# Packet Reordering is Not Pathological Network Behavior: And it never will be again.

Jon C. R. Bennett
Harvard University &
BBN Technologies
jcrb@eecs.harvard.edu
jcrb@bbn.com

Craig Partridge
BBN Technologies
craig@bbn.com

Nick Shectman
BBN Technologies
shectman@bbn.com

**Abstract**

It is a widely held belief that packet reordering in the Internet is a pathological behavior, or more precisely, that it is an uncommon behavior caused by incorrect or malfunctioning network components. A number of recent studies of Internet traffic [13, 10] have reported seeing occasional packet reordering events and ascribed these events to "route fluttering", router "pauses" or simply to broken equipment.

We have found, however, that parallelism in Internet components and links is causing packet reordering under normal operation and that the incidence of packet reordering appears to be substantially higher than previously reported.

More importantly, we observe that in the presence of massive packet reordering TCP performance can be profoundly effected. Perhaps the most disturbing observation about TCP's behavior is that large scale and largely random reordering on the part of the network leads to poor and in some cases, self-reinforcingly poor, performance from TCP.

## 1 Introduction

That there *is* packet reordering in the Internet is not going to come as a great surprise to most people. What may come as a surprise is the frequency and magnitude of the packet reordering that is occurring over many routes in the Internet. A further surprise is that the cause of much of this reordering is local parallelism.

By local parallelism, we mean that a packet can follow multiple paths within a device or logical link. Examples of local parallelism are switches which allow packets traveling between the same source and destination to take different paths through the switch, and link-level striping. One of the challenges of understanding reordering due to local parallelism is the parallelism is not made externally visible (except through anomalous behavior like reordering). So, unlike Paxson's experience with multi-path routing[12], where the datagrams' different paths were clearly indicated by the different addresses of the routers they traversed, the only hint of local parallelism may be that a particular hop exhibits varying delays and reordering.

Local parallelism is on the increase in the Internet because it reduces equipment and trunk costs. It is often more cost effective to put two components in parallel than to use one component that has twice the speed. This rule is especially true when purchasing long-haul serial links because tariffed options often don't have nice increments. In addition, parallel links are a very useful way to improve reliability. If the parallel links follow different physical paths, the virtual link they implement is less vulnerable to backhoes. Not surprisingly, large businesses, ISPs, and their vendors are aggressively pursuing parallel links. In a survey of 38 major ISPs conducted in mid 1997 [2] only 2 of the smaller ISPs did not have parallel paths between nodes.

Compounding this trend is the fact that interface speeds typically trail customer needs, especially ISPs needs. So ISPs are forced to cobble together the link rates they need by aggregating interfaces. An example of an ISP-like situation where bandwidth demands have led to parallelism is the MAE-East peering point, where we have observed extensive reordering.

In the rest of this paper, we will examine the reordering problem further. Section 2 reports on tests done to measure the prevalence of reordering. Section 3 reports in detail on why reordering occurs at MAE-East and then discusses why it is difficult to get rid of reordering. Section 4 presents how reordering affects TCP performance. We conclude, in Section 5, with a discussion of how this work may affect the construction of networking equipment and the design of networking protocols.

# 2 Observing and Measuring Reordering

We performed a number of tests between a variety of hosts to determine average frequency and severity of the reordering that any given session might expect to experience. What we discovered is that reordering is a very complex phenomenon. In particular, we found that reordering is strongly a function of the following properties of the path taken by a session: (1) the existence of parallel links between nodes on the path; (2) the exact configuration of the hardware and software in the nodes; and (3) the load on the nodes.

## 2.1 What We Measured

Because we saw signs that reordering was a function of load and configuration, we looked for measurement paths for which we could get detailed information about load and configuration. In the end, we chose to measure sessions over paths that traversed the MAE-East exchange.

The MAE-East was an excellent example case, for a number of reasons: (1) we have full control of, and knowledge about all the routers between the source of our tests and the MAE-East exchange, allowing us to insure that they are not effecting our results; (2) there is a wide diversity of equipment and hosts only 1 or 2 hops on the other side of MAE-East from our ISP. (3) the topology and makeup of MAE-East are publicly known; (4) the traffic statistics at MAE-East are publicly available; and (5) it is the most heavily loaded site we are aware of having the properties which can play a crucial role in causing reordering.

The reader is strongly cautioned not to infer, however, that reordering at MAE-East is somehow special. We have found many other paths upon which reordering occurs[1].

## 2.2 The First Test: How Many Sites Experience Reordering?

We chose 140 Internet hosts which we had reason to believe were topologically close to MAE-East such as server machines of the ISPs peering at MAE-East To each host we sent a burst of five 56 byte ICMP ping packets to insure that there were no problems caused by route cache misses, followed 5 seconds later by a back-to-back burst of 50 ping packets[2] . We report here the results of two tests spaced a month apart. Each test was run from 6pm to 7pm EST on a Monday evening.

We then measured the number of sessions that experienced out-of-order delivery. To avoid having to define 'out-of-order' in the presence of packet loss, we measured the number of probes which received all of the first 10 packets, then how many probes that received all the first 10 packets received them in order. We also measured the number of probes which received all of the first 20 packets, and how many probes that received all the first 20 packets received them in order.

## 2.3 The Second Test: Reordering and Load

Detailed daily traffic load information is available for MAE-East so we decided to conduct a more detailed test of the relationship between traffic load and reordering. We chose a site which demonstrated a particularly high degree of reordering and sent it a 100 packet burst of 512 byte packets every minute for four days.

In this test we also developed a metric for how much reordering took place. The metric we chose was the average number of SACK blocks required to cover the out-of-order ping replies received, if the session were a TCP connection. Losses were eliminated from the traces by renumbering of sequence numbers (i.e., if packet 8 was lost, packet 9 was renumbered as packet 8 for the purpose of computing the SACK block metric). This is particularly important to keep in mind that all the holes in the SACK scoreboard are due to reordering and not to packet loss.

We chose the SACK block metric after trying several other metrics. Our goal was to find a metric which had the following properties: (1) generally matched an intuitive assessment of the amount of scrambling in a given data set; (2) had a higher metric if data passed through two scramblers than if passed through a single scrambler (and ideally, would yield a value equal to the sum of the metrics for the scramblers separately); (3) be reasonably independent of the size of the data set; (4) be reasonably independent of data loss; (5) had a minimal value for in-order data; and

[1] For example a quick check at the routing data of the *Traceping* page at Oxford University in England http://av1.physics.ox.ac.uk/ will show large numbers of parallel links visible at the IP level.

