_ip, cn local_cn, cn remote_cn ) {
    TCP entity for physical address local_p executes:
        TCB[local_cn].DEST_CN = remote_cn;
        TCB[local_cn].DEST_IP = remote_ip;
}
```

A.4.2 TCP Sending Entity EFSM

Extended Finite State Machine

Event	TCB[i].SND.STATE	
	IDLE	SENDING
Send[i,ml]	A1(i,ml)/SENDING	
IntSEvent[i]		P1(i)∧P2(i): A2(i)/~ P3(i): A3(i)/IDLE (P1∧P2)∨P3: ~/~
S_Ack[i,seg]	~/~	P6(i,seg): A4(i,seg)/~ P6(i,seg): ~/~
ReTO[i,s,seg]		A5(i,s,seg)/~

P1(cn i): TCB[i].SND.NXT ≤ TCB[i].SND.UNA + W - 1
//Window is currently open

P2(cn i): TCB[i].SND.NXT ≤ TCB[i].SND.LST
//At least one segment of message has never been sent

P3(cn i): TCB[i].SND.UNA > TCB[i].SND.LST
//All segments of message were sent and acknowledged

P6(cn i, segment* seg):
(TCB[i].SND.UNA < seg->Seq()) && (seg->Seq() ≤ TCB[i].SND.NXT)
//Incoming ack is for sent but unacknowledged segment

A1(cn i, length ml):
// Respond to Send[i,ml] event scheduled by application
//
TCB[i].SND.FST = TCB[i].SND.NXT;
TCB[i].SND.LST = TCB[i].SND.FST + ml - 1;
A2;

A2(cn i): //Send data segments via IP entity until window closes
while (P1∧P2) {
Boolean PSH = TCB[i].SND.NXT==TCB[i].SND.LST;
segment SG(DATA, TCB[i].DEST_CN, PSH, TCB[i].SND.NXT);

Schedule(IP_ToNet[TCB[i].DEST_IP, SG], 0, my_ip);
TCB[i].SND.Timer[TCB[i].SND.NXT] = Schedule(ReTO[i,TCB[i].SND.NXT, SG], TO, self);
TCB[i].SND.NXT++;
}
Schedule(IntSEvent[], 0, self);

A3(cn i): Schedule(Send_Done[i], 0, my_app);

A4(cn i, segment* seg):
//Process incoming and valid acknowledgement
for (int k = TCB[i].SND.UNA; k ≤ seg->Seq()-1; k++) Cancel(TCB[i].SND.Timer[k]);
TCB[i].SND.UNA = seg->Seq();
Schedule(IntSEvent[], 0, self);

A5(cn i, ssn s, segment* seg):
//Retransmit data segment due to timer expiration
Schedule(IP_ToNet[TCB[i].DEST_IP, *seg], 0, my_ip);
TCB[i].SND.Timer[s] = Schedule(ReTO[i,s,*seg], TO, self);

A.4.3 TCP Receiving Entity EFSM for Connection i

Note
TCP may receive and queue one or more messages before the application schedules Receive[...].

Extended Finite State Machine

Event	TCB[i].RCV.STATE	
	$\overline{RP}$	RP[1]
Receive[i]	P4(i): ~/RP P4(i): A6(i)/~	
S_Data[i,seg]	P5(i,seg): A7(i,seg); A8(i)/~ P5(i,seg): A8(i)/~	P5(i,seg): A7(i,seg);A8(i);A9(i)/~ P5(i,seg): A8(i)/~
IntREvent[i]		P4(i): ~/~ P4(i): A6(i)/$\overline{RP}$

P4(cn i): TCB[i].RCV.MsgsAvail==0 //All complete messages received by TCP were received by user

P5(cn i, segment* seg): //Incoming segment lies in receive window
(TCB[i].RCV.NXT ≤ seg->Seq()) && (seg->Seq() < TCB[i].RCV.NXT+W)

A6(cn i): //Application wants message; return message that already arrived
TCB[i].RCV.MsgsAvail--;
Schedule(Receive_Done[i], 0, my_app);

A7(cn, segment*):
//Segment with data arrived; copy it into out-of-order list, then delete all contiguous segments
//from out-of-order-list. If PSH bit of a contiguous segment is set, increase count of available
//messages
//
TCB[i].RCV.OOO.Insert(seg);
while (! TCB[i].RCV.OOO.IsEmpty() && TCB[i].RCV.OOO.Head()->Seq() ==
TCB[i].RCV.NXT) {
TCB[i].RCV.NXT++;
if (TCB[i].RCV.OOO.Head()->PSH()) TCB[i].RCV.MsgsAvail++;
RemoveHead(TCB[i].RCV.OOO);
}

A8(cn i):
//Acknowledge received, contiguous segments by creating and sending an acknowledgement
//segment; acknowledgement must take reverse path of data segments it acknowledges
//
segment Seg(ACK, TCB[i].DEST_CN, TCB[i].RCV.NXT);
Schedule(IP_ToNet[TCB[i].DEST_IP, Seg], 0, my_ip);

A9(cn i):
Schedule(IntREvent[], 0, self);

[1]RP denotes "RECEIVE from application is pending."

A.5. IP_EFSM

IP events

Event scheduled by TCP entity

