TCP Behavior of a Busy Internet Server: Analysis and Improvements

Hari Balakrishnan*, Venkata N. 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 concurrent 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 several hundred thousand 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. Our two most important results are: (1) short Web transfers lead to poor loss recovery performance for TCP, and (2) concurrent connections are overly aggressive users of the network. We then discuss techniques designed to solve these problems. To improve the data-driven loss recovery performance of short transfers, we present a new enhancement to TCP's loss recovery. To improve the congestion control and loss recovery performance of parallel TCP connections, we present a new *integrated* approach to congestion control and loss recovery that works across the set of concurrent connections. Simulations and trace analysis show that our enhanced loss recovery scheme could have eliminated 25% of all timeout events, and that our integrated approach provides greater fairness and improved startup performance for concurrent connections. Our solutions are more general than application-specific enhancements such as the use of persistent connections in P-HTTP [13] and HTTP/1.1 [7], and addresses issues, such as improved TCP loss recovery, that are not considered by them.

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) [5], which uses TCP [15] 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 could exacerbate network congestion.

A solution that has been proposed in the literature is to have a single, long-lived TCP connection onto which several logical data streams are multiplexed by the application. Examples include persistent-connection HTTP (P-HTTP) [13], Session Control Protocol (SCP) [14] and persistent connections in HTTP/1.1 [7]. While these schemes do help improve performance, they have drawbacks. They are specifically tied to a single application and/or application-level protocol (HTTP). Also, by multiplexing data streams onto a single TCP connection, they introduce undesirable coupling between the streams that might otherwise be independent.

The performance problems mentioned above and the limitations of existing solutions motivate our work. We analyzed a large packet-level trace of TCP connections to a busy server to determine exactly what the performance limitations were. We then designed new transport-level mechanisms to address these performance problems. Since these mechanisms are application-independent, they benefit all applications that use TCP. We used simulations to evaluate the performance benefits of these mechanisms. We are currently in the process of implementing these in a real network stack.

The goal of the trace 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:* How effective are TCP's fast retransmission and recovery mechanism in avoiding timeouts?
2. *Receiver bottleneck:* How often is a TCP receiver's advertised window size the limiting factor for performance?
3. *Ack compression:* How often does ack compression [11,18] occur and how does it impact the packet loss rate?

Parallel Connection Behavior:

1. *Throughput:* How does the throughput seen by a client vary with the number of connections it has open to a server?
2. *Congestion control*: How does a set of parallel connections as a whole respond to congestion-induced packet losses?
3. *Loss behavior*: How are packet losses distributed across multiple parallel connections as a function of the congestion windows of the individual connections?

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.* Almost 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.
- *Ack compression is an observed phenomenon and can be correlated with subsequent packet losses.*
- *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 parallel connections. Further, the set of parallel connections is less responsive to congestion-induced losses than a single connection.
- *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 new TCP mechanisms to overcome or eliminate these observed problems. These mechanisms are completely compatible with existing TCP on-the-wire protocol and only require modifications to the sender-side TCP implementation. Therefore, current Web servers (and other TCP senders) can benefit from these mechanisms while leaving Web clients untouched. This facilitates incremental deployment in the Internet. Our proposed mechanisms fall into two classes — those designed to improve the performance of individual TCP connections and those 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.

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

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 briefly describes the details of the trace collection and the post-processing we performed on the packet traces. In a related paper that used the same data [4] we describe the setup in more detail.

2.1 Data Collection Site and Methodology

The server complex consisted of a cluster of IBM RS/6000 machines connected via a 16 Mbit/sec token ring network, running the Reno version of TCP [15]. This complex was connected to the internet at each of the 4 US Network Access Points (NAPs). A WWW request entering the server complex passed through a load-balancing connection router [1, 17] that sent it across the token ring network to a single server node. After fetching the appropriate web object, the server node transmitted it across an internal ATM network and through the Internet to the clients. Note that this is an asymmetric topology — web requests and responses traversed different network links.

Our trace collection machine was on the token ring network connecting the server nodes, capturing all traffic on port 80 (the HTTP port). This traffic consisted of the HTTP request as well as all TCP acknowledgements for the HTTP transfers. We also modified the server nodes to indicate when a TCP retransmission occurred by sending a packet to our trace collection machine. Using this information combined with the techniques described in the next section, we were able to reconstruct the dynamics of the TCP connections.

The following tables and figures summarize a variety of interesting information about the collected data. 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 [4].

2.2 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.

In order to faithfully reproduce the state of the sender-side implementation of TCP-Reno, 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 done by the 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.

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.