[2] This was done using the ping *preload* command

| Hosts Receiving Pkts | 1-10 | 1-10 | 1-20 | 1-20 |
|---|---|---|---|---|
| Date of Test | | In order | | In order |
| Early Dec 1997 | 117 | 7 | 97 | 5 |
| Early Jan 1998 | 60 | 4 | 46 | 1 |

Table 1: Reordering Tests

(6) was reasonably easy to calculate. Among the metrics we tried were displacement metrics, number of monotonic sequences required to include all data, and the number of spurious retransmissions that would have been generated by a standard TCP implementation. Of these, SACK blocks proved to be the best metric, although it is not independent of the size of the data set.

## 2.4 Results

Table 1 shows the results of the first test, pinging 140 sites. We show the number of sessions that received all of the first 10 and first 20 packets, and then the number the number of these sessions that received all the packets in order. The table suggests the extraordinary result that probability of a session experiencing reordering is over 90%.

The reader may find this result hard to believe. If packet reordering were this common, surely it would have been seen before. We believe there are two reasons why reordering has not been seen before. First, and foremost, the reordering at MAE-East is as a function of network load (as shown in the next test), and is almost non-existent below a certain load.

Prior to middle of 1997 the load at MAE-East almost never exceeded this threshold. Second, many people who study networking traffic are at major sites attached to major ISPs. The major ISPs all have private peering points which, by and large, do not use parallel links and thus do not suffer reordering.

Figure 1 shows some of the results from polling one site regularly. The particular results are for three days in early 1998. The figure should be compared with figure 4, which shows the load curve at MAE-East over the same three days.

The particular dates were chosen because the vividly illustrate both the relationship of load and reordering and confirm that reordering was occuring at MAE-East. On the third day, between 10 and 11 PM on February 2nd, a private peering relationship was established between the two ISPs on the path, bypassing MAE-East. Observe that the load curve tracks the SACK curve until the private link was put in place, after which reordering largely (but not entirely) went away.

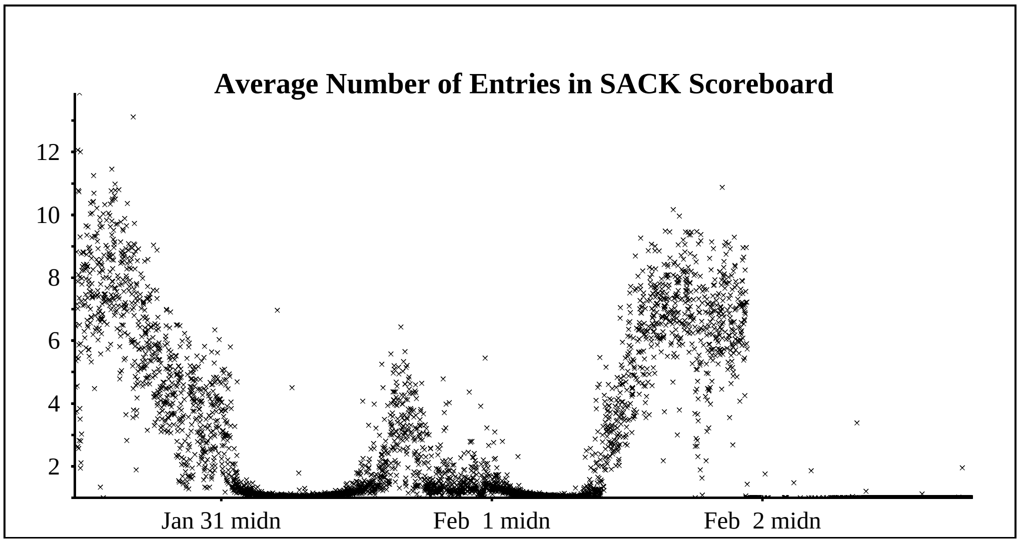


Figure 1:

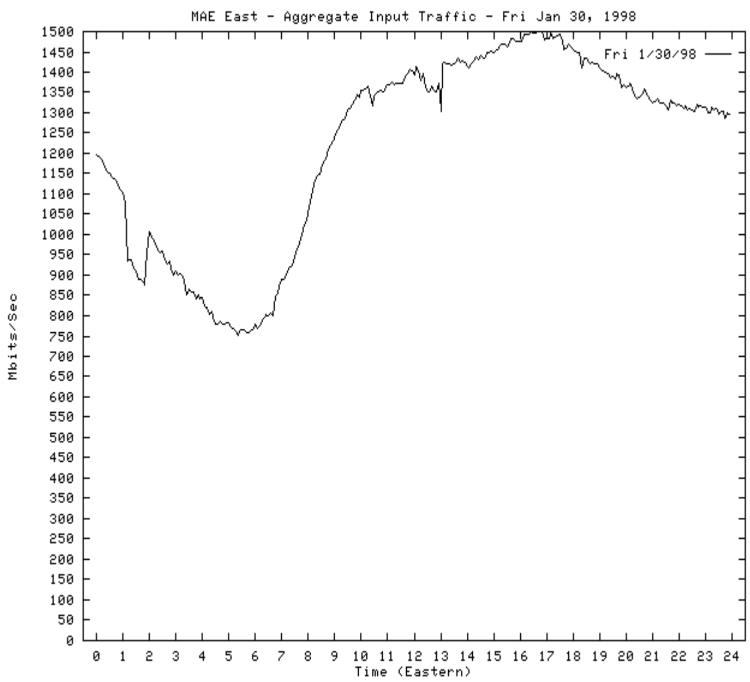
MAE East - Aggregate Input Traffic - Fri Jan 30, 1998
Fri 1/30/98
Mbits/Sec
Time (Eastern)
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
1150
1200
1250
1300
1350
1400
1450
1500
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Figure 2:

