Improving the Start-up Behavior of a Congestion Control Scheme for TCP

Janey C. Hoe
Laboratory for Computer Science
Massachusetts Institute of Technology
janey@ginger.lcs.mit.edu

Abstract

Based on experiments conducted in a network simulator and over real networks, this paper proposes changes to the congestion control scheme in current TCP implementations to improve its behavior during the start-up period of a TCP connection.

The scheme, which includes Slow-start, Fast Retransmit, and Fast Recovery algorithms, uses acknowledgments from a receiver to dynamically calculate reasonable operating values for a sender's TCP parameters governing when and how much a sender can pump into the network. During the start-up period, because a TCP sender starts with default parameters, it often ends up sending too many packets and too fast, leading to multiple losses of packets from the same window. This paper shows that recovery from losses during this start-up period is often unnecessarily time-consuming.

In particular, using the current Fast Retransmit algorithm, when multiple packets in the same window are lost, only one of the packet losses may be recovered by each Fast Retransmit; the rest are often recovered by Slow-start after a usually lengthy retransmission timeout. Thus, this paper proposes changes to the Fast Retransmit algorithm so that it can quickly recover from multiple packet losses without waiting unnecessarily for the timeout. These changes, tested in the simulator and on the real networks, show significant performance improvements, especially for short TCP transfers. The paper also proposes other changes to help minimize the number of packets lost during the start-up period.

1 Introduction

Since the specification of Transmission Control Protocol (TCP) in 1981 [14], implementations of TCP have been augmented with several performance-enhancing algorithms, such as congestion control. Implementations are now expected to include Jacobson's congestion control algorithms as described in [8] and fast retransmit and Fast Recovery algorithms [9, 10, 17]. (We refer to these algorithms collectively as the congestion control scheme.) Based on experiments conducted in a simulator and over the networks, this paper analyzes the behavior of the congestion control scheme, focusing on the start-up period. It also proposes changes to improve the performance during this period. The changes require only modifications to a TCP sender, and a TCP with these changes is interoperable with any existing TCP implementation.

In a TCP connection, the congestion control scheme uses acknowledgments (ACK's) from the receiver to dynamically calculate reasonable operating values for TCP parameters, which determine when and how much the sender can pump into the network. When the sender starts up, without any information about the network capacity and the receiver, the scheme uses default values for the parameters. Since these default values are arbitrary, the sender usually ends up outputting too many packets [1] too quickly and thus losing multiple packets in the same window.[2] We show that recovery from these losses is often unnecessarily time-costly.

More specifically, using the current Fast Retransmit algorithm, when multiple packets are lost, only one of the packet losses may be recovered by the algorithm; the rest are often recovered by Slow-start after a usually lengthy retransmission timeout. Based on these observations, we propose several changes to improve this algorithm. One proposed change makes better use of existing parameters to define a Fast Retransmit *phase* so that the algorithm can recover multiple packet losses without waiting unnecessarily for the timeout. The other changes define how packets are retransmitted during the Fast Retransmit phase to allow faster recovery of multiple packet losses. Another change proposes using an estimated value instead of the default value for an important parameter, *ssthresh*,[3] to minimize the number of packets lost during the start-up period.

Tested in a network simulator and over real networks, these simple changes (i.e. translates into a few lines of

[1] In this paper, we use the terms packets and segments interchangeably.

[2] Although the paper focuses on the start-up period, the experiments over real networks show that multiple losses of packets from the same window can occur several times throughout a data transfer. The proposed changes can apply to those cases as well.

[3] In this paper, we keep TCP parameter names as close to those in current TCP implementations [18] as possible.

code) to a sender's TCP allow for significant performance improvements during the start-up period. TCP's performance during this period is important because this period is a significant portion of the duration of most data transfers over TCP connections for two reasons. First, with the increased complexity and size of networks, this start-up period of probing the network for reasonable operating parameters lasts longer in duration. Second, a large number of application protocols, e.g. FTP, HTTP, etc., use TCP for short transfers, which deliver relatively small number of data segments and terminate before TCP settles into its steady-state behavior.

We start by discussing closely related work. After describing the congestion control scheme using graphs from simulations, we show that similar behavior occurs in real networks. Finally, we propose changes to improve the performance. We show the results of testing these changes in the simulator and over real networks.

1.1 Related Work

Van Jacobson's important paper [8] defines his congestion avoidance and control scheme, generally known as Slow-start. The scheme was later modified to include Fast Retransmit and Fast Recovery algorithms [10, 9]. We refer to this scheme in Section 2.

Selective acknowledgment (SACK) options [12] have been proposed to be added to TCP standards. Using selective acknowledgments, a receiver can inform the sender about the segments that have arrived successfully, so a sender needs to retransmit only the missing segments. Because SACK can give a sender more information about which segments are actually cached at the receiver, algorithms designed to be used with SACK, such as Forward Acknowledgment congestion control algorithm [13], may allow more precise control over the data flow in the networks. Fall and Floyd [3] used simulations to compare the performance of Tahoe, Reno, SACK TCP, and New Reno, which is Reno TCP with an earlier version of our proposed changes[7]. Although selective acknowledgments and related algorithms can resolve the issues in the recovery from multiple packet losses, they require cooperative receivers. The proposed algorithms in this paper require simple changes only to TCP implementation at the senders and are consistent with current TCP specification.

The sender-side congestion avoidance algorithm of TCP Vegas [2] changes Slow-start's linear growth. Instead, Vegas can increment, decrement or not adjust the window by one segment every roundtrip time. Work in [1] evaluates Vegas' performance compared against Reno.

Shenker and Zhang [15] use simulations to make some observations about the behavior of the congestion control algorithm in the 4.3-Tahoe BSD TCP implementation. Floyd and Jacobson [5] discusses bias against certain traffic as a result of traffic phase effects in networks with Drop Tail gateways. They show that Random Early Detection (RED) gateway, a gateway congestion avoidance algorithm that monitors the average queue size and drops packets when the average queue size gets too large, can correct the bias against bursty traffic.

Our previous work [7] discusses some observations of the congestion control scheme in the BSD Net/2 implementation in detail, describing the effects of congestion window adjustments, delayed acknowledgment, damped exponential phase of Slow-start after a timeout, and Fast Retransmit and Recovery algorithms. It proposes some changes to the congestion control scheme. This paper mainly focuses on the fast retransmit and Fast Recovery algorithms, and presents revised changes to the algorithms, based on our earlier work.