```
IP_ToNet[ip_addr, segment]    //Described in Section A.3.2
```

Events scheduled by IP entity for itself

```
IP_FromNet[datagram d]        //Datagram d arrives to IP entity from network
IP_End_Service[]              //End of processing and transmission of datagram at head of queue
```

Constants

```
usint MyPhysAddr;             //Physical address of node running this IP entity
```

Data Types

```
struct datagram {             //Datagram of passed between IP entities
    ip_addr DestIP;           //DestIP field of datagram(dest_IP,seg) call
    segment Seg;              //Seg field of datagram(dest_IP,seg) call
    datagram(ip_addr* dest_IP, segment seg);
                              //Create a datagram destine to IP address dest_IP containing
                              //segment seg
    ~datagram();              //delete datagram, but not segment that datagram points to
};

class dq {                    //FIFO queue of datagrams
public:
    dq();                     //Creates instance of class dq with empty queue
    void Insert(datagram& d); //Insert copy of d into queue in FIFO order
    usint Length();           //Returns number of datagrams in queue
    usint DataLength();       //Returns number of datagrams with segment type DATA in queue
    datagram* Head();         //Return head of queue, but don't remove it from queue
    datagram* RemoveHead(); //Return head of queue and remove head from queue
};

typedef usint netid;          //A portion of an IP address specifying a network id
```

Local variables

```
dq TransQ;                    //FIFO queue of datagrams that arrived either from TCP entity, if
                              //any, on this node or from network awaiting transmission. See
                              //assumption 7 in Section 5 for further information. Note that
                              //TransQ may contain at most B datagrams received from network
                              //and containing data segments is limited
datagram* DG;                 //Holds datagram currently propagating over comm. medium
```

Extended Finite State Machine

Event	Action
IP_FromNet[DG]	P0(DG): ~ P0(DG): A2(DG)
IP_ToNet[dest,seg]	A1(dest,seg);A2(DG)
IP_End_Service[]	A4()

```
P0(datagram* dg):
        dg->Seg()->Type()==DATA ∧ TransQ.DataLength() ≥ ((this is a gateway) ? B_G : B_H)
                                        //No room for datagram dg in transmission queue

DirectDelivery(usint j): ROUTE[MyPhysAddr, j] == ∅        //j is a netid

A1(ip_addr dest, segment* seg):
        DG = new datagram(dest, seg);  //Create datagram with destination dest

A2(datagram* dg):
        //Insert copy of datagram dg into transmission queue.  If queue was empty prior to insertion, then
        //immediately begin transmission of dg.
        TransQ.Insert(dg);
        if (TransQ.Length()==1) A3();

A3():   //Could be labeled event "IP_Begin_Service": Beginning of processing and transmission of
        //datagram at head of IP entity's queue to TCP on same node or to IP on another node.
        //
        DG = TransQ.Head();

        if ( DG->DestIP()∈IP[p] )
            //Datagram destination is this IP entity.
            //
            if (no TCP entity present at this node) Raise error:  datagram destination is not a host
            else
                //Extract segment from datagram and pass to connection specified by segment in TCP
                //entity served by this IP entity
                //
                if (DG->Seg().Type()==DATA)
                      Schedule( S_Data[ *(DG->Seg()).DestCN(), DG->Seg() ], Sample(P_H), my_tcp_rcv);
                else  Schedule( S_Ack[ *(DG->Seg()).DestCN(), DG->Seg() ], Sample(P_H), my_tcp_snd);
        else {
            //Datagram destination is another IP entity.
            //
            usint DestNetId = DG->DestIP().NetId;
            ip_addr NextHop =
                if ( DirectDelivery(DestNetId) ) then DG->DestIP() else ROUTE[MyPhysAddr, DestNetId];
            Schedule( IP_FromNet[*DG], Sample(P_G) + Sample(PROP[NextHop.NetId]),  NextHop );
        }
        Schedule( IP_End_Service[], IP_Delay, self );