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)

3.1.1 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. A coarse timeout also occurs 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*. 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 Acknowl-

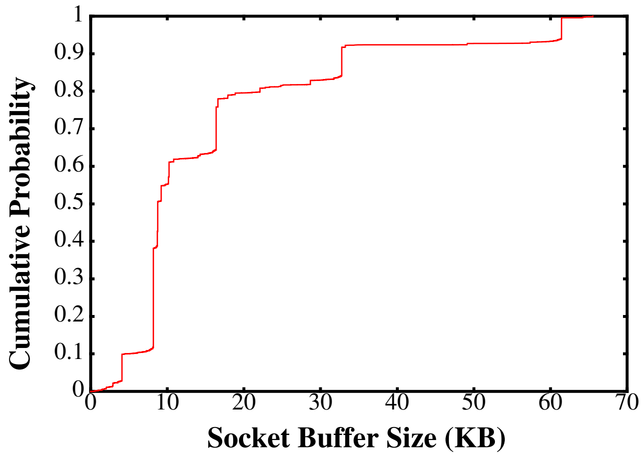

Figure 1. Cumulative distribution function (CDF) of receiver advertised window size.

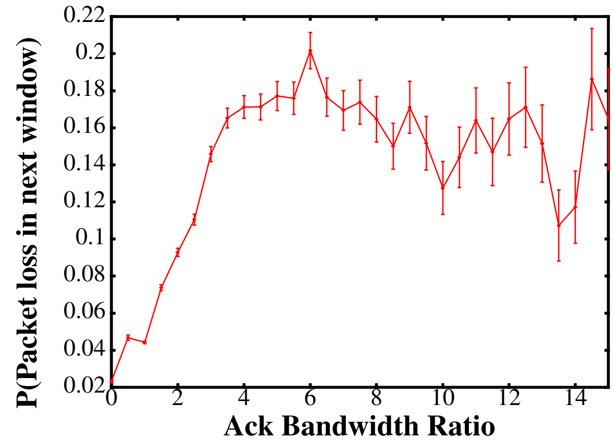

Figure 2. The probability of packet loss in the next window versus the Ack Bandwidth Ratio

edgment (SACK) option in reducing the number of timeouts, and describe an enhancement to the loss recovery procedure to significantly improve its performance.

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 1 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 [18] 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 [10], 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 2 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. We make two important observations. First, the probability of loss increases sharply for small values of ack bandwidth ratio. But beyond a point, it flattens out because the network has already been placed into a state of congestion.

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 *data bandwidth* during the window is computed by dividing the amount of data sent by the time to transmit the window[1]. Similarly, the *ack bandwidth* is computed 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 because acks are used to clock out data, this ratio is the same as the ratio of the ack bandwidths of consecutive windows.

Figure 3, which is analogous to Figure 2, shows the impact of ack compression factor during a window on the packet loss rate during the next window. We observe that the loss probability grows while the ack compression factor increases up to about 3, and then levels off.

---

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

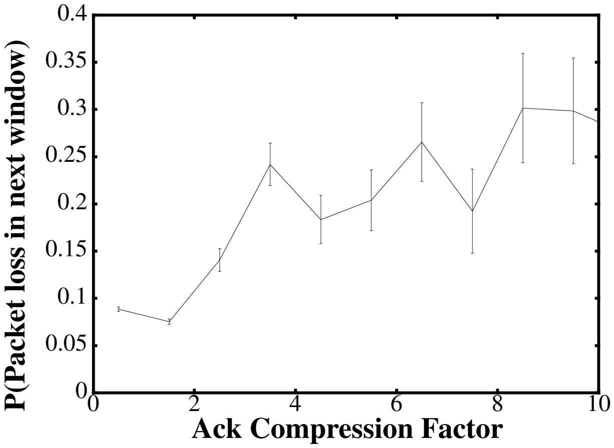

Figure 3. The probability of packet loss in the next window versus the Ack Compression Factor

Thus we see that both the ack bandwidth ratio and the ack compression factor can serve as good indicators of packet losses induced by ack compression. However, one significant advantage of the latter is that it can easily be computed in a real TCP implementation. Unlike the ack bandwidth ratio (which is derived from [11]), the ack compression factor does not require computing the median.

From this analysis, we see that *ack compression is an observed phenomenon and can be correlated with subsequent packet losses.* Therefore, a TCP sender would benefit by keeping track of the severity of ack compression (for instance, by dynamically computing the ack compression factor) and taking corrective action (such as slowing down its data rate) when there is a significant danger of packet loss.

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

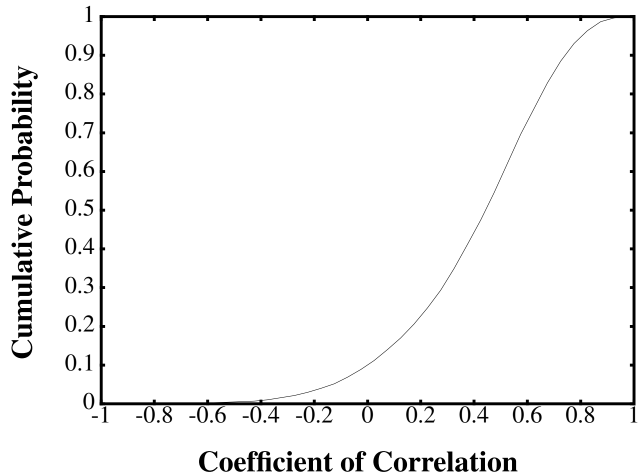

Figure 4. CDF of the coefficient of correlation between throughput and the number of simultaneous connections. The CDF is obtained by aggregating across all hosts.

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 4. 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 degraded congestion control behavior. 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 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.