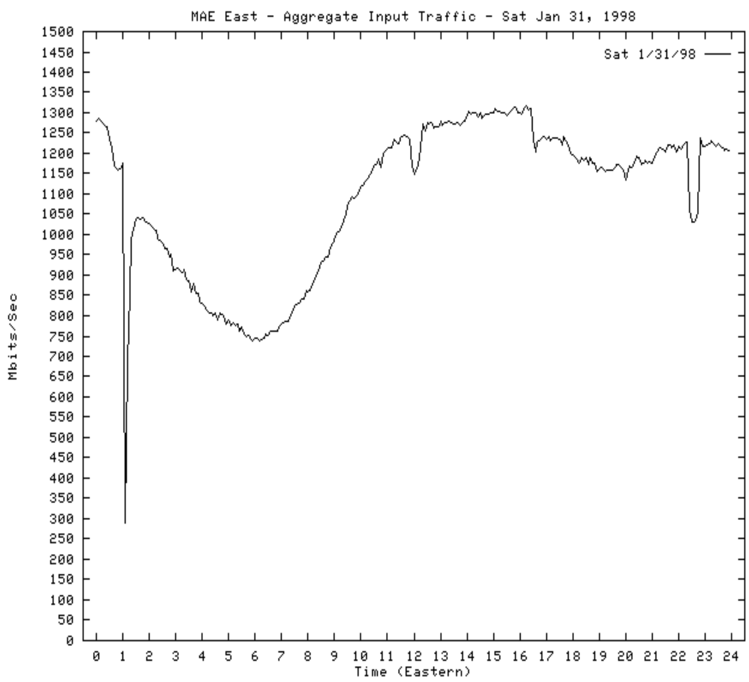
MAE East - Aggregate Input Traffic - Sat Jan 31, 1998
Sat 1/31/98
Mbits/Sec
Time (Eastern)
1500
1450
1400
1350
1300
1250
1200
1150
1100
1050
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Figure 3:

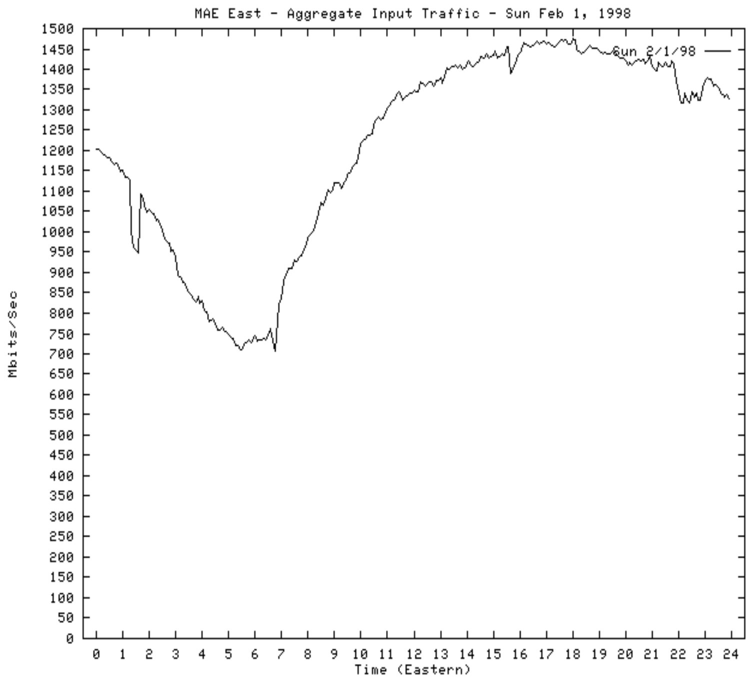
MAE East - Aggregate Input Traffic - Sun Feb 1, 1998
Sun 2/1/98
Mbits/Sec
Time (Eastern)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950 1000 1050 1100 1150 1200 1250 1300 1350 1400 1450 1500

Figure 4:

# 3 Understanding the Causes of Reordering

We believe that reordering, eventually, will be a fact of life on the Internet. It is a natural result of the increasing use of parallelism in network devices and surprisingly hard to prevent.

To explain our views, this section start with a practical example: why reordering occurs at MAE-East. The section then proceeds to discuss the challenges in eliminating reordering.

## 3.1 Hunt Groups at MAE-East

To the best of our knowledge MAE-East is currently the largest ISP peering point in the world in terms of number of bits per second of traffic[3] and handling perhaps as much as a third of *all* long-haul Internet traffic [2] The primary network device used to switch this volume of traffic is the DEC Gigaswitch/FDDI which is a multiport FDDI switch with a crossbar based switch fabric. A particular feature of the Gigaswitch is it's support of what DEC calls *hunt groups*. A hunt group is a collection of ports which operate as a virtual link, that is they allow two layer 2 bridges to be connected by multiple parallel links for increased bandwidth capacity.

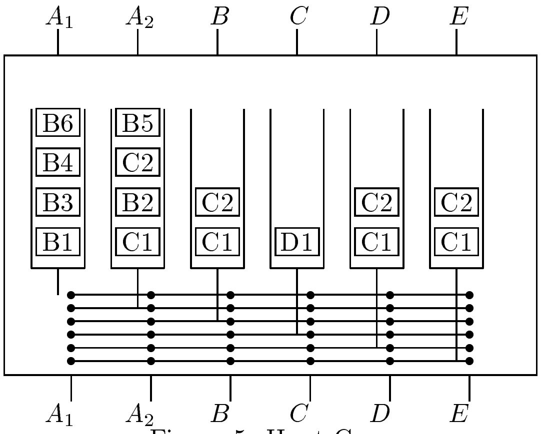

Figure 5: Hunt Groups

Figure 5 is a simplified diagram of a Gigaswitch with six links labeled $A_1, A_2, B, C, D, E$. The first two links have been labeled with subscripts to indicate that they are members of the same hunt group, so any packet which wants to go over virtual link '$A$', can take either link $A_1$ or link $A_2$. The input buffers on the Gigaswitch/FDDI are single FIFOs which results in *head of line blocking* which, while it increases reordering, it is not the fundamental cause of reordering. In the figure each buffer contains a sequence of packets labeled with the letter of the output port for which they are destined and a transmission sequence number. In this example, assume that all packets between a given input port/output port pair belong to the same 'flow'. Thus one can see that packets belonging to the flow between the input hunt group ports $A_1, A_2$ and the output port $B$ has been split between the two hunt group ports, but that the packets were still effectively received in order.

Because the input buffers are FIFO, they always try to send the packet at the front of their queue to its output port. Switch allocation is done through a bidding system, in which the input requests to connect to the output and waits for the output port to to *grant* the connection. The output port may receive requests from multiple input ports but may only grant one because the crossbar can only connect one output port to one input port at a time. While an input port is waiting for a grant it can not send any packets due to the FIFO nature of the buffer.