A4():   //If queue is not empty, schedule next departure event
        //
        TransQ.RemoveHead();          //Datagram was already sent
        if (TransQ.Length()>0) A3();
```

Appendix B: Specification of Client/Server Workload Model

B.1. Overview

The application layer is specified as two EFSM's, one for the client and one for the server in Sections B.2.2 and B.2.3, respectively.

The following relationship must hold between the parameters for the client/server model (Table 2) and those for the TCP/IP model (Table 1):

NT = NC+NS
Clients have physical addresses 0..NC-1
Servers have physical addresses NC..NC+NS-1

$$\text{Conn}[p] = \begin{cases} \text{NS if } 0 \le p < \text{NC}-1 \\ \text{NC if } \text{NC} \le p < \text{NC}+\text{NS}-1 \\ \text{undefined if } \text{NC}+\text{NS} \le p \end{cases}$$

B.2. Declarations Common to Multiple EFSM's

B.2.1 Definitions

All clients (respectively, servers) have the same association of connection numbers to IP addresses. Specifically, connection number i is the connection to server S[i-1] (respectively, client C[i-1]).

B.2.2 Global Initialization

```
Create NC instances of the Client EFSM;
Create NS instances of the Server EFSM;

//Establish connection from each server (respectively, client) to all clients
//(respectively, servers)

  for (int p=0; p<NC; p++) {
      //TCP runs on a client; establish connection from client p to all servers
      //
      for (int k=NC; k<NC+NS; k++) Connect( p, ip(k), p, k-NC );
  }

  for (int p=NC; p<NC+NS; p++) {
      //TCP runs on a server; establish connection from server p to all clients
      //
      for (int k=0; k<NC; k++) Connect( p, ip(k), p-NC, k );
  }
```

B.3 Client EFSM

Data Types

enum client_state { S1, S2, S3 };

External functions called

```
usint Uniform(usint a,b)      //Returns uniformly distributed random variate between a and b
cn RndLocalServer()           //Returns connection number chosen randomly and uniformly from
                              //the set of local servers in array S
cn RndRemoteServer()          //Returns connection number chosen randomly and uniformly
                              //from the set of remote servers in array S
```

Variables

```
client_state State;           //State of client EFSM
cn I;                         //Connection from this client to current server
time TLastSend;               //Time at which last message was sent
length LastMsgLen;            //Length of last message sent
```

Events

Scheduled by Client:

```
Think_Done[]                  //Internal event to capture client think time
Send[cn, length]              //See Section A.3.1
Receive[cn]                   //See Section A.3.1
```

Scheduled by TCP entity:

```
Send_Done[cn]                 //See Section A.3.1
Receive_Done[cn]              //See Section A.3.1
```

Initialization

```
State = S1;
Schedule( Think_Done, 0, self );
```

Extended Finite State Machine

Event	State		
	S1	S2	S3
Think_Done[]	A1()/S2		
Send_Done[i]		A2(i)/S3	
Receive_Done[i]			A3()/S1

```
A1(): //Randomly select a server that is either on the same network or on another network, and send the
      //server a random length message via TCP send entity
      //
      if (no remote servers exist)     I = RndLocalServer();
      else if (no local servers exit)  I = RndRemoteServer();
      else if (Uniform(0,1)<p)         I = RndLocalServer();
      else                             I = RndRemoteServer();
      LastMsgLen = Sample(L);
      Schedule ( Send[ I,LastMsgLen ], 0, my_tcp_snd );
      TLastSend = TNow;

A2(cn i):  Schedule( Receive[I], 0, my_tcp_rcv );
           RTD.Observe(TNow-TLastSend);
           TPUT.Observe( LastMsgLen/(TNow-TLastSend) );

A3():  Schedule( Think_Done[], Sample(T), self );
```

B.4 Server EFSM

Data Types

```
enum server_state { S1, S2 };
```

Variables

```
server_state State;          //State of server EFSM
cn I;                        //Specifies which client the server is answering
time TLastSend;              //Time at which last message was sent
length LastMsgLen;           //Length of last message sent
```

Events

Scheduled by Server:

