The Effect of Packet Reordering in a Backbone Link on Application Throughput

Michael Laor and Lior Gendel, Cisco Systems, Inc.

Abstract

Packet reordering in the Internet is a well-known phenomenon. As the delay and speed of backbone links continue to increase, what used to be a negligible amount of packet reordering may now, combined with some level of dropped packets, cause multiple invocations of fast recovery within a TCP window. This may result in a significant drop in link utilization and hence in application throughput. What adds to the difficulty is that packet reordering is a silent problem. It may result in significant application throughput degradation while leaving little to no trace. In this article we try to measure and quantify the effect of reordering packets in a backbone link that multiplexes multiple TCP flows on application throughput. Different operating systems and delay values as well as various types of flow mixes were tested in a laboratory setup. The results show that only a small percentage of reordered packets, by at least three packet locations, in a backbone link can cause significant degradation of application throughput. Long flows are affected most. Due to the potential impact of this phenomenon, minimization of packet reordering as well as mitigating the effect algorithmically should be considered.

Packet reordering in the Internet is a well-known phenomenon [1]. Packets may be reordered as they traverse different paths in the network or by the networking gear itself.
As measurements of Internet traffic suggest [2], the majority of the traffic sent over the Internet is transported by TCP. Although TCP is a reliable transport mechanism, dropped packets and packet reordering can and do affect its performance, and hence the end application throughput. Even in networks with a large number of flows, packet reordering causes a drop in throughput and response time for some applications. This drop in application throughput is due to TCP's reduction of its window size, resulting in less bandwidth for the application.

TCP implementations today use fast recovery [3, 4] and fast retransmission [3, 4] algorithms when responding to dropped packets and packet reordering. One of the properties of fast retransmit is to filter out reordering of one or two packet positions within a flow, allowing the operation to continue normally in those cases. However, the reordering of three or more packet positions within a flow, as well as packet drop, trigger both retransmit and fast recovery algorithms. While the fast recovery algorithm provides a very efficient recovery mechanism when a packet is dropped, once two or more packets are dropped or reordered in a window, invoking fast recovery multiple times, a significant drop in bandwidth may result.

Today, an ever increasing bandwidth of links is leading to an increase in the bandwidth-delay product and to an increase in the potential TCP window size. What historically amounted to only negligible instances of packet reordering or drop rate in the backbone may now become more significant given the larger window size, and thus may affect application throughput more severely. While some algorithms like selective acknowledgment (SACK) [5] have been proposed to mitigate this problem, most are considered experimental and have not yet become standard TCP implementation in most operating systems.

In light of the potential impact of packet reordering, the question arises of what is the actual effect of reordering in the backbone on application throughput.

In this article we try to measure the effect of packet reordering on application throughput when packets are reordered in a backbone link carrying multiple flow types. We emulated a network in which several clients and servers communicated over a Gigabit Ethernet backbone link running multiple flow types. We took measurements of application throughput while varying the reorder rate on the backbone link and the delay along the path. Different TCP implementations were used for both the clients and the servers.

In the following section we provide an overview of the TCP algorithms that govern TCP window size during data transfer as well as during the recovery operations that occur when packets are dropped or reordered. A discussion of reordering in a backbone link is presented next. We continue with a description of our test setup and procedures followed by a presentation of our results and our interpretation of them. We conclude with a summary, a recommendation for avoiding packet reordering, and a description of potential future work.

An Overview of TCP Behavior During Packet Reordering and/or Packet Dropping

This section describes TCP behavior during data transfer, packet reordering, and packet dropping. Detailed descriptions can be found in Allman *et al.* [3] and Stevens [6].

TCP uses two main flow control mechanisms during data transfer: the receiver's flow control and the sender's flow control. The receiver's flow control is based on the receiver's advertised window size; with each ACK it sends to the sender, the receiver includes its available buffer size (indicating how much data the sender can send at any point).

A variable called the *congestion window* manages the sender's flow control. The congestion window defines the maximum number of segments (specified in bytes) the sender can send without getting an ACK. This value (*cwnd*) is controlled and changed locally by the sender according to the algorithms described below. The actual amount of data the sender can send to the receiver without getting an ACK is the minimum of the receiver's advertised window, and the value of *cwnd*. Thus, at any point in time during a TCP data transfer, either the receiver's or the sender's flow control is dominant.

Several algorithms control the size of the congestion window. Two of them, slow start and congestion avoidance [7], come into effect during every data transfer. Two other algorithms, fast retransmit and fast recovery [4], come into effect during packet dropping or packet reordering. A brief description of these algorithms and how they interact with each other follows.

Slow Start and Congestion Avoidance Algorithms

Slow start defines the way in which *cwnd* is initially set and accumulates its value. Initially, when a TCP connection is established, the congestion window is set to a default value that is defined to be no more than two segments [3].

Because the initial *cwnd* is set to a low value and the receiver's advertised window is normally set to a higher value, it is *cwnd* that controls the amount of data the sender sends, thus forcing the sender into a slow start due to the small window size. The sender increases its congestion window size by one segment with each ACK (*cwnd* is actually specified in bytes, but for simplicity here we refer to its size by number of segments).

The value of *cwnd* grows from two segments to four when the first two segments are ACKed. It then grows from four segments to eight when the four segments are ACKed. This process continues and results in an exponential increase over time of *cwnd*.

At some point a limit may be reached: *cwnd* may grow larger than the receiver's advertised window, or a limit along the path may be reached beyond which the network cannot send the amount of data being sent. Once a path capacity limit is reached, a packet may be dropped. While it makes sense to start slow and increase the window exponentially to achieve maximum throughput quickly, it would be advantageous to reduce the growth rate of the window size before the path capacity limit is reached. This is the point at which the congestion avoidance algorithm takes control of *cwnd*.

The congestion avoidance algorithm was invented and described by Van Jacobson in [7]. The transition from slow start to congestion avoidance is controlled by a variable called *slow start threshold* (*ssthl*). When $cwnd \le ssthl$, slow start is in effect. When $cwnd > ssthl$, congestion avoidance is in effect. Under congestion avoidance, the *cwnd* growth slows down to a rate of 1/*cwnd* per each arriving ACK, resulting in a linear growth rate of one segment for each round-trip time (RTT).

The goal is to try to open the window to the optimal size of the delay-bandwidth product. However, because the sender has no up front information on the actual link characteristics, if *cwnd* was to continue to open the window exponentially and then back off when a packet loss was detected, it could create unneeded fluctuations. This could result in overloading the network at times and underutilizing it at other times. Instead, it would make more sense to slow down the growth rate of the window as path capacity is approached, thus changing exponential growth to linear.

