

connections. For example, with our network with three connections, with a buffer size of 32 and a pipe size of 50 packets, the relative shares of throughput were 31%, 39%, and 30%.

This anomalous behavior can be removed by simply changing the congestion avoidance increase algorithm to read:

```
    cwnd += 1 / wnd
```

With this change, `cwnd` always increases by one in every epoch.

For those who are interested in experimenting with the above algorithm, the modified TCP code that we are currently exploring is described below. In real BSD TCP code, `cwnd` is in terms of *bytes*, instead of *packets*. The above modification requires a one-line change in `tcp_input()` procedure: replace

```
u_int incr = tp->t_maxseg;
if (tp->snd_cwnd > tp->snd_ssthresh)
    incr = MAX(incr * incr / tp->snd_cwnd, 1);
```

by

```
u_int incr = tp->t_maxseg;
if (tp->snd_cwnd > tp->snd_ssthresh)
    incr = MAX(incr / (tp->snd_cwnd / incr), 1);
```

References

- [1] R. Braden (editor). *Requirements for Internet hosts - communication layers*, RFC-1122, October 1989.
- [2] J. Davin and A. Heybey. *A Simulation Study of Fair Queueing and Policy Enforcement*, In this volume, 1990.
- [3] A. Demers, S. Keshav, and S. Shenker. *Analysis and Simulation of a Fair Queueing Algorithm*, In **Proceedings of SIGCOMM '89**, September 1989.
- [4] E. Hashem. *Analysis of Random Drop for Gateway Congestion Control*, In **Report LCS TR-465, Laboratory for Computer Science, Massachusetts Institute of Technology**, 1989.
- [5] V. Jacobson. *Congestion Avoidance and Control*. In **Proceedings of SIGCOMM '88**, August 1988.
- [6] V. Jacobson. *Berkeley TCP evolution from 4.3-tahoe to 4.3-reno*. In **Proceedings of the Eighteenth Internet Engineering Task Force**, Vancouver, British Columbia, August, 1990.
- [7] A. Mankin and K. Thompson. *Limiting Factors in the Performance of the Slow-Start TCP Algorithms*, In **Proceedings of USENIX Winter'89 Conference**, 1989.
- [8] A. Mankin. *Random Drop Congestion Control*, To appear in **Proceedings of SIGCOMM '90**, September 1990.
- [9] J. Nagle. *Congestion Control in TCP/IP Networks*, **ACM Computer Communications Review**, 14(4), October, 1984.
- [10] J. Postel. *DoD Standard Transmission Control Protocol*. Network Information Center RFC-793, SRI International, September 1981.
- [11] L. Zhang. *A New Architecture for Packet Switching Network Protocols*, In **Technical Report TR-455, Laboratory for Computer Science, Massachusetts Institute of Technology**, 1989.
- [12] L. Zhang and D. Clark. *Oscillating Behavior of Network Traffic*, in preparation, 1990.

in settings far more general than those investigated in this paper. Rather, these modifications are presented with the intent of illuminating the root causes of the phenomena we observed and how these phenomena are effected by various algorithms.

Clearly the most straightforward way to prevent the separation of packets is not to allow the sender TCP to send back-to-back packets. We have experimented with several algorithms in which the sender introduces delays between the packets of approximately $\frac{rtt}{wnd}$. Pacing out the packets in this manner will create some mixing, but each connection still loses a single packet in each congestion epoch.

Another possible modification to TCP is the change being contemplated for the forthcoming 4.3-Reno BSD release[6]. Here, the window adjustment algorithm has been modified so that, upon a single packet loss, each connection maintains at least $cwnd/2$ outstanding packets.¹⁰ In this case, even though the packets are still separated, and each connection loses a single packet in each congestion epoch, one can expect less or even no bandwidth being wasted because the total network load does not decrease as dramatically as before.

We can also contemplate changing the switch algorithm. One possible way to prevent every connection from losing a packet during a congestion epoch is to implement some form of preemptive dropping, whereby packets are dropped from the queue even before the buffer is full. Conceivably, one could construct such an algorithm in which only one of the connections has a packet dropped in each congestion epoch. An example of such an algorithm might be to simply drop a single packet (which is randomly chosen from the packets in the queue) whenever the queue size passes some threshold. While this might eliminate synchronized packet losses, the packets from different connections will still not be mixed.

The mixing of packets can also be done directly at the switch itself. The Fair Queueing switch algorithm, which originated from a suggestion by Nagle [9] and is described and analyzed in [3], is roughly equivalent to giving round-robin service to the packets from the various connections (see also [2] for further simulations of this algorithm). Thus, regardless of the order in which the packets arrived at the queue, they will leave fully mixed. Furthermore, the fact that the various connections are somewhat decoupled suggests that the connections need not all lose packets in the same epoch.

We are currently in the process of experimenting with algorithms like those suggested above, and will report

¹⁰A simple-minded approach of reducing $cwnd$ to half would not work properly. For details see [6].

on them in a future publication¹¹. It is important to note that the phenomena observed in this paper are specific to the simple network model considered here. We have only investigated single bottleneck networks with all traffic having the same latencies and flowing in the same direction. As such, we have neglected the effects of random processing times, different round-trip times for the various connections, and cross traffic. In particular, the dynamics are much more complicated once two-way traffic is introduced since our assumption about the ACK packets never queueing is no longer valid.

Acknowledgments

We would like to thank Van Jacobson, Sally Floyd, and Dan Swinehart for their extensive comments on an early draft. We would also like to thank Abhijit Khale for his help in producing the graphs used in this paper.