2 The Congestion Control Scheme

The congestion control scheme in current TCP implementations has two main parts: 1) Slow-start and 2) Fast Retransmit and Fast Recovery algorithms.

2.1 Slow-Start

Using Slow-start, a TCP sender starts with a congestion window of one segment and exponentially increases the congestion window. When the congestion window hits a threshold, *ssthresh*, the sender continues to increase the congestion window linearly, probing the network capacity as it becomes available. Evidently, in this scheme, the choice of the threshold, which is an estimation of the equilibrium operating point, i.e. a packet leaves the network as a sender puts a packet into the network, is key to the performance of the algorithm.

In Figure 1,[4] we show a segment number versus time graph of a 1-Mbyte transfer in the Netsim simulator [6], running BSD Net/2 TCP implementation. All the simulator experiments in this paper use a segment size of 1024 bytes, and the maximum window size for the connection is 50 segments. (For details about the simulation environment and how to read the graph, see the appendix.) In this implementation, *ssthresh* is initialized arbitrarily to 64 segments. In Figure 1, this high initial threshold allows the sender to exponentially ramp up the sending rate, leading to multiple packet losses very soon after the connection starts.

A packet loss can only be recovered by either the Fast Retransmit algorithm or Slow-start after a retransmission timeout. In this case, the Fast Retransmit is unable to recover multiple packets, and thus the connection has to idlely wait for a timeout and finally recover the lost packets with Slow-start at 1.5 sec. In the next section, we look at the Fast Retransmit algorithm in more detail by zooming in on each of the circled regions in Figure 1.

2.2 Fast Retransmits and Fast Recovery

Since a receiver acknowledges the highest *in-order* sequence number it has seen so far, when it receives out-of-order packets, it generates acknowledgments for the same highest in-order sequence number (i.e. duplicate ACK's). Thus, when

[4] To facilitate detailed discussion, we use segment numbers instead of sequence numbers in the graphs of simulator results. In the graphs of the results from real networks showing general behavior, we use sequence numbers.

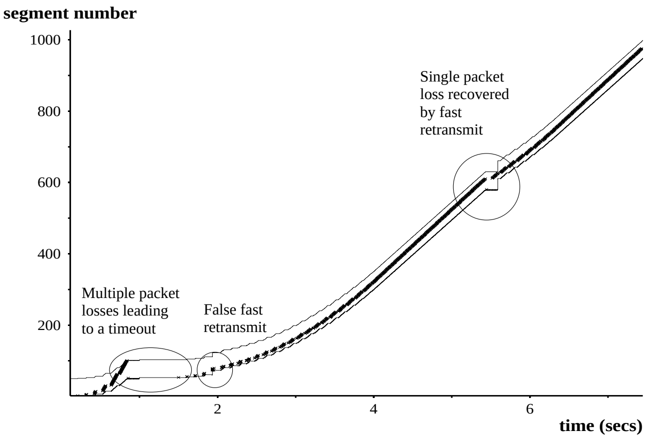

Figure 1: A 1-Mbyte transfer in Netsim simulator

a sender receives three duplicate ACK's, the Fast Retransmit algorithm deduces that a segment has been lost. It assumes that the missing segment starts with the sequence number immediately after the number acknowledged by the duplicate ACK's, and the presumed missing segment is retransmitted. In addition, *ssthresh* and the congestion window are lowered to approximately half of the congestion window size prior to the Fast Retransmit to slow down the sending rate. The Fast Recovery algorithm refers to the way the congestion window and *ssthresh* are adjusted so that after a Fast Retransmit, the sender slows down and enters a mode that linearly instead of exponentially increases the congestion window.

In this section, we discuss the Fast Retransmit algorithm being invoked under two circumstances: (1) single packet loss and (2) multiple losses of packets from the same window. We also look at a case in which a Fast Retransmit is falsely invoked. We show that the Fast Retransmit algorithm works well in the first case, but is unable to recover from the second case.

2.2.1 Single Packet Loss

Figure 2 shows the connection successfully recovering from a single packet loss using Fast Retransmit. At time right before 5.43 sec (in the circled region), the connection starts receiving duplicate ACK's for segment 579. Upon receiving three duplicate ACK's for segment 579, segment 580 is Fast Retransmitted, and the congestion window size is halved. Right before the Fast Retransmission occurred, the connection has a congestion window of size say *old_cwnd*, and therefore, *old_cwnd* segments are outstanding. In the bottom graph, [5] we see that with each duplicate ACK that

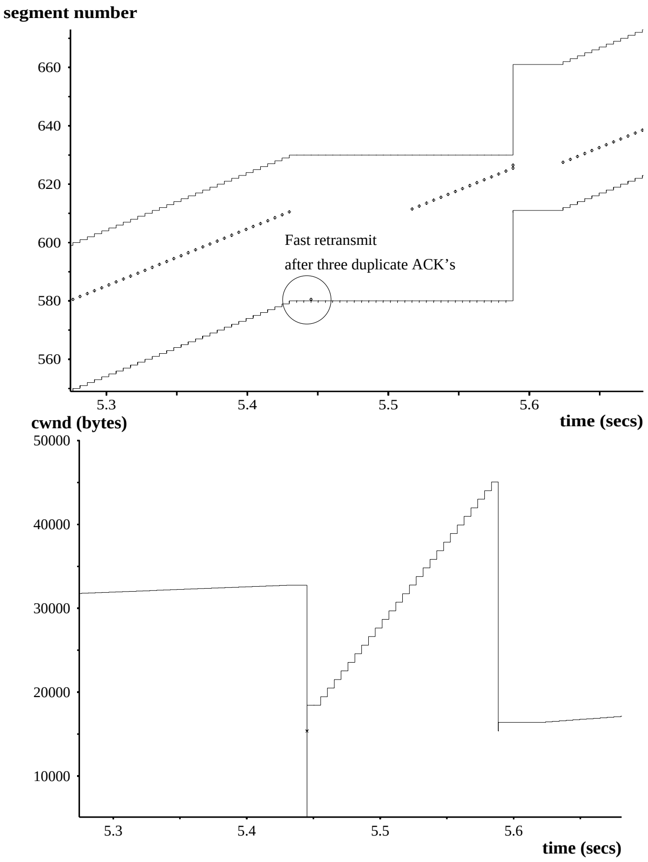

Figure 2: The connection recovers from a single packet loss after a Fast Retransmit. The bottom graph shows the corresponding congestion window size.

continues to come in after the Fast Retransmit, the congestion window is increased by one segment. At time 5.52 sec, we see that the congestion window is increased to the value *old_cwnd*. As the congestion window continues to open, the connection is able to have more segments outstanding and send new segments. At time right before 5.6 sec, an ACK for a large number of segments, including segment 579, is received. This ACK causes the inflated congestion window (accounting for cached segments at the receiver side) to be retracted to *ssthresh*, and the sender enters the linear mode as a result of the Fast Recovery algorithm.

