[13] ns Network Simulator. http://www-mash.cs.berkeley.edu/ns/, 1996.

[14] Official 1996 olympic web site - home page. http://www.atlanta.olympic.org, 1996.

[15] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proc. Second International WWW Conference*, October 1994.

[16] S. Spero. Session control protocol (scp). http://www.w3.org/pub/WWW/Protocols/HTTP-NG/http-ng-scp.html, 1996.

[17] W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, MA, Nov 1994.

[18] J. Touch. *TCP Control Block Interdependence*. RFC, April 1997. RFC-2140.

[19] WOM Boiler Room. http://www.womplex.ibm.com, 1996.

[20] L. Zhang, S. Shenker, and D. D. Clark. Observations and Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proc. ACM SIGCOMM '91*, pages 133–147, 1991.

- *Of a group of parallel connections, ones with small outstanding windows could experience a larger number of losses than their share of the total outstanding window would warrant*. This means that it may be harder to initiate a new connection than to keep an existing connection going.

We then proposed sender-side TCP modifications that improve the performance of TCP loss recovery and the use of parallel connections from individual clients. To reduce the occurrence of timeouts, we present an enhanced loss recovery scheme that improves performance when window sizes are small and when insufficient duplicate acknowledgments arrive for multiple losses in a window. Analysis of the trace data shows that over 25% of the coarse timeouts could be avoided by this scheme, resulting in significant performance improvements. Simulation results shows that under test circumstances this can lead to a dramatic reduction in the number of coarse timeouts. To address the problem of parallel connections from individual clients, we presented an *integrated connection* approach to congestion control and loss recovery that allows a TCP sender to aggregate simultaneous connections from individual clients and treat them as a single unit. Simulation results show that our approach achieves much-improved start-up behavior, loss recovery, and bandwidth sharing amongst the parallel connections from a number of hosts.

In future work, we plan to implement our enhanced loss recovery and integrated connection improvements and examine their performance in real-world busy server environments. In addition, as a part of our data collection, we also used traceroute to collect network topology information for a large fraction of the clients that visited the Web server. We plan to use this topology information to examine the possibility of extending the integrated congestion control and loss recovery methods to share information across connections from nearby hosts as well as from connections originating from the same host, expanding on the results presented in [3].

7. References

[1] IBM AlphaWorks Home Page. http://www.alphaworks.ibm.com, 1996.

[2] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, Affordable, High-Performance Statistics Collection. http://www.nlanr.net/NA/Oc3mon/, Aug 1996.

[3] H. Balakrishnan, S. Seshan, M. Stemm, and R.H. Katz. Analyzing Stability in Wide-Area Network Performance. In *Proc. ACM SIGMETRICS '97*, June 1997.

[4] T. Berners-Lee, Cailliau, and et al. The World Wide Web. *Communications of the ACM*, 37(8):76–82, Aug 1994.

[5] K. Fall and S. Floyd. Comparisons of Tahoe, Reno, and Sack TCP. ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z, December 1996.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, Jan 1997. RFC-2068.

[7] J. Gettys. Mux protocol specification, wd-mux-961023. http://www.w3.org/pub/WWW/Protocols/MUX/WD-mux-961023.html, 1996.

[8] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *Proc. ACM SIGCOMM '96*, August 1996.

[9] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 88*, August 1988.

[10] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proc. Winter '93 USENIX Conference*, San Diego, CA, January 1993.

[11] J. C. Mogul. Observing TCP Dynamics in Real Networks. Technical Report 92/2, Digital Western Research Lab, April 1992.

[12] Netscape Corp. http://home.netscape.com/eng/mozilla/3.0/handbook/plugins/index.html, 1997.

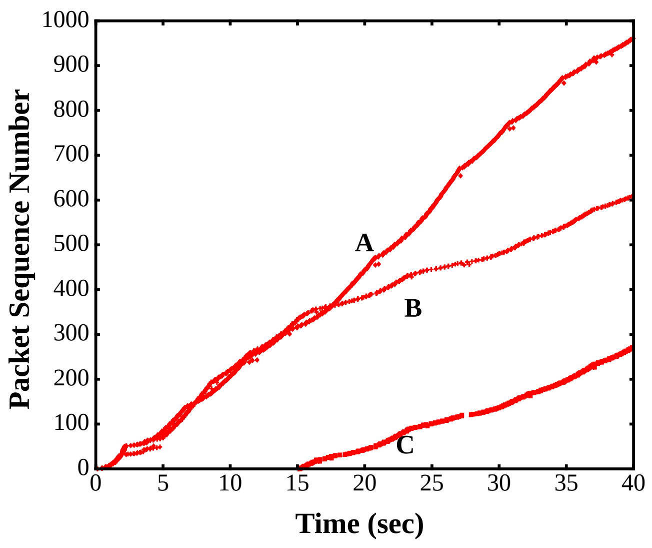

Figure 18. Three TCP-INT transfers from two hosts through a single bottleneck router. Connection A originates from the first host and starts at time 0. Connections B & C originate at the second host and start at time 0 and 15 sec respectively.

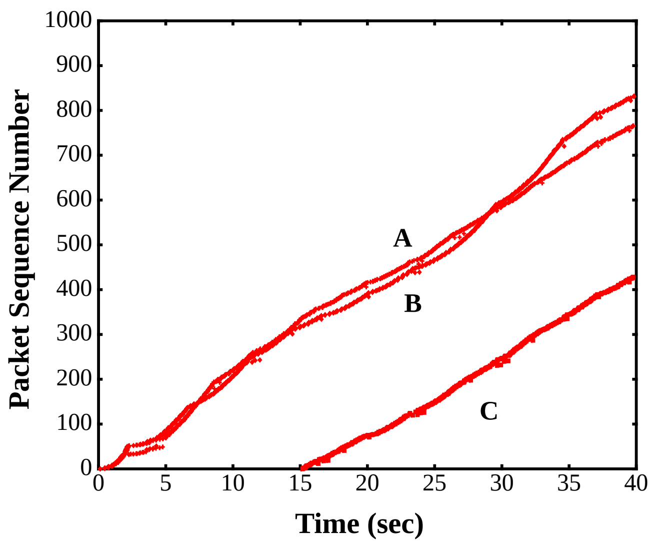

Figure 19. Three TCP-INT transfers using the congestion control algorithm for proxies from two hosts through a single bottleneck router. Connection A originates from the first host and starts at time 0. Connections B & C originate at the second host and start at time 0 and 15 sec

mately half the bandwidth of the transfer on node R1. This is because each *host* independently performs the same congestion control algorithm. Therefore, each host receives an equal portion of the bottleneck link. However, as discussed in Section 5.5, there are often situations where a host should acquire a greater portion of the link bandwidth. The sequence plot of connections using the congestion control policy described in that section are shown in Figure 19. The plot shows that the integrated congestion control algorithm can be adjusted to provide equal shares to each of the connections, while still maintaining the benefits of integrated loss recovery for the two connections, B and C, between nodes S and R2.

6. Conclusions

In this paper, we have presented a detailed analysis of TCP behavior from a busy Web server. Our analysis has focused on two main areas: examining the performance of individual TCP connections that carry HTTP payloads, and examining the detrimental effects of how Web browsers use parallel TCP connections on overall network performance. We found that:

- *Existing loss recovery techniques are not effective in dealing with packet losses and new techniques must be developed to handle them*. Over 50% of all losses require a coarse timeout to recover. Fast retransmissions recover from less than 45% of all losses. The remainder of losses are during slow start following a timeout.
- *Future network implementations should increase their default socket buffer size to avoid the receiver window from becoming a bottleneck.* The socket buffer size limited the throughput of approximately 14% of all observed connections.
- *Ack compression is an observed phenomenon and can be correlated with subsequent packet losses.* Our analysis indicates that a dynamic comparison of data and acks bandwidths is an effective indicator of ack compression, and is also easy to implement in the TCP layer. One area of future work is the investigation of mechanisms, such as traffic shaping, to combat the adverse effects of ack compression.
- *A client using a collection of parallel connections between a client and server is a more aggressive user of the network than an application that uses a single TCP connection.* Throughput is positively correlated with the number of active connections. When multiple connections are concurrently active and one of them experiences a loss, only half of the remaining ones on average experience a loss. The combined congestion window of a group of parallel connections does not decrease as much as the congestion window of a single connection after a loss epoch.

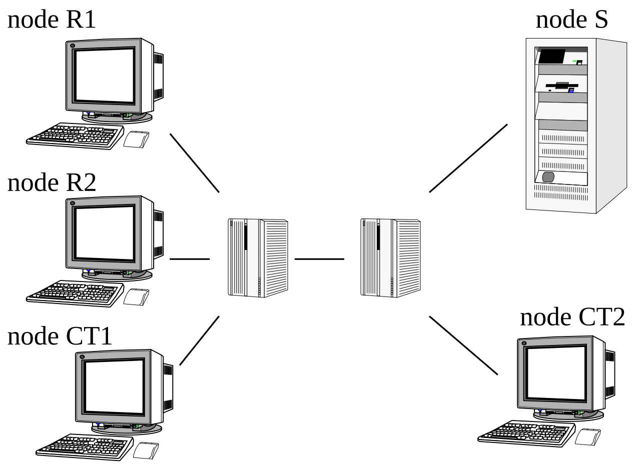

Figure 16. Topology for simulation tests.