Once congestion is encountered, as indicated by a dropped packet, the value of *ssthl* is set to half of the current window size while *cwnd* is set to one, causing the sender to go back to slow start. As the window size increases beyond the link's capacity, it backs off and then starts again, only slower.[1]

A dropped packet is identified when the time to receive an ACK for a segment times out. During this period the pipe drains. By dropping *cwnd* to one segment the algorithm returns to slow start, underutilizing the pipe.

A more optimized way would be to detect the missing packet as soon as possible and "fill the hole" created by the missing packet while continuing to send the data at the optimal rate according to the congestion avoidance algorithm. This is what fast retransmit and fast recovery aim to achieve. These two algorithms were invented by Van Jacobson as modifications to congestion avoidance and were first described in [4]. Good descriptions can also be found in Allman *et al.* [3] and Stevens [6].

Fast Retransmit and Fast Recovery Algorithms

Any out-of-order packet arriving to the receiver will trigger a duplicate ACK in which the receiver repeats the ACK of the last in-order segment received (hence a duplicate ACK is sent). Any additional out-of-order segment will cause another duplicate ACK. Packet reordering may result in one or two duplicate ACKs. The sender, however, considers three duplicate ACKs to be an indication that the packet was actually dropped and consequently triggers a retransmit of the missing segment. The retransmit occurs without waiting for a timeout. This behavior is defined as fast retransmit [4]. The number of duplicate ACKs that trigger a fast retransmit is defined to be three in [3]. This is the default value in all implementations that support fast retransmit. All implementations also allow the administrator to manually set this value.

Once retransmit takes place the sender enters fast recovery. Fast recovery differs from congestion avoidance in the behavior of the sender during the period between the retransmit of the missing packet and the time a new segment is ACKed (implying the arrival of the retransmitted segment).

Fast recovery differs from congestion avoidance in two primary ways. First, instead of closing the window setting *cwnd* to a value of one, the window's size is set to equal the value of *ssthl* + the number of duplicate ACKs. (Note that *ssthl* is halved in the same way as during congestion avoidance). Second, as the sender continues to receive duplicate ACKs for each packet that was in flight ahead of the retransmitted packet and is now arriving at the receiver, it opens *cwnd* by one segment for each received duplicate ACK. (This is sometimes referred to as the window being *inflated* as it resumes exponential growth). This behavior continues until the retransmitted segment is ACKed. At this point, the sender exits fast recovery, the *cwnd* value is set to equal *ssthl*, congestion avoidance is initiated, and *cwnd* returns to a linear growth pattern.

Fast recovery is designed to achieve several goals. First, as *cwnd* is kept open it avoids emptying the "pipe" while recovering from a lost packet. Once a retransmitted packet is sent, each additional duplicate ACK represents a packet that was in flight, has left the network, and is now in the receiver's buffer. Thus, there is capacity to send another packet. If we were to go back to slow start, the window would be closed, and no new packets could be sent for each additional duplicate ACK until the window size increased again.

Second, after *cwnd* is halved it is set to a value that initially points to packets that were already transmitted. Therefore,

[1] *Note that with each packet drop we switch to congestion avoidance earlier, due to halving* ssthl*, and thus the growth rate is reduced because the point of switching to linear growth is earlier each time.*

when duplicate ACKs arrive the window grows by one segment with each ACK, but no new packets are sent because they were sent already. This allows for a period of time during which the sender does not send any more data into the network and any short-term congestion in the network is cleared up. When the window grows beyond the points of packets that were already sent, new packets are sent with each duplicate ACK. This continues until a new segment is ACKed, indicating the arrival of the retransmitted packet. At this point the sender exits fast recovery and enters congestion avoidance.

The amount of data sent during the fast recovery period is identical to the amount of data that should have been sent if we were halving the window and continuing with congestion avoidance following the packet drop. Hence, we fill the hole while continuing to send the same amount of data. This also allows avoidance of a burst of packets when exiting fast recovery because no catch up is required in order to fill the pipe.

High Reorder and Drop Rate Effects in the Presence of a High Bandwidth-Delay Product

A known property of fast recovery [4] is that while it recovers smoothly from a single packet drop within a window, a drop of more than one packet in the same window causes the throughput to degrade sharply because the window size decreases too. This may occur due to one of the following mechanisms:

- A sender exits fast recovery as a new segment is ACKed, indicating the arrival of a retransmitted segment. However, a sender has no way of knowing if two packets were actually dropped. In this case, when the retransmitted segment arrives at the receiver, the receiver ACKs all the way up to the second missing segment. This is called a *partial* ACK. When the sender receives this partial ACK, it exits fast recovery because it does not know another packet is missing. When the receiver continues to send duplicate ACKs for the second missing packet, the sender soon enters fast recovery again, halving the *cwnd* window again. This results in a reduction of *cwnd* to one quarter of its original size, slowing down the sender.
- When more than two packets are dropped from within the same window, the sender can actually stall such that it has to wait for a timeout for the third dropped packet. During this period the pipe gets emptied because no packets are sent and the *cwnd* goes back to one segment and slow start. This leads to significant degradation in throughput. Examples of such cases can be found in Bennett *et al.* [1] and Fall and Floyd [8].

Several algorithms have been proposed to overcome this problem such as SACK [5, 8, 9], which tells the sender exactly which blocks were received in order to avoid resending segments that were received already. Duplicate SACK (D-SACK) [10, 11] which is an extension of SACK, defines a method in which the receiver actually reports duplicate received segments, allowing the sender to infer reorder and potentially increase the threshold for fast retransmit value, and readjust the window that had been unnecessarily reduced. Other suggested approaches rely on retransmit during partial ACK when a partial ACK signals that a segment is missing. Common to all these approaches is the attempt to avoid exiting fast recovery before all the missing segments are ACKed, thus avoiding a reslicing of the window size for each missing segment.

D-SACK allows the sender to not only avoid reslicing the window but adds the ability to actually reopen the window when it detects that it was closed due to a packet reordering rather than a packet drop. D-SACK also provides the ability to respond to a heavily reordered network by increasing the fast retransmit threshold. However, some of these algorithms are considered experimental and are not a standard default in existing TCP implementations.[2] As such, with the increase of bandwidth*delay, there is an increased likelihood of experiencing a sharp drop in throughput in networks that are congested or in which packet reordering is introduced.