In Figure 3, we make the observation that the group of segments sent between 5.52 sec and 5.6 sec triggers the group of ACK's to come back in a similar pattern, triggering the next group of packet to be sent out in a similar pattern, and so on. This "grouping" effect becomes less obvious after 6.2 sec, as the pipeline is being filled and the ACK's are coming in continuously. This effect is a good demonstration of TCP using the acknowledgments as a self-clocking signal discussed in [8].

2.2.2 Multiple Packet Losses

In this section, we show that multiple losses of packets in the same window cannot be recoverable by the Fast Retransmit

[5] The sudden deep fade of congestion window size is a result of the implementation details of Fast Retransmit. When Fast Retransmit is activated, *ssthresh* is set to half of the minimum of the congestion window and the send window. The congestion window drops to one segment and tcp_output() is called so that only one segment is fast retransmitted. The congestion window is then increased to *ssthresh* plus one segment so that the sender enters linear increase mode immediately after the fast retransmit. Note that the graph shown is a zoomed-in view of a 1-Mbyte transfer, and thus the lowest value of the y axis is not 0 bytes

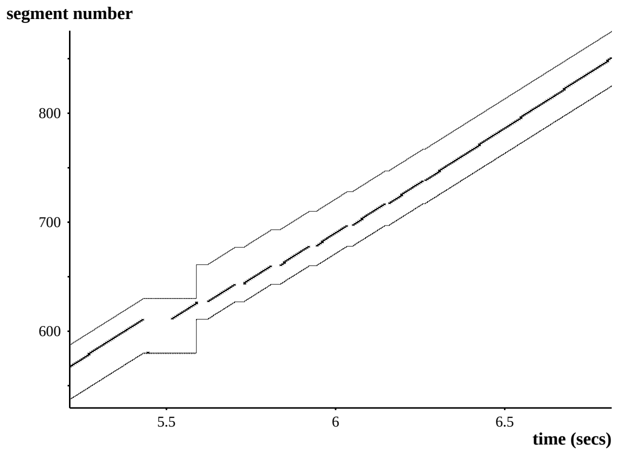

Figure 3: The "grouping" effect during the loss recovery after a fast retransmit triggered by a single packet loss

algorithm, and thus the connection must wait for a retransmission timeout. We show one particular case in Figure 4. As seen in the figure, the large congestion window at time right before 0.8 sec allows a burst of closely spaced packets into the network overflowing the queue in the bottleneck switch.

To take a closer look at the Fast Retransmit algorithm, we will focus on segments 50 through 54. The packet trace in the simulator indicates that segments 51 and 53 were dropped at the bottleneck switch and segments 50 and 52 were received. Figure 4 shows how the current scheme recovers packet 51 and 53. Around the time 0.83 sec (in the circled region), the sender receives a duplicate ACK for segment number 50. At this point, the sender is not able to send any new segments, since a full window of segments is outstanding. As time proceeds, a group of closely-spaced duplicate ACK's for segment 50, triggered by the large surge of segments sent earlier, is received. On the third duplicate ACK, segment 51 is Fast Retransmitted, causing an ACK for segment 52 to return at time 1 sec.

We discuss the above sequence of events in further detail. When the sender received three duplicate ACK's for segment 50, it can deduce that at least two segments following segment 50 have been received (assuming that the network is not duplicating packets) and that segment 51 has *not* been received. When the sender receives the ACK for segment 52 as a response to the Fast Retransmit of segment 51, this implies that the receiver already had segment 52. Since segment 52 is confirmed as one of the two received segments following segment 50, the other received segment is either segment 53 or 54. However, because the receiver only ACKed up to segment 52 not 53, this ACK is a clear cue that segment 53 may have been lost. Although it is possible that segment 53 has been reordered in the network and may still arrive at the receiver, the series of ACK's imply that 53 has been delayed by more than one entire roundtrip time relative to segment 54, which strongly suggest that segment 53 is lost. As we will see, this clear cue is not used to allow fast recovery of lost segments.

At this point, only two events can cause the retransmission of segment 53 to occur: (1) another activation of the Fast Retransmit algorithm and (2) a retransmission timeout. A Fast Retransmit to recover segment 53 at this point

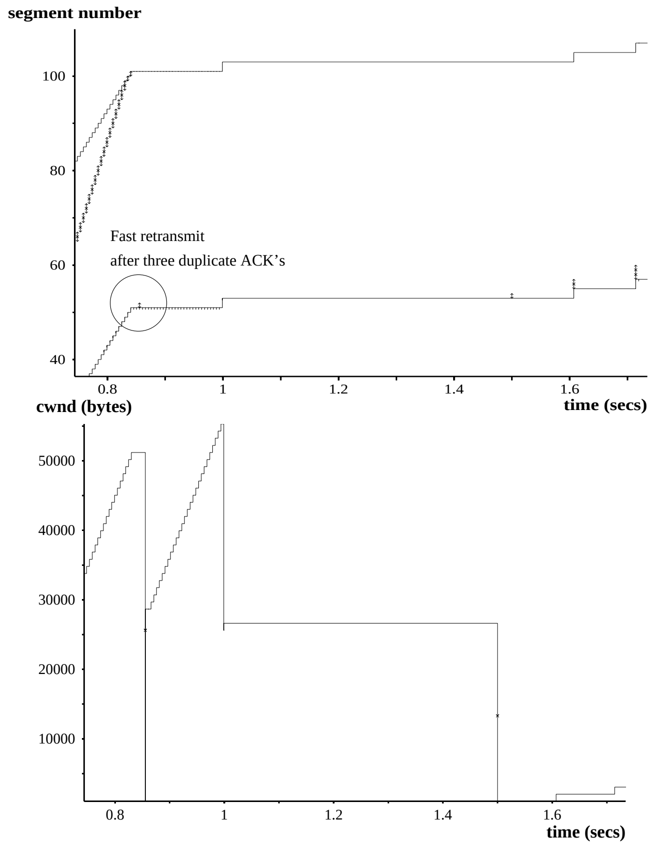

Figure 4: As a result of multiple packet losses, Fast Retransmit did not work here. The sender has to wait for a retransmit timeout. The bottom graph shows the corresponding congestion window size.

