A TCP Simulator with PTOLEMY

Dorgham Sisalem
GMD-Fokus Berlin
(dor@fokus.gmd.de)

June 9, 1995

1 Introduction

Even though lots of TCP simulators and TCP traffic sources are already implemented in different programming languages, e.g., REAL [1], the *x*-Kernel [2], tcplib[3] , we have decided to implement our own simulator. Building a simulator with PTOLEMY [4] would not just ease the integration and handling of the simulator as a TCP traffic source in general, but would add a very useful galaxy to PTOLEMY as well. For this reason two versions of a 4.3BSD Tahoe based TCP simulator were implemented[1].

The basic version provides the user with the usual TCP-window based control mechanism, as well as the slow start, congestion avoidance and round trip estimation algorithms suggested by Jacobson [5]. In the enhanced version of the simulator the fast retransmission algorithm was implemented as well.

It will be shown that with this enhancement the throughput and performance of the protocol increases considerably since the number of retransmitted packets decreases.

To verify the simulator and to compare the two versions a network configuration has been chosen that has already been used in another study [6]. Whereas the results obtained from both simulators show a great similarity to the ones reached in [6], the basic version shows a reduced efficiency.

2 A Brief History of TCP

Before moving to the actual simulator implementation and the description of its different features this section summarizes the basic development of TCP.

1. **4.2BSD** (1983): The first widely available release of TCP/IP based on RFC 793 [7].

2. **4.3BSD Tahoe** (1988): The version implemented here. The main protocol improvements introduced were slow start and congestion avoidance algorithms.

3. **4.3BSD Reno** (1990): This implementation increased the efficiency of the protocol through a better implementation. The main changes like TCP header prediction, and more effective silly window handling code increased the speed of the sender but did not alter the protocol itself. Other changes aimed at reducing the spurious retransmissions through invoking slow

[1]The source code of the simulator and some usage example can be obtained as a compressed tar file from ftp://ftp.fokus.gmd.de/pub/step/tools.

start on idle links and a better accounting for the variance of the round trip time in the round trip time estimation, for more information see [8].

4. **4.4BSD (1993)**: To allow TCP to perform well over the so called *long fat pipes*, i.e., links with large *bandwidth-delay products*, a few new options had to be included. These options allow for window size scaling, protection against sequence number wrap up and a better round trip time estimation, as described in RFC 1323 [9].

5. **4.3BSD Vegas** (1994): Here an improved round trip time estimation algorithm is used and the congestion avoidance and slow start mechanisms were modified [10].

3 4.3BSD Tahoe TCP Congestion Control Algorithm

The Reno TCP release was mainly intended to improve the performance of the hosts. This was done without altering the protocol or adding any new algorithms to the Tahoe version. Since we can assume the simulated hosts to be perfect hosts, i.e., hosts that consume no time while processing packets or executing the control algorithms, there was no need to implement the changes introduced by Reno. The options introduced in RFC 1323 allow for a 32 bit long representation of window sizes and sequence numbers, which is more adequate for high speed networks. As window sizes and sequence numbers are represented as *integers* in the simulator these options are already implicitly contained in the implementation. The improved round trip time estimation introduced in RFC 1323 is on the other hand discussed in Appendix 1 of [11]. With Vegas the behavior of the sending host has been altered in order to improve the overall performance. These changes are however still under discussion and are not included in what can be called a typical TCP implementation. As our aim was to implement a basic TCP model that includes the main features of the congestion control algorithm without wasting time on unnecessary or not yet standardized improvements or extensions we chose to implement the Tahoe version. This version includes explicitly the main services of the congestion control algorithm of TCP. Implicitly some of the desired extensions like the usage of windows larger than 64 kbytes are added to the simulator through the implementation itself.

In this section a brief description of the services provided by the simulator are introduced and the differences between the two versions are mentioned.