Now consider the behavior of the switch shown in Figure 5. For simplicity, assume that all the packets are the same size so that we can speak in terms of *time slots*[4] In time slot 1 only port $A_1$ has a packet for port $B$, so it will certainly get a grant to send packet $B1$ to port $B$. In time slot 1 ports $A_2, B, D$ and $E$ all have packets bound for

[3] As of this writing the average load at MAE-East is *continuously* greater than 1Gbps for the better part of the working day as reported on the MAE-East statistics web page http://www.mfst.com/MAE/east.html

[4] In fact if we consider the general case of different sized packets we see another case where even without FIFO queuing we can see very large degrees of reordering. If an input $A$ has packets bound for two outputs $C$ and $D$ and is sending a long packet to $C$ when $D$ finishes sending a packet it received from a different input, $B$ which has a long stream of packets bound for output $D$, the only way that $A$ can send its next packet to $D$ is if $D$ waits for $A$ to finish sending its packet to $C$. If $D$ starts accepting a new packet from $B$ while $A$ is busy sending to $C$ then in fact $A$ may never send a packet to $D$. In such a switch the only way to fairly arbitrate amongst the different ports is for the outputs to be *non-work conserving*.

port $C$. Assuming a fair arbitration scheme there is only a 1 in 4 chance that port $A_2$ will get to send its packet to port $C$. If port $A_2$ does not get to send a packet to port $C$ then in time slot 2 port $A_1$ will send packet $B_3$ to port $B$ while packet $B_2$ is still sitting in the queue at port $A_2$!. Even if port $A_2$ sent packet $C_1$ during the first time slot, there is only a 1 in 2 chance that port $A_2$ will get to send packet $B_2$ during time slot 2.

The reordering can be quite dramatic. For instance, the DEC Gigaswitch Operation Manual version 3.2 notes that hunt groups may cause packet reordering which may cause TCP to incorrectly retransmit packets and that to prevent this users may wish to set TCP's duplicate ack retransmission threshold *tcprexmtthresh* to 100! (Compare this value with the more common value of 3 that is widely used).

At this point, one may be tempted to simply declare that the Gigaswitch has a bug. That would be a mistake. Reordering is, in fact, a natural consequence of parallelism. To explain why this might be so, the next several subsections discuss various approaches to solving the reordering problem, and why they generally fail.

## 3.2 Fixing the Switch

Suppose we try to fix the Gigaswitch's switch to maintain packet ordering. In particular, we want to find a way to allow traffic to be split across two input ports of the switch and yet stay in the order they were received. (We leave to later the question of problem of ensuring that the order packets are received is the same as the order in which they were transmitted). Further, we would prefer to do it in a work conserving way. By *work conserving* we mean that no part of the switch is idle when there is data that could be sent through that part of the switch. We prefer a switch to be work conserving because a switch which is non-work conserving typically has to run faster to handle the same load of traffic. Head-of-line blocking is an example of a non-work conserving system.

Packet switch architectures can be generally classified into one of four types: (1) shared buffer; (2) output buffered; (3) multistage; and (4) input buffered. Multistage switches look particularly unsuited to this problem. They have buffering and queues inside the switch and, as a result, packets from different inputs can experience wildly different delays through the switch. But the other three switches, at least at first glance, appear amenable to retaining order.

### 3.2.1 Shared Buffer and Output Buffered Switches

In a shared buffer switch the switch fabric is comprised of a central memory system. In the case of a *non-blocking*[5] switch, must have an I/O bandwidth equal to the sum of the *bi-directional* link capacities.

In an output buffered switch, the switch fabric is a trivial copy network, which delivers a copy of every packet received to all the output ports. Each output port has a device called a *concentrator* which is responsible for filtering out packets not destined for the output. The concentrator's input must have an I/O bandwidth equal to the sum of the inbound link capacities. Because several packets for the output may arrive at the same time, the concentrator's output must have an I/O bandwidth which is much greater than the output link bandwidth.

Clearly both of these switches can maintain packet order. Packets will enter the shared memory in the order received on a shared buffer switch, and with appropriate management, can be encouraged to leave in order. Similarly, packets arrive in the output concentrator in the order received and presumably the concentrator can maintain the order. Unfortunately, neither switch design scales well.

The limiting factor in building shared memory switches is the speed of memory chips and the physical/electrical difficulties of multiplexing all the inputs into the memory. For example consider a shared memory ATM switch using SRAM with a 10ns cycle time. Even if the memory is 424 bits wide so that an entire cell can be written into the memory in one cycle, and an entire cell can be read out from the memory during the next cycle, the switch would only be able to process 50 million cells per second for a total capacity of about 20Gbps. While this could be doubled with dual ported memories and an IP router might be able to use some what wider memories, these are only constant factor improvements.

While the challenge of building a concentrator is substantially simpler, we are still bound, on the outbound side of the concentrator by the limits of memory speeds.

In short, even though they can maintain ordering, these switch designs are not expected to scale up to the speeds the Internet requires.

### 3.2.2 Input Buffered Switches

Input buffered switches have the majority of their buffer capacity located on the input side of the switch fabric. The primary reason for a switch to be input buffered is so that the switch fabric and buffers only need to run at,or slightly

[5]It is important to understand the definition of the term *non-blocking* since outside of academic circles (read "in marketing literature") this term has been greatly abused in recent years. We use the classical circuit switch definition of *non-blocking*, which is that at a given instant a non-blocking switch is capable of connecting all inputs to all outputs in an arbitrary *permutation*.

above the speed of the fastest link as opposed to output buffered switches where the fabric and the buffers must run at $n$ times the link speed for an $n$ port switch. Currently the most popular switch fabric for implementing input buffered switches is the *crossbar*.

The challenge in constructing a fast (or large) crossbar is designing the switch *arbiter*. The arbiter controls the connection of input ports to output ports. Using the best algorithms, the complexity of the arbiter is a roughly linear function of the product of the number of ports and the number of bids each port may submit. To eliminate head-of-line blocking, most input buffered switches require each input port to bid in parallel for all output ports for which it holds packets. A good arbiter also has the property that the way it services competing bids appears, for practical purposes, to be near random. In particular, no one port should be predictably favored over another.