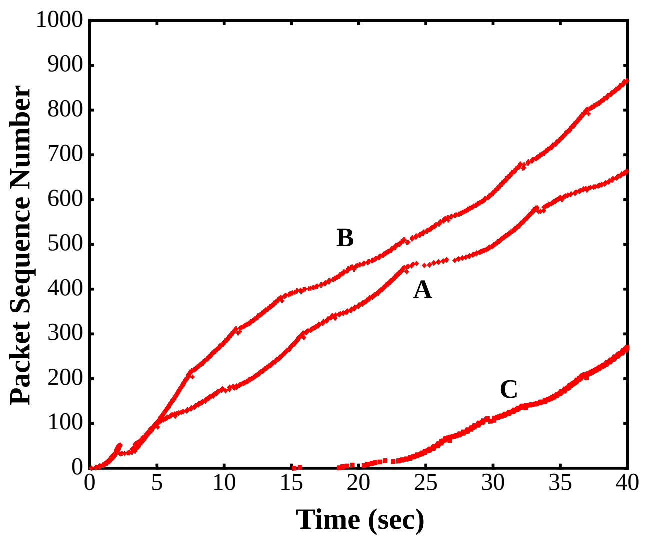

Figure 17. Three TCP-SACK transfers from two hosts through a single bottleneck router. Connection A originates from the first host and starts at time 0. Connections B & C originate at the second host and start at time 0 and 15 sec respectively.

However, unlike the case of independent TCP connections, the penalty, which is equal to half the window of the affected connection, is spread across all the connections. Given the prevalence of FIFO queuing in the network, which can lead to cases where a burst of traffic in one connection causes losses in another connection, there is no reason to single out the connection that experienced a loss for all the penalty. Therefore, we spread the penalty across all the connections between the proxy and the server, to the same end host, while effecting the same decrease in the combined congestion window as when only one of the connections is penalized. Spreading the penalty also smooths out the performance observed by the individual connections. Note that round-robin scheduling described in Section 5.3 automatically achieves the desired spreading.

As an illustration of the ideas in the previous paragraph, suppose that there are two connections between the proxy host and the server, one each on behalf of two end clients. Further, suppose that when the total congestion window is 8 KB (4 KB for each connection in logical terms), connection #1 suffers a loss. As a result, the combined congestion window decreased by half of 4 KB, yielding a total of 6 KB. With the penalty spread out, each connection ends up with an effective congestion window of 3KB. In contrast, if the connections had operated independently, they would have ended up with the unequal congestion windows of 2 KB and 4 KB, respectively.

We note that none of the above alters the integrated data-driven loss recovery algorithm described in Section 5.3

In terms of a practical implementation, there are two requirements. First, the sender (often the Web server) should be able to distinguish between connections on behalf of different end client hosts (or perhaps even users). Different flows could be marked by using IPv6 flow identifiers or by creating a new TCP option type. Our focus is not on the mechanism to define priorities or the policy to assign priorities, but rather on techniques for enforcement of a policy once it has been determined and communicated to a TCP sender.

5.6 Simulation Results: Multiple Client Hosts Case

We now investigate how the bottleneck link bandwidth is shared by connections initiated from more than one host. Our test uses the network topology shown in Figure 16. At time 0, a single TCP transfer is started from node S to each of nodes R1 and R2. Some time later (in this case, fifteen seconds), a second transfer is started between node S and R2. In addition, several cross traffic connection are maintained between nodes CT1 and CT2. This is intended to make the simulation more realistic.

Figure 17 shows the sequence plot for a test where node S uses TCP with selective acknowledgments. This graph shows that each connection receives approximately the same share of the bottleneck link bandwidth. As a result, node R2 receives approximately twice the bandwidth of node R1 after the second connection starts up. This is because each *connection* independently performs the same congestion control algorithms and are each equally likely to experience congestion. Figure 18 shows the same test using TCP-INT. After the second connection from node R2 starts, each of the transfers from R2 receive approxi-

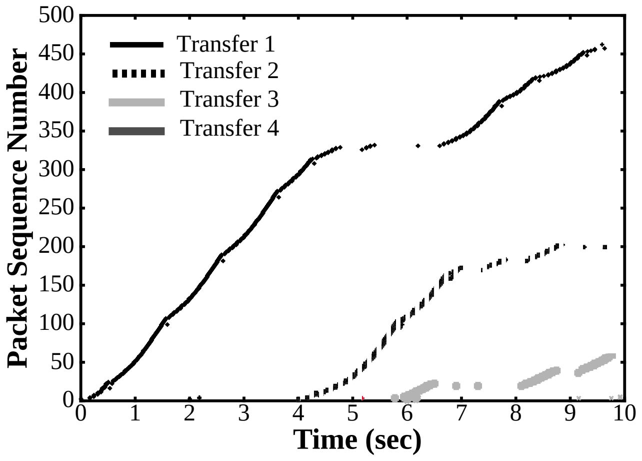

Figure 14. Four TCP-SACK transfers through a router with buffer size 3. Transfers start at 0, 2, 4 and 6 seconds.

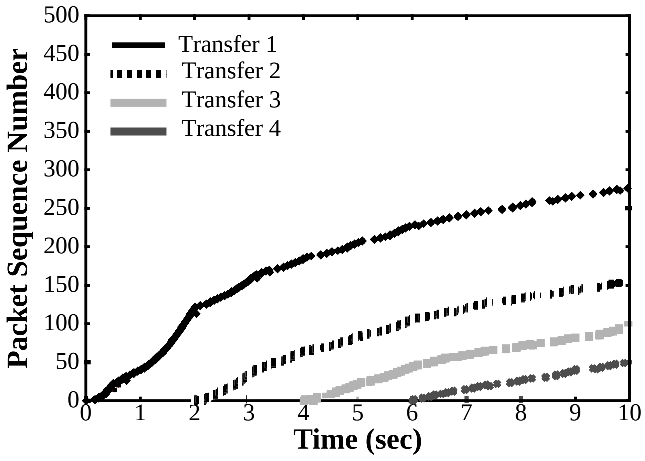

Figure 15. Four TCP-INT transfers through a router with buffer size 3. Transfers start at 0, 2, 4 and 6 seconds.

unpredictability is undesirable because it may happen that connections carrying critical data get slowed down while others carrying less important data do better.

Figure 15 shows the sequence plot for the same test with the senders using TCP-INT. Integrated loss recovery helps this TCP variant avoid coarse timeouts. Integrated congestion control allows the different connections to each obtain an equal share of the total bandwidth. Although the total number of bytes transferred here is actually slightly less than in the case with the TCP-SACK protocol, the performance of the transfers is much more consistent and predictable. Also note that since the slow-start window growth happens in an integrated manner for the ensemble of connections, later connections can build on the window size attained by earlier connections rather than initiate slow start with an initial window size of one segment (which happens with vanilla TCP). This can greatly help reduce the connection start-up latency for Web clients with short overlapping parallel connections.

In the second test, we investigated how the bottleneck link bandwidth is shared when there are competing connections from more than one host. Before discussing the results, we describe an extension to the integrated congestion control algorithm to handle connections from proxy hosts.

5.5 Adapting Integrated Congestion Control to Proxy Hosts

Treating simultaneous connections from single hosts as a single unit with respect to loss recovery and congestion control reduces any advantage that individual applications may gain from using multiple TCP connections. All IP hosts that connect to the TCP sender are treated equally. There are situations, however, where IP hosts should not be treated equally. Many clients today connect to the Web using proxy or caching hosts. Although seen as a single host, these proxies represent a much larger number of end client hosts (and consequently a larger number of people). By reducing the advantage that comes about from multiple connections, we may be penalizing clients that connect via proxies. Therefore, we investigated mechanisms that allow unequal treatment of hosts receiving data from the TCP sender.

We adapt our algorithm for integrating congestion control to address this problem. For simplicity, suppose that the desired policy is that all *end* client hosts be treated as equal. Then the combined congestion control window for the set of connections between a proxy host and server would be apportioned equally to the connections initiated on behalf of each end client host involved. So again for simplicity, suppose that there is only one connection between the proxy host and the server on behalf of each end client.

The basic idea now is that congestion control for the set of connections from the proxy is done in such a way as to mimic the case when the connections are independent (because each connection is on behalf of a different end client). In the slow start phase, the combined congestion window is increased by 1 segment for each ack received. In the congestion avoidance phase, the congestion window is increased by 1 segment per RTT for *each* connection. If one of the connection's experiences a loss, only the corresponding portion of the congestion control window is halved; the window apportioned to the other connection remains unchanged.

to be sent by adding an entry to the tail of the list of outstanding packets. This entry contains the sequence number size and timestamp of the transmitted packet. When the packet is sent, the connection increments the ownd by the size of the packet.

New recv ack routine: When a new ack arrives, the sender increases the cwnd variable as appropriate. We experimented with a variety of congestion window increase algorithms to support different bottleneck link sharing goals. These algorithms are described in Section 5.5. Also upon the arrival of a new ack, the sender removes any packets from the pkts[] list that have reached the receiver. The sender decrements the ownd variable by the size of packets removed from the list.

Upon arrival of any ack, new or duplicate, the sender uses the ack timestamp[2] to increment the later_acks field of any packet that was transmitted earlier than the one just acknowledged. The sender then traverses the list of packets in the "pipe" from oldest to most recent to identify any candidates for retransmission. In a simple situation where the delayed acknowledgment algorithm is not being used, a retransmission candidate can be identified by the following rules:

1. The later_acks field is greater than 3. This is used to avoid unnecessary retransmission as a result of network reordering of packets, **and**
2. The packet is the lowest sequence number, unacknowledged packet on a connection, **and**
3. The connection associated with the packet does not have a pending retransmission.

The presence of delayed acknowledgments complicates these rules by effectively reordering the transmission of acknowledgments. Receivers implementing delayed acknowledgments must acknowledge the receipt of at least every other packet or at most 200 ms after the receipt of a packet. This necessitates the following additional rules to identify lost packets:

4. A candidate for retransmission must also have one other packet on the same connection with 3 or more later acks. This compensates for the requirement of only acknowledging every other packet, **or**
5. The packet being acknowledged is well over 200ms more recent than the possibly lost packet. In our simulation, we use 200 * 2 ms to provide a conservative bound. This is based on the requirement to acknowledge a packet at least 200 ms after its reception.

