Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic

Lixia Zhang[1], Scott Shenker[1], David D. Clark[2]

Abstract

We use simulation to study the dynamics of the congestion control algorithm embedded in the BSD 4.3-Tahoe TCP implementation. We investigate the simple case of a few TCP connections, originating and terminating at the same pair of hosts, using a single bottleneck link. This work is an extension of our earlier work ([16]), where one-way traffic (i.e., all of the sources are on the same host and all of the destinations are on the other host) was studied. In this paper we investigate the dynamics that results from two-way traffic (in which there are data sources on both hosts). We find that the one-way traffic clustering and loss-synchronization phenomena discussed in [16] persist in this new situation, albeit in a slightly modified form. In addition, there are two new phenomena not present in the earlier study: (1) ACK-compression, which is due to the interaction of data and ACK packets and gives rise to rapid fluctuations in queue length, and (2) an out-of-phase queue-synchronization mode, which keeps link utilization less than optimal even in the limit of very large buffers. These phenomena are helpful in understanding results from an earlier study of network oscillations ([19]).

1 Introduction

One of the longstanding problems with datagram networks is that it is difficult to control congestion. However, in the past decade, tremendous progress has been made on this problem. One particularly noteworthy success is the congestion control algorithm developed by Jacobson ([6]), which is currently embedded in the BSD 4.3-Tahoe TCP implementation and is similar in spirit to the one Jain, Ramakrishnan, and Chiu ([8]) designed for the DECnet architecture. Jacobson's congestion control algorithm has resulted in a dramatic reduction in congestion in the Internet[3] and has become an Internet standard ([1]). Thus, it is important to understand the behavior of this algorithm. We hope that increased insight into this particular algorithm can both lead to better understanding of the behavior of the current Internet and provide guidance for the design of new congestion control algorithms.

While the practical benefits of this congestion control algorithm are clear, its behavior is not yet fully understood. There is a small, but rapidly growing, set of simulation studies of this algorithm; for example, see [2, 3, 4, 5, 10, 16, 18, 19]. In several of these studies the focus is on the effect of various gateway disciplines such as Fair Queueing ([2, 3]) and Random Drop ([4, 5, 10, 18]) on network performance in the presence of traffic sources using the

[1]Palo Alto Research Center, Xerox Corporation

[2]Laboratory for Computer Science,Massachusetts Institute of Technology

[3]While we are not aware of anyone who disputes this statement, the evidence for improved overall Internet performance due to this congestion control algorithm is mostly anecdotal.

BSD 4.3-Tahoe TCP congestion control algorithm. Several other studies ([5, 18]) present aggregate throughput, loss, and delay characteristics of this congestion control algorithm in various complicated network configurations.

In contrast, in this paper we examine only a few simple network configurations with standard FIFO gateways. Our focus is instead on the detailed dynamics of the congestion control algorithm in rather simple settings. Aside from the seminal work of Jacobson [6], these dynamics have received little attention. Some aspects of these dynamics were studied in [19], where simulations on two different network configurations revealed the presence of rapid queue length fluctuations. While preliminary explanations were offered, the complexity of the network configurations precluded a systematic analysis. One of the purposes of this paper is to reproduce the essential elements of that fluctuating behavior in the simplest possible network configuration so that it can be more fully analyzed.

This paper is a continuation of the research effort initiated in [16]. There we examined the dynamics of the BSD 4.3-Tahoe TCP congestion control algorithm in the simplest of network configurations: one or several TCP connections, originating and terminating at the same pair of hosts, sending traffic through a single bottleneck link, with all of the connections transmitting in the same direction (i.e., with all of the sources of the connections on one host and all of the destinations of the connections on the other host). Since all of the data packets travel in one direction and all of the ACK packets travel in the other, the dynamics are relatively tractable in this configuration. In the present paper, we retain the same network topology of a single bottleneck link but progress to the slightly more complicated situation of having one source on each host, so that both data and ACK packets travel in each direction. The seemingly innocuous modification of introducing two-way traffic greatly complicates the dynamics. More surprisingly, it appears that all of the essential elements of the behavior reported in [19] are present in this simple configuration.

In the one-way traffic configuration, there are two notable aspects of the behavior: the clustering of packets from each connection and the synchronization of packet losses (these effects will be described in more detail later). Two-way traffic exhibits similar phenomena. However, there are two dynamic phenomena that are singular to two-way traffic. The first, labeled ACK-compression, gives rise to rapid fluctuations in the queue length at the bottleneck gateway. In contrast to one-way traffic, where the ACK's provide a reliable clock to regulate traffic and keep the queue length variations modest, in two-way traffic the ACK's can become compressed together and hence lose their clocking properties. The second phenomenon is an out-of-phase synchronization mode. In the one-way traffic configuration, all of the connections are synchronized in-phase in that the flow control windows of the various connections all increase and decrease at the same time. With two-way traffic, under certain conditions the connections in different directions are synchronized out-of-phase in that the flow control window of one connection is increasing while that of the other is decreasing. This phenomenon has the effect of keeping the bottleneck utilization below optimal, even in the limit of infinite buffers.

Our detailed analysis is only applicable to the single specific network configuration we consider. However, the basic phenomena of ACK-compression and synchronization modes seem to be present in the dynamics of much more complicated configurations. Similarly, while the present study is limited to investigating the behavior of one specific congestion control algorithm, we think that the effects described above apply to a wider class of algorithms. In fact, we conjecture that any nonpaced[4] window-based congestion control algorithm will exhibit these two phenomena.

We hasten to note, however, that our study is rather incomplete. The BSD 4.3-Tahoe TCP congestion control algorithm gives rise to a wealth of complicated dynamical behavior; we have only examined the most accessible of these. Even in the extremely simple configurations examined here, there are effects that we do not yet fully understand. More importantly, for those behaviors which we do understand, we have yet to determine how relevant the insight gained from examining relatively simple network topologies is to more complicated and realistic network configurations. This is the subject of future work.

This paper has 6 sections. In the next section, we briefly describe the BSD 4.3-Tahoe TCP congestion control algorithm and discuss our network model and simulator. To provide context for the two-way traffic simulations presented here, in Section 3 we review the one-way traffic results from [16] and the rapid queue length fluctuation results from [19]. In Section 4 we examine the novel behavioral aspects of two-way traffic, focusing on ACK-compression and synchronization modes. The relationship between this data and that in [19] and the effect of various other factors, such as the delayed-ACK option and other network topologies, are discussed in Section 5. We summarize our results in Section 6.

2 Network: Algorithm, Configuration, and Simulator

In this section we first give a brief overview of the BSD 4.3-Tahoe TCP congestion control algorithm, then discuss the network configurations considered, and lastly describe the network simulator used.

2.1 BSD 4.3-Tahoe TCP Congestion Control Algorithm

The following is a very abbreviated and oversimplified description of the BSD 4.3-Tahoe TCP congestion control algorithm. For further details, see either [6] or the BSD 4.3-Tahoe code itself (which has sufficient comments to render it a useful text). At TCP connection set-up the receiver specifies a maximum window size $maxwnd$.[5] 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 ([14]), the window used by the sender, which we will denote by wnd, is the receiver advertised window $maxwnd$ regardless of the load in the network. In the BSD 4.3-Tahoe 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

[4] Pacing will be discussed later, but for now it suffices to define a nonpaced algorithm as one in which the source sends data packets immediately upon the receipt of an ACK, without introducing any artificial delay which would spread out packets.

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

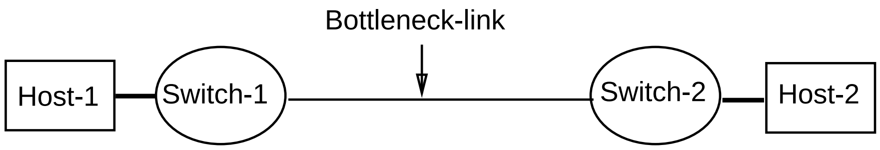

Figure 1: Network Topology.

decreased whenever a packet loss is detected.[6] The actual window used by the sender is the floor of the minimum of the congestion window and the receiver advertised window:[7]

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

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. A control threshold, $ssthresh$, determines which phase a connection is in. 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 in the real BSD 4.3-Tahoe TCP code 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 = MAX[ MIN(cwnd/2, maxwnd), 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*, is an important measure of how rapidly the window size is changing. Notice that when $cwnd < ssthresh$, $cwnd$ doubles during an epoch so $acceleration \approx cwnd$. In contrast, when $cwnd > ssthresh$, $cwnd$ increases by approximately 1 during an epoch: $acceleration \approx 1$.

[6]Packet losses are detected by either the receipt of duplicate acknowledgments or the expiration of a timer.

[7]Since 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.

Note that the congestion control algorithm presented above has the occasional anomaly that after a full window's worth of packets have been acknowledged the value of $\lfloor cwnd \rfloor$ will remain unchanged even when $cwnd < maxwnd$. As discussed in [16], this anomalous behavior can be removed by simply changing the congestion avoidance increase algorithm with $cwnd \geq ssthresh$ to read:

```
cwnd += 1 / ⌊cwnd⌋
```

With this change, $\lfloor cwnd \rfloor$ increases by one in every epoch in which $cwnd < maxwnd$. In order to simplify the results in our simulations, we will use this modified algorithm. None of the qualitative conclusions we reach will be affected by the change. Removing the anomaly does make the analysis of the dynamics much more straightforward.

The BSD 4.3-Tahoe TCP implementation has a *delayed-ACK* option. With the option off, the arrival of each new data packet at the receiver triggers the transmission of an associated ACK packet. With the option on, upon receiving the first data packet after an ACK has been sent, the receiver does not send out an ACK immediately. The receiver instead waits for either a data packet transmission in the other direction on which the ACK can be piggy-backed, or the arrival of another data packet so that two ACK's can be combined, or the expiration of a timer. In our simulations, the delayed-ACK option is off. We will discuss the effect of turning the delayed-ACK option on in Section 5.