The GigaSwitch is an input buffered switch and as sections 2 and 3.1 showed it suffers from reordering. The reason (beyond head-of-line blocking effects) is that the arbiter will randomly mix traffic from $A_1$ and $A_2$. And there isn't an obvious way to solve this problem. (The one apparently obvious solution, namely having the transmitter sequentially number the packets it sends on $A_1$ and $A_2$ doesn't work very well. It reintroduces head-of-line blocking, in a new form, and makes the arbiter hideously complex to boot).

Observe the problem here is cost and complexity. We might be able to build a input buffered switch that maintained ordering, but it would be more expensive.

### 3.2.3 Make the Switch Ports Faster

Another approach to fixing reordering is to make the switch ports faster, and attach multiple interfaces through one interface. So the links for $A_1$ and $A_2$ come into the same interface board. (Note that speeding up the ports does not imply the overall switch must be faster; we can have fewer ports that are clocked at a higher bandwidth).

By keeping all the links in a hunt group on the same board, the problem of reordering becomes a purely local problem between the transmitting and receiving interface boards. It is still not a simple problem (see Appendix I), but good algorithms to preserve order exist [1], including one that is work-conserving [11].

Unfortunately, this solution is also somewhat limiting. The maximum bandwidth that can be bundled together is the amount that can be fit on one interface card. In situations like MAE-East where a few core switches may need a large amount of bandwidth between them, this solution may not serve.

### 3.2.4 What Can IP Do?

Yet another approach is to ask whether the IP layer can do anything to help solve this problem. It turns out that IP can help the problem a little bit, but also that current trends in forwarding engine design may actually cause more problems.

If IP is made aware of the presence of the parallel links, and can direct which link each packet is sent over, then IP can ensure that traffic between a particular source and destination always goes over the same link. This trick is easy to do (the common approach is to use a simple hash of the source and destination, modulo the number of parallel links) and ensures that no conversation sees reordered packets. But it has a number of pitfalls. First, it means that no one conversation can use the entire bandwidth of the parallel links. Second, depending on the mix of conversations and the quality of the hash function, there may be uneven assignments of conversations to links. In an extreme case, one might find all the conversations sharing one link while the others are idle.

Looking forward a bit, we can also see a potential problem for the IP layer. In the past year, several vendors have developed inexpensive ASIC implementations of IP forwarding, each capable of forwarding one to eight million packets per second. As OC-48c interfaces become available, some vendors may need to use two or more forwarding ASICs to get enough forwarding power to keep pace with OC-48c's 2.4 Gb/s data rate. That situation creates a new opportunity for reordering, if two successive packets are handed to different ASICs. There are plenty of scenarios in which, depending on the state of the two ASICs, one ASIC may process the same header faster than another, especially if the ASICs keep a recently-used route cache. The hashing trick works here too. We can hash the headers to allocate them to ASICs. But the same problems exist too  we can easily poorly utilize the ASICs.

## 3.3 Numbering Packets

So far we have demonstrated that getting a parallel system to naturally keep packet order, without introducing inefficiencies and higher costs, is very hard.

We conclude this section by considering the obvious question of why not just number the packets on transmission and then sort them back into order upon receipt? (And do the analogous thing for IP forwarding ASICs).

That approach works, and indeed, there even exist good algorithms which do the numbering implicitly, at some risk of reordering[1]. But there's an annoying limitation. As Appendix I proves, for a broad range of situations, a scheduling system that keeps packets in order over parallel links must be non-work conserving. That means there's some inefficiency in the system; we are wasting some system capacity to retain ordering.

All is not completely lost. There exist work conserving algorithms to keep the packets in order. But they require us divide the packet processing into uniform units of work. To illustrate this idea, suppose we striped packets bit-by-bit onto two parallel links and removed them bit-by-bit at the opposite end. Ignoring the problems of errors and the inevitable slight variations in link delays, this scheme delivers packets in-order and is work conserving. An approach based on this idea exists for multiplexing parallel links[11]. Applying this idea to forwarding ASICs, or switch schedulers remains an unsolved problem.

# 4 Impact of Reordering on TCP

At first glance it may be unclear why packet reordering should adversely effect the performance of TCP. TCP is a reliable transport protocol[14], designed to recover from all sorts of misbehavior at the IP layer. As a form of misbehavior, reordering looks pretty tame since all TCP segments eventually get to the right place. The problem is that large scale packet reordering tends to drive TCP into two destructive modes. In one mode, it has great difficulty opening its congestion window and makes inefficient use of available link bandwidth. In the other mode, it loses self-clocking and becomes highly bursty.

There are two types of packet reordering that we will examine, data reordering or *forward path reordering*, and ack reordering or *reverse path reordering*. We consider these separately for two reasons. First, the asymmetric nature of Internet routing [12] frequently results in connections which only experience reordering in one direction. Second, the impact of the reordering is different depending on which direction it takes place over. When data is reordered the result is the transmission of duplicate acks, but when acks are reordered the result is the reception of out-of-order non-duplicate acks.

## 4.1 Forward Path Reordering

In forward path reordering, TCP segments containing data arrive at the receiver out of order. Reordering of data has five effects: (1) reordering causes unnecessary retransmissions; (2) reordering makes it difficult or impossible to grow TCP's congestion window (*cwnd* and *ssthresh*); (3) reordering can cause actual packet losses to be obscured, so that TCP retransmits lost packets later than it normally would; (4) reordering may cause the round-trip estimator to poorly estimate the round-trip time; and (5) reordering may reduce the efficiency of the receiving TCP. After briefly presenting the general behavior of a TCP receiver, we discuss each of these effects in turn.

### 4.1.1 What a TCP Receiver Does

When TCP is receiving data segments, its behavior is very simple. It acks the data it has received. If the data is in order, the acknowledgment is a simple ack which indicates the receiving TCP has received all data through the last byte of the most recent segment.

If the data is out-of-order, things get more complicated. At minimum, the receiving TCP sends a duplicate acknowledgment, which repeats the acknowledgment of the last in-order byte received. If the TCP implements selective acknowledgments (SACK [9]) the ack will both acknowledge the last in-order byte, and also selectively acknowledge the new data received.

The effects of reordering in the forward path are determined by how the TCP sender responds to these various acks.