The sender retransmits the single oldest lost packet and marks the connection as having a retransmission pending. The sender then adjusts the congestion window, cwnd, to perform the appropriate congestion control following a loss.

Finally, the sender uses the ownd and cwnd variables to identify if additional packets can be introduced into the "pipe". If there is space for new packets, the sender chooses a connection to transmit packets. The choice of connection is done via a round-robin algorithm across all connections to the same host.

5.4 Simulation Results: One Client Host Case

In this section, we describe results from an ns simulation designed to examine integrated congestion control and loss recovery across simultaneous TCP connections.

The first test used the topology in Figure 11. The router's buffer size was set to 3 packets. This is small enough to force transfers to have small congestion windows and experience frequent losses. Once again, the topology and parameters were chosen to recreate situations that frequently occur in our traces, and not to mimic an actual network. In this test, the transmitting node performs 4 TCP transfers to the receiver. The transfers start at 0, 2, 4 and 6 seconds and all end at 10 seconds. The actual choices of the values 0, 2, 4, and 6 are not important, just that the start times of each connection are slightly shifted in time.

Figure 14 shows the sequence plot for the test using a SACK-based transmitter. It shows that typically only one connection performs satisfactorily at any one time. For example, at time 2 seconds, the connection starting up experiences several early losses and is forced to recover them via coarse timeouts. In fact, this connection does not send a significant amount of data until 4 seconds later (at time 6 sec). Similarly, the first connection experiences many losses at time 4 seconds and spends several seconds recovering from these losses. Over the 10 second period, the connection starting at time 2 sec. and time 6 sec. account for a minuscule (<10%) fraction of the total bytes transferred. Although the combination of the four connections transmit data at a reasonable rate, the bandwidth is clearly not evenly split among the active connections. In addition, the numerous coarse timeouts make the performance of individual connections unpredictable. From an application's point of view, such

2. In absence of the timestamp option, the ACK sequence number combined with transmission order of all packets can be used to perform the same actions.

```
struct chost {
  Address addr   // address of host
  int cwnd;      // congestion window for host
  int ownd;      // total bytes in pipe to host
  int ssthresh;  // slow start thresh for host
  int count;     // count of pkts for cwnd increase
  Time decr_ts;  // time of last window decrease
  Packet pkts[]; // pkts sent in order of xmission
                 // these are pkts in the “pipe”
  TCPConn conn[];// set of tcp connections to host
}

struct packet {
  TcpConn *conn; // connection that sent pkt
  int seqno;     // seqno
  int size;      // size of pkt
  Time sent_ts;  // time sent
  int later_acks;// # of acks for later
                 // pkts on any conn
}
```

Figure 13. New structures necessary for shared congestion windows and shared error recovery.

5.2.1 HTTP/1.1 with Persistent Connections

The HTTP/1.1 protocol [6], which has recently been standardized, recommends the use of persistent connections. So it seems likely that persistent connections will find wide support in future client and server software. Therefore, the first point made in Section 5.2 about applications having to be rewritten is not an issue. However, the other two drawbacks remain valid.

The new HTTP protocol does *not* integrate other applications’ TCP connections (such as FTP or other TCP connections initiated by Netscape plug-ins). We believe that this is symptomatic of a larger problem. If in the future HTTP is replaced by a different protocol, special efforts would have to be made again (in terms of framing format, etc.) to ensure that the same drawbacks do not recur. In contrast, our approach has been to identify the fundamental problem, namely the lack of efficient support for numerous short data transfers between a pair of hosts, and then provide a solution at the transport layer that provides efficient support for short connections regardless of the application-level protocol.

The third drawback mentioned in Section 5.2 also affects persistent connections in HTTP/1.1. Consider the simultaneous transfer of several images over a persistent connection. Suppose that one of the images is of particular interest to the user. Because of the ordered byte-stream abstraction that TCP provides, the rendering of this image gets held up whenever *any* packet gets dropped and requires retransmission. In contrast, with our solution, only the loss of a packet on the connection corresponding to the image of interest (which is clearly only a subset of all packet loss events) can stall the rendering of the image.

We advocate that TCP connections already provide a perfectly useful abstraction to applications, and rather than making the application or socket library more complex by adding the concept of a “session”, the transport level should maintain the same abstractions with a more efficient *integrated connection* implementation.

5.3 TCP-INT Implementation

We now describe an the implementation of integrated congestion control and loss recovery scheme that only modifies the TCP at the sender. For each host with which a single machine is corresponding with, the TCP/IP stack creates a structure (Figure 13) to store information about any communication. This new structure enables the desired shared congestion control and loss recovery by providing a single point of coordination for all connections to a particular host. The chost structure contains the standard TCP variables associated with the maintenance of TCP congestion windows (cwnd, ssthresh and count). The structure also introduces some new variables to aid in congestion control (ownd, decr_ts) and other variables to support integrated loss recovery (pkts[]). In the following subsections, we describe how the various TCP routines use and update this new information.

New send data routine: When a connection desires to send a packet, it checks to see if the number of bytes already in the “pipe” from the sender (ownd) is greater than the desired size of the “pipe” (cwnd). If not, the connection prepares the packet

By using a separate TCP connection for each transfer, an application can avoid unnecessary coupling between parallel transfers multiplexed on the same connection. Each connection provides a reliable, ordered byte-stream abstraction independently of the others. The flow control for each connection happens independently of the others, so the delivery of data to the receiving application also happens independently for each connection.

At the same time, congestion control is integrated across the TCP connections. There is a single congestion window for the set of TCP connections between a client and a server that determines the total amount of outstanding data the set of connections can have in the network. When a loss occurs on any of the connections, the combined congestion window is halved, thereby achieving the same effect as when a single, persistent TCP connection is used.

Data-driven loss recovery is also integrated across the set of TCP connections. When a packet is lost on one connection, the successful delivery of later packets on other connections allows the sender to reliably detect the packet loss without resorting to a timeout, thereby improving performance.

With such a transport-layer solution, existing unchanged applications can set up as many TCP connections as they wish without having an adverse effect on either congestion control or loss recovery. We call the TCP resulting from this set of modifications, *TCP-INT*.

Before describing our implementation and simulation results, we discuss alternative solutions that have been proposed by other researchers and point out their shortcomings vis-a-vis our solution. A majority of these solutions operate at the application level; we discuss these in Section 5.2. One notable exception is [18], which proposes that information, such as the congestion window size and the round-trip time estimate, be shared across TCP connections. This proposal is similar to our integrated congestion control and loss recovery scheme in some respects. However, [18] does not describe how such a scheme may be implemented nor does it evaluate the potential improvement in performance.

5.2 Alternative Solution: Application-level Multiplexing

In contrast to our solution which involves changes only to the network stack, alternative solutions typically operate at the application level. The main goal of application-level solutions is to multiplex several logically-separate data streams onto a single TCP connection, thereby avoiding the use of multiple, parallel TCP connections. Application-level solutions fall into two categories: those implemented as part of applications and those that are application-independent. An example of the former in the context of Web applications is Persistent-connection HTTP (P-HTTP) [15]. Examples of the latter include the Session Control Protocol (SCP) [16] and the MUX protocol [7].

In P-HTTP, each of the Web client and the server open a single TCP connection between them and multiplex several transfers one after another onto it. Separate transfers are demarcated using content-length prefixes. The TCP connection between the server and the client is persistent so that the same connection can be used for transfers initiated at a later point in time. The use of persistent connections results in a significant decrease in latency of Web accesses, especially when the individual transfers are short and the bandwidth-delay product of the server-client path is large [15].

The SCP and MUX protocols also allow multiplexing multiple transfers, but in an application protocol-independent way. They define framing schemes that can be used to demarcate multiple transfers multiplexed onto a single TCP connection. The framing scheme has a low overhead in terms of bytes, so fine grained multiplexing is possible.

In spite of the benefits, application-level solutions have their drawbacks:

1. They require existing applications to be rewritten or at least relinked. Moreover, this is necessary both at the server and the client.
2. They do not allow multiplexing of transfers initiated by more than one application.
3. Since the transfers are multiplexed over a TCP connection which provides an ordered byte-stream abstraction, there may be undesirable coupling between the transfers. For example, the loss of a packet carrying data for one transfer could hold up the delivery of data belonging to entirely different transfers.

We discuss these in more detail in the context of HTTP/1.1.

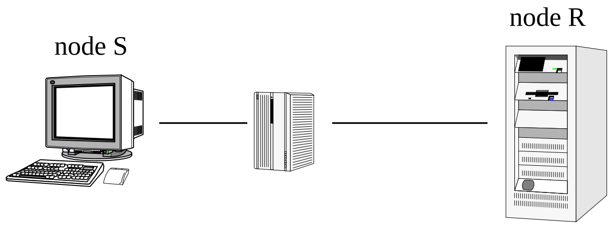

Figure 11. Topology for simulation tests.