2.2 Network Configuration

All of the simulation results reported on here will involve the simple network topology consisting of a single bottleneck duplex link connecting two switches, as depicted in Figure 1. Attached to each switch is a single host (Host-1 and Host-2). The bottleneck link has a bandwidth μ of 50 Kbps, and a propagation delay denoted by τ. We will consider two values for τ, .01 sec and 1 sec. The links connecting the hosts to the switches have bandwidths of 10 Mbps and a propagation delay of 0.1 msec. All links are modeled as giving error-free transmission. The host processing time of each data or ACK packet is 0.1 msec.

Each switch has a packet buffer associated with each outgoing link and uses the FIFO service discipline and the drop-tail discarding algorithm.[8] There is no buffer sharing between the different outgoing lines. All the configurations discussed in this paper have a buffer size of 20 packets, except the one used to reproduce the simulations in [19] where the buffer size is 30 packets, and the one used to investigate the dynamics of connections with fixed window sizes where the buffer size is assumed to be infinite.

The traffic sources will consist of some number of TCP connections which have an infinite amount of data to send. Each TCP connection has a $maxwnd$ value of 1000, with a constant packet size M of 500 bytes.[9] ACK packets are 50 bytes each. We assume that each TCP connection preexists, so the connection set-up exchange is not simulated.

[8] When the buffer is full and a new packet arrives, the arriving packet is dropped.

[9] For our network configurations the value of $cwnd$ never exceeds 50, so that the actual value of the maximum window size will not be a factor in any of our simulations.

We define the bandwidth-delay product or *pipe* size P as the number of data packets that can be in flight in one direction along the bottleneck link: $P \equiv \mu\tau/M$. With our particular choices in parameters, the propagation delays of 0.01 sec and 1 sec represent pipe sizes of 0.125 and 12.5 packets, respectively.

The configurations considered in this paper are distinguished by the number and location of the TCP connections. The one-way configurations considered in [16] and reviewed in Section 3.1 involve several TCP connections all having their data sources located on Host-1. The configurations in [19], reviewed in Section 3.2, and the configurations in Section 4 have sources on both hosts.

2.3 Simulator

All of the simulations reported on here were done with a simulator written by one of us (LZ), and has been used in several previous simulation studies ([16, 18, 19]). The TCP code was taken directly from the BSD 4.3-Tahoe 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. Also, in order to remove the anomaly in the measure of acceleration, a single line in the window adjustment algorithm was modified as discussed in Section 2.1.

3 Previous Work

In this section we review our two previous studies on the dynamics of the BSD 4.3-Tahoe TCP congestion control algorithm. We begin with the work on one-way traffic in [16], where the dynamics are relatively well understood. We then continue on to the more complicated configuration of [19], where the dynamics are significantly more mystifying. Our goal is to increase our understanding of these more complicated dynamics by applying the insights from the simple configurations in [16] to more complex configurations.

3.1 One-Way Traffic

Consider a network configuration as in Figure 1 with three TCP connections, all with their sources located on Host-1 and their destinations on Host-2. Figure 2 shows the queue length and *cwnd* behavior of this configuration. A detailed analysis of this and other relevant data is given in [16]. Here we only briefly review the aspects of the dynamics most relevant to our subsequent discussion.

First we note that there are periods when the queue length is alternating between two adjacent values at a very high frequency (on the order of a data packet transmission time[10]); in what follows, let q denote the smaller of these two values. Due to the limited resolution, these

[10]Here, as elsewhere in the paper, transmission time refers to that on the bottleneck link.

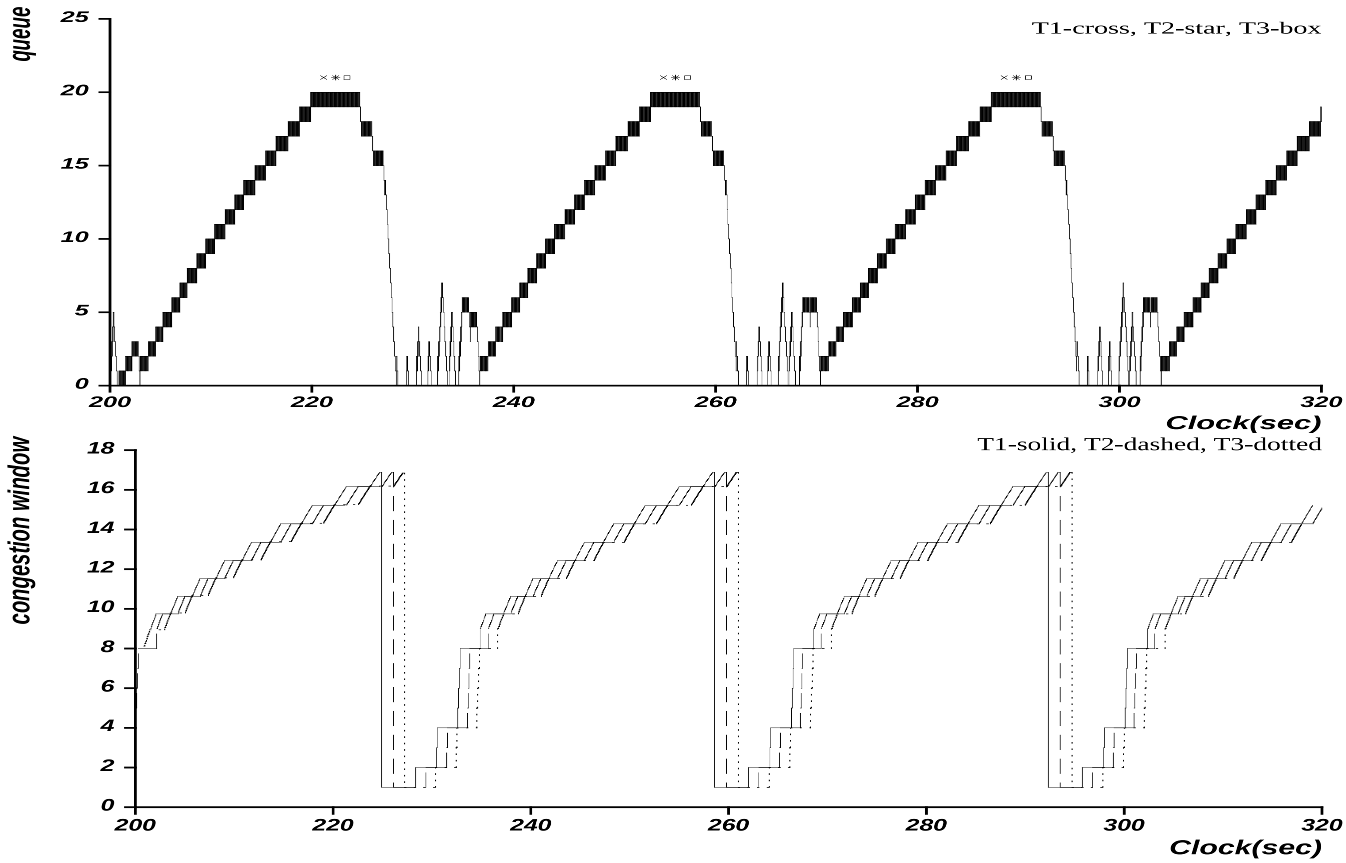

Figure 2: Packet queue at the switch and congestion window sizes for a configuration with three connections, all having sources on Host-1, and with $\tau = 1$ sec. The switches have a buffer size of 20 packets. The marks above the graph of the queue length show the times when packets from the various connections are dropped; the darkened regions are due to the queue length alternating between two adjacent values as packets arrive and depart in an interleaved fashion.

rapid variations appear as darkened areas in Figure 2. These variations are not associated with any rapid variations in the *cwnd* values and instead are due to the alternation of packet arrivals and departures at the queue. The value of q changes rather smoothly with time, and these changes are associated with variations in the values of *cwnd*. This smoothness in the behavior of q is due to the fact that the ACK packets are serving as effective clocks for the connections. That is, between every two ACK arrivals at the source of a connection, there has been a packet departure at the queue. Thus, transmitting a single packet in response to each ACK (as would happen if *wnd* were fixed) ensures that the queue length will alternate between the two adjacent values q and $q + 1$. If we fix the window sizes, the queue length will obey the equation

$$q = \lfloor MAX[0, wnd_1 + wnd_2 + wnd_3 - 2P] \rfloor$$

We can define the capacity C of the path, the maximal total window size that will not result in dropped packets, as $C = \lfloor B + 2P \rfloor$ where B is the size of the switch packet buffer and P is the pipe size. Since q is a function of the total window size, each change in q is associated with a change in the window sizes wnd_i.

In addition to the extremely high frequency alternations in the queue length, there are relatively low frequency oscillations (with a period of roughly 34 seconds) in both the *cwnd* values and the queue length. These oscillations are on the scale of many round-trip times, and reflect the nature of the congestion window adjustment algorithm; *cwnd* is increased until a dropped packet is detected, at which point *cwnd* is decreased to one and the cycle starts over again. The *cwnd* oscillations in the three connections are completely synchronized in-phase. This is because the packet losses of the various connections (which are depicted in Figure 2 by symbols over the graph of the queue length) are synchronized. A note on terminology; we refer to the synchronization of the *cwnd* values as window-synchronization and the synchronization of the packet losses as loss-synchronization. Each connection loses a single packet during the congestion epoch. The number of packets each connection loses is exactly the *acceleration* (as defined in Section 2.1) of the window adjustment algorithm. At the beginning of the congestion epoch, the total window size has reached the capacity of the path, $wnd_1 + wnd_2 + wnd_3 = C$, and any further increase in the window sizes will result in dropped packets. A source, upon receiving the ACK that causes the value of *wnd* to increase by one, immediately sends two packets out; however, there is only room for one packet in the queue (remember that the ACK signaled the departure of a single packet from the queue) so the second packet is dropped. The amount by which a connection increases its value of *wnd* during a congestion epoch is exactly the number of extra packets the source will transmit in response to incoming ACK's while the queue is full, and thus directly determines how many of its packets will be dropped.