Packet Reordering in Backbone Links

Packet reordering is a phenomenon that exists in large networks. Measurements of a high degree of reorder rate and an analysis of its cause and impact are described in Bennett *et al.* [1].

Reorder may be the result of packets from the same flow traversing different paths (due, e.g., to per-packet load balancing in a router), where each packet experiences a different propagation delay. Reorder may also be caused by networking gear along the path (routers, switches, etc.) that reorders the packet stream.

Given a certain probability for a packet to be reordered along a single link, and given that the backbone link multiplexes a large number of flows, the probability of a particular flow being affected is low. Even if a packet location is changed within the packet stream, only a few of the affected packets, if any, change location within their own flows; exchanged packets may be from different flows.

Furthermore, even when a packet is reordered within its own flow, given the high degree of multiplexing involved, the packet is likely to move only one packet position. This means it has changed places with the packet next to it in the flow. Even fewer packets will move by two packet positions, because this implies three packets from the same flow in close proximity in a stream in which a reorder mechanism exists to create such a shift. The probability for a larger shift in packet position within the same flow continues to decrease.

Shifting one or two positions causes only one or two duplicate ACKs, which does not affect the sender's operation (other than not growing *cwnd* when a duplicate ACK is received). Only the much smaller number of packets shifted by three or more locations within their flows will trigger fast retransmit and fast recovery, and actually affect throughput.

On the other hand, some factors contribute to an increase in the reorder rate. Per-packet load balancing, in which a router makes use of parallel paths to a certain destination to load balance the incoming traffic among those paths, may result in reorder because each path may have a different propagation delay.[3]

Reorder may also be additive in the sense that within a large network we may encounter several places in which packets are reordered. This may result in the same packet shifted several times and thus several locations by the time it arrives at its destination host. This can also act to cancel out the effect, however.

Thus, while reorder clearly exists in the network [1], the

[2] *None of the operating systems in our test laboratory used D-SACK algorithms. Only one used SACK (Linux 2.2.6 Red Hat 7.0). See below for more details.*

[3] *For this reason some network operators often deploy a per source/destination load balance that guarantees that the same flow will always use the same path. Most modern routers support this feature. This approach may present other inefficiency issues, but it solves the reorder problem.*

actual magnitude of reorder in backbone links that affect application throughput is hard to evaluate.

The difficulty arises with the significant increase in bandwidth of backbone links as well as that of "last mile" technology, along with large delay values (e.g., the availability of Gigabit Ethernet on servers, faster trans-Atlantic links, 10 Gb/s links across the continental United States, and faster last mile technologies). These developments contribute to a significant increase in the bandwidth-delay product and, in turn, TCP window size. While the growth of TCP window size is limited by receiver buffer size, the availability of larger memories in hosts along with various TCP tuning tools that increase receiver buffer size contribute further to the increase of the TCP window.

As the TCP window becomes much larger, a small percentage of three or more packet reordering, which at lower bandwidth is not significant, may now combine with some incidence of dropped packets to constitute a scenario of several instances of three duplicate ACKs within the same window. This may lead to a significant drop of throughput, as discussed in the previous section.[4]

Another difficulty when evaluating the effects of packet reordering is that reordering is a silent problem. While most routers count packet drops and can show this information to the network operator, packet reordering is invisible to routers and leaves no trace, so the operator cannot identify the potential source of throughput degradation. Thus, packet reordering can affect the throughput of some applications without leaving much of a trace, if any.

Measuring Effects of Packet Reordering

In this article we have tried to measure the actual effect of packet reordering in a backbone link on end application throughput. Using a laboratory setup, we forced packet reordering in a backbone link that multiplexed multiple flows.

We applied different packet reordering rates and varied the delay of the link and the mix of flows. The effect on the end application throughput was then measured for each case.

To isolate the effects of reordered packets, we tested in an essentially lossless environment. This means that no element in the sender to receiver path dropped packets due to congestion. Therefore, the only reason for duplicate ACKs in the test scenario would be due to packet reordering, which we forced artificially on the backbone link.

Various operating systems and TCP implementations were used in the end hosts in order to ensure that we observed the actual effects of packet reordering rather than the behavior of a particular implementation. Using multiple operating systems also allowed us to evaluate the efficiency of the various implementations in handling packet reordering and created a more realistic environment.

[4] *It is important to note that the bandwidth in the bandwidth*delay product refers to the "bandwidth of the narrowest pipe" along the path between two hosts (which dictates the actual bandwidth). Thus an increase in bandwidth in the backbone or any other point will only effect the bandwidth*delay product of a particular pair of hosts if it increases the bandwidth bottleneck along the path between them.*