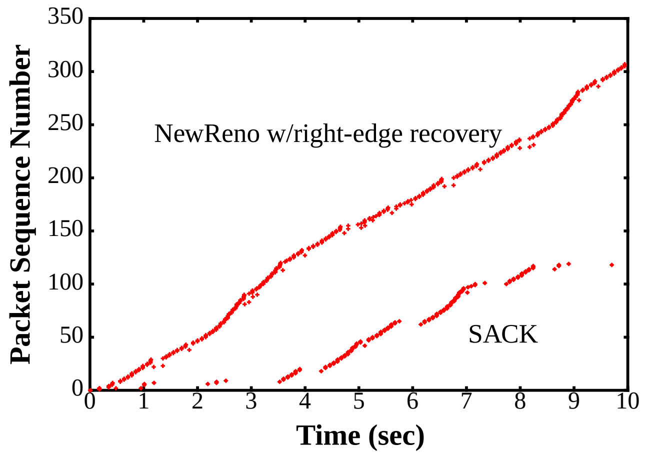

Figure 12. TCP with and without right-edge recovery.

and not to simulate any existing or typical network topology. The simulation tests consisted of a single TCP transfer from node S to node R for a duration of 10 seconds. Each test used a different variant of TCP sender protocol on node S.

Figure 12 shows the simulated sequence plots for TCP with SACK and our enhancement to TCP-NewReno (TCP-Reno with certain bug fixes described in [8]) with right-edge recovery. As indicated by the large gaps in the sequence plot, the SACK transfer experiences many more coarse timeouts than the transfer using right-edge recovery. These coarse timeouts are a result of frequent losses that generate two or fewer duplicate acknowledgments. These duplicates are not sufficient to trigger the fast retransmission algorithm. Therefore, in a traditional TCP implementation the loss is only retransmitted after a coarse timeout occurs. When the right-edge recovery algorithm is used, the sender transmits a new packet when each these duplicate acknowledgments arrive. These new packets trigger the transmission of additional duplicate acknowledgments from receiver. These additional duplicates allow the sender to use fast retransmissions to recover from the losses. By eliminating the coarse timeouts, the sender using right-edge recovery performs the transfer more than twice as fast as a standard transmitter.

5. Improving the Performance of Multiple TCP Connections

Current applications often use multiple logically separate data streams between a Web server and client. We showed in Section 3.2 that the use of independent TCP connections for each transfer has a significant impact on network behavior. In this section, we present the design and implementation of a scheme that eliminates the adverse effects of concurrent TCP connections and actually improves TCP loss recovery in such situations. This modified TCP uses integrated congestion control and loss recovery that exploits temporal parallelism and shares state across concurrent connections.

5.1 Integrated Congestion Control/Loss Recovery

The motivation behind transport-layer, kernel-level integrated congestion control and loss recovery is to allow applications to use a separate TCP connection for each transfer (just as they do today), but to avoid the problems mentioned in Section 3.2 by making appropriate modifications to the network stack. We divide TCP functionality into two categories: that having to do with the reliable, ordered byte-stream abstraction of TCP, and that pertaining to congestion control and data-driven loss recovery.

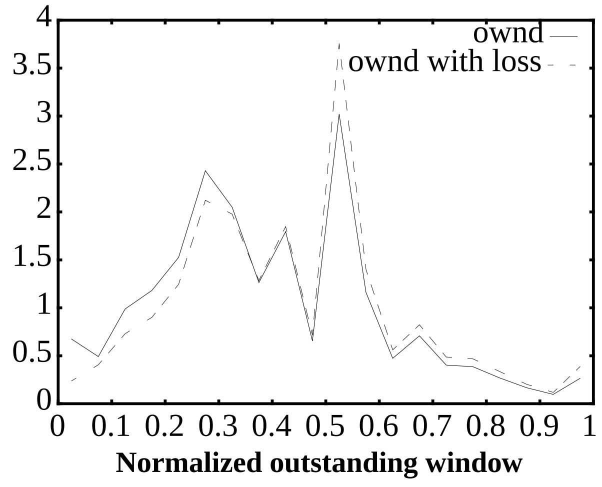

Figure 9. Probability density function (PDF) of normalized ownds of all connections and that of the connections that saw a loss during a loss epoch.

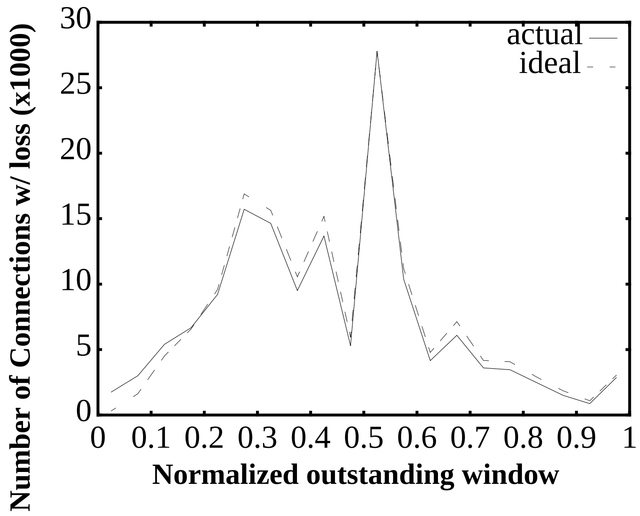

Figure 10. The number of connections, with specific values of normalized ownd, that experience a loss during a loss epoch in a 4 hour long trace (~100,000 loss epochs). Both the actual and ideal numbers are shown.

4.1 Enhanced Loss Recovery

As mentioned in Section 3.1, over 55% of all retransmissions on the Olympic Web server happened after one or more coarse timeouts kept the link to the client idle for periods from hundreds of milliseconds to seconds. Our analysis showed two main reasons for the occurrence of these timeouts:

1. Fast retransmit followed by a timeout: The TCP Reno sender was unable to recover from multiple losses within the same transmission window. This situation can be recognized by the occurrence of a fast retransmission shortly before the coarse timeout.
2. Insufficient duplicate acknowledgments: Either the number of outstanding packets was too small, or most of the packets in the window were lost, preventing the sender from receiving enough acknowledgment information to trigger a retransmission.

The use of TCP selective acknowledgments (SACKs) has often been suggested as a technique to improve loss recovery and avoid timeouts unless there is genuine persistent congestion in the network [5]. However, windows are often too small for this alone to help. A detailed analysis of the trace data showed that out of the 422,025 coarse timeouts over a 3-hour period, SACKs could have helped avoid at most 18,713 (4.43%) of them. *In other words, current approaches to TCP Reno enhanced with SACKs do not really avoid most timeouts*. It is clear that an alternative technique is needed to recover from the bulk of these losses.

During the same 3-hour period, approximately 403,312 (95.6%) coarse timeouts occurred as a result of insufficient acknowledgment information (i.e., an insufficient number of duplicate acknowledgments arrived to trigger a retransmission). Of these timeouts, no duplicate acknowledgments arrived at all for 70% of them. In these situations the network is most likely experiencing severe congestion. The best solution is for the sender to wait for a coarse timeout to occur before transmitting any packets. For the remaining timeouts (about 25% of them, in which at least a single duplicate acknowledgment arrives), we propose that a single new segment, with a sequence number higher than any outstanding packet, be sent when each duplicate acknowledgment arrives. When this packet arrives at the receiver, it will generate an additional duplicate acknowledgment. When this acknowledgment later arrives at the sender, it can be assured that the appropriate segment has been lost and can be retransmitted. We call this form of loss recovery *enhanced* or *"right-edge"* recovery. This scheme can also be effectively combined with SACK information from the receiver.

4.1.1 Simulation Results

An ns [13] based simulation, using the topology shown in Figure 11, was performed to test the enhanced loss recovery algorithm. A significant amount of additional cross traffic was generated to force the transfers through the router to cope with fre-

Thus, the throughput and congestion control analyses show that *a client using several parallel connections to connect to a server is a more aggressive user of the network than one that uses a single TCP connection.* This increases the chances that the network gets into a congested state and stays there, which adversely affects overall network performance.

3.2.3 Loss distribution analysis

We now analyze the pattern of losses in greater detail to determine how losses are distributed across parallel connections with different window sizes. If we found that during a loss epoch, connections with small outstanding windows were more likely to experience losses than connections with large outstanding windows, then that would indicate some bias in the network for or against connections with small or large window sizes. Our results show that the actual probability of a loss in a window is close to the expected probability of loss (proportional to the size of the window), with the exception that connections with a small share of the total window suffer a disproportionate number of losses.

To see this, we analyze the distribution of losses across the connections in a loss epoch. For each connection, we record its outstanding window size (*ownd*) at the start of the loss epoch and count the number of packet losses it suffered during the epoch. Since the absolute value of the outstanding window could depend on the characteristics of the network path from the server to the specific client in question, we normalize it by dividing by the total outstanding window of all the simultaneous connections put together. For plotting the graphs discussed below, we quantize the normalized ownd values into multiples of 0.05.

In Figure 9, we plot the probability density function (PDF) of the normalized ownds of all connections and that of the subset that experience a loss during a loss epoch. We observe that the latter is skewed to the right with respect to the former, which is what we would expect because having a larger outstanding window should increase a connection's chances of experiencing a loss.

To analyze the situation in more detail, we do a simple calculation. Suppose there are a total of k packet losses across all connections during a loss epoch. Then, if the network does not discriminate against or favor any connection, the probability that a connection having a normalized ownd x experiences a loss during that epoch is $(1-(1-x)^k)$.

Figure 10 shows the number of connections with specific values of normalized ownd that experience a loss during a loss epoch. These numbers are obtained by aggregating over about 100,000 loss epochs in an 4-hour long trace. We plot both the actual numbers from the trace and the ideal ones computed using the formula mentioned above.

We observe that the actual and the ideal curves match each other quite well. However, the main discrepancies are that connections with a small share of the total window suffer losses significantly more often than they should, and the total number of connections (summed over all values of normalized ownd) that experienced a loss is about 5% smaller than that predicted by the ideal model. The former means that a new connection that starts up while other connections are going strong might suffer an unfairly large number of losses. The latter implies that loss events have a tendency to be concentrated in a subset of the connections.

From this analysis, we see that *of a group of parallel connections, ones with small outstanding windows could experience a larger number of losses than their share of the total outstanding window would warrant*.