Note that there are some periods during which the queue is empty, and thus the line is idle. The level of utilization for the $\tau = 1$ sec case depicted in Figure 2 is about 90%; for the $\tau = 0.01$ sec case which is not shown here but is treated in [16], the level of utilization is nearly 100%. Thus, the utilization level decreases as the size of the pipe increases. Furthermore, the utilization increases as the size of the buffer increases. Intuitively, the time period during which the queue remains empty during the cycle is an increasing function of the pipe size, and the length of the oscillatory cycle is an increasing function of the buffer size. Thus, the utilization level decreases with an increased pipe size, and increases with an increased buffer size. One can show that asymptotically the link idle time decreases with increasing buffer size as B^{-2}.

There is an important aspect to the behavior in this configuration that is not readily apparent from the data presented here. All of the packets from a single connection are clustered together; the entire window's worth of packets passes through the switch consecutively, uninterrupted by packets from another connection. This clustering is reflected in the graphs of *cwnd* in Figure 2 where the curves alternate constant regions with increasing ones, with only one connection increasing at a time and only one increase period per epoch; since *cwnd* only changes upon the receipt of ACK's, this indicates that all of a connection's ACK's arrive in a cluster. The following is a brief explanation of the clustering effect; for a more complete explanation see [16].

Consider a connection immediately after it has reduced *cwnd* to one. At every point at which this connection increases *wnd*, the extra packet is sent out immediately following the preceding packet. Thus, as a connection increases its window size, each new packet is attached to an already existing cluster. The clusters from the various connections do not intermingle, since only one connection receives an ACK in each packet transmission interval making it impossible for a connection to transmit during another connection's cluster. This analysis is valid as long as every packet transmission is in response to an ACK; this can be violated only when the connections are recovering from packet loss(es), in which case they retransmit after some essentially random interval. However, since all connections lose at the same time in this particular configuration, the retransmissions don't interfere with the clustering. In contrast, if only one connection were to lose during a congestion epoch, then its retransmission would likely occur during another connection's cluster.

Let us introduce the terminology that a *pacing* congestion control algorithm is one in which either data packets are not always transmitted immediately by the source upon receipt of an ACK, or ACK packets are not always transmitted immediately by the receiver upon receipt of a data packet; instead, either data or ACK packets are *paced* out according to some other criteria (such as, for example, an estimate of the network bottleneck's transmission rate). Nonpaced algorithms are ones in which both data and ACK packets are always transmitted immediately upon receipt of an ACK or data packet, respectively. One can show that, under fairly general assumptions, this clustering effect will exist in this network configuration with any nonpaced congestion control algorithm. This follows from noting that as long as every packet transmission follows immediately upon receipt of an ACK, the number of packets which are followed in the queue by a packet from a different connection is a nonincreasing function. Experimentally, the same clustering effect has been reported ([11]) in a very different congestion control algorithm (see [12] for a description). Note, however, that the analysis of the clustering effect depends in detail on the round-trip times of the various connections being identical. See Section 5 for further discussion of this point.

While we cannot claim to fully understand every detail of the dynamics in this configuration, most of the relevant phenomena here do seem comprehensible within the framework we developed in [16]. A natural progression to more complicated configurations would lead to consideration of having the sources of the TCP connections located on both hosts. Such a configuration was considered in [19], to which we now turn.

3.2 Rapid Queue Fluctuations

Reference [19] discussed the dynamics in two network configurations significantly more complicated than the one discussed in [16] which was reviewed above. The simpler of the two configurations considered in [19] is similar to that of Figure 1 with ten TCP connections, five having their source on Host-1 and five having their source on Host-2. The actual configuration considered in [19] had somewhat different line speeds, and did not use the slight modification to the congestion control algorithm (described in Section 2.1), but those differences have no qualitative impact on the results. To facilitate direct comparisons, we have chosen to show simulation results from such a configuration based on the network in Figure 1 with $\tau = 0.01$ sec and the congestion control algorithm described in Section 2.1 rather than reproducing the data from the original paper. In this configuration, both switches have a buffer size of 30 packets. The graphs of the total queue length at the two switches are shown in Figure 3.

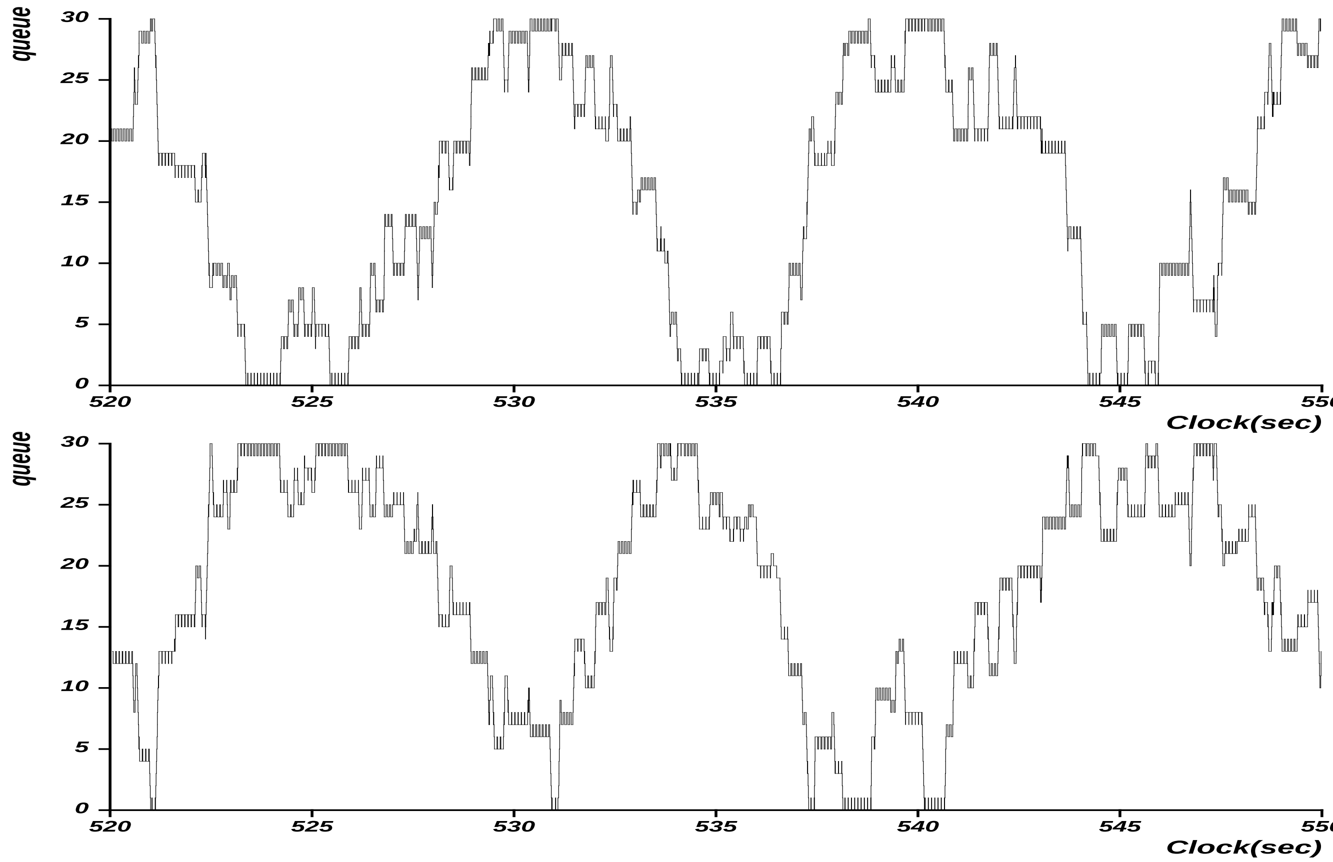

Figure 3: Packet queue at switches 1 and 2 for a configuration with ten connections, five having their source on Host-1 and five having their source on Host-2, with $\tau = 0.01$ sec. The switches have a buffer size of 30 packets.

The data bears some resemblance to that for the previous configuration. The queue lengths still exhibit a low frequency oscillation. However, the most striking feature of this data is the rapid fluctuations in the queue size. The fluctuations are on the order of 5 packets and occur on a time scale smaller than that of a single data packet transmission time. These fluctuations are not associated with correspondingly large changes in the *cwnd* values. This behavior is in sharp contrast to what we saw in the one-way traffic case, where the queue length varied smoothly. These rapid fluctuations in queue length are the central mystery of the dynamics of this configuration.

Another intriguing aspect to the behavior is that there is significant idle time on the bottleneck lines even though the pipe P is quite small. The utilization on the line is roughly 91%, compared to nearly 100% utilization with only one-way traffic. Furthermore, when one increases the buffer size the fraction of idle time does not decrease; in fact when the buffer size is increased to 60 the utilization decreases to roughly 87%. In one way traffic, there is significant idle time only when P is large and, for any given P, the fraction of idle time asymptotically vanishes in the limit of large buffers. This one-way behavior has given rise to the commonly-heard assertion that increasing buffers is a reliable way to increase throughput. The data here contradicts that assertion.

One issue that arises only when there are two congested queues, rather than the single congested queue in the one-way traffic case, is the relative synchronization of the queue lengths in the two switches. In Figure 3 the queue lengths both go through similar low frequency oscillations, but they are out-of-phase with each other. One queue reaches its maximum while the other queue is at its minimum.