is probably out of the question, since the connection is idle and no duplicate ACK's is expected. More specifically, the value of the congestion window before the last Fast Retransmission is 51 segments, which is the beyond the maximum send window size. The surge of segments the sender transmits before 0.83 sec triggers the duplicate ACK's seen between time 0.83 sec and 1 sec. After the Fast Retransmit of segment 51, the congestion window is reduced to approximately half. With each duplicate ACK the sender receives, the sender is able to open the congestion window by one segment, as seen in Figure 4. However, since the sender is not allowed to have more than a full window of data outstanding, it cannot send any more segments until non-duplicate ACK's comes in. Because many roundtrip times have already passed since the last transmission, the likelihood of any packets being stuck in the network and still having the possibility of making it to the receiver is minimal. Thus, without any further transmission, no further ACK's can be triggered, and therefore fast retransmission cannot be used to retransmit segment 53. The only other way for retransmission to occur is to wait for the retransmission timeout, which finally occurs at time 1.5 sec.

The underlying issues of this time-consuming loss recovery are (1) the initial arbitrary value of *ssthresh*, which allows the sender to clock out a large surge of packets in exponential slow-start mode leading to multiple packet losses and (2) the failure of the Fast Retransmit algorithm to re-

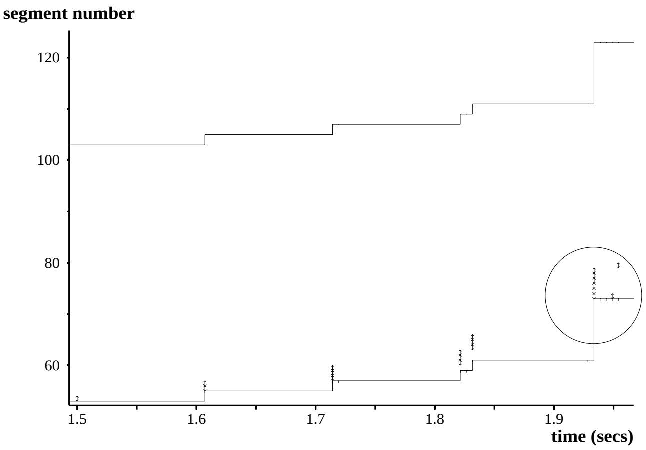

Figure 5: False Fast Retransmit

cover the lost packets. The result is that the sender must wait for the long retransmission timeout, which drastically reduces the performance during the start-up period. We mentioned the ACK for segment 53 is a cue not used by the Fast Retransmit algorithm. In the proposed changes, we suggest making better use of existing information so that we can take advantage of the cue.

2.2.3 False Fast Retransmits

In some cases, false Fast Retransmits can occur. We observe the case at time 1.95 sec (in the circled region) in Figure 5. The sender is in the exponential mode immediately after a retransmission timeout, recovering from an episode of multiple losses of packets in the same window. More specifically, odd numbered segments between segment 50 and segment 62 are lost. When the out-of-order segments (segments 52, 54, 56, etc.) arrive at the receiver, they are stored in the receiver's reassembly queue until the missing odd-numbered segments arrive. To explain Figure 5, we tabulate the beginning of the recovery process from the sender's perspective in Table 1.

We see that the retransmission of a segment that was not lost, i.e. already cached at the receiver, generates a duplicate ACK. Around time 1.94 sec, the earlier retransmission of segments that are cached at the receiver leads to a series of duplicate ACK's and thus a Fast Retransmit in the circled region. We call this a false Fast Retransmit, since the algorithm is activated even though there is no segment loss. The false retransmit mistakenly forces the sender to go into the less aggressive linear mode, when there is really no congestion.

In the same circled region, we also note a large spike right before the False Retransmit. This spike is due to the large ACK that arrived, which acknowledged most of the outstanding segments. We discuss this further in Section 3.3.

3 Proposed Changes

In Section 2, we made some observations on the start-up dynamics of TCP. To improve TCP's performance during this period, it is useful to review the observations and note the episode of events that is time-costly. We repeated the same experiments from Section 2 many times over real networks from a MIT machine to a Berkeley machine to show that the previous observations in the simulations also occur in real networks. (Details of the experiments are in the appendix). We show in Figure 6 one of the transfers over real networks that closely resemble our simulation results. We see the familiar episode of a surge of packets being sent, leading to multiple packet losses and an unsuccessful attempt to recover those packets using Fast Retransmission. The end result is the long wait for the retransmission timeout, which is shown in the figure as the flat portion of the graph during which the sender is not able to send any segments. Note that during the first 3 seconds of the transfer, about half of the time was spent waiting for the retransmission timeout.

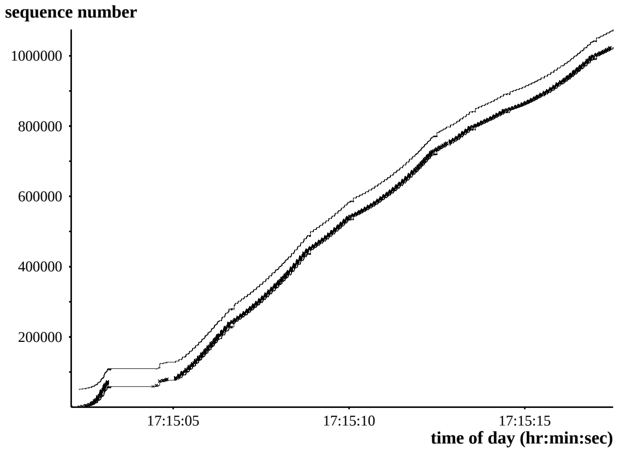

Figure 6: A 1-Mbyte transfer from a east coast machine to a west coast machine using the original congestion control scheme

To deal with these issues, there are two simple approaches: (1) curtail the surge of packets that lead to multiple packet losses and (2) change the Fast Retransmit algorithm so that it recovers from multiple losses of packets in the same window and thus reduces the need to wait for the retransmit timeouts. The first approach can be implemented by finding a better initial value of *ssthresh*, and the second requires a more aggressive Fast Retransmit algorithm to recover lost segments.