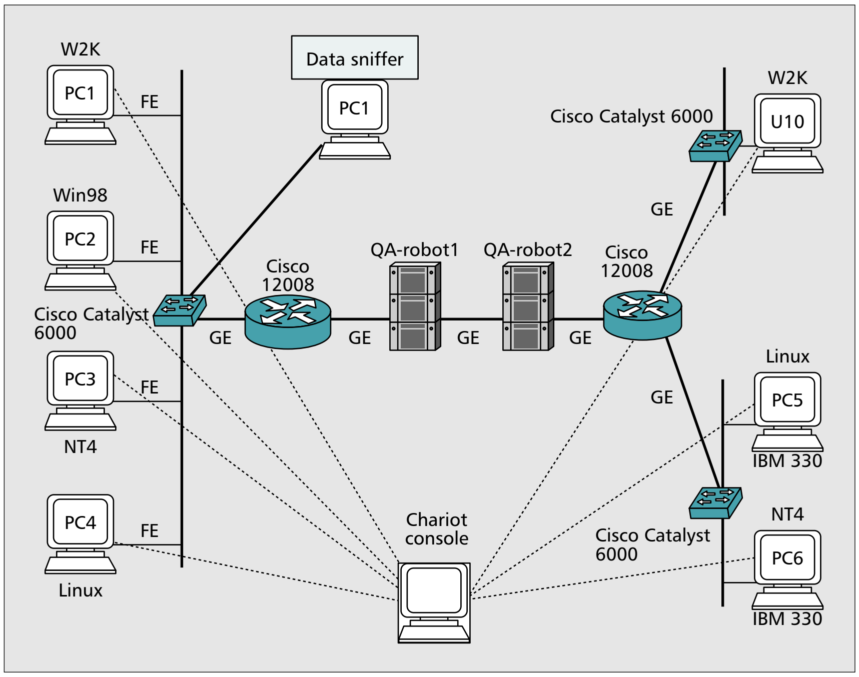


Figure 1. *The test setup.*

Test Setup

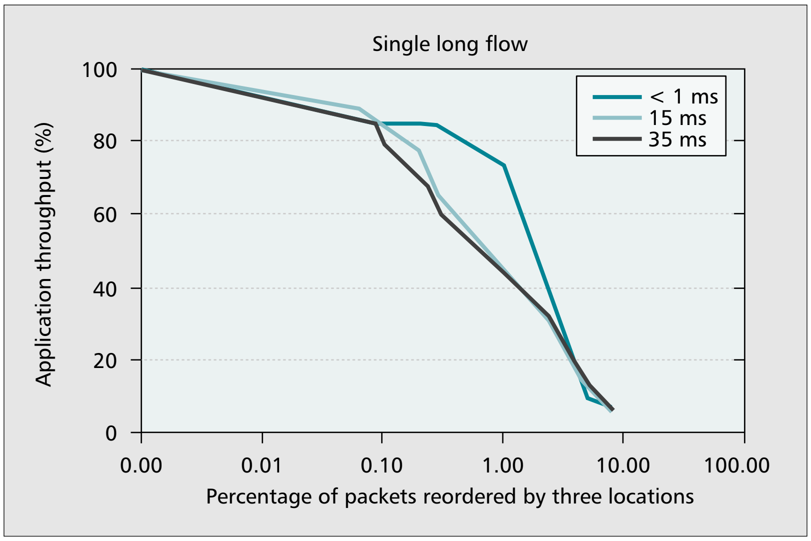
The test setup (Fig. 1) involved four 866 Mhz PCs running as clients connected over a 100 Mb/s onboard Fast Ethernet interface via a Cisco Catalyst 6000 switch. Each of the four client PCs ran a different operating system: Linux, NT4, Windows 2000, and Windows 98. The switch, in turn, was connected to a Cisco 12008 router via a Gigabit Ethernet uplink. The Cisco 12008 was connected to a second Cisco 12008 router with another Gigabit Ethernet link. This link represented the backbone link in our setup, and carried all the traffic between the clients and servers. The second Cisco 12008 router was connected via a Cisco Catalyst 6000 switch to three servers: a Sun Ultra10 running Solaris 2.6, an IBM 330 dual CPU 1 GHz running Linux 2.2.6, and an IBM 330 dual CPU 1 GHz running NT4 (Table 1). All servers used a Gigabit Ethernet interface.

Two Agilent QA-robot devices were connected as part of the backbone link. A data sniffer was used to observe the actual packets sent.

Note:
- Linux was the only operating system in the setup with SACK support.
- With the exception of Windows NT, all operating systems supported window scaling (RFC 1323) [12].

The QA-robot allowed for the introduction of impairments into the traffic flows on the Gigabit Ethernet backbone link. The types of impairments are defined as follows:
- Insert delay — allowed to introduce delays of any number of milliseconds on all traffic to emulate delay along the path.
- Packet drop — allowed to drop a packet and define how many and at what frequency.
- Reorder packets — allowed to reorder one packet at a time (the amount currently supported by QA-robot); it accepted two variables: skip and reorder. For example, skip N reorder M means take packet $N + 1$ and put it after packet $N + M$. Reorder can be applied in either direction separately, thus creating reorder on data, ACKs, or both.

Additionally, the QA-robot could be programmed to operate in a flow-through mode in which no impairment was introduced. In the context of this test the two QA-robots were used to introduce delay as well as to reorder packets.



■ Figure 2. *A single long flow, UNIX to Linux.*

To measure end-to-end application throughput we used a program called Chariot by NetIQ. The main components of Chariot include a console that sets up an application at two end hosts. Any type of application could be defined as well as the amount and type of data to be sent. In our case, TCP sessions were invoked to transfer data. The two endpoints of a TCP session were set up by the Chariot program to take measurements while transferring data. Those measurements were then sent to the Chariot console. The information included, among other data, the average, minimum, and maximum throughput each application actually received. Thus, Chariot enabled us to measure the actual throughput available to the application while filtering out potential network inefficiencies (e.g., data that was retransmitted).

Test Procedure

We defined three variables for our test cases and varied each to approximate a complete set of all combinations. The three variables were a mix of flow types on the backbone link, delay, and host operating systems. We applied a series of different types and rates of packet reordering to each combination of these variables, as described below.

Description of variables:

Mix of flow types on the backbone link. Three types of flow mixes were used:

- Single long flow: A long file transfer of ~400 Mbytes in which a TCP session was established and transfer was set up. At completion the session closed with an abortive reset, and a new session was established for another transaction.
- Multiple flows between a single client-server pair: We added two other flow types to the long file transfer above: a short file transfer and a short HTTP text session. The short file transfer was 100 kbytes long and closed with an abortive reset after the file transfer completed. A new session was then started for another file transfer of 100 kbytes. In the case of the short HTTP text session the client set up a TCP session and sent a request of 300 bytes. The server responded with either 5 or 20 kbytes. After completion of the data transfer the session closed with an abortive reset, and another session began for the same transaction type.

  The multiple flows case consisted of a mix of all three types of flows running simultaneously where the ratio of flow types was as follows: 12.5 percent of all sessions were long file transfers, 12.5 percent short file transfers, and 75 percent short HTTP text sessions. The HTTP was evenly divided between the two different HTTP types (which were 5 or 20 kbytes server's response).

  To avoid a case where all sessions start simultaneously, a uniform distribution within an interval of (0, 200 ms) was applied to determine the actual starting time of each run.
- Multiple flows between a server and multiple clients. Here we used the multiple flows mix described above and ran it simultaneously between a server and all clients. This was repeated for all different servers.

Host operating systems. Both the single long flow and multiple flow cases were repeated between all different pairs of client-servers shown in Table 1. The third flow type was repeated for all different server types, each running multiple flows for all four clients simultaneously.

Delay. Each run was repeated with four different delay values on the backbone link: 0, 15, 35, and 70 ms. Note that these represent one-way delays, so the RTT was twice that value. This was an additional delay, introduced by the QA-robot, added to the link's inherent delay.

We ran close to a complete set of all combinations of the defined mixes of flows, end host operating systems (i.e., the different TCP implementations), and the four delay values specified above.