Appendix

In the body of this paper we concluded that, since we assumed connection increases wnd by one every epoch, every connection loses exactly one packet every congestion epoch. However, if we look at the 4.3-Tahoe TCP code, we see that this assumption is not valid. Define $cwnd(i)$ to be the value of $cwnd$ after the i 'th packet has been acknowledged. In the congestion avoidance regime, $cwnd(i) = cwnd(i-1) + 1/cwnd(i-1)$. The increase in $cwnd$ in an epoch starting with the k 'th packet, call it $\Delta cwnd$, can be calculated as $\Delta cwnd = \sum_{i=1}^{wnd} \frac{1}{cwnd(k-1+i)}$. This sum is always less than one, so wnd need not increase by one on every epoch; wnd increases only if the fractional part of $cwnd$ plus $\Delta cwnd$ is greater than one. The closer to one $\Delta cwnd$ is, the more likely the increase will happen. Note that asymptotically as $cwnd \rightarrow \infty$, $\Delta cwnd \rightarrow 1 - \frac{1}{2cwnd}$.

If, during a congestion epoch, a connection happens to be in a state where wnd does not increase, then that connection will not lose a packet in that epoch. While the other connections will reset their windows to one, this lucky connection will continue to increase its window thereby getting more of the bandwidth. This pattern of a particular connection avoiding a drop in a congestion epoch can be repeated cyclically, leading to an unfairness in the long-term throughput of the

¹¹Sally Floyd is also experimenting with preemptive random drop algorithms.

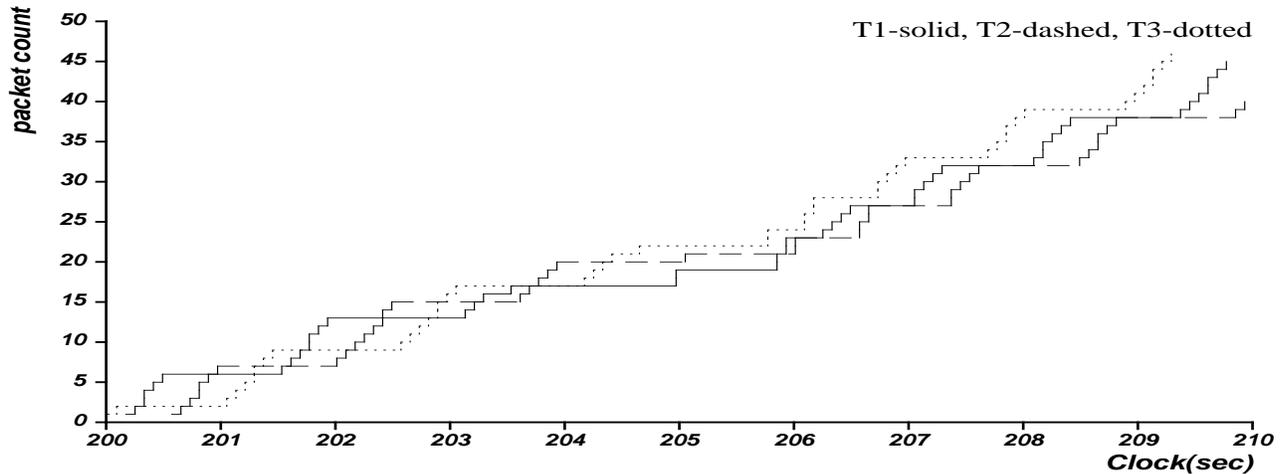


Figure 6: The number of packet generated vs. time for each of the three connections, with $\tau = 0.01$ sec. The flat regions indicate the complete separation of packets.

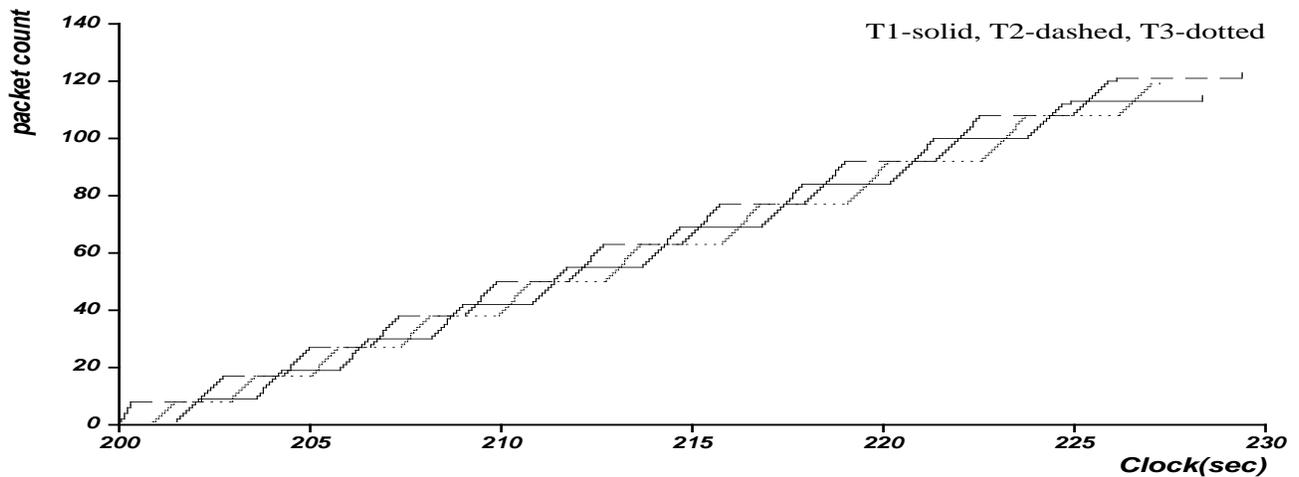


Figure 7: The number of packet generated vs. time for each of the three connections, with $\tau = 1$ sec.

atively large pipes. If only one connection lost a packet during a congestion epoch, then the total window size would not vary so suddenly. Note that we have yet to determine how general this phenomena of synchronized packet losses is. While we can identify topologies where this synchronization can be less strong, such as when the round-trip times of the various connections are dramatically different, we do not know if synchronization of packet losses is common in today's Internet.