As there is no interest in using the simulator as a real TCP source, some simplifications have been taken during the implementation:

- Instead of using the usual TCP clock with a resolution of 500 msec the time information were taken from the timestamps of the packets. These timestamps had to be added to the packets anyway as the discrete event scheduler of PTOLEMY uses these stamps to determine the right sequence of scheduling. This not only leads to an easier implementation but to more precise results as well.

- All packets have the same length which is the maximum segment length (MSS).

- A packet consists only of a header that contains fields describing the source and destination addresses, size of the packet, a packet sequence number, a byte sequence number and an acknowledgment field.

- A connection always exists, i.e., there is no need for a connection establishment and termination phase.

- Only congestion related and necessary mechanisms were implemented, i.e., no PUSH-FLAG, urgent mode, or persist timer.

In what remains of this section the implemented algorithms are briefly described. For more detailed information, see [12].

1. **Sliding window**: On the arrival of a data packet at the receiver, an acknowledgment for the last correctly received byte, is generated. The acknowledgment also contains information about the amount of data the source could still transmit without congesting the receiver, the so called *advertised window*. The source can now keep on sending data until the sequence number of the last sent byte equals the sum of the last acknowledged byte number plus the advertised window size.

2. **Silly window handling**: The silly window handling mechanism is used to avoid the exchange of small data segments. This can occur due to the advertisement of small windows instead of waiting until a larger window can be advertised. At the source side this can occur due to the sending of small segments instead of waiting for additional acknowledgments. Whenever the size of the advertised window drops below the MSS a window of size zero is advertised. An update packet is then sent when the size of the advertised window reaches half of the maximum window size.

3. **Slow start**: The slow start algorithm adds another local window to the TCP source, a congestion window (cwnd) that is measured in segments. When a packet is lost or when a new connection is established, this window is set to the size of one segment. Each time an acknowledgment is received the cwnd is increased by the size of the acknowledged segments. The source can now transmit up to the minimum of the cwnd and the advertised window.

4. **Congestion avoidance**: Through the exponential increase of the sending rate caused by the slow start algorithm, the capacity of the network will be reached at some time and an intermediate router will start discarding packets. In order to avoid this the slow start algorithm was supplemented through the congestion avoidance algorithm. With this algorithm the congestion window is increased for each acknowledgment as follows:

$$\text{cwnd} += \frac{1}{\text{cwnd}}$$

   This way the congestion window increases by at most one segment each round trip. The relation between the congestion avoidance and the slow start will be explained when discussing the retransmission scheme used in TCP.

5. **Exponential backoff and round trip estimation**: Jacobson described in [5] a method, in which the round trip time (RTT) estimation is based on calculating both the mean and the variance of the measured RTT. To simplify the calculation Jacobson suggested that only the mean deviation should be used instead of the the standard deviation. This leads to the following equations:

$$\begin{aligned} \text{Err} &= M - A \\ A &= A + g\text{Err} \\ D &= D + h\left(|\text{Err}| - D\right) \\ \text{RTO} &= A + 4D \end{aligned}$$

where M is the measured RTT, A is the smoothed RTT, D is the smoothed mean deviation and RTO is the current estimated RTT. The gain g is for the average and is set to $1/8$, h is the gain for the deviation and is set to $1/4$.

In the original 4.3BSD Tahoe TCP implementation only one round trip time is measured per connection at any time. When a measurement is started, a counter is initialized to zero and the sequence number of the sent packet is remembered. The counter is incremented every 500 msec until the acknowledgment for the sent packet is received. During this time no other measurement can start. After receiving the acknowledgment for the sent packet the value of the counter is used for updating the RTO value and a new measurement can start with the sending of the next packet. Thereby, only one measurement is done per sent window.