For the sake of brevity, we have not shown any data on the *cwnd* or packet-drop behavior, which are not nearly so simply characterized as in the one-way traffic case; we now briefly summarize those results. There is still some degree of loss-synchronization, in that the majority of the connections lose packets during the same congestion epoch. One remarkable occurrence is that 99.8% of the dropped packets are data packets, even though during a congestion epoch all connections are sending packets to a nearly full queue with some of the packets being data packets and some being ACK packets. Since only one of the two queues is full during a congestion epoch (the other is nearly empty), this tendency to drop only data packets implies that only connections sending data packets through the congested queue experience drops during that congestion epoch. In addition, the *cwnd* data indicates that the connections sending in the same direction are window-synchronized in-phase, but the connections with sources on Host-1 are synchronized out-of-phase with the connections on Host-2. This is reflected in the out-of-phase synchronization of the queue behavior discussed above. The total number of packet drops per congestion epoch varies, but the average is approximately ten, the same as the total *acceleration* in this configuration.

It is clear that the insights gained in [16] are not sufficient to explain the behavior in this more complicated configuration. However, the presence of ten connections makes a detailed analysis difficult. Thus it seems natural to consider the simplest case of two-way traffic: that of two TCP connections with sources on different hosts. The dynamics of that configuration is the focus of the rest of this paper.

4 Features of Two-Way Traffic

In this section we analyze the dynamics of the configuration consisting of a network as in Figure 1 with two connections, connection 1 having its source on Host-1 and connection 2 having its source on Host-2. We present our analysis in three parts. We first give an overview of the dynamics, then examine more closely the phenomena of ACK-compression and synchronization modes.

4.1 Overview

Figures 4-7 depict the queue lengths and *cwnd* values from the two-way traffic configuration with propagation delay values $\tau = 0.01$ sec and $\tau = 1.0$ sec. The queue lengths exhibit the familiar low frequency oscillatory pattern. In each congestion epoch two packets are dropped. This is consistent with the *acceleration* analysis which predicts that the total number of packet drops in a congestion epoch is equal to the total acceleration during that epoch; in this case the acceleration of each connection is one so the total acceleration is given by the number of connections. Furthermore, as in the one-way traffic case, the packets from each connection are completely clustered together. This fact is not evident from the figures but was gleaned from a more detailed examination of the dynamics.

However, there are two obvious features of these two-way traffic results that are not present in the one-way traffic results. First, there are high frequency square waves superimposed on the low frequency oscillations of the queue size. These high frequency square waves are similar to, but much more regular than, the rapid fluctuations in queue size seen in [19] and discussed in Section 3.2. Second, the synchronization behavior when $\tau = 0.01$ sec is different from that in Figure 2. Note that the window-synchronization is out-of-phase in Figure 5, in that one window is increasing while the other is decreasing. Also, there is no loss-synchronization in Figure 4; instead, the two dropped packets in each congestion epoch are always from the same connection. We will discuss each of these phenomena in turn in the sections that follow.

The data from the two-way traffic case is rather complex, which not only interferes with the analysis but greatly complicates our presentation. To simplify the situation, we disentangled the effects of the congestion control algorithm from the effects of two-way traffic by considering a configuration in which the window sizes were fixed. Figures 8-9 show the behavior of queue length in configurations in which the two TCP connections have values of wnd which are held constant at 30 and 25 respectively. The switches have infinite buffers. The two connections started at random times and the packets of each connection are completely clustered.

While these fixed-window graphs look quite different than the corresponding graphs in Figures 4 and 6, they share enough of the essential dynamics to render them useful. In fact, as we shall see, both of the phenomena alluded to above have little to do with the details of any particular congestion control algorithm, and are really a feature of clustered two-way traffic under window flow control when the data and ACK packets are of different sizes.

4.2 ACK-Compression

Consider Figures 8-9 which depict the fixed-window data. They exhibit square wave oscillations similar to those in Figure 4, except that here the amplitude of the oscillations is constant which facilitates the analysis. Note that a similar simulation with one-way traffic would have yielded a constant queue length (modulo the alternation between adjacent values as packets arrived and departed). This is because, with one-way traffic, the ACK packets serve as a reliable clock in steady state: between every two ACK arrivals at a source there has been a packet departure at the queue (if the queue is nonempty). This clocking depends critically on the fact that, upon arriving at the source, ACK packets are separated in time by at least the transmission time of a data packet. Because data packets leaving a queue are spaced by the transmission time of a data packet, the receiving host generates ACK packets with this proper spacing. This spacing will remain constant as long as the ACK packets never encounter a nonempty queue on their way back to the source host, which holds true for one-way traffic. When we have two-way traffic, however, ACK packets do encounter nonempty queues. Recall that packets from each connection are clustered together; when

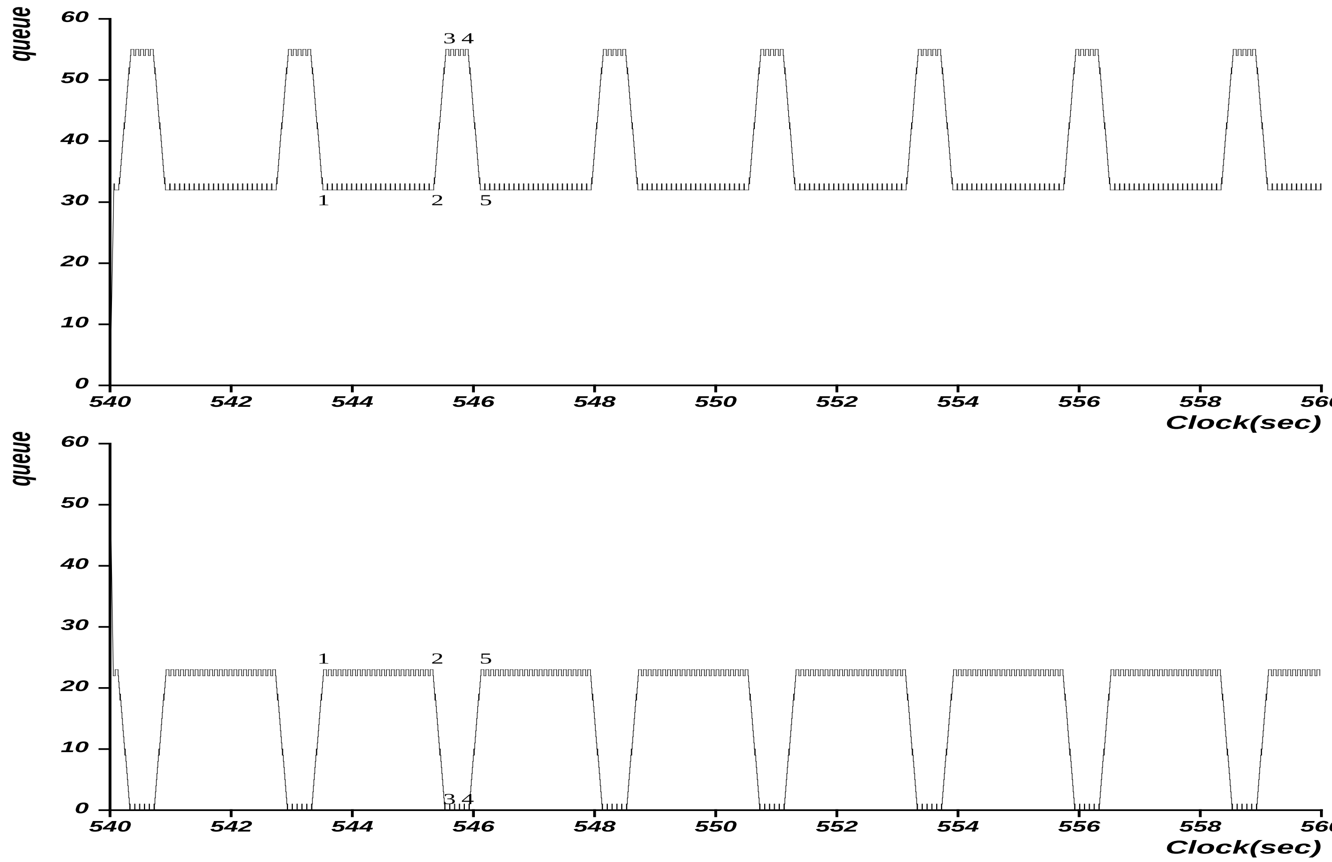

Figure 8: Packet queues at switches 1 and 2 for a configuration with $\tau = 0.01$ sec. There are two connections, one having its source on Host-1 and the other having its source on Host-2, with fixed window sizes of 30 and 25 respectively. The switches have infinite buffers. Note that the two queues have different maximum heights.

a cluster of ACK packets encounter a nonempty queue, their spacing in time upon leaving the queue is no longer the transmission time of a data packet but rather becomes the transmission time of an ACK packet. Since ACK packets are typically much smaller than data packets (in our simulations ACK packets are 1/10 the size of a data packet), this causes the ACK packets to arrive at the source much more closely spaced. The ACK packets are thus no longer reliable indicators of departures of data packets from the queue.

The arrival at the source of each ACK packet triggers the immediate transmission of a new data packet. Thus, a group of closely spaced ACK packets generates a similar group of closely spaced data packets. The sharp increases in queue length seen in Figures 8-9 are the result of these closely spaced group of data packets hitting the queue. The rapid decreases in the queue seen in the data is the result of the ACK packets leaving the queue (this reflects the fact that the queue length is measured in the number of packets rather than in bytes).