3.3 Key Results of Trace Analysis

Our analysis of the behavior of individual and parallel connection behavior has led to the following important results:

- Existing loss recovery techniques are not effective in avoiding timeouts when packet losses occur and new techniques must be developed to handle them.
- A client using a collection of parallel connections between a client and server is a more aggressive user of the network than an application that uses a single TCP connection.

In the remainder of the paper, we present sender-side modifications to TCP to solve these problems. In Section 4, we present an enhanced loss recovery scheme to improve the performance of individual connections. In Section 5, we present a new *integrated connection* approach to congestion control and loss recovery to improve the performance of parallel connections.

4. Improving Single Connection Performance

In this section, we describe techniques designed to improve the loss recovery performance of TCP transfers.

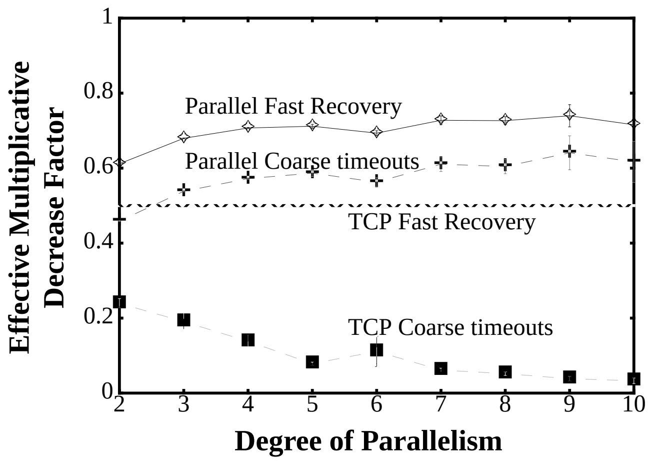

Figure 8. Effect of Parallelism on Outstanding Window size

that each of the connections has at this point in time. The time when the outstanding data of all the connections has been acknowledged marks the end of the loss epoch. For each loss epoch, we record the number of simultaneously open connections, the outstanding window size of each connection and the distribution of packet loss events across the connections. We aggregate this data across all hosts.

Suppose that of the *n* simultaneously open connections, the average number that sees a loss during a loss epoch is *m*. If *m* is on average less than *n*, then an application that has multiple open connections is more aggressive than if the connections were treated as a single unit with respect to congestion control. Since packets belonging to all the connections are very likely to traverse the same network path, a loss on any one connections should cause the entire stream of data flowing towards the client host to be slowed down. Assuming that all the TCP connections have the same congestion window size at the start of the epoch and that each TCP connection that experiences a loss halves its window, the effective multiplicative decrease in a host's total transmission window is *(1 - m/2n)*, which is no less than the factor 1/2 for a single connection. From Figure 7, we see that typically half the total number of open connections see a loss during a loss epoch, i.e., *m* is approximately *n/2* — the slope of the best-fit line through the set of points is 0.50 and the intercept is 0.1. So, the effective multiplicative decrease factor is *3/4*, which represents much more aggressive behavior than halving the window.

The above analysis makes two assumptions that may not be true in practice — that each connection has the same congestion window size, and that each halves its window in response to a loss. The *m* connections that experience losses may have had larger windows than the others. Furthermore, the loss of a packet may cause a connection to drop its window down to 1 segment (due to a retransmission timeout) rather than just halve it. Therefore, it is still possible that the congestion behavior is not as aggressive as the factor *3/4* indicates.

We determine the actual combined congestion window size after a loss by using the traces and the TCP emulation engine. We make one of two extreme assumptions about the way in which the set of connections that experience a loss during a loss epoch respond: either all do fast retransmissions (window halved) or all undergo a retransmission timeout (window dropped to 1). For each case, we compute the ratio of the combined congestion window after and before the loss event. The two cases give us bounds on the effective multiplicative decrease factor, which is shown as a function of the degree of parallelism in Figure 8 (the curves labelled “Parallel Fast Recovery” and “Parallel Coarse Timeouts”). As a baseline comparison, we also compute the congestion window size ratio that would result if the connections were treated as a single unit and (a) a fast retransmission, or (b) a coarse timeout occurred. These are shown by the “TCP Fast Recovery” and “TCP Coarse Timeouts” curves in the figure. The effective multiplicative decrease factor (backoff in response to congestion) of a system using parallel connections is typically in the range 0.6 to 0.75 for various degrees of parallelism. This is clearly much more aggressive than normal TCP which backs off by a factor of 0.5 or less.

In addition to the effective multiplicative decrease in the face of congestion losses being much milder than that of a single TCP connection, we also note that the rate at which a set of parallel connections increase their aggregate window is *n* times that of a single connection. A single connection in the congestion avoidance phase increases its congestion window by 1 segment per round trip time, so a collection of *n* parallel connections increase their combined congestion window by *n* segments per round trip time.

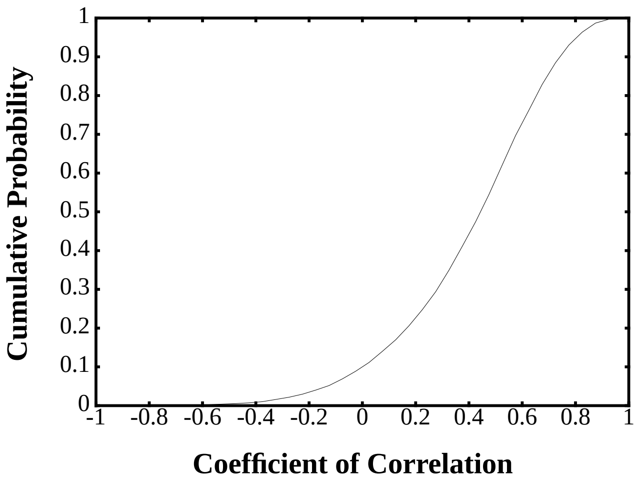

Figure 6. CDF of the coefficient of correlation between throughput and the number of simultaneous connections. The CDF is obtained by aggregating across all hosts.

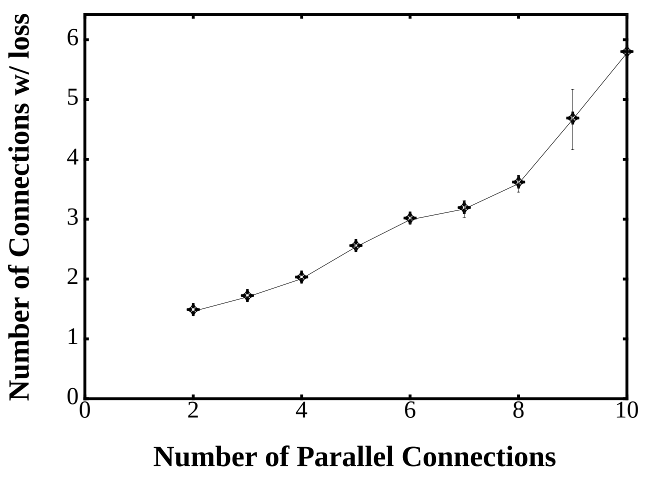

Figure 7. Average number of connections that experience a loss within the same window when parallel connections are used.

3.2 Parallel Connection Behavior

In this section, we study the effect that multiple, simultaneous connections (*parallel connections)* have on end-to-end performance and the network. We first investigate how throughput varies with the number of parallel connections (the *degree of parallelism*). We then investigate how a group of parallel connections react to a loss. We look at the number of parallel connections that experience a loss when one of the parallel connections experiences a loss, and the resulting combined congestion window size after a loss. We look at how losses are spread (in time) across parallel connections and how they are spread out as a function of the number of unacknowledged bytes (the *outstanding window*), over all parallel connections.

3.2.1 Throughput analysis

We analyze the traces to determine if there is any significant correlation between the throughput seen by a client host and the number of simultaneous connections (*n*) it has open to the server. This is important because if there is a positive correlation, then it provides a mechanism that allows applications to obtain more than their “fair share” of bandwidth on a network path’s bottleneck link.

For each host, we divide the entire duration of its interactions with the server into periods during which *n* is constant. A transition from one period to the next happens either when a connection terminates (*n* decreases by 1) or when a new connection starts up (*n* increases by 1). During each such period, we compute the throughput as the ratio of the total number of useful bytes transferred during the period by all the connections put together to the duration of the period. We only consider periods with at least 5 KB transferred so that the throughput numbers are more meaningful. After considering at all such periods, we compute the coefficient of correlation between throughput and the number of simultaneous connections. We only compute this for hosts for which we have at least 10 pairs of (throughput, *n*) samples.

By aggregating the coefficients of correlation for all hosts, we obtain the cumulative distribution function (CDF), which is shown in Figure 6. It is clear from the figure that there is a substantial positive correlation. There is a positive correlation for about 90% of the hosts and a coefficient of correlation larger than 0.5 for about 45% of the hosts. Thus, clients do help themselves by opening more simultaneous connections.

3.2.2 Congestion control analysis

The apparent benefit that an individual client derives by launching many simultaneous connections in parallel comes at the cost of global network performance. To demonstrate this, we analyze the pattern of losses seen across the connections from each individual client.