We discuss these approaches below. We also mention briefly a way to deal with the large spike in the circled region of Figure 5. These approaches give us a basis for future work on TCP performance during the start-up period.

3.1 A Better Initial Value of *ssthresh*

From the previous section, we see that the initial value of *ssthresh* is critical. One way to avoid the large surge of packet that leads to multiple packet losses is to pick a better *ssthresh* than the arbitrary initial value. A lower *ssthresh* would allow the congestion window to open exponentially, aggressively up to *ssthresh* and then open additively (one segment per window), probing the capacity of the "pipe" instead of overfeeding it. However, if the initial *ssthresh*

Time (sec)	Segments Sent	ACK's Received
1.5	Segment 53 is retransmitted.	
1.61		The retransmission of segment 53 at time 1.5 sec leads to an ACK of two segments: segments 53 and 54. The receiver acknowledges segment 54 as well, since segment 54 is already cached in the reassembly queue at the receiver.
1.61	Segments 55 and 56 are retransmitted.	
1.72		The retransmission of segment 55 leads to an ACK of two segments: segments 55 and 56, since segment 56 is already cache in the reassembly queue.
1.72	Segments 57, 58, 59 are retransmitted.	
1.725		The retransmission of segment 56 at time 1.61 sec leads to a *duplicateACK* of segment 56.
1.82		The retransmission of segment 57 leads to an ACK of two segments: 57 and 58, since segment 58 was already cached at the receiver.
1.82	Segments 60, 61, and 62 are retransmitted.	
1.83		The retransmission of segment 58 leads to duplicate ACK of segment 58.
1.835		The retransmission of segment 59 leads to an ACK of segments 59 and 60 since segment 60 was already cached at the receiver.
1.835	Segments 63, 64, and 65 are retransmitted.	
1.93		The retransmission of 60 leads to duplicate ACK of segment 60.
1.935		The retransmission of segment 61 leads to an ACK of segment 61 through 72 since they were cached at the receiver.
1.935	Segments 73 through segment 78 are retransmitted.	
1.94		The retransmission of 61 leads to a duplicate ACK of segment 72.
1.945		The retransmission of 62 leads to the second duplicate ACK of segment 72.
1.95		The retransmission of 63 leads to the third duplicate ACK of segment 72.
1.95	Segment 73 is Fast Retransmitted even though it hasn't been lost.	

Table 1: Tabulation of events leading up to a false Fast Retransmit

is set too low, the sender would prematurely switch to the additive increase mode, and the performance would suffer. As a result, although there is no packet loss, the sender would be sending so slowly that the transfer would take significantly longer.

So, we need to find an estimate of the threshold at which the sender is closely approaching the full network capacity and thus should slow down and probe the remaining capacity. An example of such an estimate would be the bandwidth-delay product. We discuss one simplified way to estimate this value and show some preliminary results.

Data packets, which are sent closely spaced, arrive at the receiver at the rate of the bottleneck link bandwidth. If the ACK's arrive at the sender with approximately the same spacing, using the ACK's and the time at which they arrive, we can calculate an approximation of the bandwidth. The round-trip delay can be approximated by timing one segment, reading the timestamp upon sending the segment and reading the timestamp again once the ACK for the segment is received. We can then calculate the bandwidth-delay product. We set *ssthresh* to the byte-equivalent of this product.

In Figure 7, we show the simulation results from initializing *ssthresh* with the bandwidth-delay product. We allow *ssthresh* to be initialized to 64 segments as before. We time the $SYNC$ segment, the first segment transmitted by a sender for synchronization between the receiver and the sender, to get an approximation of the round-trip delay. Once the connection is established, the bandwidth is calculated by using the least-squares estimation on three closely-spaced ACK's received at the sender and their respective time of receipt. (This is similar to the concept of Packet-Pair algorithm in [11].) *Ssthresh* is then adjusted to the resulting estimate. As a result, we see a very smooth transfer without retransmission timeouts, since the estimate of the *ssthresh* estimate prevented the episode of a large surge of packets that leads to multiple packet losses. The occasional single packet loss is effectively recovered by the Fast Retransmit algorithm.

This estimation method makes the fairly conservative assumption that *ssthresh* can be at least four segments, so that there would be enough time to get the three ACK's needed for the estimation and pull the *ssthresh* value down to the estimated value in time to prevent the excessive ramp-up that leads to multiple packet losses.

Since our topology is simple and involves only one unilateral transfer, the estimation method works well for our transfer In reality, we may not be able to obtain an estimate as good or as quickly. In particular, problems such as ACK-Compression causing ACK's to return to the sender differently spaced as the segments they ACK may lead to inaccurate estimations. However, the important issue is not that the bandwidth estimation be completely reliable but that the value of *ssthresh* chosen using this estimate be no worse than and frequently better than the arbitrary initial value in current implementations. Even if the resulting estimate is higher than the "right" value, it would be better than the arbitrary maximum value of 64 segments in our case, which allows the sender to open the congestion window too aggressively and leads to many packet losses. In the worst case, if the estimate is too low, the sender may end up being too conservative, and the resulting performance for that one connection can be worse than losing multiple packets and waiting for a retransmit timeout to recover. However, a conservative sender is more desirable than an overly aggressive one in the interest of the collective performance of the network.

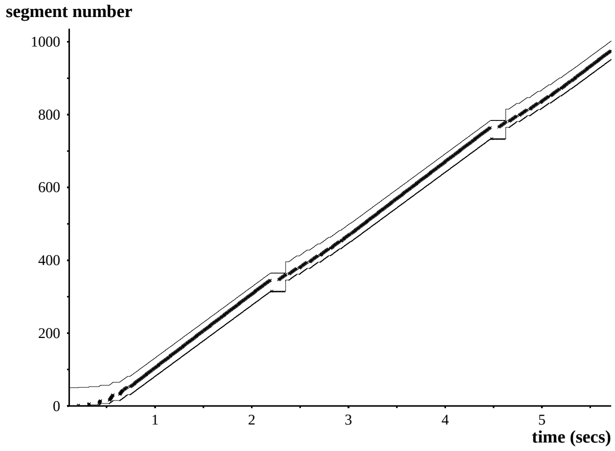

Figure 7: A 1-Mbyte transfer initializing the *ssthresh* with the byte-equivalent of bandwidth-delay product

3.2 Recovery from Multiple Packet Losses

We propose a change to the Fast Retransmit algorithm so that it may recover from multiple packet losses more effectively by making better use of the cues and implications of the ACK's discussed earlier so that the recovery process using Slow-start begins sooner, instead waiting for a retransmission timeout. We introduce a new concept of Fast Retransmit phase.