If the timer goes off before receiving the acknowledgment for that packet, the packet has to be retransmitted and the value of the timer is doubled. After that the value of the timer is doubled for each retransmission with an upper limit of 64 seconds. In the original specification of this algorithm RTO was calculated as $A + 2D$. However, this mechanism was not flexible enough to account for fast changes in the round trip time and occasionally lead to early and unnecessary retransmissions. In the 4.3Reno TCP a factor of 4 was used instead of the original 2 for the variance and almost all spurious retransmissions disappeared.

The retransmission could lead to the so called *retransmission ambiguity problem* .This problem stems from the fact that when an acknowledgment for a retransmitted packet arrives it is not possible to determine whether the acknowledgment is for the lost packet or the retransmitted one. As this would introduce a certain error to the estimation of the round trip time, Karn [13] specifies that when the acknowledgment for a retransmitted packet arrives the RTT estimators should not be updated.

6. **Retransmission**: There are two ways for the source to detect the loss of a packet:

    (a) *Duplicate acks*: The TCP receiver can detect packet loss through keeping track of the sequence numbers in the headers of the received packets. After detecting a gap in the sequence numbers the receiver generates an immediate acknowledgment, a so called *duplicate ack*, for the last correctly received byte. This is repeated whenever an out-of-order packet is received.

    (b) The expiration of the retransmission timer.

    After getting informed about the loss, the source has to retransmit the lost packets and the following steps are taken:

    (a) A variable called *ssthresh* is set to half the current cwnd.

    (b) The congestion window is set to the size of one segment.

    (c) The size of the congestion window is then increased for each received acknowledgment. While cwnd is less or equal to ssthresh slow start is performed and cwnd is increased by the size of the acknowledged segments. Otherwise, cwnd is only increased by 1/cwnd

    Depending on the version of the implementation the source can now send data either starting from the byte number that follows the lost packet or from where it stopped before the retransmission (fast retransmission).

    - Fast Retransmission: When an out of order segment is received TCP generates an acknowledgment of the last correctly received byte and the segment is cached in the local

TCP receiver buffer. As duplicate acks could also be generated due to a reordering of the data packets the source waits for three duplicate acknowledgments to arrive before retransmitting the lost packet. On receiving the lost packet the receiver does not just acknowledge the lost packet but all other segments saved in the local buffer as well. Those buffered segments can now be handed over to the user. If more than one packet got lost, the acknowledgment packet will only acknowledge the retransmitted packet and the correctly received packets. Upon that the sender will have to retransmit the other lost packet.

As all the segments that were sent after the lost packet were also acknowledged the source could just go on sending data starting from the byte number which it reached before the loss of the packet.

7. **Delayed acknowledgments**: To reduce the amount of the sent data, the sending of acknowledgments is delayed until another packet is received or until an acknowledgment timer goes off.

4 The TCP Simulator

The simulator consists of two components that can be simply attached to the network end systems. The two versions have identical structures and can thus be used interchangeably without having to modify the environment in which they are used.

4.1 TCP Source

As can be seen from Fig. 1 the source consists of three main parts that implement the TCP source side protocol. It has two input terminals on which it receives the size of the packets to be sent and the acknowledgments of the sent packets, and an output terminal on which it sends the packets.

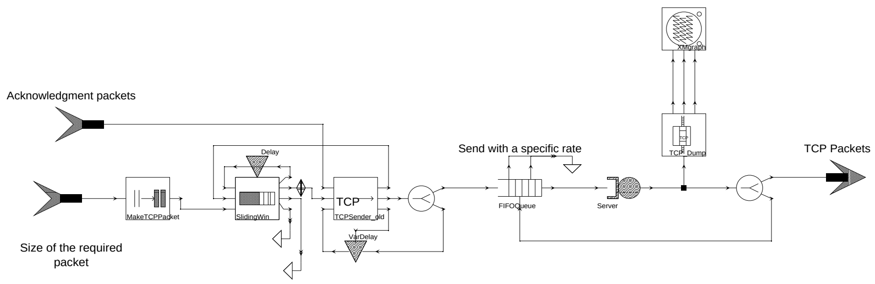

Figure 1: TCP Source

What follows is a brief description of the main stars and their function:

1. **MakeTCPPacket**: Whenever the source end system wishes to send a TCP packet, it must inform this star of the size of the packet to be sent. Upon this a packet is generated that contains information fields for the source and destination addresses, the byte sequence number, the size of the packet and a packet sequence number. Whereas these fields are filled before handing the packet to the next star the additional acknowledgment field stays empty.