Consider a period of time when a client host has *n* simultaneously open connections. Suppose that one of the *n* connections experiences a packet loss. We use this event to mark the beginning of a *loss epoch*. We record the amount of outstanding data

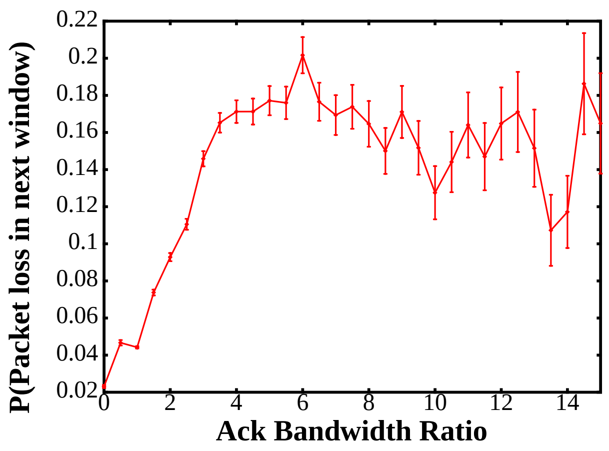

Figure 4. The probability of packet loss in the next window versus the Ack Bandwidth Ratio

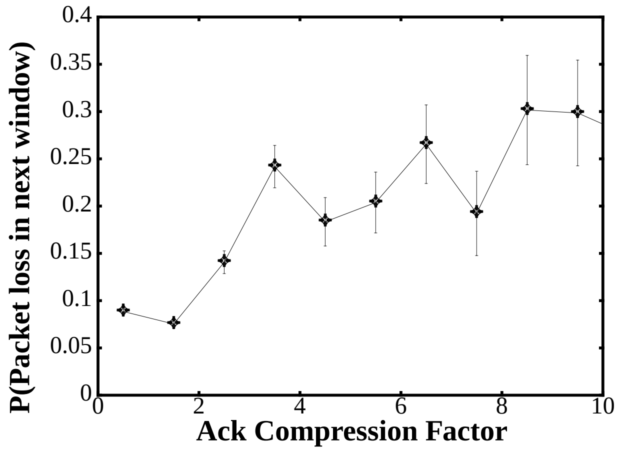

Figure 5. The probability of packet loss in the next window versus the Ack Compression Factor

A significant fraction of ratios seem large enough to cause excessive burstiness at the sender. To quantify this effect, we also calculate the probability that there is a loss in the subsequent window. If this excessive burstiness is indeed bad, we would expect that higher ack bandwidth ratios lead to an increased probability of packet losses. Figure 4 shows the results of this analysis. The *x*-axis is the (quantized) ack bandwidth ratio, and the *y*-axis is the probability that the next window contained a packet loss. Two things are important to observe from this graph:

- Even small degrees of burstiness (ack bandwidth ratio > 3) lead to a dramatic increase in the probability of a loss.
- After a threshold amount, however, additional burstiness does not lead to increased probability of packet loss, as the network has already been placed into a state of congestion.

We now discuss an alternative way of characterizing the adverse effects of ack compression, and then compare the two methods.

Method 2: Dynamic Comparison of Data and Ack Bandwidths

The second method takes into consideration the flow of both data and acks. Consider a window of data transmitted by the sender. (The window ends when an acknowledgment covering the start sequence number of the window is received.) The *data bandwidth* during the window is computed by dividing the amount of data sent by the time to transmit the window[1]. Similarly, we compute the *ack bandwidth* by dividing amount of data acknowledged by the time difference between the first and last acks. We define the *ack compression factor* as the ratio of the ack bandwidth to the data bandwidth during the same window. Note that this ratio is the same as the ratio of the ack bandwidths of consecutive windows.

A large ack compression factor during a window would cause data to be sent out at a much higher rate during the next window. As a consequence, we would expect an increase in the probability of packet loss during the next window. Figure 5 shows the probability of packet loss in the next window versus the ack compression factor in the current window. We only compute the data and ack bandwidths when the amount of data sent and acked is more than 5 KB. From the figure, we observe that the loss probability goes up while the ack compression factor increases up to about 3, and then levels off.

One advantage of this different way of defining ack compression is that the ack compression factor can easily be computed in a real TCP implementation. Unlike the first method described above and in [11], no median computation is required. When it detects a high enough level of ack compression, a TCP sender could choose to slow down its data transmission rate to decrease the chances of packet loss.

From this analysis, we see that *ack compression is an observed phenomenon and can be correlated with subsequent packet losses. Future work should investigate mechanisms for traffic shaping and sender adaptation in the presence of ack compression.*

1. Note that the data bandwidth could well exceed the quantity window size/RTT if the data is sent out as a burst.

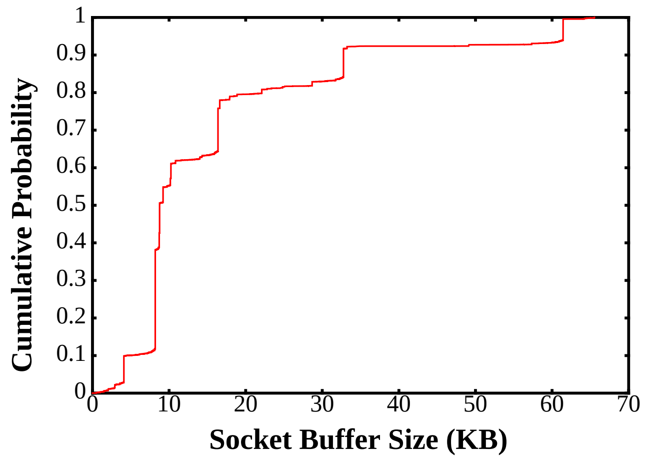

Figure 2. Cumulative distribution function (CDF) of receiver advertised window size.

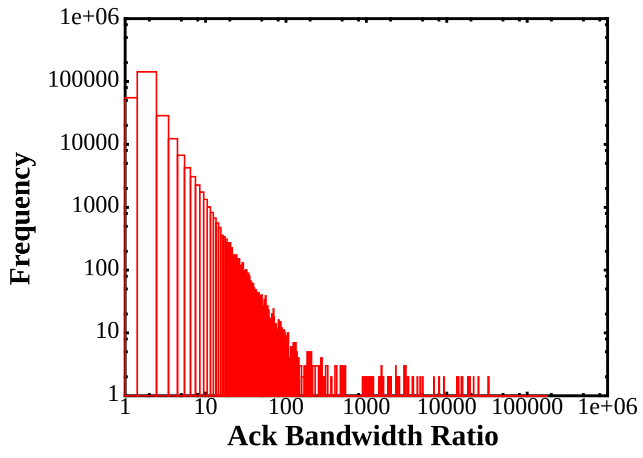

Figure 3. Histogram of Ack Bandwidth Ratios

3.1.2 Receiver-advertised window

Assuming that the source always has data to send, the amount of data transmitted by TCP at any time is primarily governed by the minimum of two parameters: the sender's congestion window, which tries to track the available bandwidth in the network, and the receiver's advertised window, which handles flow control. The maximum possible value for the latter parameter is equal to the socket buffer size chosen by the application when the connection is established. An excessively small value of this parameter could result in sub-optimal end-to-end performance if the receiver's advertised window is consistently smaller than the sender's congestion window.

Figure 2 shows the cumulative distribution function of receiver advertised window for transfers to 32000 hosts (over 1650103 connections) during a 3-hour period. The CDF shows distinct upswings at window sizes of 4 KB, 8 KB, 16 KB, etc., values that are commonly used by receivers.

For each connection, we compared the receiver advertised window with the congestion window size (as computed by the TCP engine), and determined that in approximately 14% (233906 connections) of all connections the latter grew to be larger than the former, i.e., the receiver advertised window limited the amount of data the TCP sender could have outstanding. In these cases, the Web client (the receiver) could potentially have obtained a higher throughput had it employed a larger socket buffer (and consequently advertised a larger window).

From this analysis, we make the following recommendation. *Future network implementations should increase their default socket buffer size to avoid the receiver window from becoming a bottleneck.* Default values of 4 KB are often too small.

3.1.3 Ack Compression

Ack compression [20] occurs when the spacing between successive acknowledgments is compressed while they are in transit between the receiver and sender. The acknowledgments then arrive at the sender at a higher rate than they were generated by the receiver. This disturbs the ack-clocking nature of TCP [9], causing the sender to transmit a burst of packets. This burst of packets is undesirable as it increases the likelihood of overflowing a queue at a network router, leading to packet loss. We wanted to determine the degree to which acknowledgments are compressed in the network and to quantify the negative effect that this has on TCP senders. We did this analysis in two different ways.

Method 1: Ack Bandwidth Ratio [11]

In this method, we calculate the *ack bandwidth* for TCP windows for all connections in the trace. The ack bandwidth with respect to a starting acknowledgment *A* is defined as the number of outstanding bytes at the time *A* was received divided by the time to receive the ack for the last outstanding byte. Each individual ack bandwidth sample is then compared to the median ack bandwidth for all windows in the connection. This ratio (ack bandwidth)/(median ack bandwidth) allows us to quantify the degree to which acks are compressed.

Figure 3 shows the histogram of ack compression ratios for the trace on a log-log scale. The *x*-axis is the (quantized) ack bandwidth ratio, and the *y*-axis is the frequency with which that ratio was observed in the trace. The ack bandwidth ratio crosses several orders of magnitude, but the probability of large ratios decreases as the ratio increases.

Trace Statistic	Value	%
Total connections	1650103	
With packet re-ordering	97036	6
With receiver window as bottleneck	233906	14
Total packets	7821638	
During slow-start	6662050	85
# of slow-start pkts lost	354566	5
During congestion avoidance	1159588	15
# of congestion avoidance pkts lost	82181	7
Total retransmissions	857142	
Fast retransmissions	375306	44
Slow-start retransmissions	59811	7
Coarse timeouts retransmissions	422025	49
Avoidable with SACKs	18713	4
Avoidable with enhanced recovery	104287	25

TABLE 3. Summary of Analysis Results (percentages are relative to the category above it)

3.1 Single Connection Behavior

Table 3 summarizes the results of the analysis of single connection behavior. In the rest of this section, we discuss the performance and behavior of retransmissions, loss recovery, receiver-advertised window sizes, and ack compression.