For each combination, three series of reordered packets were applied. The three differed in the number of packet positions a packet shifted in the backbone packet stream. Position change involved either a shift of a single packet location, a shift of two packet locations, or a shift of three packet locations. For each, a series of different rates of packet reordering was applied to the backbone link. We started from 0 percent of the packets on the backbone and increased, in steps, to a reorder rate of 20 percent of the traffic. Measurement of each application throughput was taken for each case. It is important to note that packet reordering was forced in the backbone link and applied to the multiplexed stream of flows in this link.

Results

Figures 2–8 show results of our measurements. Because of the relatively large number of cases that we tested, only a sample of the results is provided here.

Description	Hardware type	Operating system
PC1	HP VL400 866 MHz; 128 Mbytes	Windows 2000 Service Pack 1
PC2	HP VL400 866 MHz; 128 Mbytes	Windows98 SE
PC3	HP VL400 866 MHz; 128 Mbytes	Windows NT4 Service Pack 6
PC4	HP VL400 866 MHz; 128 Mbytes	Linux 2.2.6 Red Hat 7.0
U10	Sun U10	Solaris 2.6
PC5	IBM 330 Dual CPU 1 GHz; 256 Mbytes	Linux 2.2.6 Red Hat 7.0
PC6	IBM 330 Dual CPU 1 GHz; 256 Mbytes	Win NT4 Service Pack 6

■ Table 1. *Test setup configuration.*

Figure 2 shows the typical behavior of a single long flow. While Fig. 2 shows the measurements for UNIX to Linux as a sender and receiver pair, all other operating system pairs behaved in a similar way. Three delay values are shown: less than 1 ms,[5] 15 ms, and 35 ms. In the less than 1 ms case, no delay was introduced. We applied the delays on the backbone link for the 15 and 30 ms cases, respectively. Note that these are one-way delays, so the RTT amounts to twice this value. The X axis in Fig. 2 uses a logarithmic scale showing the percentage of packets that were reordered by three locations on the backbone link. Assuming the case of zero packets reordered as a reference (100 percent on the Y axis), the Y axis shows the percentage of the bandwidth relative to the zero packets reordered case that was measured by the receiving applications for each packet reorder rate.

Figure 3 shows the aggregate average throughput of multiple flows, consisting of a mix of flows as described in the test procedure above, between a server and a single client. The X axis scaling is the same as in Fig. 2. Figure 3 shows measurements with Linux as a server exchanging data with each of the four clients, a different client in each test. Similar behavior was observed with the two other servers, UNIX and NT, as well as when we vary the delay. At high delay values, most notably at 70 ms, the rate of degradation in application throughput is larger.

Figure 4 shows the results of a server connected simultaneously to multiple clients and running multiple flow mixes to each one. Results for Linux and UNIX servers are shown for both the 15 and 35 ms delay cases.

We observed a similar behavior with NT as a server. The percentage degradation of bandwidth relative to the zero packets reordered is similar for both the 15 and 35 ms delays. Figure 5 shows the actual aggregate average throughput, for the cases shown in Fig. 4, as measured in megabits per second. While the higher delay results unsurprisingly in smaller throughput, the behavior in terms of the degradation rate relative to the reorder rate is very similar.

Figure 6 shows the response of application throughput, for each application type, to packet reordering. The value shown is the aggregate average throughput as measured by all applications of the same type: long file transfer, short file transfer, and HTTP.

While Figs. 2–6 show behaviors for three or more duplicate ACKs, Figs. 7 and 8 show behaviors for one and two duplicate ACKs, respectively. We tested a single server to multiple clients, because this configuration is the most indicative of real-life behavior. Results for all three servers are shown for both the 15 and 35 ms delays.