2. **SlidingWindow**: This is a FIFO queue that realizes the sliding window in TCP. It has two input terminals: one on which it receives the packets from the MakeTCPPacket star and

another one on which the actual sending instance signals the number of the last byte that can be sent before the window is closed.

When a signal arrives that allows the star to send packets, more than one packet will be sent simultaneously. To avoid sending packets with the same time stamp a delay is introduced between subsequent packets. This delay could either represent the sending rate or the minimal delay between two packets.

3. **TCPSender**: This is the actual sending instance of the source. The arriving acknowledgments and the execution of the different services provided by TCP enable this star to determine the allowed number of bytes that can be sent and if retransmissions are necessary.

   The sender in the two versions differs only in the way this star is implemented. For the basic version this star has to first retransmit all packets that were saved in the local window after a retransmission. Only then can new data be sent. For the enhanced version on the other hand new data can be taken out of the sliding window and be sent directly after receiving the acknowledgment for the lost packet and the packets that were sent after it.

4.2 TCP Receiver

The receiver side of the simulator is responsible for generating the acknowledgments, detecting the loss of packets and simulating the user. This is mainly done in the following two stars:

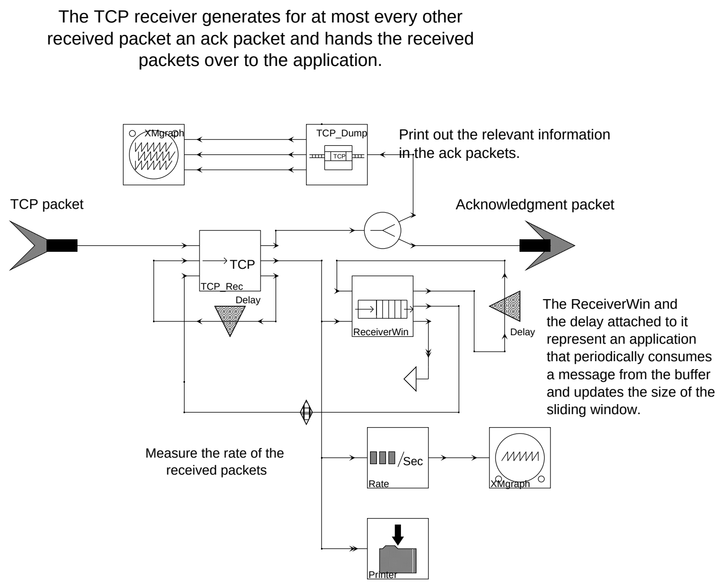

Figure 2: TCP receiver

1. **TCP_Rec**: This is the actual receiver that generates the acknowledgments on receiving data packets and duplicate acknowledgments when out of order packets are received. In the case of fast retransmission, this star must also buffer the out of order packets until the lost packet is retransmitted. Only then are those packets passed to the user.

2. **ReceiverWin**: This star represents a TCP user that consumes a packet at regular intervals. Here, it was built as a self triggering FIFO queue. When the first packet arrives it is passed

directly to a delay star. All the following packets are then queued. When the first packet reenters the star it is discarded and another packet can leave the queue.

Whenever a packet is added or removed from the queue a message containing the current size of the queue is sent to TCP_Rec. With this information the receiver can determine the appropriate size of the advertised window.

5 Using the Simulator

The simulator consists of the two galaxies for the receiver and sender that were mentioned in the previous section. The whole simulator was implemented in the DE domain of PTOLEMY and offers the user the possibility of changing a set of TCP options.