In Figure 8, we outline the Fast Retransmit phase algorithm. Note that step (i) in the setup is the same as that in the original congestion control scheme. Step (ii) marks the highest sequence number, *snd_high*, the TCP sender has sent so far. This is used to define the Fast Retransmit phase. The phase begins when 3 duplicate ACK's are received, and it ends when the sender receives an acknowledgment for a sequence number that is greater than or equal to *snd_high*. Being in the Fast Retransmit phase means that multiple losses of packets in the same window have not been all recovered. So, while a sender is in the phase, it continues to retransmit packets using Slow-start until all packets have been recovered. Although this approach may cause unnecessary retransmissions, the cues in the ACK's as discussed in Section 2.2.2 do give the sender a strong indication of which segments have not been received, and the number of unnecessary retransmissions should be low. Step (vi) is simply an attempt to keep the "flywheel" going; with every two duplicate ACK's we receive, we force out a new data segment that has never been sent before. This keeps new data flowing in the pipe and thus triggers additional acknowledgments.

We tested the new algorithm (with the exception of step (vi)) in many experiments over real networks. A typical sequence number versus time graph is shown in Figure 9.

Compared to Figure 6, we see significant improvements during the start-up period. The transfer was able to proceed smoothly without unnecessary timeouts. The new algorithm was triggered 4 times (in the 4 circled regions of the figure) in this particular experiment, and we zoom in on one in Figure 10. We see that once in the Fast Retransmit phase, the sender continues to retransmit using Slow-start until all of the lost packets are recovered.

Compared to the original algorithm, the Fast Retransmit phase algorithm seems more aggressive. One may argue that the long retransmission timeout after multiple packet losses is a good idea, since it allows the queues in the network to drain. In the fast retransmit phase algorithm, a retransmission timeout serves only as the backup plan, used only when all else fails. Waiting for queues in the network to drain is necessary when nothing is getting through the network. However, while in the Fast Retransmit phase, acknowledgments are returning to the sender, indicating that data is still flowing in the network. Thus, as long as the sender proceed with precaution (never exceeding half of its original rate that led to multiple packet losses, as dictated by step (iv)), the sender can continue to retransmit packets until all packets losses have been recovered.

The concept of a *Fast Retransmit phase* can also be used to fix the problem of false Fast Retransmits as pointed out by [4]. Since duplicate ACK's that acknowledge segments from the same window as the segments from a previous Fast Retransmit are not an indication of continued congestion, the sender can ignore all duplicate ack's as long as it is in the *Fast Retransmit phase*. Thus, step (v) in Figure 8 eliminates false Fast Retransmits.

3.3 Limiting the Output of Successive Packets

As mentioned in Section 2.2.3, during the recovery of multiple packet losses, a large ACK can trigger a sudden surge of segments to be sent as shown in the circled region of Figure 5. This sudden surge may lead to further loss of segments. One way to deal with this problem is to limit the number of segments TCP can output successively in such a situation. We leave the analysis of this issue for future work.

4 Discussion

We have to be careful when comparing the performance of TCP's with and without the proposed changes. In particular, it is not useful to compare the overall throughput of Figure 9 and Figure 6, since the overall throughput is dependent on a number of factors that are hard to keep constant in all of the experiments over real networks. We have repeated the experiments numerous times, and each time, we get a graph that looks a little different in the slope and in the number of packet loss episode. For example, the throughput can be dependent on the time of the day, the forward and reverse path, the number of data flows on the same path, etc. What is useful to observe in these graphs is the behavior of the congestion control scheme in response to the packet losses. And it is clear that the proposed algorithms perform better in the start-up period since it is able to recover