The complete separation of the packets, while not particularly a problem in this simple topology, could possibly cause a problem in a more complicated network. Consider the case where the three connections, rather than terminating at the same host, had different paths. Then, the switches on these paths would see a very bursty traffic pattern from these connections: a period with no packets transmitted followed by a period of

packets coming at the rate of the shared line. This very bursty traffic source might harm the connections sharing lines with it. However, we do not know if the packet separation phenomenon will arise in these more complicated networks. We would urge others to look for the presence of this separation effect, either in their simulation experiments or in real networks. If packet separation is a widespread effect, then modifying the protocol to reduce its occurrence may be worthwhile.

We conclude this paper by briefly discussing several algorithmic modifications which will effect these phenomena. The first class of modifications can be implemented in the TCP connection itself, while the second class involve changing the switch queue control algorithm. These modifications are not suggested a general improvements to congestion control; such a proposal would require an investigation of their behavior

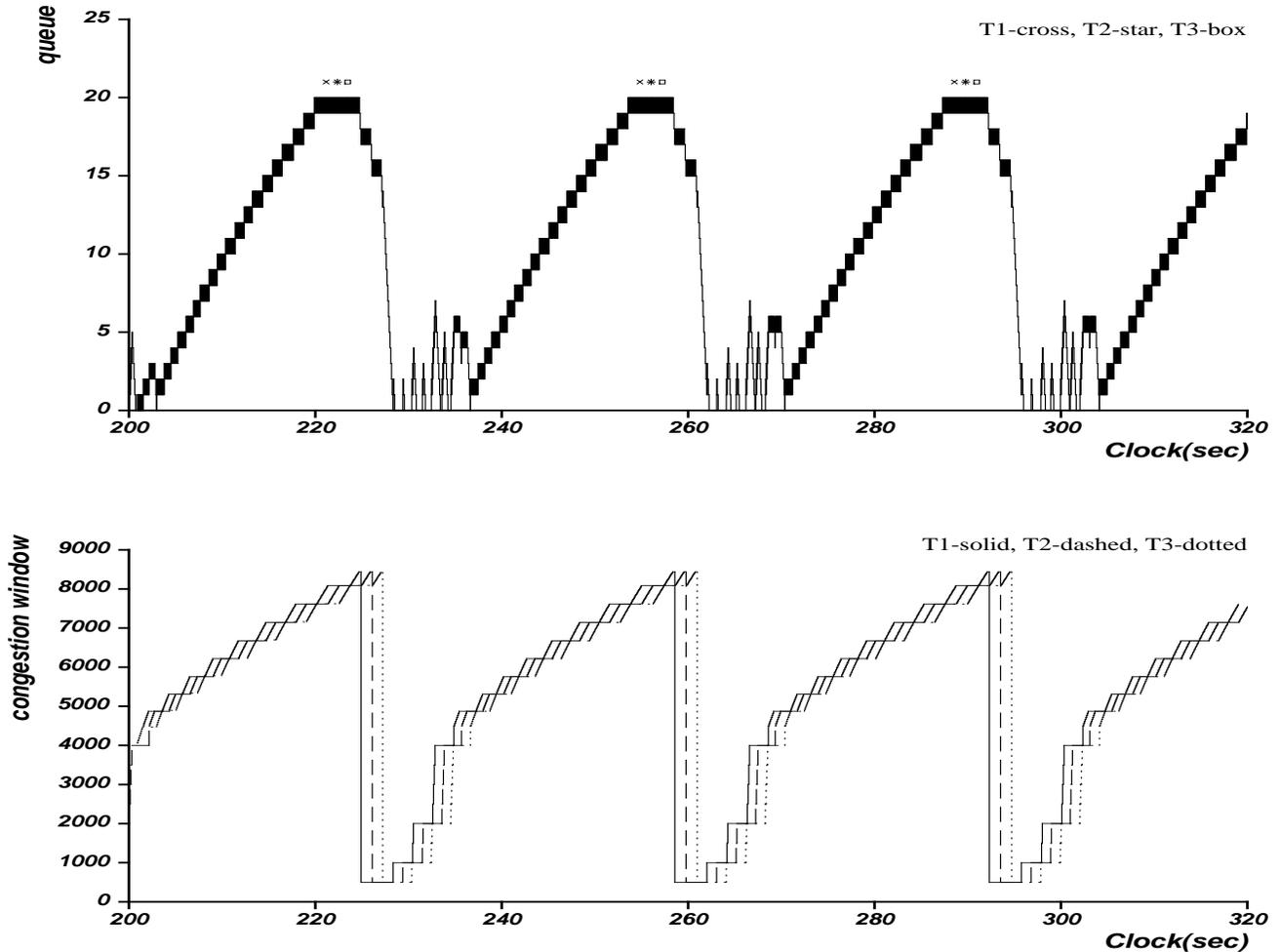


Figure 5: Packet queue at the switch and the congestion window sizes of the three connections, with $\tau = 1$ sec. In this case the network capacity of 45 is a multiple of the number of connections, so all connections have the same value of wnd at a packet drop. The order of packet losses is the same for all congestion epochs.

show the *cumulative packets sent vs. time* for the three connections, with propagation delays τ of 0.01 sec and 1 sec, respectively. Notice that each connection sends out a full window's worth of packets in one burst and then waits until the next epoch. This separation occurs for two reasons. First, whenever the window is increased, the extra packet is sent immediately following another packet. When they arrive at the switch, these two packets are adjacent in the queue. Second, these adjacent packets will, through their acknowledgments, always generate pairs of adjacent packets in future epochs;⁹ no packet from another connection will ever come between them. This is because all packets, except for retransmissions, are generated in response to acknowledgments. Retransmissions are not an issue here because they always occur when all of the connections have $wnd = 1$. Therefore, in the process of building the window up from 1 after a packet loss, a

⁹Of course, if wnd increases as a result of one of the ACK's, there will be additional adjacent packets in the cluster.

connection creates a monolithic clump of packets that are always back-to-back in the queue. There are no mechanisms present in our network that can break up this separated structure.