3.1.1 Retransmissions and Loss Recovery

We classify the retransmissions of lost segments in TCP Reno into three categories — *fast retransmissions*, which are triggered when a threshold number of duplicate acknowledgments (three in TCP Reno) are received by the sender, *timeouts*, which are retransmissions triggered by the expiration of a timer before the arrival of an acknowledgment for the missing segment, and *slow-start retransmissions*, which are retransmissions performed by the sender immediately after a timeout, for subsequent packets that were presumed lost in the window. There are typically two situations that result in coarse timeouts. In TCP Reno, the loss of multiple segments in a window usually leads to a coarse timeout. Coarse timeouts also occur when the number of duplicate acknowledgments is insufficient to trigger a fast retransmission.

Running the TCP emulation engine against the observed acknowledgment trace showed that timeouts and the subsequent slow-start retransmissions are the predominant mechanism for loss recovery in TCP Reno. Specifically, over a 3 hour trace involving 1,650,103 connections and 285,979 individual retransmission events, we found that 49.3% of all retransmissions were due to timeouts, 43.8% were the result of a TCP fast retransmission, and 6.9% were the result of slow start retransmissions. That is, 56.2% of all retransmissions occurred soon after a coarse timeout.

We also characterized the state of the sender at the time a loss occurred into slow-start periods and congestion avoidance periods, to determine if either mode shows inherently different behavior from the other. Both congestion avoidance and slow-start have similar frequencies of loss: 82,181 (7%) out of 1159588 packets were lost in congestion avoidance and 354566 (5%) out of 6662050 packets were lost in slow-start.

From this analysis, we can see that *existing loss recovery techniques are not effective in dealing with packet losses and new techniques must be developed to handle them*. This result is somewhat disappointing, because it implies that in practice, the more sophisticated data-driven loss recovery mechanisms are not being used and that there is a heavy dependence on the timeouts for loss recovery. In Section 4.1, we discuss the effectiveness of the standard TCP Selective Acknowledgment (SACK) option in reducing the number of timeouts, and describe an enhancement to the loss recovery procedure to significantly improve its performance.

In order to faithfully reproduce the state of the sender-side TCP implementation, we wrote a *TCP emulation engine* that took the captured acknowledgments as input and reproduced the evolution of the sender-side state variables (e.g., congestion window, round-trip time estimate, maximum packet sequence transmitted, etc.). The captured acknowledgments are sufficient for this because the change in any sender-side state variable is triggered by the reception of an acknowledgment or the expiration of a timer.

In essence, the engine is an implementation of a subset of the functions of a TCP sender at user-level, driven by the sequence of acknowledgment traces. If the evolution of the sender's state were completely driven by the arrival of acks (as is usually the case if a connection experiences no losses), it is possible to reconstruct it with complete accuracy. We handled the complications due to timer events by using a number of heuristics, in addition to estimating the round-trip time as faithfully as done by the real TCP sender.

We validated our engine by comparing the times at which it predicted a retransmission event with the actual times that retransmissions occurred, as reported by the modified server nodes and captured by the trace collector. When completed, the engine predicted the number of coarse timeouts, fast retransmissions, and slow-start retransmissions to within 0.75% of the correct number (even assuming that no retransmission notifications were lost by the packet capture process). As a result, all the window size calculations are also accurate to within this bound plus the underlying loss rate in the packet capture process (about 0.5% on average).

In summary, the TCP emulation engine is a tool that emulates the evolution of a TCP sender's state with a very high degree of accuracy. The emulation engine, combined with the collected traces, allow us to analyze how TCP connections as used by Web browsers behave in today's Internet.

3. Analysis

In this section, we present the results from our analysis of the real-world behavior of TCP connections. We first consider single-connection behavior (analyzing TCP connections independent of each other), analyzing how well the different TCP loss recovery mechanisms work in practice and looking for the presence of receiver bottlenecks and ack compression. We then analyze the effects that multiple simultaneous connections have on the network, focusing on the loss recovery behavior and congestion window evolution of parallel connections. In all the graphs that have error bars, the error bars represent a confidence interval of plus or minus one standard error.

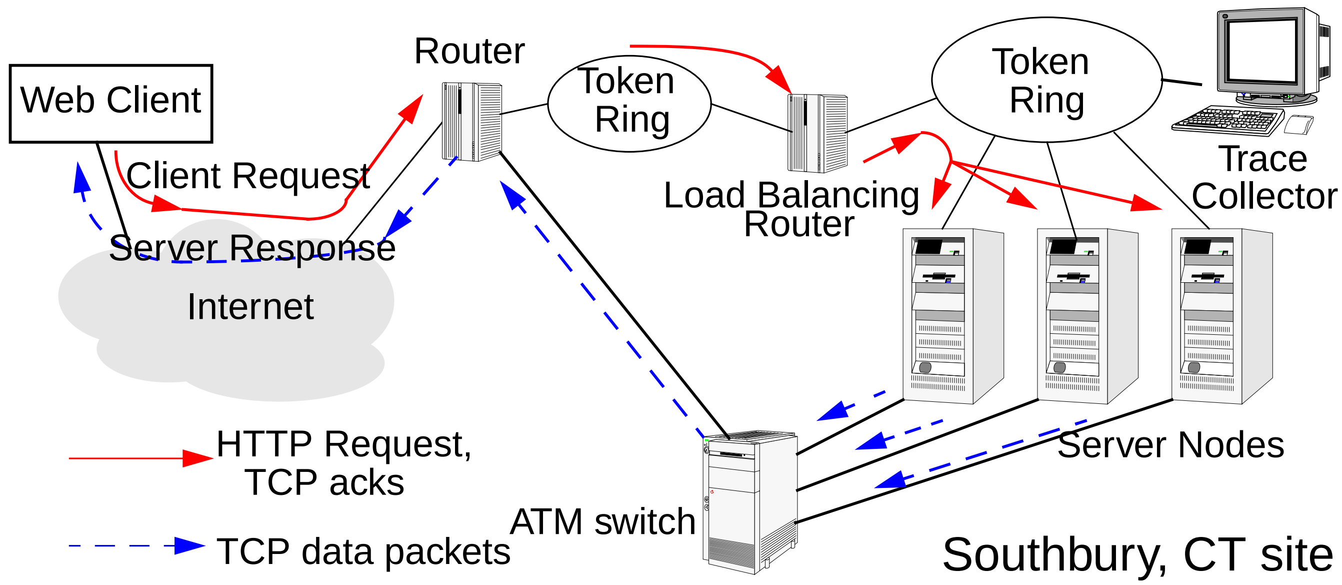

Figure 1. WWW Server and Monitoring Setup

Trace Statistic	Value
Packets captured	1,540,312,422
Packets dropped	7,677,854
Average packet drop percentage	0.498%
Distinct client addresses	721,417
Total Bytes Collected	~189 GB

TABLE 1. Trace Statistics

Site Statistic	Value
Total server hits during Olympics	190,000,000
Max hits/day	16,955,000

TABLE 2. Site Statistics

we collected approximately 1 GB of compressed packet headers per hour. Periodically, the packet trace files were dumped onto tape. While dumping the trace files to tape, however, there were occasional periods during which the traffic was not monitored.

Due to the location of the trace collection machine and the asymmetric path of request and response data, we could only capture packets coming from the clients *into* the Web server complex. These packets include the initial TCP SYN packet from the client, the HTTP request packets, as well as all TCP acknowledgments for data sent from the server. In particular, this implies that we did not explicitly capture any data packets from the server to the client. However, this does not prevent us from determining the packet sequence trace as a function of time, since we can estimate this from the sequence of TCP acknowledgments. In addition, we modified the TCP stack on a subset of the servers. These modified server nodes transmit the TCP/IP header, together with information on the current sender congestion window size, smoothed round-trip time estimate, and number of duplicate acknowledgments causing the retransmission, of any retransmitted packet on the token ring interface to our trace collection machine. This retransmission information combined with the acknowledgement traces and the knowledge of the sender's TCP algorithms at the server allow us to reconstruct the outgoing data stream with high accuracy.

The following tables and figures summarize a variety of interesting information about the data collected. Table 1 describes various statistics about our trace data while Table 2 contains data about activity at the site during the Olympics. More details about these traces and a study of the stability of wide-area network performance based on them can be found in [3].

2.3 TCP Emulation Engine

We post-processed the packet traces to generate a one-record summary of the progress of each TCP connection, including the times and sequence numbers of acknowledgments as well as the server-side retransmission notifications. We then organized the connection summaries into a database that allowed us to cluster together connections from a single client and compare the TCP performance of different clients at different times.

- *Ack compression is an observed phenomenon and can be correlated with subsequent packet losses.* Future work should investigate mechanisms for traffic shaping and sender adaptation in the presence of ack compression.
- *A client using a collection of parallel connections to connect to a server is a more aggressive user of the network than one that uses a single TCP connection.* Throughput is positively correlated with the number of active connections. When multiple connections are concurrently active and one of them experiences a loss, only half of the remaining ones experience a loss on average. The combined congestion window of a group of parallel connections does not decrease as much as the congestion window of a single connection after a packet loss.
- *Of a group of parallel connections, ones with small outstanding windows could experience a larger number of losses than their share of the total outstanding window would warrant.* This means that it may be harder to initiate a new connection than to keep an existing connection going.