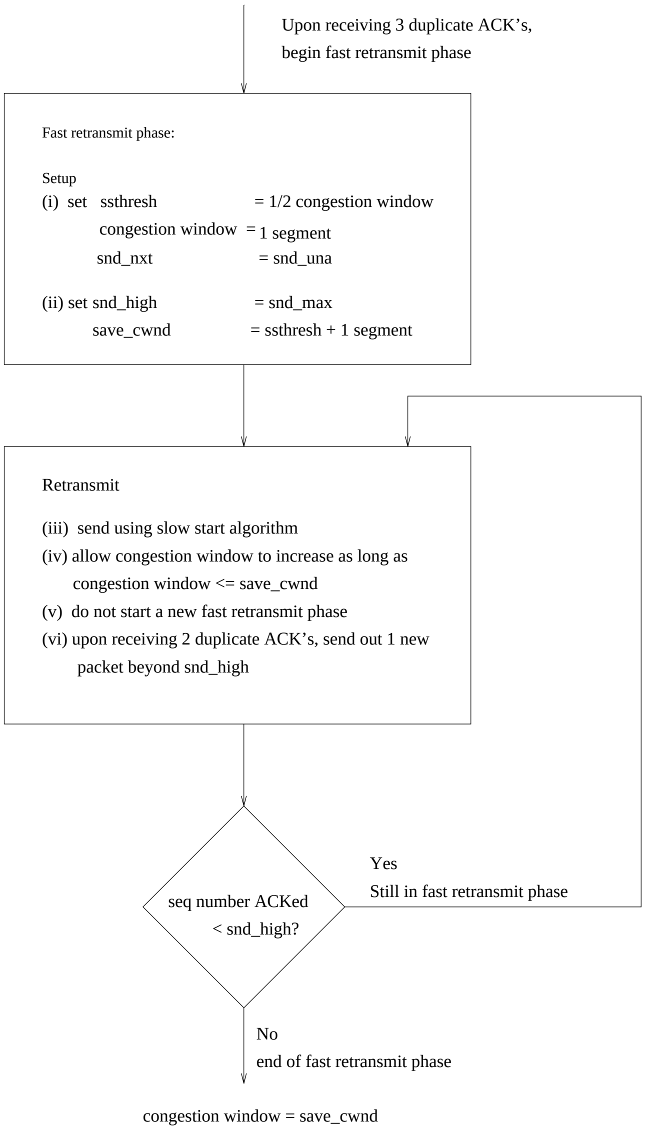

Figure 8: Fast Retransmit phase algorithm with proposed changes

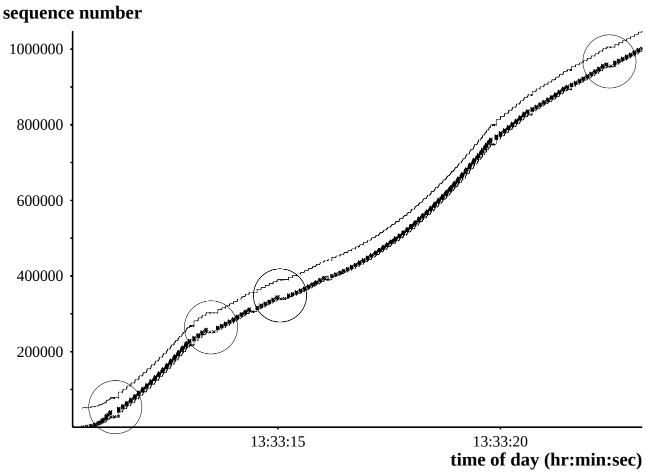

Figure 9: Testing the proposed new fast retransmit algorithm over real networks

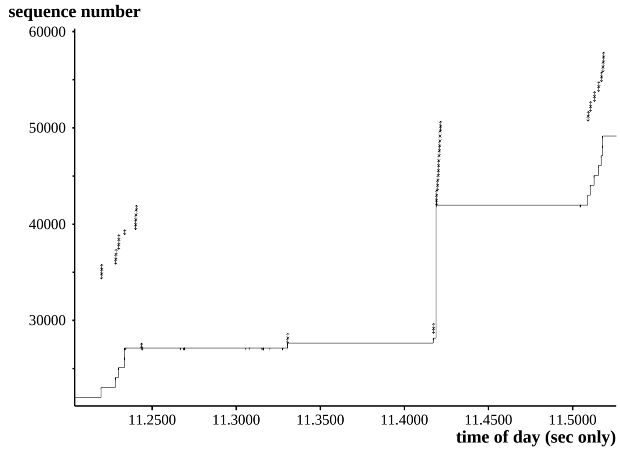

Figure 10: A blowup of a section of the previous figure

from multiple packet losses. The performance during this period is especially important because many, if not most, TCP connections do not last very long.

The proposed algorithms require very simple changes to a sender's TCP implementation. An example of an application is to implement the proposed algorithms in a busy FTP server to allow speedup of frequent short data transfers to any TCP receiver.

5 Summary

This paper proposes several possible changes to improve the start-up behavior of the congestion control scheme. Changes proposed include using an estimated value instead of default value for the important parameter, *ssthresh*, at start-up. More notedly are the changes to the Fast Retransmit algorithm. Tested in simulations and over real networks, these changes can significantly improve TCP's performance during the start-up period by eliminating the wait for unnecessary timeouts. These improvements are especially noticeable in short TCP data transfers, which are becoming increasingly common. More extensive experiments, incorporating all the proposed changes, are being conducted.

6 Acknowledgment

The author would like to thank David D. Clark for generously sharing his knowledge and guiding this work and AT&T Bell Laboratories GRPW program for funding this research. The author also thanks David Clark, Sally Floyd, and the anonymous reviewers for their detailed and helpful criticisms and suggestions on earlier drafts of this paper. Finally, the author would like to thank Andrew Heybey and others who contributed to the implementation of the original Netsim simulator, Garrett Wollman and Nathan Yee for helping with the setup of the experiments over real networks, and Tim Shepard for many helpful discussions and technical assistance.

A Methodology

This section describes the simulations and the experiments conducted over real networks. We first discuss the simulator, Netsim, and the network topology and parameters used in the simulations. We briefly describe the setup for experiments over the real networks and give a short tutorial on how to read the sequence number versus time graphs. Finally, we evaluate the limitations of the simulations.

A.1 The Simulations Environment

A.1.1 The Simulator

The results presented in Section 2 are based on simulations in Netsim[6], a packet-by-packet, event-driven simulator. The simulator is based on the model that a computer network consists of an interconnected set of communication and switching components. The switching component in the simulator simulates a Drop Tail gateway. When the queues in the switching components overflow, the last packet that arrived at the queue is dropped.

We introduced a new TCP component to the existing Netsim simulator. To create this TCP component, we made minor modifications (which do not affect the behavior of TCP) to the actual BSD Net/2 code to conform with the simulator environment and requirements. We eliminated the 1/8 factor in the congestion window and acknowledge every packet as mentioned in [7].

A.1.2 Network Topology and Parameters

For all the simulations in this paper, we use a very simple topology shown in Figure 11. The buffer size of each switch is 10 packets. The segment size for transfer is 1024 bytes, and the maximum window size of the connection is 50 segments (51200 bytes). We transfer 1 Mbyte across a simple one-way TCP connection.

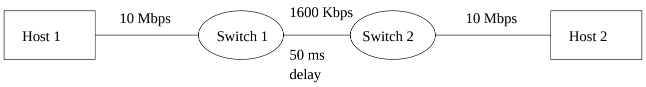

Figure 11: Network topology used in the simulations

The parameters chosen may limit the relevance of the results to real situations. For example, a more reasonable number for the buffer size parameter, may be the bandwidth-delay product of the bottleneck link ($\frac{bandwidth \times delay}{packet\ size\ used\ for\ transfer}$), which is 20, in our case. However, although this paper does look at some performance issues, the focus is on TCP's start-up transient *behavior*, i.e. how TCP reacts when congestion emerges. Since such behavior occurs over a wide range of parameters, as long as the parameters are within the range, the exact values of the parameters used are not of utmost importance. In most cases, varying the parameters within the range only varies the timing and duration of such behavior. A more detailed discussion on these issues is in [7]. Given that simulation has its limitations under certain circumstances, in our case, we show that behavior observed in simulations also occur in real networks.

A.2 Experiments over Real Networks

Figure 6, Figure 9, and Figure 10 are based on 1-Mbyte, one-way transfers from a east coast machine running Free BSD to a west coast machine running HP-UX. The TCP's at both machines implement the congestion control scheme that we discuss in Section 2. We ran *tcpdump* during the transfer and translate collected data into segment number versus time graphs.