### 4.1.2 Unnecessary Retransmissions

Most TCPs implement an algorithm known as fast retransmit [15] to try to recover quickly from occasional packet loss. The fast retransmit algorithm is triggered by a series of duplicate acks. If the TCP sender sees three duplicate acks, it assumes that the data immediately after the byte being acked has been lost, and retransmits that data. The idea is that the duplicate acks indicate a packet has been lost, and by quickly retransmitting the data, the sending TCP can repair the loss before it causes the current pipeline of data to drain.

If the duplicate acks are caused, however, by reordering, then the fast retransmission is unnecessary and wastes bandwidth by sending the same data over the wire twice. TCP implementations with FACK behave even less well because FACK does a fast retransmit if the sender receives an ack that suggests that three segments worth of data are outstanding.

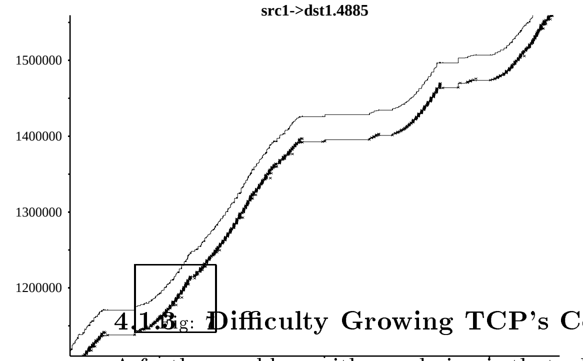


Figure 6:

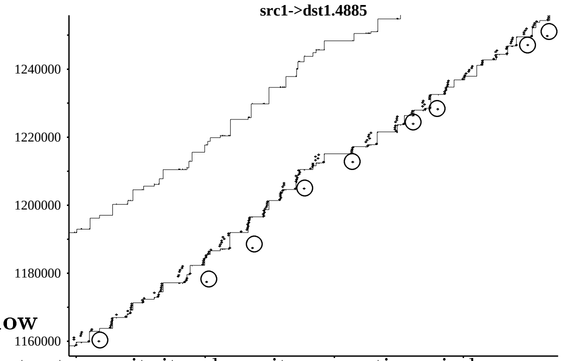


Figure 7:

#### 4.1.3 Difficulty Growing TCP's Congestion Window

A further problem with reordering is that when a TCP does a fast retransmit, it reduces its congestion window *cwnd*, in the belief that it has seen packet loss. Unfortunately, if there are multiple fast retransmits in a single window (which reordering can trigger), the congestion window is quickly driven down to a very low value. The result is that TCP has difficulty opening its congestion window and properly utilizing the link.

This behavior is illustrated in Figures 6 and 7. Both are plots of part of a 2.1 MB file transfer passing through MAE-East. Observe that in Figure 6, the usual neat exponential curves of TCP congestion control are interrupted by an extended period of slower growth. Figure 7 shows part of this period in detail. The TCP is suffering from forward path reordering, and the duplicate acks are causing regular fast retransmits that reduce the congestion window. The particular behavior is that any time the sender opens its window to about 8 segments, segments are reordered, by three or more positions. The resulting spurious fast retransmissions are circled. Almost immediately after most of them, the original transmission is acked, and TCP emits new segments but into a reduced congestion window. It is sobering to observe that in the half second this graph covers, of the 10 fast retransmissions observed, all but one (that at time 46.475) are unnecessary.

#### 4.1.4 Obscured Packet Loss

What happens when along with reordering, a segment actually gets lost? The unfortunate answer is that TCP often does not initially see the loss because the reordering conceals it. The sending TCP is forced to wait for a timeout to retransmit the loss.

An example may be useful. Suppose the sender transmits segments 1 through 6, and that segment 3 is discarded, and segment 2 is reordered so it arrives after segment 6. The receiver will generate an in-order ack for segment 1, and then three duplicate acks for segment 1 as it receives 4, 5 and 6. These duplicate acks will cause the sender to perform a spurious fast retransmit of segment 2. When segment 2 finally arrives at the receiver, the receiver generates an ack for segment 2, but since there are no more segments in flight, no duplicate acks will be generated, and the sender will not be aware, until it times out, that segment 3 is missing. If the segments had not been reordered, then the duplicate acks would have been for segment 2 and spurred a fast retransmit of segment 3. Figures 8 and 9 show examples of obscured packet losses. In each figure the reordered packet and its false retransmission are circled. In both cases there is a packet lost before the reordered packet is received as a result all the acks generated are duplicate acks for the reordered packet and not the packet which was actualy lost, which in both cases must be recovered by timeout. One of the modifications proposed for FACK TCP, moving the duplicate ack retransmission to the SACK scoreboard so that fast retransmit can be performed for each hole in the scoreboard, would be able to recover from this scenario, but in cases with more severe reordering it would suffer from many more false retransmissions.

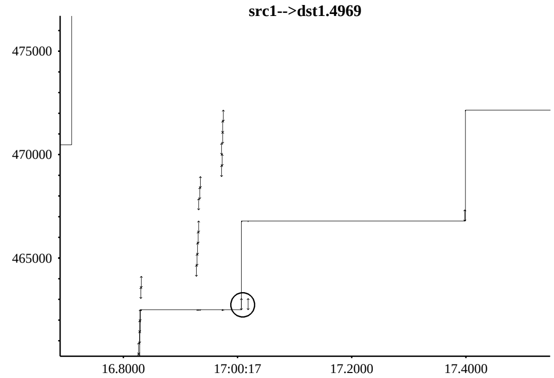


Figure 8:

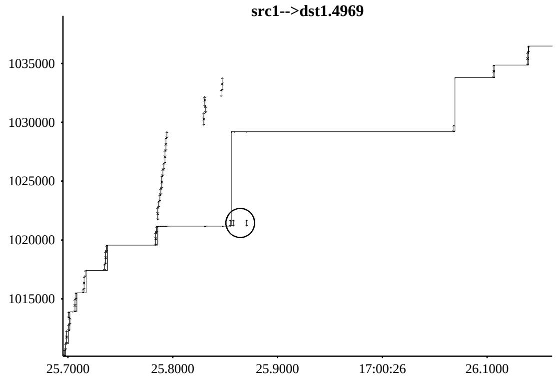


Figure 9:

### 4.1.5 Poor Round-Trip Time Estimation

Another impact of reordering is that it can, at least in theory, cause some TCP round-trip time estimators to stop working due to lack of round-trip time samples.