4 Implications and Modifications

The previous section described two dynamic phenomena that were contrary to our naive expectations. These phenomena, while of intellectual interest, also have practical implications for the functioning of the network. The fact that each connection loses a packet during the same congestion epoch means that all of the connections are decreasing their congestion windows at the same time. Because the window adjustment algorithm resets the window to one, the sum of the window sizes after a congestion epoch may be too small to fully utilize the bandwidth of the line when the network has rel-

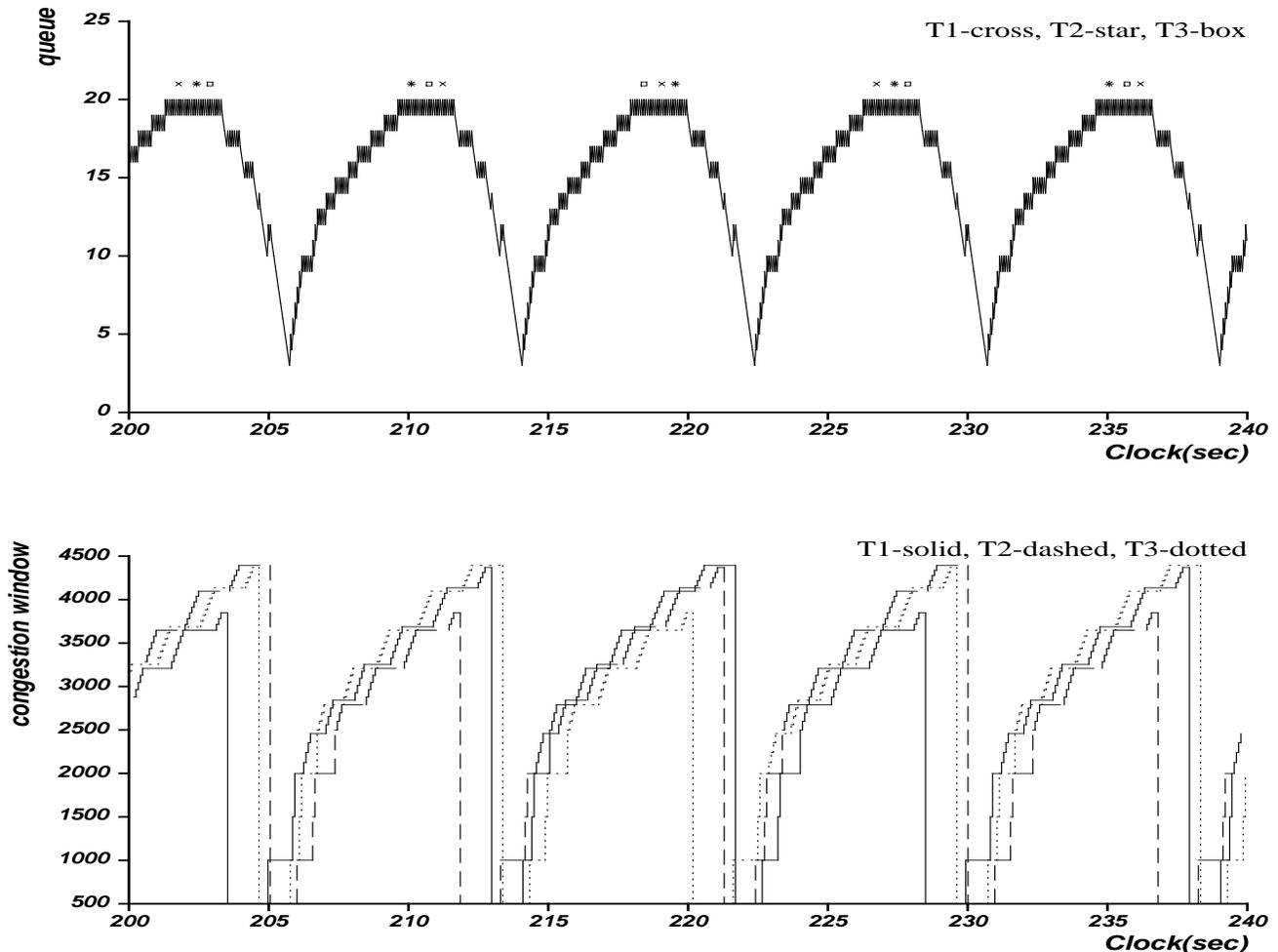


Figure 4: Packet queue at the switch and the congestion window sizes of the three connections, with $\tau=0.01\text{sec}$. The marks above the graphs of the queue length indicate when packets from the various connections are dropped by the switch. Note that the period of oscillation is slightly more than a third of what it was in the single connection case. Furthermore, observe that the connections do not all have the same value of wnd at the time their packet is dropped; in each congestion epoch, the first connection to have a packet dropped has $wnd = 7$ while the other two connections have $wnd = 8$. This is because the capacity of the path, 20 in this case, is not a multiple of the number of connections. However, since the role of having the first packet drop rotates among the connections, the long-term bandwidth allocations are equal.

nection loses exactly one packet.⁸ To understand this, consider the epoch in which the sum of the windows is equal to the capacity. In response to each ACK, the connections send out a new packet and also increase $cwnd$ by $1/cwnd$. wnd will not increase until $cwnd$ has passed the next integer. Until then the connections will be clocking their packets with their ACK's, keeping the path completely full. Each incoming ACK is a signal that a data packet has left the path, and the sender then responds by filling that temporary hole with another data packet. At the point that wnd increases, the connection will send out two packets back-to-back. There is no room in the path for this second packet;

⁸There are rare cases where some connections don't lose a packet, which we discuss in the Appendix.

when it reaches the switch, the buffer will be full and the packet dropped. Since the path becomes full for all the connections at the same time, and every connection will have increased $cwnd$ by roughly 1 during a single round-trip time, they will all have a single packet dropped during this epoch. This is consistent with our earlier analysis using *acceleration*. The acceleration of the total traffic is N_c , the number of connections, since each connection increases $cwnd$ by one during each epoch. Thus, we would expect to see N_c packet drops during the congestion epoch.