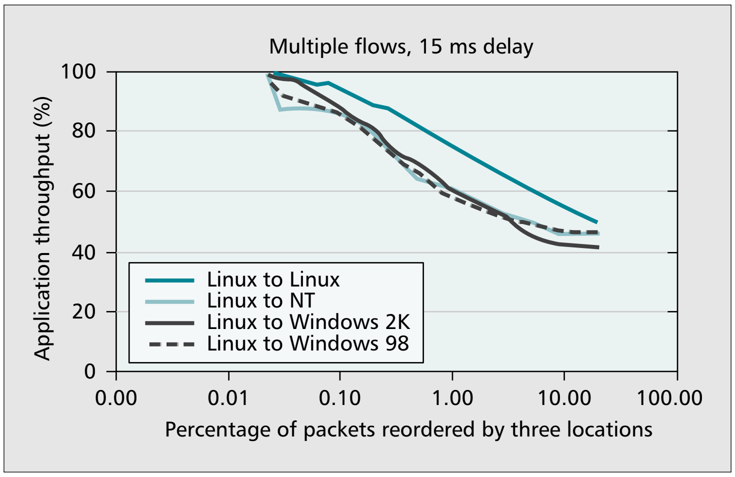


Figure 3. *Server to a single client, multiple flows.*

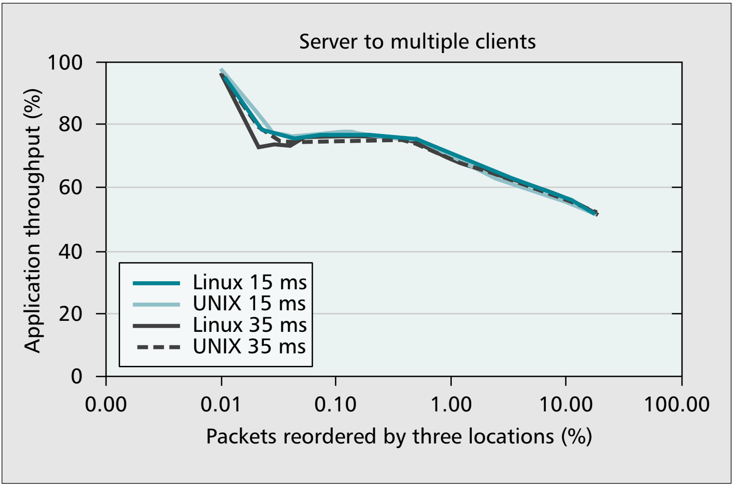


Figure 4. *A server to multiple clients, multiple flows.*

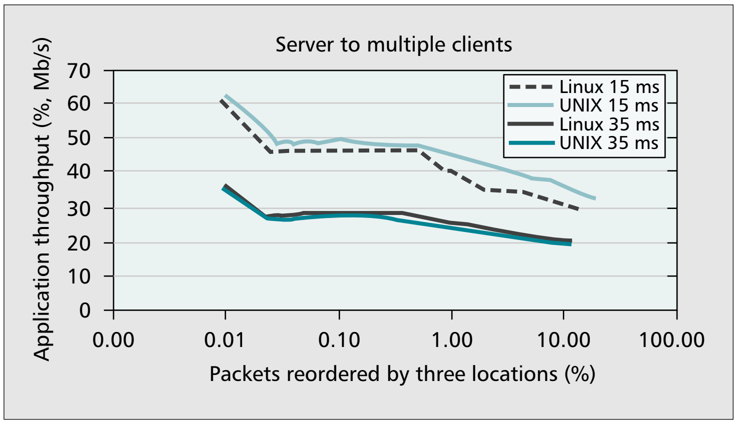


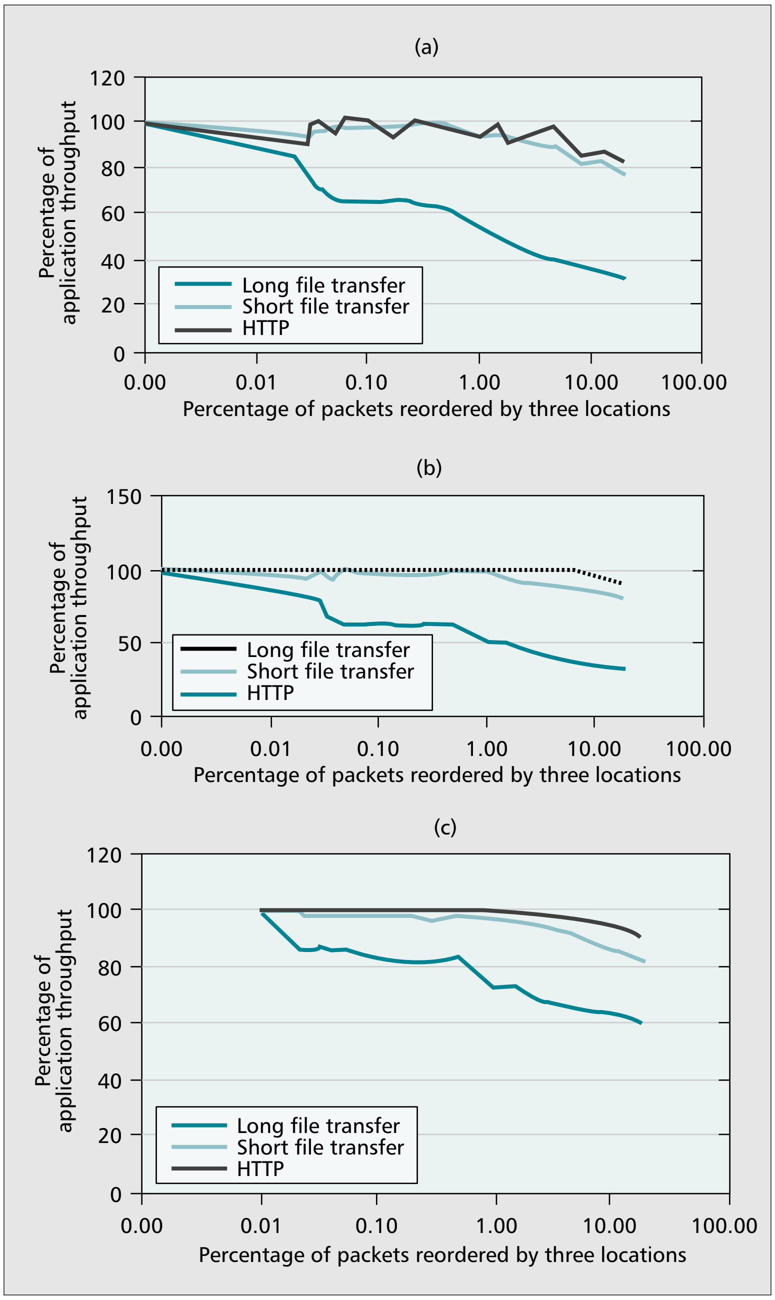
Figure 5. *A server to multiple clients, aggregated throughput.*

Discussion

In the single long flow shown in Fig. 2, there are three regions on the graph. The first region is the area of low reorder rates. In this region the drop rate was relatively low, although there was a drop in throughput. Here, *cwnd* was halved (because the reorder caused triple duplicate ACKs and thus triggered fast recovery) and then built up again. Because the reorder rate was low, it allowed the window to reopen before it was halved again. In Fig. 2 this region started at zero reordered packets and increased to a 0.1–0.2 percent reorder rate for the higher delay cases and up to approximately 0.5–1 percent for the low delay case.

Above those reorder rates it is evident that the bandwidth drop rate increased significantly. As the reorder rate increased, the rate of halving the window also increased.

[5] *In the less than 1 ms case, zero delay is forced by the QA-robot. We specify less than 1 ms since some inherent delay (e.g., traversing network devices, receivers' delay) exist.*



■ Figure 6. *Throughput per transaction type, server to multiple clients: a) 15 ms delay; b) 35 ms delay; c) 70 ms delay.*

This resulted in a faster degradation of bandwidth. At these rates the window was not allowed to reopen to its original size before it was halved again. There was a sharper fall in bandwidth from about 85 percent of the original application throughput to around 10 percent. The lower delay case reopened its window much faster than the higher delay cases. Therefore, the lower delay case resumed its lost throughput faster and maintained a higher bandwidth longer. It started the sharper degradation in bandwidth at a higher reorder rate (0.5–1 percent in Fig. 2 compared to 0.1–0.2 percent for the 35 ms delay case). Links exhibiting higher delays start to see a sharp degradation in bandwidth at lower reorder rates.

As the reorder rate continues to increase, a third region can be observed in which the bandwidth drop rate reduces again as it asymptotically approaches the point in which the window is kept at its minimum size of one to three segments (starting at around an 8 percent reorder rate in Fig. 2). Here, the reorder rate was high enough to not allow the window to start growing at all, and it was thus kept at a minimum value, resulting in minimum utilization.

The case shown in Fig. 2 was of a single long flow. This means we kept affecting the same flow again and again with each packet reordering, causing its window to continue to reduce its size, which would not allow it to grow. This explains why the degradation in bandwidth is so high, down to approximately 10 percent of the original bandwidth.