Most TCP implementations do round-trip time estimation in one of two ways. If the TCP implements the TCP Timestamps Option and the option has been negotiated at connection setup, then the sending TCP estimates the round-trip times by subtracting the timestamps echoed in the acks from the present time to get an accurate round-trip time sample [5]. This algorithm works correctly even in the face of reordering.

However, if TCP timestamps are not in use, most TCPs use an algorithm that samples one round-trip time per window of data. The usual implementation is to note the sequence number of the first byte of the segment currently being sent, start a timer, and then sample the timer when an ack that includes the sequence number of that byte is received. However, if the byte being sampled is retransmitted, the sample must be discarded, to avoid a recurrence relationship [6, 7].

Unfortunately, if we are spuriously doing fast retransmits, we are likely to have to discard most of our potential samples. As a result, the round-trip time estimator will not sample the round-trip time very frequently and will not keep a good estimate of the round-trip time. And keep in mind that due to obscured packet loss, performance will be affected by the goodness of the sender's round-trip time estimate.

### 4.1.6 TCP Receiver Efficiency

TCP requires that the receiving application receive data in order. As a result, network reordering of data segments places serious burdens on the TCP receiver.

First, the receiver must buffer out-of-order data until data is received to fill gaps in the sequence space. At some stage this process requires sorting data into proper order, an expensive operation that is never required if data arrives in order.

Second, the data being buffered is being withheld from the receiving application. Rather than getting its data in a smooth series of modest chunks, the application gets the data in bursts. The overall efficiency of the system suffers because it is not work conserving.

Third, many TCP implementations use the *header prediction*[4] algorithm to reduce the costs of TCP processing. Header prediction, however, only works for in-order TCP segments. So if segments are reordered, most TCP implementations do far more processing than they would for in-order delivery.

## 4.2 Reverse Path Reordering

In reverse path reordering, the acks traveling back to the sender are reordered. For the purposes of this section, we will assume that no reordering affects the forward path, since that simplifies the discussion.

It is important to observe that the ack patterns are very different for forward and reverse path reordering. In forward path reordering, the reordering of data segments causes the receiver to send duplicate acks. In reverse path reordering, the receiver is sending cumulative acks (because data is arriving in order), but the acks are being reordered on their way back to the sender.

Reverse path reordering has one major effect: a loss of self-clocking leading to highly bursty transmission patterns.

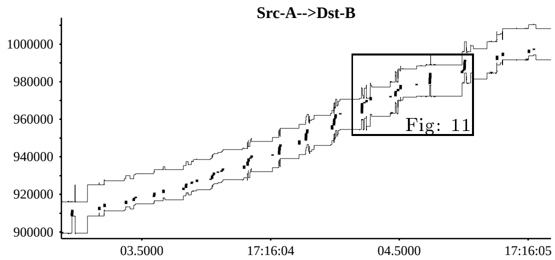


Figure 10:

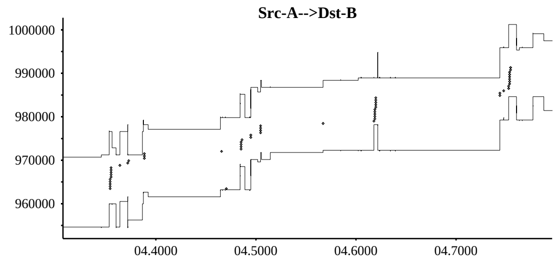


Figure 11:

### 4.2.1 Loss of Self-Clocking

In the late 1980s TCP was found to have a nice self-clocking property: the acks roughly reflected the rate at which data could transit the network and be removed from the network by the receiver[3]. The sender sees a nice sequence of acks, each acking one or two segments worth of data and notifying the sender it is free to inject one or two more segments into the network.

Reordering acks scrambles self-clocking. The problem is that, because all the acks are cumulative, reordering acks means that an ack for a later byte is arrived early. What the sender sees is an occasional ack, acknowledging a large amount of data, followed by a number of apparently redundant acks which acknowledge data that the first ack already acknowledged. This ack pattern has at least two harmful effects.

First, it affects the rate at which the sending TCP opens its congestion window. The sender opens its congestion window for each ack that acknowledges previously unacknowledged data. The window is opened by one maximum segment size per ack during slow start, and by a fraction of the maximum segment size during congestion avoidance. The goal is to roughly double the window size per congestion window's worth of segments during slow start, and to grow the congestion window by one segment per window's worth of data after slow start.

But if acks are reordered, there may be only one or two acks that acknowledge new data, even if tens or hundreds of segments have been sent. The unfortunate result is that the congestion window will grow far too slowly (e.g., one tenth or hundredth of its intended rate). Particularly severely hurt by this behavior are long delay bandwidth paths like satellite paths and high-speed transcontinental links.

The second effect of ack reordering is to encourage bursts of segments. An in-order ack frees up space within the window, and permits the sender to transmit an amount of equal to the amount acknowledged. So if acks are reordered and one ack arrives early and acknowledges a large amount of data, the sender is likely to launch a volley of new segments all at once. Some spacing of the segments will occur at bottleneck links, and because most receivers ack only every other segment, but it does led to closely spaced acks and if ack compression occurs on the reverse path, makes more ack reordering likely. In short, this behavior is self-reinforcing at least for modest periods. If the sender gets an out-of-order ack, the sender will inject a burst of data that makes receiving another out-of-order ack more likely. Note that this effect has almost nothing to do with the congestion window (which the first effect modifies).

Figures 10 and 11 show this self-reinforcing behavior in different levels of detail. The trick is to look at the bottom line, which shows the sequence number acknowledged by the latest ack. Cases where the line goes up and then down again reflect acks which have been reordered, such that they arrive early. Figure 10 shows a long interval of reordering. Figure 11 illustrates a particular period of clearly self-reinforcing behavior, where four successive bursts were triggered by the reordering of acks from a prior burst.

## 4.3 Forward and Reverse Path Reordering

What happens when a TCP experiences both forward and reverse path reordering? The traces have so far suggests that the TCP tends to oscillate between forms of misbehavior. In some situations the effects of forward path reordering dominate, and the TCP has trouble doing slow-start. In other situations reverse path reordering dominates and we see bursty behavior. One can see some of this style of behavior in Figure 10 where on the left side of the figure we see the TCP having difficulty opening its congestion window due to forward path reordering, while on the right side we see bursts due to reverse path reordering.