Our second assumption, that the packets from the different connections are mixed, is also wrong. Instead, the packets are completely *separated*. Figures 6 and 7

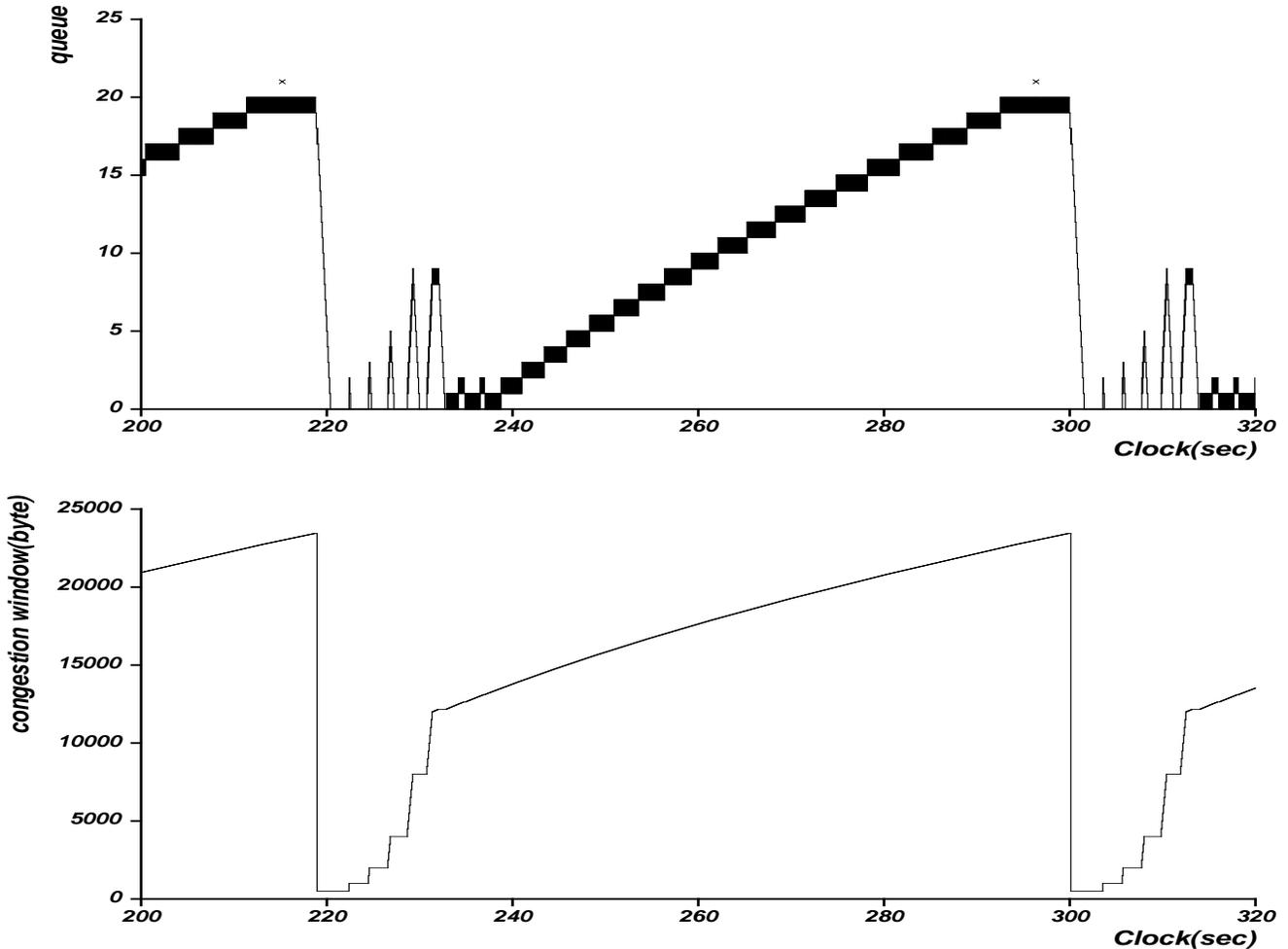


Figure 3: Queue and congestion window for one connection, $\tau = 1$ sec. The black regions in the graph of *queue* vs. *time* are due to rapid queue oscillations between two different values as packets arrive and depart. The structure visible in the early part of the oscillatory cycles reflect the clumps of packets arriving at the switch, which create a temporary queue. Once the pipe is full, the queue then gradually increases.

exponential growth regime is not present due to the small pipe size. Figure 3 shows all three regimes, but the linear growth regime is extremely narrow.

Notice that increasing the propagation delay τ (going from Figure 2 to Figure 3) increases the period in which the queue is empty, during which the line is underutilized. Most of this underutilization occurs during the slow-start phase of the congestion control algorithm, where the congestion window has been reset to 1 and then doubled in each subsequent epoch. The line becomes fully utilized whenever the congestion window reaches twice the pipe size. Assuming that $ssthresh > 2P$, it will take roughly a time period of $2\tau \log_2 2P$ for *cwnd* to achieve $2P$.⁷ The first term is the round-trip time of the packets when the queue is empty, and the second term is the number of epochs

⁷In our network of one connection with $\tau = 1$ sec, *ssthresh* is only slightly smaller than $2P$.

needed to achieve the desired window size.

3.2 Three Connections

In our network, the three connections have the same epoch period because they share the same communication path. They enter a congestion epoch whenever the total window size, the sum of the three connection's values for *wnd*, reaches the capacity of the path. One might initially expect that, as with the single connection case, there is a single packet loss per congestion epoch. We might also expect that the packets from different sources are mixed together in the queue. However, as is depicted in Figures 4 through 7, neither of these expectations are valid.