```
Service_Done[]               //Internal event to capture time required to service requests
Send[cn,length]              //See Section A.3.1
Receive[cn]                  //See Section A.3.1
```

Scheduled by TCP entity:

```
Send_Done[cn]                //See Section A.3.1
Receive_Done[cn]             //See Section A.3.1
```

Initialization

```
State = S1;

//Wait for a message from any client
//
for (k=0; k<NC; k++)         Schedule( Receive[k], 0, my_tcp_rcv );
```

Extended Finite State Machine

Event	State		
	S1	S2	S3
Receive_Done[i]	A1(i)/S2		
Service_Done[]		A2()/S3	
Send_Done[i]			A3()/S1

```
A1(cn i):   I=i;
            Schedule ( Service_Done[], Sample(V), self );

A2():       LastMsgLen = Sample(L);
            Schedule( Send[ I,LastMsgLen ], 0, my_tcp_snd );
            TLastSend = TNow;

A3():       Schedule( Receive[I], 0, my_tcp_rcv );
            RTD.Observe(TNow-TLastSend);
            TPUT.Observe( LastMsgLen/(TNow-TLastSend) );
```

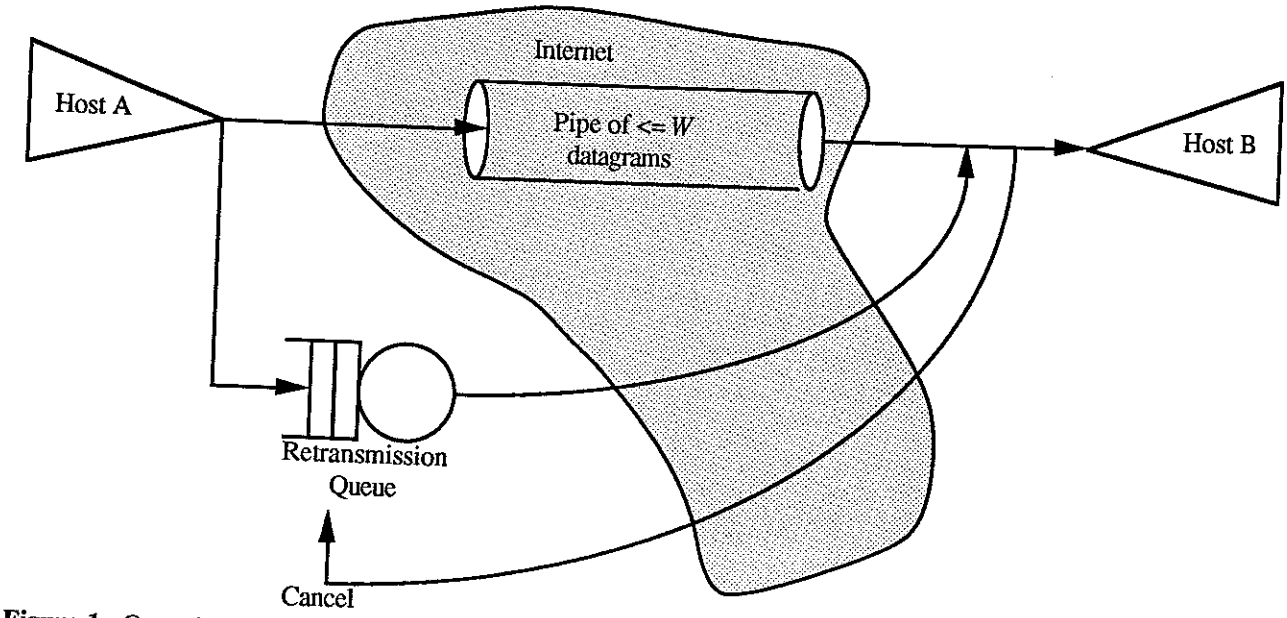

Figure 1. Queueing network model of TCP/IP based Internet.

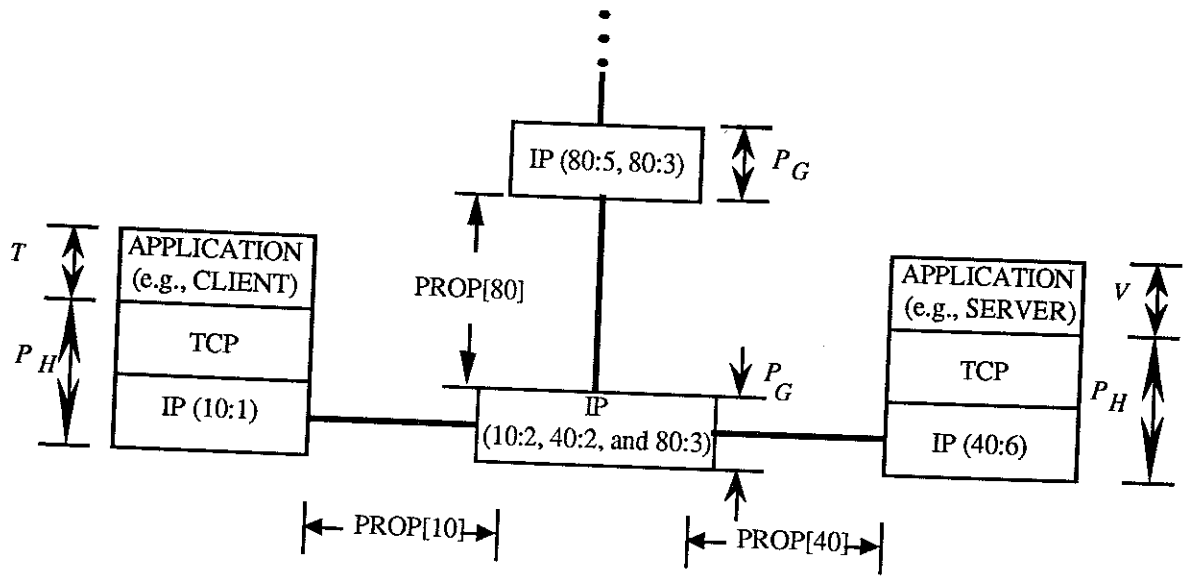

Figure 2. Illustration of time delay parameters in model. A client and server workload is also illustrated. IP addresses are denoted as x:y, where x is the network id and y is the host id.

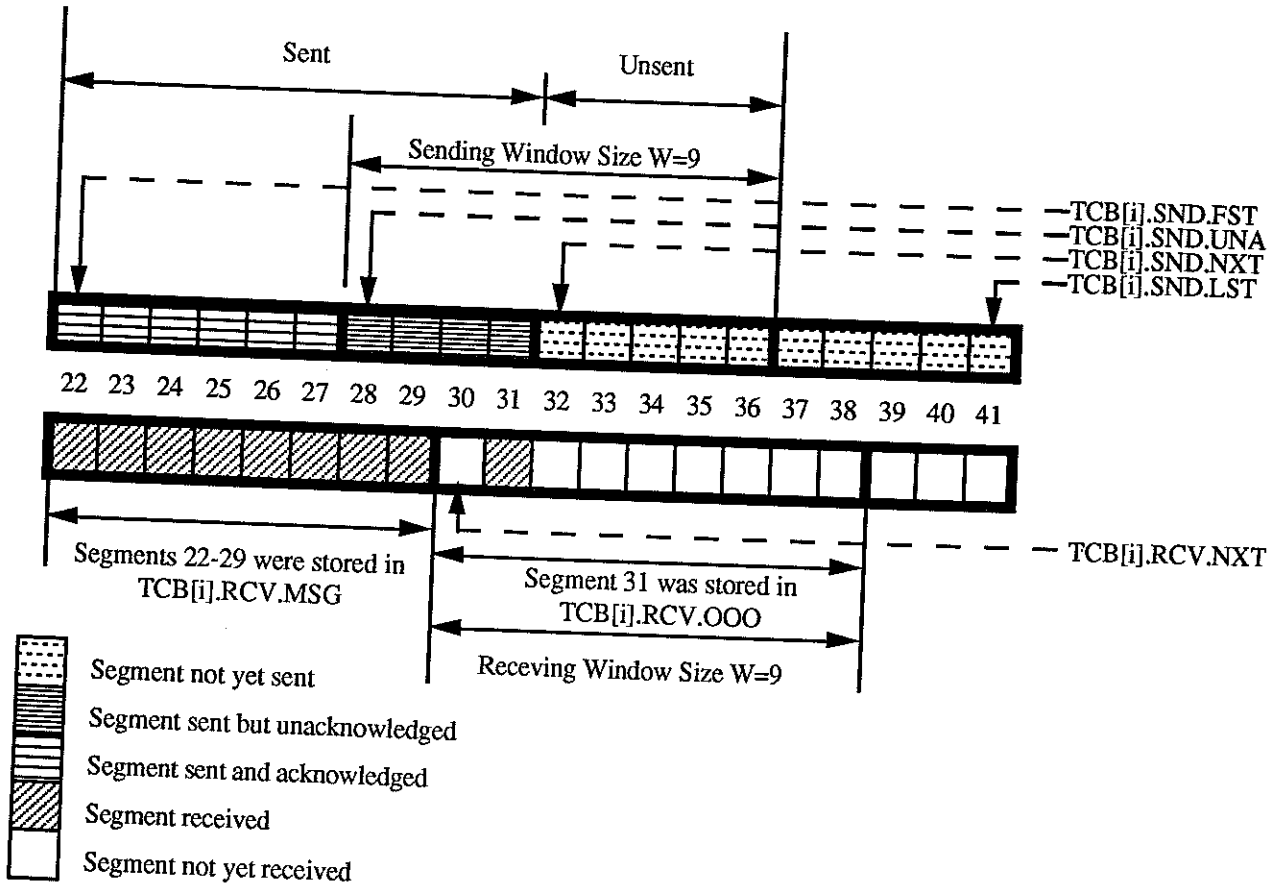

Figure 3. Illustration of send and receive windows for a single connection, i. Sending TCP entity keeps four pointers in the Transmission Control Block (TCB):

1. TCB[i].SND.FST: Sequence number that contains first byte of message passed in last SEND call for connection i