## 4.4 Some Things To Help TCP Work Better

While in the conclusion we will argue that we should do everything we can to avoid reordering, reordering is eventually part of our future. So what can TCP do to better handle reordering?

First, TCP can detect when it mistakenly does a fast retransmit and undo the congestion window changes. The obvious marker of a fast retransmit is that the ack comes back far too soon (typically just after the fast retransmit is done).

Second, the TCP can support SACK. SACKs give the sending TCP a fighting chance to figure out that reordering is taking place by more completely reflecting what is happening in the window (even with reordering). For instance, the SACK specification [9] indicates SACKs should reflect the order in which segments are received. As a result, a sender can judge, based on the ordering of SACKs, whether forward path reordering is taking place and can adjust the three segment threshold for fast retransmit accordingly. (Indeed, if no reordering is taking place, one duplicate ack might suffice, while if reordering is ongoing, it may be 30 or 40 duplicates before we can safely assume a segment is lost).

These two patches, however, are far from a complete answer to the problem.

# 5 Conclusions

We hope we have convinced the reader that reordering is now a part of the Internet's behavior. Furthermore, we hope the reader agrees that eliminating reordering is a hard problem, both economically and theoretically. Given those two observations, how should we go forward?

First, we should try to make reordering rare. Good algorithms exist [1, 11] to minimize or eliminate reordering in some situations. In other cases, algorithms like numbering packets can be used to at least minimize reordering. These algorithms should be used to reduce the frequency of reordering.

Second, we need to begin to accept that some reordering is still going to occur, and probably often enough to affect performance. In short, we need to start designing protocols to recover more gracefully from reordering. We think observing the SACK patterns to learn if reordering is occuring is a nice idea, but it is only a starting point. This topic desperately needs more attention.

# 6 Appendix: Reordering from a Theoretical Perspective

It is also useful to look at reordering from a theoretical perspective. Many (most?) switch and router architects design their equipment to be *work conserving*. Informally, a work conserving system is one where a packet is always sent as soon as the link on which it is to be sent becomes idle. (For a formal definition see [16][8]). A nice feature of work conserving systems is that they have the minimum achievable average queuing delay if we assume the system does not discard packets. Furthermore, a non-work conserving system is clearly wasting some amount of capacity (it is leaving a line idle while work for that line builds up). While the wastage can be small, in many non-work conserving schemes, the wastage (and thus reduced effective bandwidth) can be substantial.

Unfortunately, it is very easy to prove that, for a fairly simple and common set of properties, a system cannot be work conserving and maintain packet ordering while doing link level load balancing. In particular, given the following three conditions:

1. The unit of transmission is the packet (we don't try to break the packet up across links); and
2. The links across which we balance are all at the same rate;
3. Packet sizes vary by more than a factor of 2;

we can prove that no system can maintain packet ordering and be work conserving.

The trick is to observe when the last bit of each packet leaves the system. If the last bit of the second packet leaves before the last bit of the first packet, then the receiving system will either reorder the packets or be non-work conserving, because the receiving system will receive the last bit of the second packet first, and be forced to either (a) delay the second packet awaiting the first packet, which is non-work conserving, or (b) send the second packet on, causing reordering in relation to the first packet.

The proof is very simple and done by contradiction. Assume our system retains packet ordering and is work conserving. Further assume the system is currently idle, and receives packets to send at a bandwidth which is greater than or equal to the sum of the individual bandwidths $b$ of the $n$ outbound links. Now assume the system is receives two packets to transmit. The first packet is one bit more than twice the length of the second packet which is of length $l$.

The equations vary slightly, depending on whether the system starts transmitting each packet when it gets the first bit of the packet, or waits until it has the last bit of the packet. If we begin transmitting on the first bit, then when the first bit of the second packet starts to be transmitted, we will have transmitted $\frac{2l+1}{n}$ bits of the first packet. The second packet takes $l$ bit times to transmit, during which we will also send $l$ more bits of the first packet. Thus, when the last bit of the second packet departs, we will have sent $\frac{2l+1}{n} + l$ bits of the first packet. We know that $n$ is at least 2, so this value is less than $2l + 1$ and thus we will not have completed sending the first packet before the last bit of the second packet departs. The analysis is the same if the system transmits after receiving the last bit of the packet, except that we transmit even less of the first packet (just $\frac{l}{n}$ bits) before the last bit of the second packet departs. In either case, the system completes sending the second packet before completing the first packet. Thus we have a contradiction.

From this proof we can conclude that if we take the straightforward approach to link level load balancing we must either be non-work conserving or reorder packets. Neither is an appealing idea.

# References


[1] H. Adiseshu, G. Parulkar, and G. Varghese. A reliable and scalable striping protocol. In *Proc. ACM SIGCOMM '96*, August 1996.

[2] R. Gareiss. Is the internet in trouble. *Data Communications Magazine*, September 1997.

[3] V. Jacobson. Congestion avoidance and control. *Proc. ACM SIGCOMM '88*, August 1988.

[4] V. Jacobson. 4bsd header prediction. *ACM SIGCOMM Computer Communication Review*, Apr 1990.

[5] V. Jacobson, R.T. Braden, and D. Borman. Tcp extensions for high performance; rfc-1323, May 1992.

[6] R. Jain. Divergence of timeout algorithms for packet retransmissions. *Proc. 5th Phoenix Conf. on Computers and Communications*, March 1986.

[7] P. Karn and C. Partridge. Estimating round-trip times in reliable transport protocols. *ACM Trans. on Computer Systems*, November 1991.

[8] L. Kleinrock. *Queueing Systems; Volume 2: Computer Applications.* Wiley, 1976.

[9] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgement options; rfc-2018, October 1996.

[10] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion control algorithm. *Computer Communications Review*, 27(3), July 1997.

[11] C. Partridge and W. Milliken. Patent application: Method and apparatus for byte-by-byte multiplexing of data over parallel communications links, 1998.

[12] V. Paxson. End-to-end routing behavior in the internet. In *Proceedings of ACM SIGCOMM '96*, Stanford, Ca, October 1996.

[13] V. Paxson. End-to-end internet packet dynamics. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[14] J. Postel. Transmission control protocol; rfc-793, September 1979.

[15] W.R. Stevens. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms; rfc 2001, January 1997.

[16] R.W. Wolff. *Stochastic Modeling and the Theory of Queues.* Prentice Hall, 1989.