While the single long flow case does not represent much of a real-life scenario on a backbone link, the observations discussed above are indicative of application throughput response to packet reordering. Similar behavior, albeit with different values, can be observed in other cases, as shown in Figs. 3–6.

Figure 3 shows the measurements of packet throughput when applied on a backbone link carrying multiple flows between a server and a client. It reveals similar behavior, beginning with an area of low bandwidth degradation, followed by an area of a sharper decrease in bandwidth, and concluding with a reduced rate of bandwidth degradation. Note that the drop in bandwidth degradation is to the 50 percent range of the original bandwidth in contrast with the single flow case.

Figure 4 shows the results of a server running multiple flows simultaneously to multiple clients. While similar behavior is evident, note that the slope of bandwidth degradation is lower relative to the previous two cases. We show in Fig. 5 the actual aggregate average bandwidth measured for each delay case. As expected, the higher delay yields less bandwidth. Looking again at Fig. 4, however, we can observe that the application throughput response is very similar for both delay cases. In Fig. 6 we show the per transaction type response. The long flows lose most of the throughput while the degradation is smaller for the shorter file transfer and short HTTP flows. As we force packet reordering in the backbone, the longer the flow is, the more likely it is to get hit. A packet from a long flow is more likely to get reordered than a packet from a shorter flow. There is also a higher likelihood that once a packet from a long flow gets reordered it will be moved by three packet locations within its own flow.

The long flows are also affected the most. This is because long flows are long-lived flows and as such, especially in our test lab setup where there was no congestion, the sender is allowed to open its window significantly. Once reorder is introduced, the fast recovery mechanism takes effect and closes the congestion window by one half. The resulting drop in bandwidth is significant. The shorter flows, however, do not get to build their windows to this degree before the session ends. So even if a short flow experienced three duplicate ACKs, the actual amount of bandwidth lost would be smaller.

As is evident from Fig. 6, the shorter flows may even gain bandwidth in some cases. This is a result of the long flows slowing down due to the packet reordering, allowing the shorter flows to use more host resources. Note that since the link in the test setup is not congested, the shorter flows gain is due to an increase in available host resources rather than due to an increase in available link bandwidth. It is expected that in an environment with congested links, shorter flows may have additional bandwidth gain when long flows are slowing down, thus freeing the link's bandwidth. This gain, however, is limited because short flows are short-lived by nature and thus usually end before their windows can open enough in order to makes significant use of the additional bandwidth.

Lastly, the magnitude of the effect of three packet reordering implies that there is some level of grouping of packets from the same session in the backbone. That is, a number of consecutive packets from one session are followed by another group of packets from another session, as opposed to multiplexing packets where there is one at a time, each from a different flow. Otherwise, we would have observed a very minimal effect for a three packet position reordering, if there was any effect at all. This is because most packets in a session would not result in a three-packet location shift within their own flows given the relatively large number of flows that are multiplexed.

Attaching a data sniffer and observing the flows on the backbone confirmed that there were frequent groups of consecutive packets from the same flow. This grouping is a function of the sending host as well as the behavior of the router. While one might expect that as we move up through multiple levels of routers to a faster backbone link, the multiplexing level will increase, some observations suggest that a certain level of grouping is likely to remain (this is largely a function of the router's behavior and the pattern of the arriving data). Hence, we cannot assume that at faster backbone links the effect of reordering in the backbone diminishes.

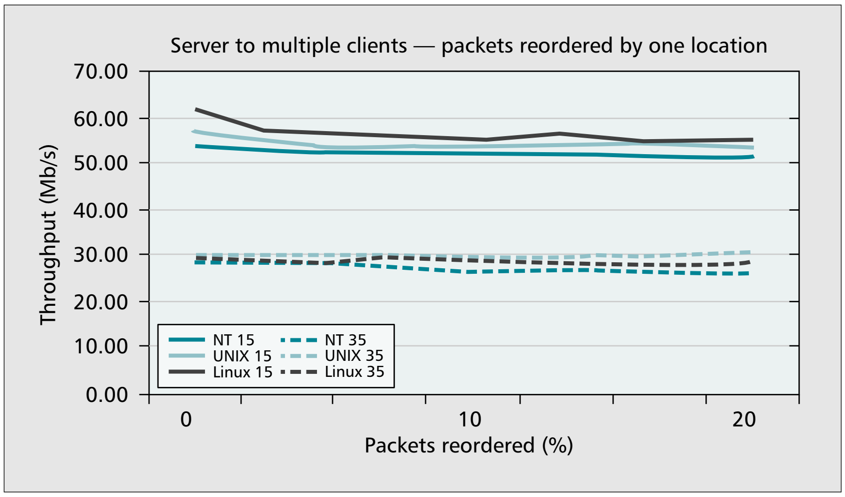


Figure 7. *Reorder of a single packet location.*

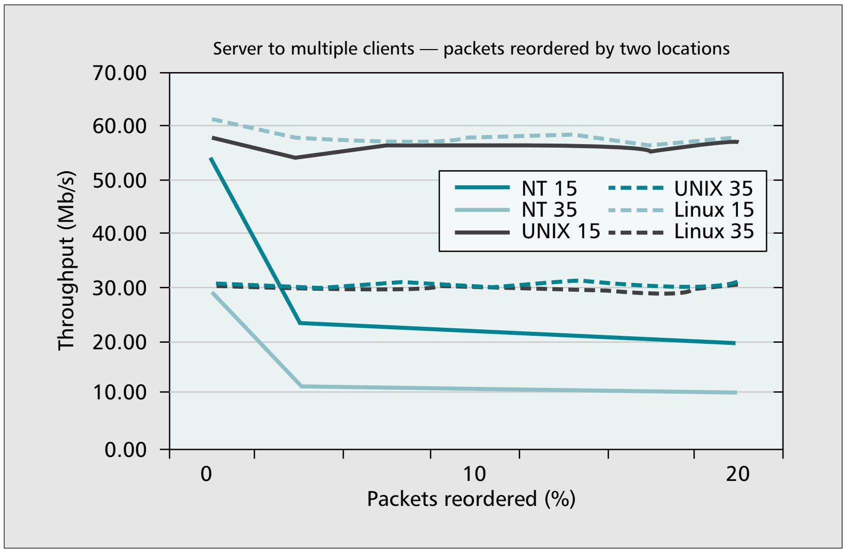


Figure 8. *Reorder of two packets location.*

Figures 7 and 8 are results of measurements with single and dual duplicate ACKs, respectively. As expected (with the exception of NT4 with dual duplicate ACKs), there is no drop in throughput. A closer look, however, reveals that at high reorder rates (above 20 percent in Figs. 7 and 8), there is a small percentage of throughput degradation. We suspect that this may be due to a combination of the following factors:

- A slowing down of *cwnd* growth because *cwnd* stays the same during duplicate ACKs. While very short HTTP sessions that finish in one to three segments do not observe slowdown in *cwnd* growth, longer flows can be affected over time when the reorder rate is high.
- As the number of duplicate ACKs increase, the overhead of the sender dealing with the duplicate ACKs comes into effect.