1. **Sender options**: At the source side of the protocol the user can control the following options:
   - *MSS*: The maximum segment size.
   - *Window*: The maximum sender window size.
   - *capacity*: The capacity of the buffer between the packet generator and the actual sender.
   - *source*: Address of the sender.
   - *destination*: Address of the receiver.
   - *initTime*: The initial value for the retransmission timer.
   - *Rate*: The minimum pause between two packets
2. **Receiver options**: At the receiver side of the protocol the user can control the following options:
   - *MSS*: The maximum segment size.
   - *Window*: The receiver window size.
   - *Ack_timer*: The maximum time distance between two acknowledgment packets.
   - *applicationDelay*: The rate at which the application can accept packets from the TCP receiver.

6 Simulator Verification

A crucial but often forgotten step that has to be done with great care when implementing a simulation model is the verification of the model. Here, a network model has been chosen that has already been tested in another study [6]. This reduced the verification requirements to a simple comparison between the results obtained in [6] and the ones obtained when using the simulator to build up the same configuration.

6.1 Network Simulator

The chosen topology is fairly straightforward as can be seen from Fig. 3. A bottleneck switch with a 20 packet buffer is connected to the source and the receiver. The bottleneck transmission between the switch and the receiver has a bandwidth of 50 kbps and a propagation delay of τ (here, it was set to 10 msec). The transmission line between the source and the switch has a bandwidth of 1 Mbps and a propagation delay of 1 msec. The TCP connection is assumed to have a maximum window size of 50 packets, with a constant packet length of 500 bytes.

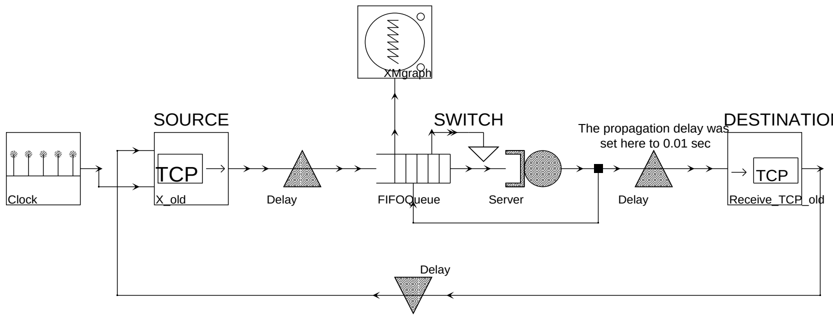

Figure 3: Network topology model

6.2 Simulation Results

As can be seen from Fig. 4 the graph showing the length of the switch buffer obtained with the enhanced version shows a great resemblance to that presented in [6]. A typical cycle consists of a short phase of exponential growth and a longer period in which the congestion avoidance algorithm is used to increase the congestion window. The exponential phase can't unfortunately be shown clear enough as is was too short. The constant period is reached when the congestion window becomes larger than the length of the switch buffer. This is also the moment where packets get lost.

The graph obtained from the basic version (Fig. 4) shows the same behavior. The differences can only be seen in the overall performance.

6.3 Performance Differences

As the fast retransmission algorithm reduces the number of retransmitted packets a higher throughput and a better performance should be expected when adding the algorithm to the TCP congestion control mechanisms.

The effects of introducing the fast retransmission algorithm were already investigated in various other studies, see [14], so there is no need to go much deeper into this subject. Still, a simple comparison between the two versions of the simulator should not just restate what we already know, but also confirm the correctness of the implementation. Fig. 5 was obtained by counting the number of packets that arrive at the user during a certain time interval. It is obvious that the enhanced version not only shows a higher throughput but a more stable one as well.

In a simulation of the length of 300 seconds the TCP receiver got 3704 packets while using the enhanced version. This equals a rate of 12.34 packets/sec. With the basic version on the other hand only 3349 packets got through in the same period. This results in a rate of 11.16 packets/sec which is nearly 10% less than the result obtained with the enhanced version.