2. TCB[i].SND.UNA: Sequence number that contains first sent but unacknowledged byte of message passed in last SEND call for connection i (zero if no segments have been sent yet)

3. TCB[i].SND.NXT: Sequence number that contains first unsent byte of message passed in last SEND call for connection i

4. TCB[i].SND.LST: Sequence number that contains last byte of message passed in last SEND call for connection i

Receiving TCP entity keeps one pointer in the Transmission Control Block (TCB), to store the sequence number of the first unreceived byte of the first incomplete message currently being transmitted over number connection i. Note that the contiguous segments received are stored in TCB[i].RCV.MSG, and the out of order segments are stored in TCB[i].RCV.OOO. All other segments received are discarded.

	Model parameter	*Notation*
Network topology	Number of nodes in model	N
	For each physical node address p, *ROUTE[p,i]* specifies how to route datagrams from p to a destination in netid i. Each entry is either NULL or an IP address. NULL means perform direct delivery. An IP address means directly deliver datagram to specified IP address (i.e., a gateway). See [COME] Fig. 8.2b.	*ROUTE[p,i]*
	Set of IP addresses corresponding to physical node address p	*IP[p]*
Network delays	Total time required by a node to process a datagram, excluding time spent queueing, when node functions as an IP gateway	P_G
	Total time required for processing one segment and the corresponding datagram at a node when the node functions as a host. Includes time required for the TCP entity, IP entity, device driver, and interfaces between protocol layers to either pass a packet from the application to the network or vice versa; excludes the time spent queueing.	P_H