First, note that in each congestion epoch *every* con-

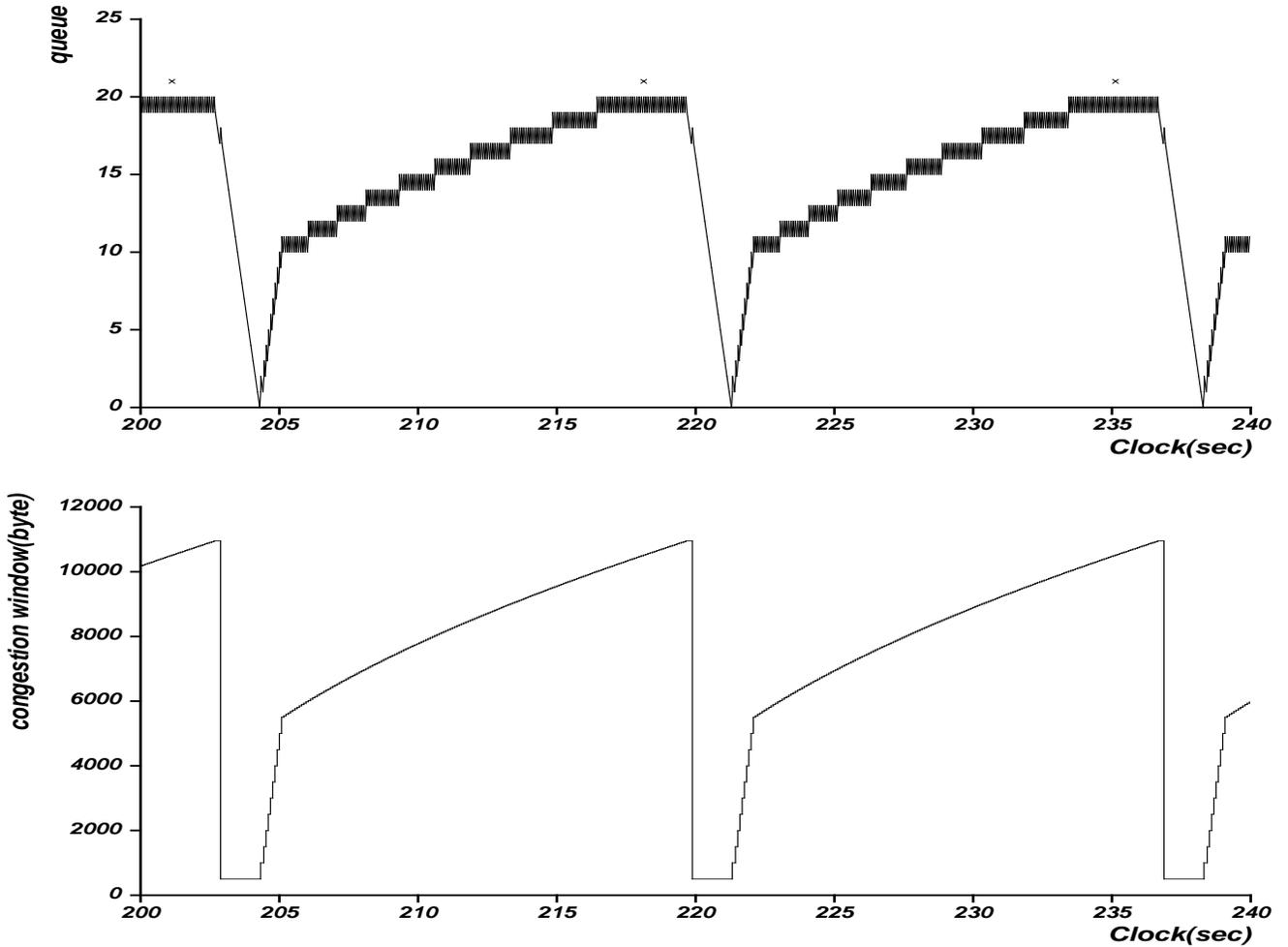


Figure 2: Packet queue at the switch and the congestion window of one TCP connection, $\tau = 0.01$ sec. The x's above the graph of *queue* vs. *time* mark when packets are dropped at the switch. The graph of *cwnd* vs. *time* show the transition from a linear increase to a square root increase. The exponential growth regime in the early part of the oscillation is not present due to the small pipe size.

a queue is starting to form, but the congestion control algorithm is still in the slow-start phase. Here, the growth rate of the congestion window is linear: $cwnd \sim t \frac{\mu}{M}$. If $ssthresh$ is smaller than twice the pipe, then the intermediate regime $ssthresh < cwnd < 2P$ also has a linear growth rate but with a different constant: $cwnd \sim \frac{t}{2\tau}$. Notice that while both intermediate regimes give rise to linear growth, the reasons are quite different. In the first case, the ACK's arrive at the maximal rate (the rate at which the switch can transmit data packets) and $cwnd$ increases by one for each ACK. In the second case the epochs are separated by the *rtt* of 2τ and $cwnd$ increases by roughly one every epoch.

3. For large windows $cwnd > MAX(2P, ssthresh)$, the window grows asymptotically as the square root of time: $cwnd \sim \sqrt{t \frac{2\mu}{M}}$. In this regime the

ACK's are arriving at the maximal rate, and the increase in $cwnd$ per each ACK decreases as $cwnd$ grows.