The simulation consumed about 62 seconds of real time on a Sun Sparc 5 machine. This equals a processing rate of around 60 packets/sec for the enhanced version. As bad as this might sound it has to be taken into consideration that about 20 events had to be handled for each packet. From these events some were used for calculating and outputting the results into temporary files to enable graphical presentation of the results at the end of the simulation.

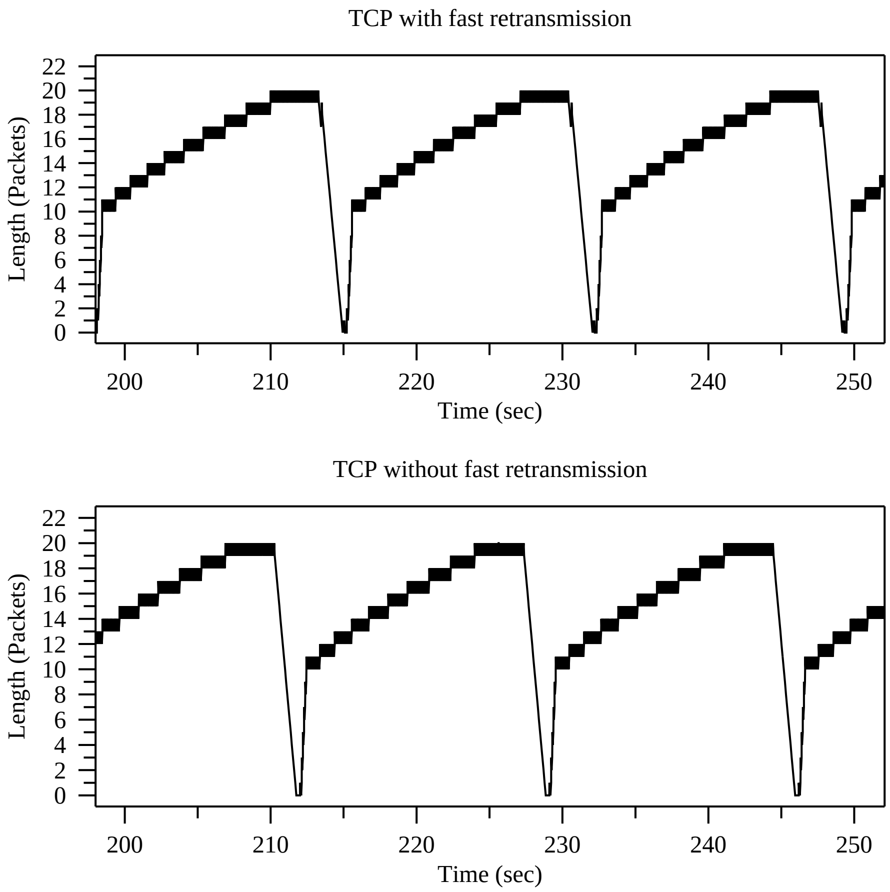

Figure 4: Queue length at the switch vs. time with the simple retransmission scheme

7 General Comments

A great part of the implementation time was spent testing and verifying the model. Still, it would not be a great surprise if the simulator did have some bugs. The results show that the simulator displays the essential dynamics of TCP's congestion control algorithm, and we believe that the simulator can be relied upon.

There is still a lot more to TCP's congestion control mechanism than the implemented parts and the points mentioned in Sec. 3, e.g., time stamp option, fast recovery. But since these points are only of minor interest and importance and have only a small influence on the performance of the protocol they will not be implemented until their necessity is shown during further tests and simulations.

References

[1] S. Keshav, "REAL: A network simulator," Tech. Rep. 88/472, Department of Computer science, UC Berkeley, 1988.

[2] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, Jan. 1991.