	Random variable denoting time required for a datagram to propagate between any two nodes attached to netid i.	*PROP[i]*
For IP	Number of buffers in finite pool of host IP entity[1]	B_H
	Number of buffers in finite pool at gateway IP entity	B_G
For TCP	Number of nodes running TCP (must be physical addresses 0..*NT*-1)	NT
	Number of connections for TCP entitity at physical address p, where $p<NT$	*Conn[p]*
	Retransmission timer timeout interval	*TO*
	Receiver advertisement window size: number of *segments* past segment containing last unacknowledged sequence number that receiving TCP entity will accept	*W*

Table 1. TCP/IP model parameters, for Appendix A.

[1]Each buffer can hold one datagram containing one data segment.

	Model parameter	*Notation*
For clients and servers	Number of clients	*NC*
	Number of servers	*NS*
	IP addresses of client hosts	*C[0..NC-1]*
	IP addresses of server hosts	*S[0..NS-1]*
	Random variable denoting length of a request or response message, in number of segments	*L*
	Random variable denoting time from when a client receives a response until client sends next request (i.e., "think time")	*T*
	Random variable denoting time from when a server receives a request until server sends response	*V*
	Probability that a client selects a local rather than a remote server to satisfy a request, for each client connected to both local and remote hosts	*p*

Table 2. Client/server workload parameters, for Appendix B.