Based on these results, we propose several modifications and improvements to TCP to overcome or eliminate these observed problems. Our proposed modifications are completely compatible with existing TCP implementations and can be incorporated with little effort. Furthermore, we consider solutions that only modify the sender-side TCP implementation. In this way, current Web servers (and TCP senders) can benefit from these modifications while leaving Web clients untouched; this facilitates incremental deployment in the Internet. Our modifications fall into two classes — ones designed to improve the performance of individual TCP connections and ones designed to improve the performance of multiple concurrent TCP connections. The major enhancements we propose are:

- *Enhanced TCP loss recovery:* An improved loss recovery technique which improves the effectiveness of TCP fast retransmission when window sizes are small. With this modification, 25% of the timeout events observed in the trace could have been avoided.
- *Integration across concurrent connections:* An integrated approach to congestion control and loss recovery across multiple simultaneous TCP connections between a pair of hosts. This allows (unmodified) applications to open as many TCP connections as they wish to another host without adversely affecting either their own performance or that of others.

The rest of this paper is organized as follows. Section 2 describes the details of the traffic collection site, data collection, and post-processing. Section 3 presents the results of the traffic analysis. Section 4 presents our modifications to TCP to improve the performance of individual connections, and Section 5 presents our modifications to improve the performance of concurrent TCP connections. Section 6 presents our conclusions and pointers to future work.

2. Data Collection and Web Server Setup

This section describes the details of the trace collection and the post-processing we performed on the packet traces. We start with a brief description of the Web server location and configuration.

2.1 Data Collection Site

The Web traces used in this paper come from the primary Web site (in Southbury, CT) for the Atlanta Olympic games, ran by IBM (http://www.atlanta.olympic.org). Figure 1 shows the topology of the Web site's network and the hardware used at the site. During the Olympics, the Web site was connected via a T3 link to each of the 4 U.S. Network Access Points (NAPs), located in Chicago (Bellcore and Ameritech), the San Francisco Bay Area (Bellcore and Pacific Bell), New York (Sprint), and Washington, D.C. (MFS Datanet). In addition, there were mirror sites at Cornell (N.Y.), Keio (Japan), Karlsruhe (Germany) and Hursley (U.K.), for which we did not collect data. More details about the site structure and software are available from [14].

Requests for data from any client arrive at the web site after being routed through the appropriate Internet NAP. These requests are passed to a load-balancing connection router [1, 19], that distributes them across several server nodes. These nodes retrieve the appropriate Web objects and transmit them across an internal ATM network and through the Internet to the clients.

2.2 Data Collection Methodology

We monitored all the traffic coming into the site and obtained packet-level traces of this traffic by running the tcpdump [10] utility on a machine placed on the token ring connecting the load-balancing router to the server nodes (Figure 1). This machine was an IBM 150Mhz Pentium Pro PC running BSD/OS 2.1 from BSDI. We extracted and stored the first 350 bytes of every packet destined to TCP port 80 (the HTTP port), compressing this data on-the-fly using the gzip utility. In addition, the server also transmitted other data (such as streaming audio) on other ports. We did not capture this data due to limited storage space of our setup and the high data packet capture rates. Even while restricting ourselves to traffic on port 80, during peak periods

The majority of solutions that have been proposed to address these problems involve the use of persistent connections. Examples include persistent-connection HTTP (P-HTTP) [15], Session Control Protocol (SCP) [16] and persistent connections in HTTP/1.1 [6]. The main idea in these is to multiplex several logical connections onto a single, persistent TCP connection between a server and a client.

While persistent connections certainly help, they have the drawback that they are specifically tied to HTTP. This means that TCP connections initiated by other non-HTTP applications running on the same machine (such as FTP applications or Netscape plug-ins [12]) would not be able to share the persistent connection with HTTP. Moreover, if HTTP were to be replaced by a different protocol in the future, special effort would have to be undertaken again to ensure that the same drawbacks do not recur. Also, the ordered byte-stream abstraction of TCP could result in an undesirable coupling between the logical connections that are multiplexed onto a persistent connection.

In contrast, our approach is to identify the basic issue — providing efficient support for numerous, short logically different transfers — and to provide a solution at the transport layer that operates independently of the application and the application-level protocol. Applications are not required to be modified in any way; any application that uses TCP connections (not necessarily persistent connections) can benefit from our transport-level solution.

In the remainder of this section, we present a brief overview of the paper. In the first part, we present a detailed analysis of the behavior and performance of TCP as currently used by HTTP. The results of this analysis highlight several problems that lead to reduced end-to-end performance. In the second part, we use these results to motivate and design a set of TCP improvements that alleviate many of these problems.

The goal of our analysis is to answer the following questions. We divide them into two categories: those concerning the behavior of individual TCP connections, and those concerning the combined behavior of concurrent connections used by Web browsers.

Single Connection Behavior:

1. *Loss recovery:* Do current TCP senders recover from packet losses using data-driven loss recovery mechanisms such as fast retransmission and recovery, or do they tend to depend on coarse timeouts when losses occur? What would be the potential benefits of using more sophisticated loss recovery mechanisms such as selective acknowledgments (SACKs)?

2. *Receiver bottleneck:* How often do receiver-advertised windows restrict the performance of a connection? Are typical receiver advertised window sizes large enough to utilize the more sophisticated loss recovery mechanisms? Are the effective window sizes available to senders limited by small receiver-advertised window sizes?

3. *Ack compression:* Ack compression [11,20] occurs when the spacing of acknowledgments is compressed, disrupting the ack-clocking nature of TCP. How often does this phenomenon occur? How is it correlated with congestion-related phenomena such as packet losses?

Parallel Connection Behavior:

1. *Throughput:* How does the throughput seen by a client vary with the number of connections?

2. *Congestion control*: When one of a group of concurrent connections experiences a loss, how many other connections of the group experience a loss around the same time? What is the effective decrease in congestion window size after a packet loss?

3. *Loss behavior*: When an application uses concurrent connections, are these connections treated equally by the network with respect to packet losses?

To answer these questions, we analyze a very large data set — packet-level traces of millions of TCP connections collected from the official Web server for the 1996 Atlanta Olympic games. Over a 3-week period, the traces included approximately 1.5 billion TCP packets from 700,000 distinct hosts from all over the world.

Our analysis of the traces yield the following answers to the questions raised above:

- *Existing loss recovery techniques are not effective in dealing with packet losses and new techniques must be developed to handle them*. Over 50% of all losses require a coarse timeout to recover. Fast retransmissions recover from less than 45% of all losses. The remainder of losses are recovered during slow start following a timeout.

- *Future network implementations should increase their default socket buffer size to avoid the receiver window from becoming a bottleneck.* The socket buffer size limited the throughput of approximately 14% of all observed connections.

TCP Behavior of a Busy Internet Server: Analysis and Improvements

Hari Balakrishnan*, Venkata Padmanabhan*, Srinivasan Seshan+, Mark Stemm*, Randy H. Katz*
{hari,padmanab,stemm,randy}@cs.berkeley.edu, srini@watson.ibm.com

**Computer Science Division*
University of California at
Berkeley Berkeley, CA 94720

+IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

The rapid growth of the World Wide Web in recent years has caused a significant shift in the composition of Internet traffic. Although past work has studied the behavior of TCP dynamics in the context of bulk-transfer applications and some studies have begun to investigate the interactions of TCP and HTTP, few have used extensive real-world traffic traces to examine the problem. This interaction is interesting because of the way in which current Web browsers use TCP connections: multiple parallel short connections from a single host.

In this paper, we analyze the way in which Web browsers use TCP connections based on extensive traffic traces obtained from a busy Web server (the official Web server of the 1996 Atlanta Olympic games). At the time of operation, this Web server was one of the busiest on the Internet, handling tens of millions of requests per day from hundreds of thousands of clients. We first describe the techniques used to gather these traces and reconstruct the behavior of TCP on the server. We then present a detailed analysis of TCP's loss recovery and congestion control behavior from the recorded transfers. The two most important results are that: (1) short Web transfers lead to poor loss recovery performance for TCP, and (2) concurrent connections cause an overly aggressive use of the network. We then discuss techniques designed to solve these problems. To improve the loss recovery performance of short transfers, we present a new technique for TCP loss recovery. To improve the congestion control and loss recovery performance of multiple parallel connections, we present a new transport-level *integrated connection* approach to congestion control and loss recovery. Simulation and trace analysis results show that our enhanced loss recovery scheme could have eliminated 25% of all timeout events, and our integrated connection approach provides greater fairness and improved startup performance for parallel connections. Our solutions are more general than a specific protocol enhancement such as the use of persistent connection in P-HTTP [15] and HTTP/1.1 [6], and include techniques, such as improved TCP loss recovery, that are not addressed by these protocols.

1. Introduction

The rapid growth of the World Wide Web in recent years has caused a significant shift in the composition of Internet traffic. Today, Web traffic forms the dominant component of Internet backbone traffic. For example, [2] reports that Web traffic constituted 50% of the packets and bytes traversing a busy backbone link. Therefore, there is significant value in understanding Web traffic characteristics and its implications for network performance and protocol design.

Web data is disseminated to clients using the HyperText Transfer Protocol (HTTP) [4], which uses TCP [17] as the underlying reliable transport protocol. The characteristics of Web traffic are significantly different from those generated by traditional TCP applications such as FTP and Telnet. Current Web transfers are typically shorter than FTP transfers, and often several logically separate transfers are active at any given time (for example, separate transfers for the text and the inlined images constituting a Web page). This, coupled with the single, reliable byte-stream abstraction provided by TCP, has resulted in two distinctive characteristics of HTTP: (i) each TCP connection initiated by HTTP tends to be short because a separate connection is used to transfer each component of a page, and (ii) Web browsers often launch multiple simultaneous TCP connections to a given server to reduce user-perceived latency.

Unfortunately, these transfer characteristics are poorly suited to current TCP mechanisms. The short duration of connections gives TCP limited opportunity to probe the network and adapt its congestion control parameters to the characteristics of the network. The use of multiple, concurrent connections by Web clients causes overall network performance to degrade and is detrimental to the performance seen by other hosts.