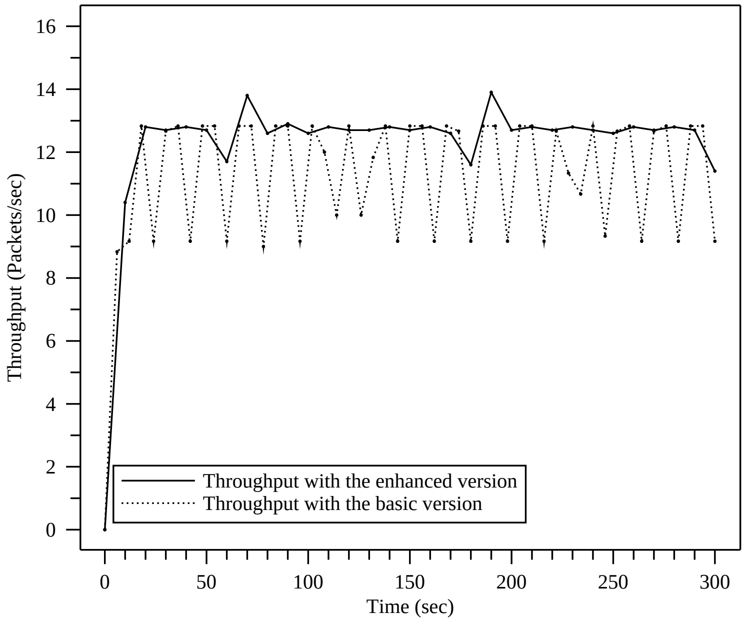

Figure 5: A comparison of the throughput of the two versions

[3] P. B. Danzig and S. Jamin, "tcplib: A library of TCP internetwork traffic characteristics," Tech. Rep. USC-CS-91-495, Computer Science Department, University of Southern California, Los Angeles, California, 1991.

[4] A. Y. Lao, "Heterogeneous cell-relay network simulation and performance analysis with PTOLEMY," Tech. Rep. UCB/ERL M94/8, Electronics Research Laboratory, UC Berkeley, 1993.

[5] V. Jacobson, "Congestion avoidance and control," *ACM Computer Communication Review*, vol. 18, pp. 314–329, Aug. 1988. Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988.

[6] S. Shenker, L. Zhang, and D. D. Clark, "Some observations on the dynamics of a congestion control algorithm," *ACM Computer Communication Review*, pp. 30–39, Oct. 1990.

[7] J. Postel, "Transmission control protocol," Request for Comments (Standard) STD 7, RFC 793, Internet Engineering Task Force, Sept. 1981.

[8] V. Jacobson, "Berkeley TCP evolution from 4.3-Tahoe to 4.3-Reno," in *Proceedings of the Eighteenth Internet Engineering Task Force*, (University of British Columbia, Vancouver), pp. 363–366, Sept. 1990.

[9] D. Borman, R. Braden, and V. Jacobson, "TCP extensions for high performance," Request for Comments (Proposed Standard) RFC 1323, Internet Engineering Task Force, May 1992. (Obsoletes RFC1185).

[10] L. S. Brakmo and S. W. O'Malley, "TCP Vegas: New techniques for congestion detection and avoidance," in *SIGCOMM Symposium on Communications Architectures and Protocols*, (London, United Kingdom), pp. 34–35, ACM, Aug. 1994.

[11] D. Sisalem, "Rate based congestion control and its effects on TCP over ATM," diplomarbeit, Technical University of Berlin, June 1995.

[12] W. R. Stevens, *TCP/IP illustrated: the protocols*, vol. 1. Reading, Massachusetts: Addison-Wesley, 1994.

[13] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," *ACM Transactions on Computer Systems*, vol. 9, pp. 365–373, Nov. 1991.

[14] S. Floyd, "Simulator tests." submitted for publication, Oct. 1994.

[15] S. Leffler, K. McKusick, M. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Massachusetts: Addison-Wesley Publishing Company, 1989.