Packet drops occur when the window size exceeds the capacity of the path: $wnd = C + 1$. Between this extra packet being sent and its drop being detected by the sender, C packets are acknowledged. The value of $cwnd$ at the time the drop is detected is roughly $C + 2 - \frac{1}{C+2}$. Thus, we expect that at steady state $ssthresh = \lfloor 1 + \frac{C}{2} - \frac{1}{2(C+2)} \rfloor$. Using these expressions as rough guidelines, the changeover from exponential to linear growth should occur at $cwnd$ values of approximately 0.25 and 23 for τ values of 0.01 and 1, respectively. Similarly, the changeover from linear to square root growth should occur at $cwnd$ values of approximately 10 and 25, respectively. Figure 2 shows clearly the transition from linear to square root growth, but the

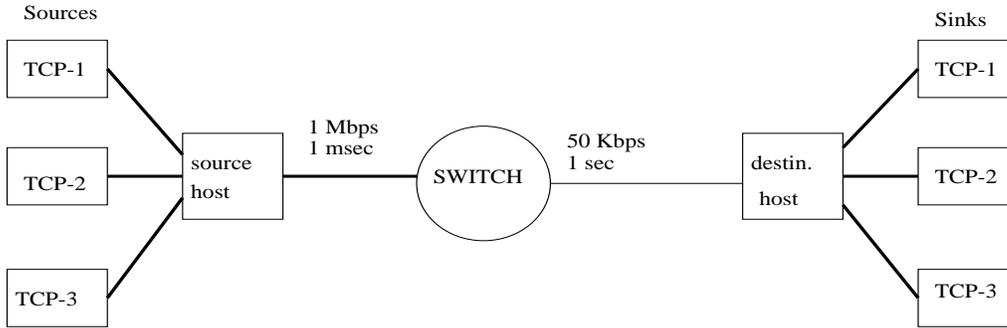


Figure 1: Network topology model.

3.1 Single Connection

While this case is relatively simple and reveals no surprises, it does help us gain a better understanding of the algorithm. Figures 2 and 3 show graphs of *queue length* and *cwnd* vs. *time* for the two different values of propagation delay τ . The graphs exhibit the oscillations common to feedback loops with binary feedback. After each congestion epoch, *cwnd* is decreased, reducing the load on the switch and causing the queue to drop. During the congestion recovery and avoidance phases *cwnd* increases, leading to an increasing queue, until the buffer becomes full and another packet is dropped. This pattern repeats itself indefinitely; the figures contain only one or a few of these oscillation periods to allow the reader see the details of the queue changes clearly.

The congestion epoch occurs when *wnd* has reached the *capacity* C of the path. The capacity of a path is the maximum possible number of outstanding packets (i.e. packets that have been sent but whose ACK's have yet to be received by the sender), assuming no packets are dropped. In our network each outstanding packet must either (1) be in the switch's queue, (2) be on the bottleneck transmission line, or (3) have its associated ACK packet on the bottleneck transmission line in the other direction.⁶ Thus for our network the capacity of the path, when measured in units of maximum sized packets, is merely the sum of the buffer size B plus twice the *pipe* size P , where P is the bandwidth-delay product $\frac{\tau\mu}{M}$ of the transmission line: $C = \lfloor B + 2\frac{\tau\mu}{M} \rfloor$. This calculation depends crucially on the observation that, in our network, the ACK packets never encounter a queue on their way to the sender and arrive at the sender with a minimum spacing equal to the transmission time of a data packet at the bottleneck link.

In each congestion epoch a single packet is dropped. This is because the acceleration α of the congestion

⁶We are ignoring the transmission time on the link between the source host and the switch, and the processing time at the receiving end.

control algorithm in the congestion avoidance phase is one. In the epoch immediately preceding the congestion epoch, *wnd* was exactly equal to the capacity C . Any further increases in the window size in the next epoch result in dropped packets, and the number of packets dropped is determined by the acceleration α of the congestion control algorithm.

The graphs of *cwnd* vs. *time* exhibit several facets of the window adjustment algorithm. We can get an approximate expression for the window size as a function of time by modeling the adjustments as a continuous process and using the differential equation

$$\frac{d\ cwnd}{d\ t} = \frac{d\ cwnd}{d\ ack} \frac{d\ ack}{d\ t}$$

where

$$\frac{d\ ack}{d\ t} \approx \frac{cwnd}{rtt}$$

and *rtt* is the average round-trip time of the packets.

The round-trip time is easily approximated. When $cwnd < 2P$ there is no long-lived packet queue at the switch, so $rtt \approx 2\tau$ reflecting the propagation delays for both the data and ACK packets. When $cwnd > 2P$, not all of the packets will fit in the forward and return pipes and an average queue of size $(cwnd - 2P)$ will develop. Thus, the average round-trip time is $rtt \approx 2\tau + (cwnd - 2P)\frac{M}{\mu} = \frac{M\ cwnd}{\mu}$.

An approximation to the term $\frac{d\ cwnd}{d\ ack}$ follows directly from the window adjustment algorithm. When $cwnd < ssthresh$, $\frac{d\ cwnd}{d\ ack} \approx 1$ and when $cwnd > ssthresh$, $\frac{d\ cwnd}{d\ ack} \approx \frac{1}{cwnd}$. Thus, there are three possible regimes of behavior.

1. For small window sizes $cwnd < MIN(2P, ssthresh)$, when the switch's buffer is usually empty and the congestion control algorithm is in the slow-start phase, the growth rate of the window is exponential: $\log\ cwnd \sim \frac{t}{2\tau}$.
2. If *ssthresh* is bigger than twice the pipe, then in the intermediate regime $2P < cwnd < ssthresh$

receiver advertised window $maxwnd$ regardless of the load in the network. In the 4.3-Tahoe BSD TCP algorithm, the window size used by the sender is adjusted in response to network congestion. The sender has a variable called the congestion window $cwnd$, which is increased whenever new data is acknowledged and is decreased whenever a packet drop is detected.² The actual window used by the sender is the floor of the minimum of the congestion window and the receiver advertised window:³

$$wnd = \lfloor MIN(cwnd, maxwnd) \rfloor.$$

The congestion window adjustment algorithm has two phases, the slow-start or *congestion recovery* phase, where the window is increased rapidly, and the *congestion avoidance* phase, where the window is increased much more slowly. Whether a connection is in one phase or the other is determined by a control threshold, $ssthresh$. Whenever a packet drop is detected, $ssthresh$ is set to half of the current $cwnd$ value, $cwnd$ is then set to one, and the congestion recovery phase begins. $cwnd$ increases rapidly until it passes the threshold $ssthresh$, then the algorithm switches into the congestion avoidance phase. The specifics of the adjustment algorithm are as follows.