Because this dynamic effect depends on the compression of the ACK packet spacing due to their smaller size, we have dubbed the square-wave-like fluctuation phenomenon *ACK-compression*. The preceding analysis indicates that this compression effect will occur only if ACK packets are in clusters; if there is a data packet in between two ACK's in the queue, the spacing between the two ACK's upon leaving the queue can never be less than

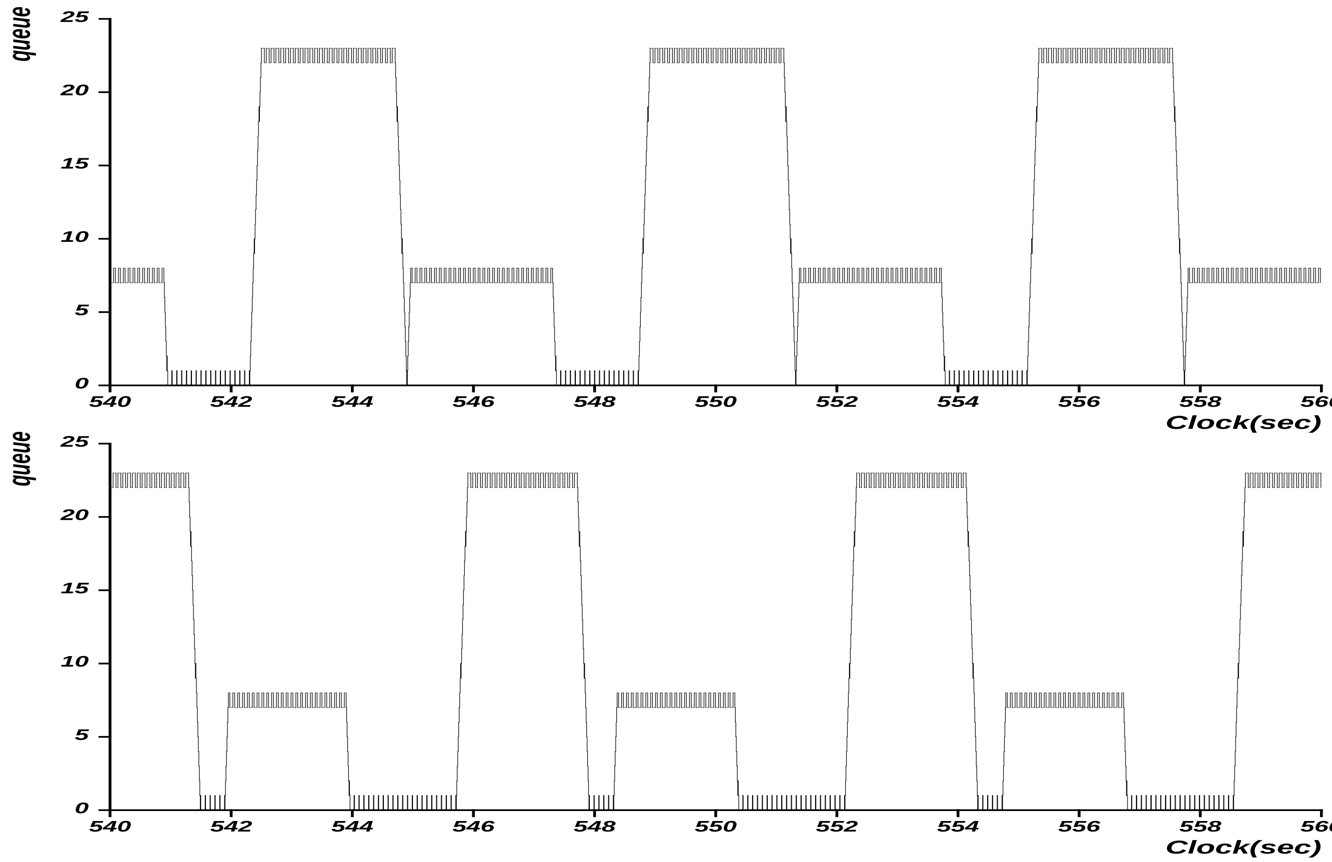

Figure 9: Packet queue at switches 1 and 2 for a configuration with $\tau = 1$ sec and with two connections, which have sources on Host-1 and Host-2 and have fixed window sizes of 30 and 25, respectively. The switches have infinite buffers. Note that the two queues have the same maximum height, and that there is an alternation pattern in the plateau heights.

the transmission time of a data packet. Thus, the existence of ACK-compression depends crucially on the packet clustering phenomenon.

Equipped with this general description, let us return to Figure 8 in greater depth. We now give a detailed chronology of one cycle of this periodic behavior. The numbers below are associated with those above the graph in Figure 8. To avoid circumlocution, we introduce the terminology that A1 and D1 refer to ACK and data packets from connection 1, respectively. Similar definitions apply to A2 and D2. Also, Q1 and Q2 will refer to the congested queue at, respectively, switch 1 and switch 2. Let RD and RA denote the transmission rates of data and ACK packets respectively; in our simulation $RA = 10RD$. Note that ACK packets always arrive at a queue at rate RD, because the data packets that the ACK packets are acknowledging arrive at the destination at exactly that rate.

Recall that the packets from the two connections are completely clustered together rather than intermingled. We start our analysis at a point when the first D2 packet in the cluster has arrived at the head of Q2. We neglect the propagation delay τ in the chronology below, as it is small compared to the transmission time of a data packet.

1. At Q2, A1's are arriving at rate RD, and D2's are leaving at the same rate. At Q1,

the situation is similar, with A2's arriving and D1's leaving. The queue lengths are essentially constant during this period.

2. The last D2 has just left Q2; the A1's are now leaving at rate RA, while the A1's are still arriving at rate RD, causing the length of Q2 to drop suddenly. At Q1, D1's are arriving at rate RA (due to the A1's leaving Q2 at that rate) while D1's are leaving at rate RD, causing the length of Q1 to increase suddenly.

3. Q2 has just emptied; now the A1's that arrive at rate RD leave at rate RD since there is no queue. At Q1, D1's are both leaving and arriving at the same rate RD. The queue lengths are essentially constant during this period. Note that all of connections 2's packets are in Q1 (as ACK's) during this phase, with D1's both ahead and behind them in the queue.

4. The A2's have reached the front of Q1; A2's are leaving at rate RA and D1's are arriving at rate RD, causing a sudden drop in the length of Q1. At Q2, D2's are arriving at rate RA (due to the A2's leaving Q1 at that rate) while D2's are leaving at rate RD, causing a sudden increase in the length of Q2.

5. All of the A2's have left Q1 and the last D2 has reached Q2; now D1's are leaving Q1 at rate RD and A2's are arriving at rate RD. At Q2, A1's are arriving at rate RD and D2's are leaving at rate RD. This completes the cycle.

The explanation of the ACK-compression phenomena used only the following two assumptions: (1) ACK packets are significantly smaller than data packets, and (2) the packets from different connections are clustered. The first of these assumptions is almost universally valid; the second, as discussed in Section 3.1, is valid in this configuration for a wide range of congestion control algorithms. Thus, we expect the phenomena of ACK-compression to be rather common in configurations like those we have described. ACK-compression is the only effect we are aware of which, in steady state, gives rise to large rapid changes in queue lengths.

The fact that not all packets are spaced out by the transmission time of a data packet renders invalid our analysis in [16] about the capacity C of the path. With two-way traffic, the number of packets that can be in flight at any one time depends on how many compressed ACK's are in the pipe. Thus, there is no longer a well defined capacity C that can reliably predict the occurrence of the congestion epochs. For one-way traffic, the line is fully utilized in the congested direction (though almost completely idle in the other direction) whenever the sum of the window sizes is larger than $2P$ (with data packets filling up the pipe in one direction and ACK packets, which are spaced out by one data packet transmission time, filling up the pipe in the other direction). One might naively expect that with two-way traffic both lines would be fully utilized whenever this condition was met. This is clearly not the case. In Figure 8 where $P = 0.125$ and the sum of the window sizes is 55, there is significant idle time when the queue for switch 2 is empty (the corresponding line has a utilization of 86%). In Figure 9 where $P = 12.5$, both queues have times when they are empty (the lines have utilization of 81% and 70% respectively). Why does this idle time occur? We do not yet have a complete explanation, and do not have room to explain what we do know, but the following remarks may provide some intuition. Whenever an ACK packet has to wait in a queue, the queueing delay has the same effect as increasing the pipe size. Thus, even though the window sizes are large enough to fill the actual pipes, they are not able to fill the *effective* pipes. Note that the size of the effective pipe seen by a connection is a function of the window size of the other connection. Thus, increasing a connection's window will increase the utilization of one line, but will decrease the utilization of the other line.

We are not yet able to precisely characterize the idle time in this fixed-window system. A system which is easier to analyze is one in which the ACK's are of zero length. We discuss this system again in Section 4.3.3, but make the following observations here. As long as the pipe has nonzero size there are *no* conditions in which *both* lines are fully utilized. In such a system, when the difference in window sizes is less than $2P$, both lines are underutilized (as in Figure 9). When the difference is greater than $2P$, only one line is underutilized (as in Figure 8).

Returning to the adjustable window case, the square wave oscillations in Figures 4 and 6 are due to the same ACK-compression phenomena just described for the fixed-window case. The only difference is that the increase in the window sizes in each epoch causes the plateau heights to increase in each epoch. The observation that ACK packets always arrive at a queue spaced out by the data packet transmission time implies that no ACK packets are ever dropped. Since a nonempty queue has decreased by at least one between every two ACK arrivals, we know that if the first ACK packet was not dropped the second cannot be dropped either. The first ACK packet in a cluster won't be dropped because it must follow the previous data packet by at least a data packet transmission time. This remark will be useful in the next section, where we focus on the behavior of *cwnd*.

4.3 Synchronization Modes

The graphs in Figures 4-7 exhibit two different synchronization modes. When the propagation delay is small (Figures 4 and 5), the connections are synchronized out-of-phase. In Figure 5, one *cwnd* value is rising while the other is falling. Similarly, the two queues are out-of-phase with each other in Figure 4. This is much like the synchronization behavior in the Figure 3. However, when the propagation delays are large (Figures 6 and 7) the connections are in phase with each other; the queue lengths rise and fall together as do the two *cwnd* values. This is much like the in-phase window-synchronization we saw in the one-way traffic case. Simulation of other configurations reveals that typically for a fixed buffer size, the synchronization is in-phase for large P and out-of-phase for small P. Similarly, for a fixed pipe size, the synchronization is usually in-phase for small buffers and out-of-phase for large buffers. We now discuss the out-of-phase and in-phase synchronization behaviors separately.