A.3 Reading the Graphs

The simulation results are presented in two types of graphs: segment number versus time graphs and simple graphs tracing the congestion window parameter of a TCP connection. Both types of graphs show data collected from the sender's perspective. The graphic convention used in the segment number versus time graphs is similar to that developed in [16]. To make these graphs, we convert sequence numbers into segment numbers[6] to make the graphs more readable and the discussions simpler. We occasionally draw circles around the regions of interest in the graph.

In a segment number versus time graph such as the one in Figure 12, each small vertical line segment with arrows (in the leftmost circled region) at the ends represents a segment sent at that time. The length of a small vertical lines gives an indication of the size of the segment represented. There are two lines that bound the small vertical line segments: the bottom one indicates the last acknowledged segment number or *snd_una*, and the top one indicates *snd_wnd* plus *snd_una* at any particular time. A small tick mark on the bottom bounding line (in the rightmost circled region) indicates that a duplicate ACK has been received.

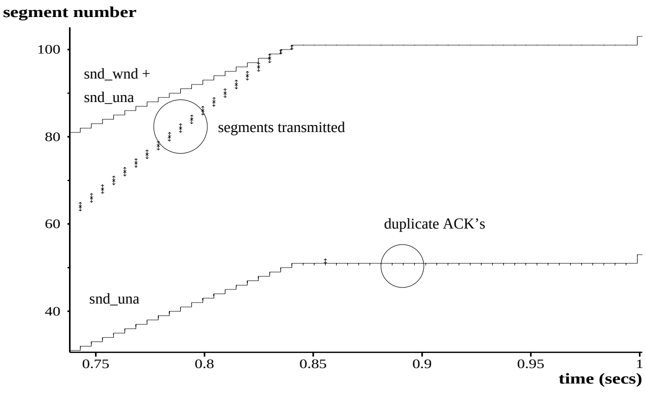

Figure 12: An example of a detailed segment number versus time graph

In Figure 13, we show another segment number versus time graph. The figure displays the same information for the entire 1-Mbyte transfer. Because of the lack of resolution, this figure does not show many details. However, we occasionally show a picture like this to display more general information about the transfer.

[6]We make the simplifying assumption that the segment number of a segment is the closest integer to the quotient of the highest sequence number in that segment divide by the maximum segment size, 1024 bytes.

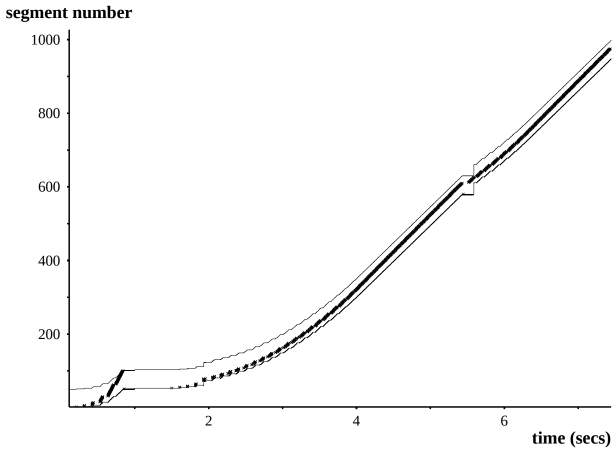

Figure 13: An example of a segment number versus time graph for an entire 1-Mbyte transfer

A congestion window graph such as the one in Figure 14 traces the size of the congestion window, *cwnd*, in bytes. The small crosses show changes in the value of *ssthresh*.

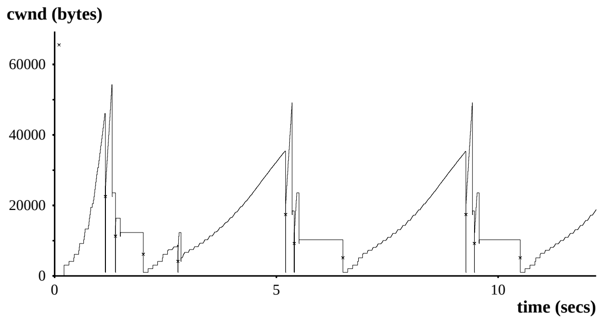

Figure 14: An example of a congestion window graph

References

[1] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: Emulation and experiment. In *Proceedings of the ACM SIGCOMM '95*, pages 185–195, August 1995.

[2] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM '94*, pages 24–35, August 1994. ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.ps.

[3] K. Fall and S. Floyd. Simulation-based comparisons of tahoe, reno, and sack TCP. ftp://ftp.ee.lbl.gov/papers/sacks.ps.z, May 1996. Submitted to CCR, July 1996.

[4] S. Floyd. TCP and successive fast retransmits. ftp://ftp.ee.lbl.gov/papers/fastretrans.ps, October 1994.

[5] Sally Floyd and Van Jacobson. On traffic phase effects in packet-switched gateways. ftp://ftp.ee.lbl/papers/phase.ps.Z.

[6] A. Heybey. The network simulator. Technical report, MIT, September 1990.

[7] J. C. Hoe. Start-up dynamics of TCP's congestion control and avoidance schemes, 1995.

[8] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM '88*, pages 314–329, August 1988.

[9] V. Jacobson. Modified TCP congestion avoidance algorithm. end2end-interest mailing list (Apr.), 1990.

[10] Van Jacobson. Berkeley TCP evolution from 4.3-tahoe to 4.3-reno. In *Proceedings of the Eighteenth Internet Engineering Task Force*, page 365, 1990.

[11] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of the ACM SIGCOMM '91*, pages 3–15, September 1991.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. ftp://ftp.ietf.cnri.reston.va.us/internet-drafts/draft-ietf-tcplw-sack-00.txt, April 1996. (Internet Draft, work in progress).

[13] M. Mathis and Jamshid Mahdavi. Forward acknowledgment: Refining TCP congestion control. In *Proceedings of the ACM SIGCOMM '88*, August 1996. To appear.

[14] J. Postel. Transmission control protocol. Request for Comments 793, DDN Network Information Center, SRI International, September 1981.

[15] S. Shenker and L. Zhang. Some observations on the dynamics of a congestion control algorithm. *ACM Computer Communication Review*, 20:30–39, October 1990.

[16] T. Shepard. TCP packet trace analysis. Technical Report 494, MIT Laboratory for Computer Science, February 1991.

[17] W. R. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley Publishing Company, 1994.

[18] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley Publishing Company, 1995.