When new data is acknowledged, the sender does

```

if (cwnd < ssthresh)
    cwnd += 1;
else
    cwnd += 1 / cwnd

```

When a packet drop is detected, the sender does

```

ssthresh = cwnd/2
cwnd = 1

```

We define an *epoch* of a TCP connection to be the time period during which an entire window's worth of packets have been acknowledged. We will focus particularly on those epochs in which packet losses occur. These will be called *congestion epochs*.

The amount by which the congestion window increases during an epoch, which we will call the acceleration α , is an important measure of how rapidly the window size is changing. Notice that when $cwnd < ssthresh$, $cwnd$ doubles during an epoch, so $\alpha \approx cwnd$. In contrast, when $cwnd > ssthresh$, $cwnd$ increases by approximately 1 during an epoch: $\alpha \approx 1$.

²Packet drops are detected by either the receipt of duplicate acknowledgments or the expiration of a timer.

³Since TCP transmits maximum size packets whenever possible to avoid the silly-window syndrome, wnd will always be an integer and is the maximum number of outstanding packets allowed.

2.2 Network Topology

We will study a simple network topology consisting of a single switch with a 20 packet buffer connecting two hosts.⁴ There are N_c TCP connections, all transmitting from the same source host to the same destination host, as in Figure 1. The bottleneck transmission line between the switch and the destination host has a bandwidth μ of 50 Kbps, and a propagation delay τ . The transmission line between the source host and the switch has a bandwidth of 1 Mbps and a propagation delay of 1 msec. The two parameters we will vary in this study will be the number of connections N_c and the propagation delay τ . First we will consider a single TCP connection, and then three TCP connections.⁵ The propagation delay τ will take on values of .01 sec and 1 sec.

Each TCP connection is assumed to have a maximum window size of 50 packets, with a constant packet size M of 500 bytes. The returning ACK packets are 50 bytes each. Thus, the propagation delays of 0.01 sec and 1 sec represent the transmission times of 0.125 and 12.5 packets, respectively. For our simple network topology the value of $cwnd$ never exceeds 50, so that the maximum window size will not be a factor in any of our simulations. We also assume that each TCP connection always has data to send and the packet flow is controlled by the congestion window only.

3 Results

We first describe the results of simulations with a single connection, and then compare this with a simulation with three connections. We concentrate on the steady-state behavior of the algorithm so the initial start-up transients are omitted from the data. When there are multiple connections, the connections are established at random times.

All of the simulations reported on here were done with a simulator written by one of us (LZ). The TCP code was taken directly from the 4.3-Tahoe BSD release and modified slightly to conform to the requirements of the simulator. In addition, the code related to TCP connection set-up, keep-alive, and close was removed.

⁴When the buffer is full and a new packet arrives, the last packet in the buffer is dropped and replaced by the arriving packet.

⁵The results for more connections are similar to those with three connections, as long as the number of connections is much smaller than the number of buffers.

Some Observations on the Dynamics of a Congestion Control Algorithm

Scott Shenker and Lixia Zhang
XEROX Palo Alto Research Center

David D. Clark
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

We use simulation to make some observations about the behavior of the congestion control algorithm currently embedded in the 4.3-Tahoe BSD TCP implementation. We investigate a simple case of a few TCP connections, originating and terminating at the same pair of hosts, using a single bottleneck link. Our simulations reveal two unexpected phenomena. First, packets from the individual connections, rather than being mixed together, completely separate into individual clusters. Second, every connection loses a single packet during each congestion epoch. As a way of exploring the cause of these phenomena, we discuss how the behavior is altered by modifications to the congestion control algorithm and to the switch queue control algorithm.

1 Introduction

The congestion control algorithm currently embedded in the 4.3-Tahoe BSD TCP implementation, which was developed by Van Jacobson and is described in [5], has had a tremendous impact on congestion in the Internet. Furthermore, it is now considered the standard Internet flow control algorithm, as spelled out in [1]. For these reasons it is important to understand the behavior of this algorithm. Unfortunately, besides the original paper [5], there have been few detailed performance studies of this algorithm (see [7]; other related work can be found in [8, 3, 11, 4]). The present paper represents another contribution, albeit a small one, to the cause. We hope that increased understanding of this congestion control algorithm can both lead to a better understanding of the behavior of today's Internet and also provide some guidance for the design of future congestion control algorithms.

We use simulation to observe the dynamics of this algorithm in the highly specialized situation of a few TCP connections, originating and terminating at the same pair of hosts, sharing a single bottleneck link. Thus,

the present work is most definitely not a comprehensive study, and the relevance of our results to more general settings is yet to be determined. However, the simulations do highlight some phenomena that surprised us.

In the next section we briefly describe the congestion control algorithm and network used in our simulation. The results of these simulations are presented in Section 3. In Section 4 we discuss possible implications of these results for network performance and also identify some possible modifications to the congestion control algorithm and switch queue control algorithm which would alter our results.

2 The Network Model: Algorithms and Topology

In this section we first provide a quick overview of the 4.3-Tahoe BSD TCP congestion control algorithm and then describe the topology of the network being considered.

2.1 The 4.3-Tahoe BSD TCP Congestion Control Algorithm

The following is a very abbreviated and oversimplified description of the 4.3-Tahoe BSD TCP congestion control algorithm. For further details, see either [5] or the 4.3-Tahoe BSD code itself (which has sufficient comments to render it a useful text). At TCP connection set-up the receiver specifies a maximum window size *maxwnd*.¹ To simplify the presentation in this paper, we will assume that all window sizes are measured in units of maximum size *packets*, instead of bytes. In the original TCP specification [10], the window used by the sender, which we will denote by *wnd*, is the

¹The variable names used here are not the same as in the 4.3-Tahoe BSD code.