4.3.1 Out-of-Phase

We first consider Figures 4 and 5 where the propagation delay τ of the bottleneck link is 0.01 sec. The symbols above the graph of the queue length in Figure 4 indicate the occurrence of packet drops. During each congestion epoch one connection loses two packets while the other connection loses none. In the next congestion epoch, the roles are reversed and the connection which escaped without packets drops in the previous congestion epoch now suffers the double packet drop. Thus, in every congestion epoch the total number of packets lost is equal to the total acceleration, but the losses are not distributed evenly. Because one connection loses while the other doesn't, the window increase-decrease cycles of the two connections are synchronized out-of-phase. The current implementation of the window adjustment algorithm is such that, if two data packets are lost in a row, the value for *ssthresh* is reduced to its minimal value of 2.[11] It takes a long time for the connection to build its window back up; during this time the other connection is getting most of the bandwidth. In fact, *cwnd* increases as the square root of time over the whole cycle, rather than having an initial exponential and then linear growth periods (see [16] for a fuller discussion of the congestion window growth laws).

The utilization of the bottleneck line is 70% (compared to nearly 100% for the one-way traffic case). Note that with two-way traffic there is considerable idle time, even here where the pipe is very small. This idle time is similar to what we saw in Figure 8; since the pipe size is so small, the windows always differ by more than $2P$ and thus only one line is underutilized at any given time.

The presence of significant idle time remains true even if we increase the buffer size; when the buffer size is increased to 60 and 120 the utilization remains at roughly 70%. For the various one-way configurations analyzed in [16] the fraction of idle time on the bottleneck line approaches zero in the limit of infinite buffers. This was because, as discussed in Section 3.1, the length of a window increase-decrease cycle increases as one increases the buffer size, but the idle time in a cycle remains constant because it is just a function of the pipe size. This is no longer true when we have two-way traffic. The idle time in a cycle is a function of the *effective* pipe size which, since it is determined by the other connection's window, increases with the buffer size. In fact, the increase in the effective pipe size is proportional to the increase in the cycle time, so that in the limit of infinite buffers the utilization remains less than optimal.

4.3.2 In-Phase

We now turn to Figures 6 and 7 where the propagation delay τ of the bottleneck link is 1 sec. Again, packet drops are indicated by the symbols above the graph of the queue length in Figure 6. In each congestion epoch, each connection loses a single packet. Thus, even though the assumption of a well defined path capacity C which underlay the analysis in Section 3.1 is no longer valid here, the results about each connection losing an acceleration's

[11] When the first loss is detected, *ssthresh* is reduced to *cwnd*/2, and *cwnd* is reduced to 1. Upon detection of the second loss, *cwnd*'s value is still 1, and *ssthresh* is set to its minimal allowed value which is 2.

worth of packets during the congestion epoch seems to hold. Since the drops are close to each other in time, the increase-decrease cycles of the *cwnd* values and the queue lengths are all synchronized in-phase. There are repeating periods of idle time when the compressed ACK's are in the pipe; the average utilization of the line is roughly 60% (compared to 90% in the one-way traffic case with the same pipe size). Note that there are times when both lines are idle, as in Figure 9. This is different from the small pipe case where only one line is idle at any moment.

4.3.3 Analysis

An obviously relevant question is: why do these two different synchronization modes arise? Surprisingly, one can see the root causes of these two modes in the fixed window data in Figures 8 and 9. Consider Figure 8; in each epoch queue 1 reaches a maximum of 55 while queue 2 reaches a maximum of 23. If one were to fix the buffer size to be 55 and then suddenly increase the window sizes of both connections by one, connection 1 would suffer two losses while connection 2 would not suffer any losses. This follows from three observations: (1) two packets must be dropped in order to fit in the buffer size of 55, (2) ACK packets are never dropped (as explained in Section 4.2), and (3) packets are never lost from queue 2 because its maximum length is well below the buffer size. This behavior resembles the out-of-phase synchronization mode.

In contrast, the queues in Figure 9 both reach the same maximal height of 23. If one were to fix the buffers sizes to be 23 and then suddenly increase both window sizes by one, both queues would overflow and thus both connections would experience a single packet loss. This is reminiscent of the in-phase synchronization mode.

We are not yet able to completely characterize the dynamics of this fixed window system. However, we do have a conjecture for a system in which the ACK packets are of zero length[12]. Let w_1 and w_2 denote the fixed window sizes and assume, without loss of generality, that $w_1 \geq w_2$. Then, we conjecture that there are only the following two cases.

1. $w_1 > w_2 + 2P$: The two queues are synchronized out-of-phase, and only one line is fully utilized.

2. $w_1 \leq w_2 + 2P$: The two queues are synchronized in-phase, and neither line is fully utilized when the inequality is strict.

This simple criterion completely characterizes the relevant behavior when we have negligible size ACK's. It appears that with nonzero-sized ACK's the system continues to exhibit only these two behaviors, but the simple criterion no longer applies.

What role does the congestion control algorithm play in determining which synchronization mode is present? The window-adjustment algorithm controls the relative window sizes during

[12] Due to space limitations, we only present the content of the conjecture here; a fuller explanation will appear in a future publication.

the congestion epoch; these window sizes determine which fixed-window case we are in. Increasing the buffers with a fixed P tends to increase the difference between the window sizes at the congestion epoch, thus producing the out-of-phase synchronization. Increasing P with fixed buffers makes the criterion $w_1 > w_2 + 2P$ harder to satisfy, thus producing the in-phase synchronization.

We have simulated other configurations. Upon varying the buffer size or the pipe size P (by adjusting the propagation delay τ), one usually sees one of the two cases described above. However, we have also observed behavior which which does not fit neatly into our in-phase/out-of-phase taxonomy. Usually these problematic behaviors are either synchronized in-phase or out-of-phase, but often the pattern of dropped packets is more complicated than we described above and violates the acceleration analysis (which we knew was not appropriate for two-way traffic). For instance, there is an in-phase mode in which both connections experience double drops every congestion epoch. Some modes alternate between the single drop and double drop behavior. Also, there is a mode in which an anomalously large number (roughly 10) of packets are dropped every few congestion epochs. We do not understand the behavior in these other, less common, modes; they are the subject of future work.

5 Discussion

One of the purposes of this paper is to understand the results in [19], which we reviewed briefly in Section 3.2. Are those results explained by what we have seen in our simple two-way traffic configurations? Compare Figure 3 with Figure 4. The rapid queue fluctuations in Figure 3 are similar to those in Figure 4, indicating the presence of ACK-compression. Furthermore, the synchronization and idle time apparent in Figure 3 resembles those of the out-of-phase synchronization mode in Figure 4. These were the key features we wanted to understand. There are some differences, however, between the data in Figure 3 and that in Figure 4; in Figure 3 the plateaus of the square-wave-like fluctuations are narrower, the queue length rise more rapid, and the dynamics significantly less regular.

The widths of the plateaus reflect the sizes of packet clusters. Recall that the configuration analyzed in Figure 3 had a buffer of size 30, with five connections in each direction and $\tau = 0.01$ sec. Thus, if the dynamics were completely regular and symmetric, each connection would have a maximum $wnd \approx 6$ during the congestion epoch. This is in contrast to the maximum wnd values of roughly 17 and 33 (see Figures 5 and 7) for the simpler configurations considered in this paper. This explains the narrowness of the plateaus.

The rate at which the queue size rises is related to the total *acceleration* and the total acceleration during a congestion epoch is just the total number of connections. Since we have 10 connections in the configuration for Figure 3 compared to just 2 for Figure 4, we would expect the queue length to rise much more rapidly in Figure 3.

The regularity in the simple configurations considered in this paper is due to the complete clustering of the packets. We have explained in [16] why this clustering occurs for one-way traffic configurations. It also holds when there is a single connection in each direction.[13] However, complete clustering does *not* always occur when there are multiple connections in each direction because not all connections lose packets during the same congestion epoch. There is still some degree of clustering, in that most packets are followed in the queue by packets from the same connection, but the clustering is no longer complete nor regular. This causes the dynamics in Figure 3 to be somewhat irregular.

We have spent considerable time focusing on the phenomena of ACK-compression and synchronization modes. It is natural to ask how general these results are. The presence of the two phenomena relied on two crucial properties: (1) ACK packets are significantly smaller than data packets, and (2) the packets from each connection are clustered together. We therefore expect that *any* configuration which satisfies these two properties will exhibit the phenomena of ACK-compression and synchronization modes. There are two aspects of a configuration; the flow control algorithms and the network topology.

In this paper we have only considered the BSD 4.3-Tahoe TCP congestion control algorithm. However, we expect our results to be more generally applicable. Other nonpaced window adjustment algorithms will also have the clustering effect, and thus we would expect to see ACK-compression and synchronization modes for those algorithms as well.

Similarly, we have only considered one very simple network topology. But, once again, as long as some degree of packet clustering exists, we expect the phenomena of ACK-compression and synchronization modes to be important aspects of the traffic dynamics. Therefore a crucial question is: for what kinds of network configurations are the packets at least partially clustered? We do not yet know the answer to this question. However, for a topology considered in [19] consisting of four switches, with a traffic pattern of 50 connections whose path lengths were roughly equally split between 1, 2, and 3 hops, the queue length data displayed both the ACK-compression and out-of-phase synchronization phenomena. Thus, even in this rather complicated topology where a detailed analysis of the dynamics is infeasible, the basic aspects of the behavior are due to the phenomena we have discussed here.