Note: NT4 seems to enter fast recovery with double duplicate ACKs. We have verified that MaxDupACKs in:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TCPIP\Parameters

is not defined (in which case it should default to 3). We repeated the test, explicitly setting MaxDupACKs to 3, and received the same results. We suspect that this behavior might be due to the way NT calculates the actual number of duplicate ACKs in the current version.

Summary

We have seen that applying a small percentage of packet reordering in a link carrying multiple flows results in a drop in application throughput. While at small reorder rates there is little effect, at rates starting from 0.1–1 percent (depending on the delay), a much faster drop in application throughput takes place. As reorder rates continue to increase, the throughput of affected applications approach minimum utilization, typically around an 8–10 percent reorder rate. Links with higher delays tend to arrive at the point of faster bandwidth degradation at lower rates of packet reordering. Long flows are affected the most. Shorter flows are affected less and do not lose as much throughput. Single and double duplicate ACKs do not affect throughput for the most part, but some penalties do exist at high reorder rates.

With the exception of NT in the case of double duplicate ACKs, we have not observed a significant difference in behavior among the various TCP implementations, although a performance difference was observed.

Conclusion

Packet reordering exists in networks. As the bandwidth grows, even a small percentage of packet reordering combined with some level of packet loss due to network congestion can cause significant degradation of link utilization as well as of application throughput. Packet reordering is a silent problem leaving no trace, and it is much harder to observe and debug than packet loss, which is usually captured by network switches and routers.

As network bandwidth continues to increase, it is likely that we will see more architectures that rely on parallel processing of packets in nodes as well as more usage of parallel transmission on links. This, along with the inherent behavior of a connectionless network, implies that some level of packet reordering is likely to always exist. Because of the potential impact of packet reordering on TCP, as shown in our tests, along with the difficulty of debugging packet reordering as the source for throughput degradation, it is important to try to minimize it. Specifically, the following steps should be considered:

- Usage of SACK and D-SACK should be encouraged in TCP implementations. Since real-life networks exhibit both packet reordering and packet loss, SACK can slow down the rate at which bandwidth is degraded in many scenarios by avoiding both retransmission of segments that were received correctly and unnecessary reduction of the sending window. D-SACK, by signaling a duplicated received segment, enables the sender to actually adapt to a high packet reorder rate as well as recover the window. This is an area that should be of continued focus in order to define and implement the sender's behavior.
- Load balancing in a router should be done per source-destination pair and not per packet, such that packets within the same flow use the same path. While efficiency issues exist, this is an area that should be of continued focus for researchers and vendors because it will allow routers to balance the load across links without creating potential reordering.
- Network designers should try to avoid using network gear whose architecture may cause packet reordering. Packet reordering is not inherent to routers' architectures, and some routers are designed to avoid it altogether. Many of these are available in the commercial market. However, if such a router must be used, the potential reorder rate should be evaluated based on the router architecture. The network designers must assess whether their network can tolerate this rate of reorder and the potential impact on application throughput.

References

[1] J. C. R. Bennett, C. Partridge, and N. Shectman "Packet Reorder Is Not Pathological Network Behavior," *IEEE/ACM Trans. Net.* vol. 7, no. 6, Dec, 1999.

[2] S. McCreary and K. C. Claffy "Trends in Wide Area IP Traffic Patterns," *13th ITC Specialist Seminar on Internet Traffic Measurement and Modeling 2000*, http://www.caida.org/outreach/papers/2000/AIX0005

[3] M. Allman, V. Paxon, and W. Stevens "TCP Congestion Control," RFC 2581, Apr. 1999.

[4] V. Jacobson, "Modified TCP Congestion Avoidance Algorithm," Email to the end2end-internet mailing list URL: ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail, Apr. 30.

[5] M. Mathis *et al.*, "TCP Selective Acknowledgment Options," RFC 2018, Oct. 1996.

[6] W. R. Stevens, *TCP/IP Illustrated*, vol. 1, Addison Wesley, 1994.

[7] V. Jacobson "Congestion Avoidance and Control," *SIGCOMM '88, Proc. ACM Symp. Commun. Arch. Protocols*, Aug. 1988, Stanford, CA, pp. 314–29.

[8] K. Fall and S. Floyd "Simulation Based Comparison of Tahoe, Reno and SACK TCP," *Comp. Commun. Rev.*, vol. 26, no. 3, July 1996, pp. 5–21.

[9] V. Jacobson and R. Braden "TCP Extensions for Long-Delay Paths," RFC 1072, Oct. 1988

[10] S. Floyd *et al.*, "An Extension to the Selective Acknowledgment (SACK) Option for TCP," RFC 2883, July 2000.

[11] S. Floyd, "Re: TCP and out-of-order delivery," Message ID <199902030027.QAA06775@owl.ee.lbl.gov> to the end-to-end-interest mailing list, Feb. 1999, http://www.icir.org/floyd/notes/TCP_Feb99.email

[12] V. Jacobson, R. Braden, and D. Borman "TCP Extensions for High Performance" RFC1323, May 1992.

Biographies

MICHAEL LAOR (mlaor@cisco.com) is a director of engineering with Cisco Systems. He joined Cisco in 1993 and has led several development programs of Cisco high-end IP routers. His main areas of interest and activity are architectures of large IP routing systems, fast IP forwarding engines, and IP core networks. He holds a B.Sc. in electrical engineering from Ben-Gurion University, Israel, and an M.B.A. from California State University.

LIOR GENDEL (lior@cisco.com) is a senior technical lead in Cisco Systems. Prior to joining Cisco he designed and implemented public IP networks. After joining Cisco in 1996 he participated in the design of many private and public IP networks, in particular in Europe. His latest activity in Cisco includes a study of applications response over large IP networks. He holds a B.Sc. in computer engineering and an M.Sc. in computer science from Ben-Gurion University, Israel.