On the other hand, we know at least two kinds of modifications to the configuration that can reduce packet clustering to some extent. First, the fact that the two connections had the same round-trip time was crucial to the complete packet clustering in our simulation. When the round-trip times of different connections differ by more than a packet transmission time at the bottleneck point, the clustering will no longer be perfect, although partial clustering may still exist. Secondly, the delayed-ACK option (see Section 2.1) in the current BSD 4.3-Tahoe TCP implementation introduces some elements of pacing, not by changing what the source does but by modifying how soon the receiver responds to arrived data. With the delayed-ACK option on, the receiver will hold back the acknowledgment to an arrived data packet until a second data packet arrives or until a timer, which has a rather conservative timeout value, expires; both of these actions effectively delay the ACK to the first packet by at least a packet transmission time. We have simulated the behavior with this option on. For small window sizes (e.g., $maxwnd = 8$), the packets in the window are cut into a few small partial clusters minimizing the effect of ACK-compression. When the window sizes are large,

[13]The argument in [16] can easily be extended to include this case; for the sake of brevity, we have omitted this argument.

however, some partial clusters are of appreciable size, and the effect of ACK-compression becomes significant again. Thus, the delayed-ACK option reduces the degree of clustering, and hence the effect of ACK-compression, to some degree but does not eliminate it.

It is reasonable to ask if the phenomena we have described here are merely artifacts of our unrealistically simple simulation model. A piece of closely related work by Wilder et al. ([17]), which we received after the completion of our work, describes some measurements on an OSI testbed network. The testbed configuration was somewhat similar to those considered here, but with longer paths and various numbers of connections going in each direction. The hosts in the testbed run an implementation of the OSI transport protocol TP4 enhanced with the CE-bit congestion avoidance algorithm ([15]). Even though this congestion control algorithm has shown fair throughput allocations in previous tests with one-way traffic configurations (on the same testbed), the two-way traffic measurements revealed extreme unfairness. This unfairness was ascribed to rapid queue length fluctuations caused by ACK-compression, which seriously interfered with the switch's load averaging algorithm. It was also noted that, as we have seen here, the lines were significantly underutilized. These measurements on a real network suggest that the phenomena we described: (1) are not simulator artifacts, (2) exist in real implementations with different window-based congestion control algorithms, and (3) can have a significant and harmful effect on congestion control algorithms which were designed with the assumption that ACK's would provide sufficient clocking to keep the queue length fluctuations minimal.

6 Summary

This paper addressed the nature of network dynamics in a simple network with two-way traffic controlled by the BSD 4.3-Tahoe TCP implementation. Two-way traffic exhibits several of the same phenomena that we found in one-way traffic. The packets from each connection are clustered together, and the number of losses can be roughly estimated by the *acceleration* analysis. However, there are two phenomena that are new to two-way traffic. First, there is ACK-compression caused by the interaction of ACK and data packets in a queue. ACK-compression produces rapid and large fluctuations in the queue length. It also renders invalid the assumption that ACK's provide reliable clocks for data transmissions. In addition, ACK-compression can give rise to significant idle time even when the flow control windows are large compared to the pipe size. Second, two-way traffic has two synchronization modes. The out-of-phase synchronization mode has the counterintuitive property that increasing the buffer size does not always result in higher throughput.

Even though our analysis was restricted to a very special case, it appears that the insight gained from these simple networks appears to apply, at least in part, to more general situations. However, there are many unresolved issues. The dynamics in more complicated networks is still very poorly understood. Also, it is not clear what relevance our results have for the Internet or other similar large-scale networks. For instance, is ACK-compression a common phenomenon in these networks? Are the packets from different connections clustered in network queues, or or are they mostly interleaved? These questions await careful measurement.

Finally, one can ask whether what we have seen here has any implications for the design of new congestion control algorithms. We have seen that a standard rule-of-thumb is not valid with two-way traffic; ACK's are not reliable clocks, even in steady state. Thus, future designs must find more reliable means to supply this clocking function. Perhaps most importantly, we have also seen that minor modifications in protocol implementations can have a profound and unintended impact on the performance. For instance, the delayed-ACK option was originally intended solely to decrease the network overhead by reducing the number of ACK's. However, this slight change to the ACK'ing behavior can significantly alter the traffic dynamics. When seemingly minor implementation changes can have unintended and unexpected consequences, how does one separate the implementation specific details of an algorithm from the essential features needed for adequate performance?

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 **ACM Computer Communication Review**, 20(4), pp. 23-29, October, 1990.

[3] A. Demers, S. Keshav, and S. Shenker. *Analysis and Simulation of a Fair Queueing Algorithm*, In **Journal of Internetworking: Research and Experience**, 1, pp. 3-26, 1990.

[4] S. Floyd and V. Jacobson. *Traffic Phase Effects in Packet-Switched Gateways*, In **ACM Computer Communication Review**, 21(2), pp. 26-42, April, 1991.

[5] E. Hashem. *Analysis of Random Drop for Gateway Congestion Control*, In **Report LCS TR-465, Laboratory for Computer Science, Massachusetts Institute of Technology**, 1989.

[6] V. Jacobson. *Congestion Avoidance and Control.* In **Proceedings of SIGCOMM '88**, pp. 314-329, August 1988.

[7] 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.

[8] R. Jain, K. K. Ramakrishnan, and D.-M. Chiu. *Congestion Avoidance in Computer Networks with a Connectionless Network Layer*, In **Innovations in Networking**, edited by Craig Partridge, Artech House, Boston, 1988.

[9] A. Mankin and K. Thompson. *Limiting Factors in the Performance of the Slow-Start TCP Algorithms*, In **Proceedings of USENIX Winter'89 Conference**, 1989.

[10] A. Mankin. *Random Drop Congestion Control*, In **Proceedings of SIGCOMM '90**, pp. 1-7, September 1990.

[11] D. Mitra and J. Seery *private communication*

[12] D. Mitra and J. Seery *Dynamic Adaptive Windows for High Speed Data Networks: Theory and Simulation* In **Proceedings of SIGCOMM '90**, pp. 30-40, September 1990.

[13] J. Nagle. *Congestion Control in TCP/IP Internetworks*, **ACM Computer Communication Review**, 14(4), October, 1984.

[14] J. Postel. *DoD Standard Transmission Control Protocol* Network Information Center RFC-793, SRI International, September 1981.

[15] K. Ramakrishnan and R. Jain *A Binary Feedback Scheme for Congestion Avoidance in Computer Networks* **ACM Transactions of Computer Systems**, Vol 8, No.2, May 1990.

[16] S. Shenker, L. Zhang, and D. Clark. *Some Observations on the Dynamics of a Congestion Control Algorithm*, In **ACM Computer Communication Review**, 20(4), pp. 30-39, October, 1990.

[17] R. Wilder, K. K. Ramakrishnan, and A. Mankin *Dynamics of a Congestion Control and Avoidance of Two-Way Traffic in an OSI Testbed*, In **ACM Computer Communication Review**, 21(2), pp. 43-49, April, 1991.

[18] L. Zhang. *A New Architecture for Packet Switching Network Protocols*, In **Technical Report TR-455, Laboratory for Computer Science, Massachusetts Institute of Technology**, 1989.

[19] L. Zhang and D. Clark. *Oscillating Behavior of Network Traffic: A Case Study Simulation*, In **Journal of Internetworking: Research and Experience**, 1, pp. 101-112, 1990.

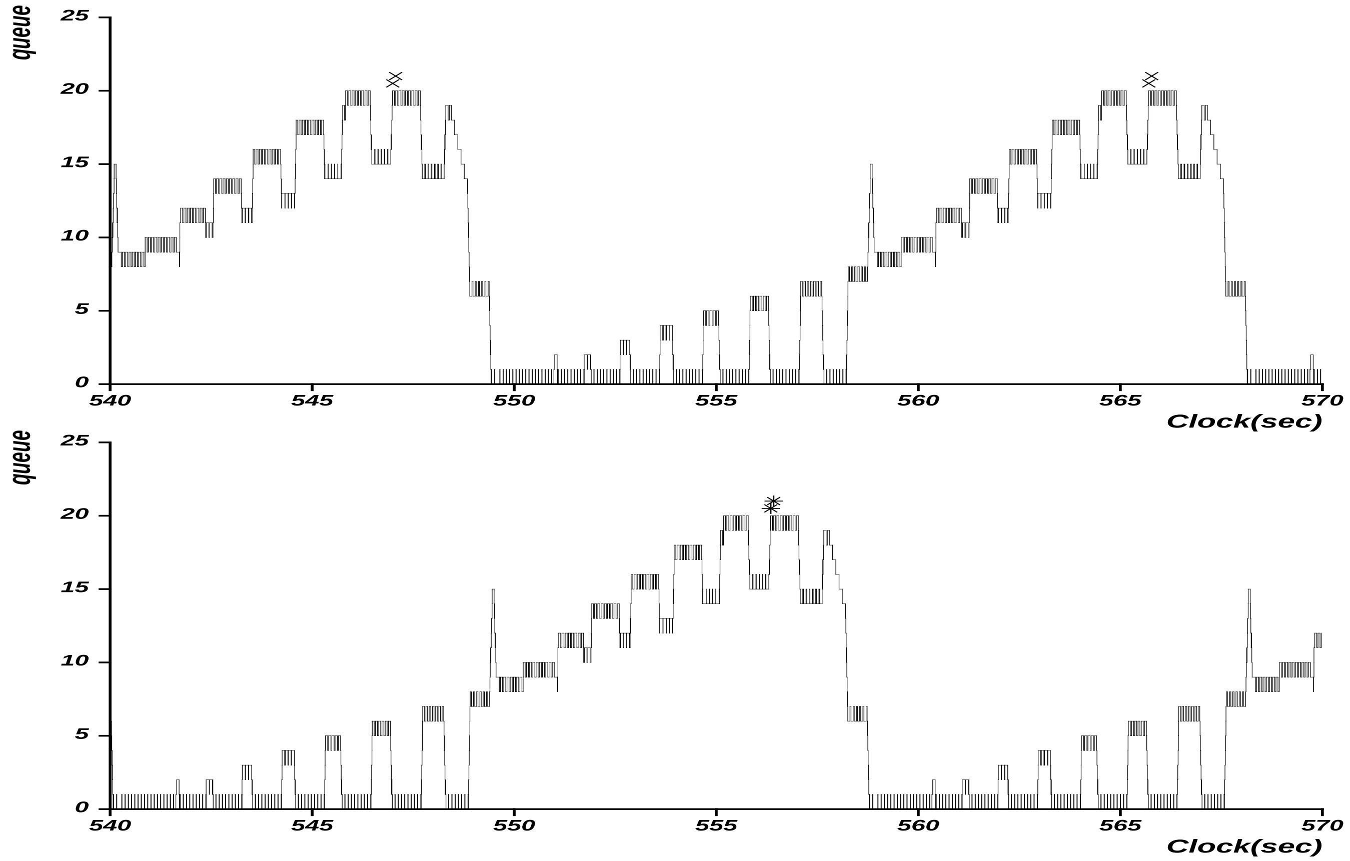

Figure 4: Packet queue at switches 1 and 2 for a configuration with $\tau = 0.01$ sec and with two connections, one having its source on Host-1 and the other having its source on Host-2. The switches have a buffer size of 20 packets. Each mark above the graphs indicates the dropping of a data packet. Note that during a congestion epoch one connection loses two packets while the other has no losses.

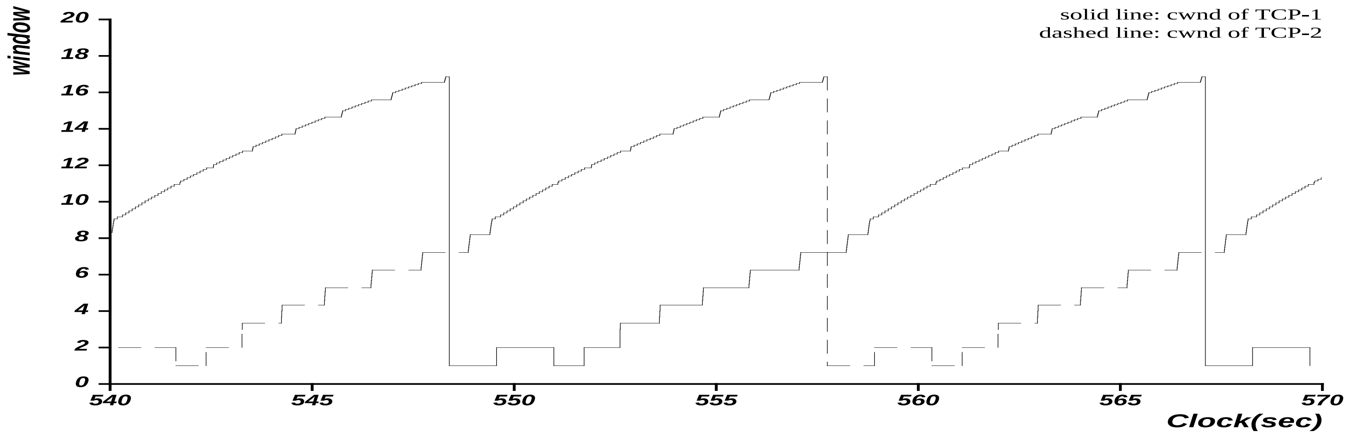

Figure 5: The congestion window sizes for the two connections in the configuration described above. The increase-decrease cycles of the two connections are synchronized out-of-phase.

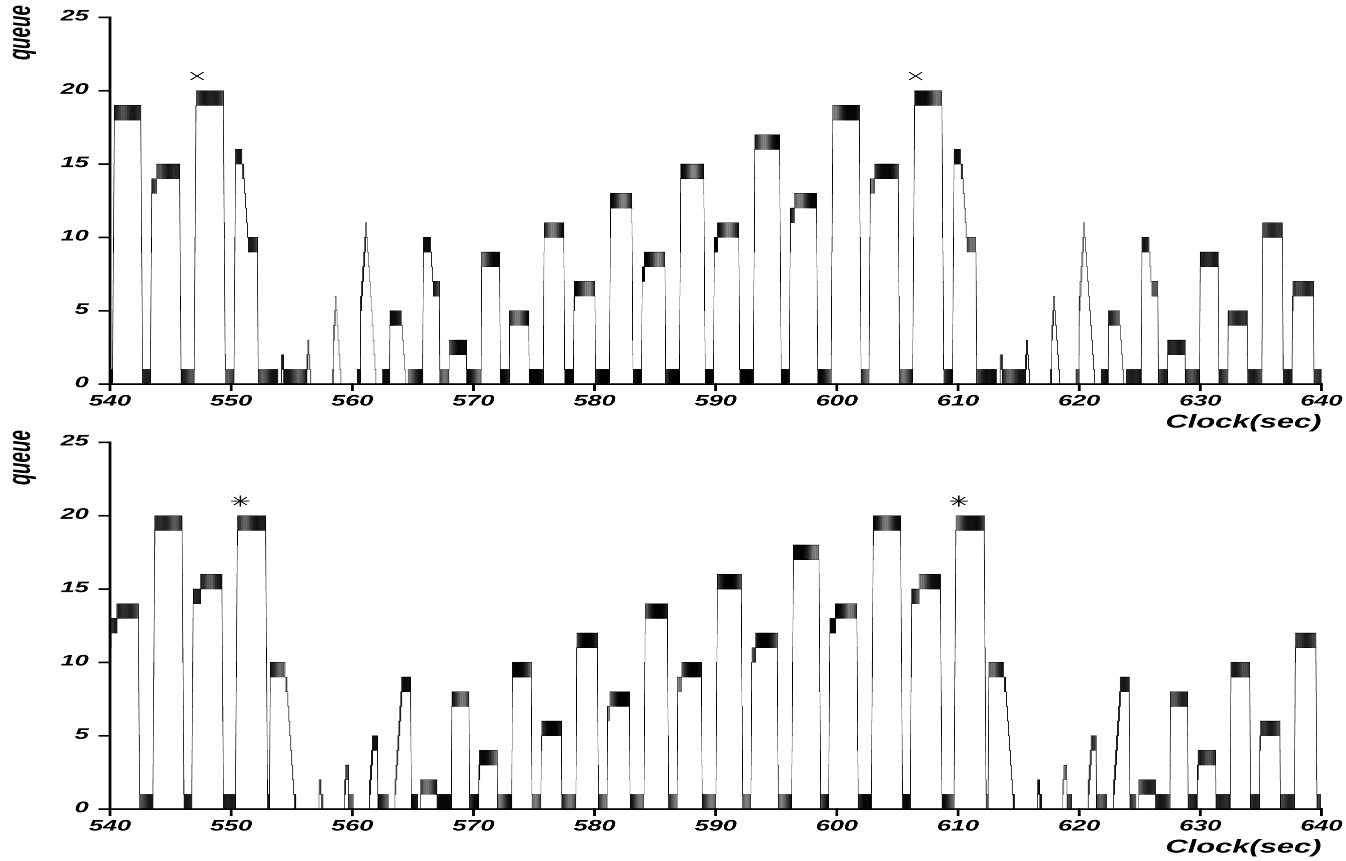

Figure 6: Packet queue at switches 1 and 2 for a configuration with $\tau = 1$ sec and with two connections, one having its source on Host-1 and the other having its source on Host-2. The switches have a buffer size of 20 packets. Each mark above the graphs indicates the dropping of a data packet. Note that during a congestion epoch both connections have a single packet dropped.

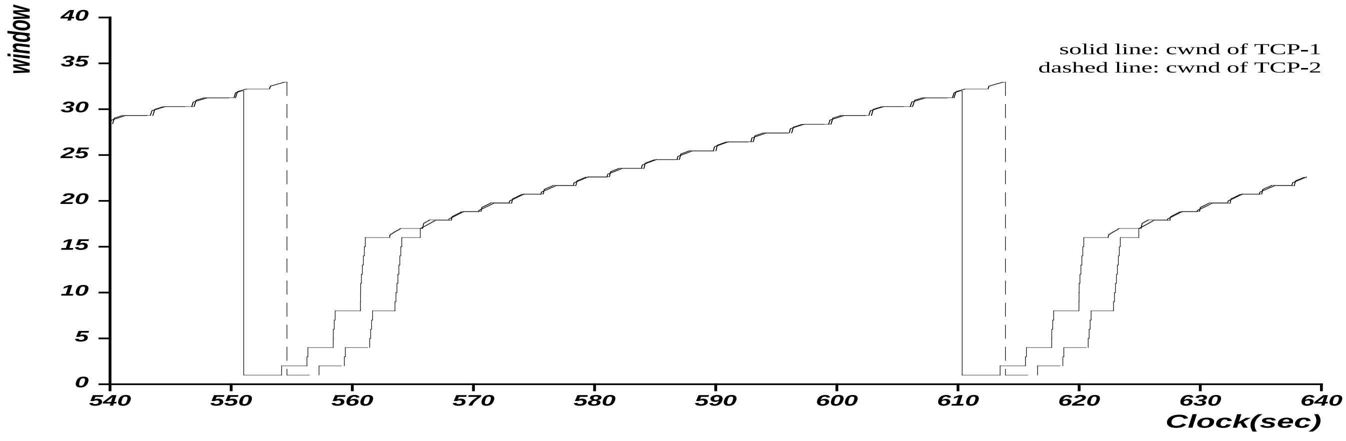

Figure 7: The congestion window sizes for the two connections in the configuration described above. The increase-decrease cycles of the two connections are synchronized in-phase.