Of the set (of size n) of connections that are active during a loss epoch, only the subset (of size m) that actually experiences a loss during a loss epoch cuts down the individual congestion windows in response. 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 $(1 - m/2n)$. From Figure 5, we see that typically half the total number of open connections see a loss during a loss epoch, i.e., m is approximately $n/2$. So the effective multiplicative decrease factor is $3/4$,

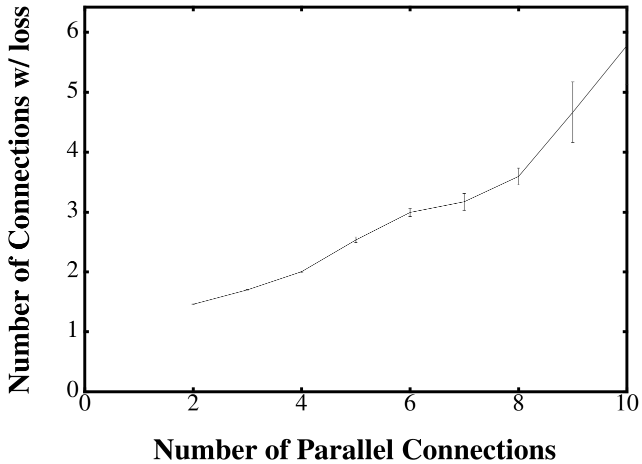

Figure 5. Average number of connections that experience a loss within the same window when using parallel connections.

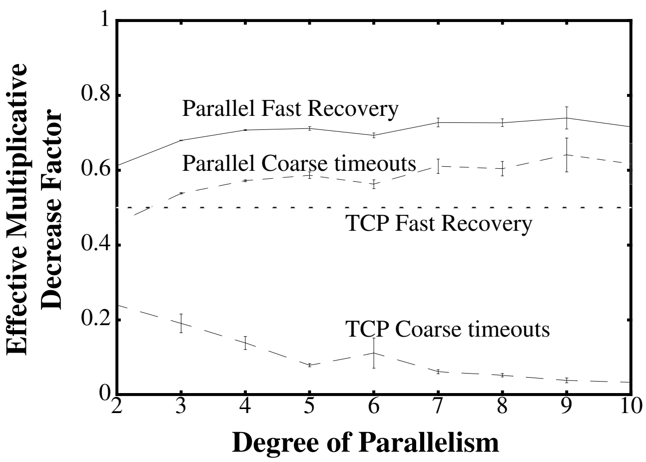

Figure 6. Effect of Parallelism on Outstanding Window size

which represents a much more aggressive behavior than halving the window. The basic reason for this is that only a subset of connections is notified of congestion occurring along path shared by all the connections.

We complement this back-of-the-envelope calculation with bounds on the actual multiplicative decrease factor. In order to compute these bounds, we use our knowledge of the individual congestion windows at the start of the loss epoch coupled with one of two extreme assumptions about how the set of connections that experience a loss during a loss epoch respond. To obtain an upper bound, we assume all do fast retransmissions and halve their windows. To obtain a lower bound, we assume that all undergo a retransmission timeout and drop their windows down to 1 segment. For each case, we compute the ratio of the combined congestion window after and before the loss event to obtain a bound on the effective multiplicative decrease factor. These bounds are shown as a function of the degree of parallelism in Figure 6 (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 significantly more aggressive than normal TCP which backs off by a factor of 0.5 or less.

Furthermore, the rate a set of n parallel connections increase their aggregate window is n times that of a single connection because each connection increments its window by 1 segment per RTT.

In short, our 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

Now we turn to the question of how packet losses during a loss epoch are distributed across the parallel connections as a function of their individual window sizes. In the interest of space, we only provide a brief summary here; a more detailed analysis can be found in [3].

For each connection, we normalized the size of its outstanding window (*ownd*) by dividing by the sum of the outstanding windows of all the parallel connections. For each loss epoch, we recorded the normalized ownds of the individual connections and whether or not they experienced a loss during the epoch. We aggregated this information across a 4-hour long trace with over 100,000 loss epochs. The main result we obtained was that *of a group of parallel connections, ones with small outstanding windows (less than 20% of the total ownd) could experience a larger number of losses than their share of the total outstanding window would warrant*. Thus a new connection that starts up while other transfers are in progress might suffer an unfairly large number of losses.

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.

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 millisec-

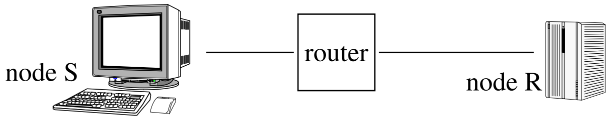

Figure 7. Topology for simulation tests.

onds 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 [6]. 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. However, in general, it is incorrect to retransmit packet prior to the arrival of at least three duplicate acknowledgments because packets can be reordered in the network.

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 is orthogonal to SACKs and can be effectively combined with SACK information from the receiver for better overall performance.

4.1.1 Simulation Results

An ns [12] based simulation, using the topology shown in Figure 7, 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 frequent losses and small congestion windows. This was done to recreate situations that occur in the traces in a controlled fashion 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.

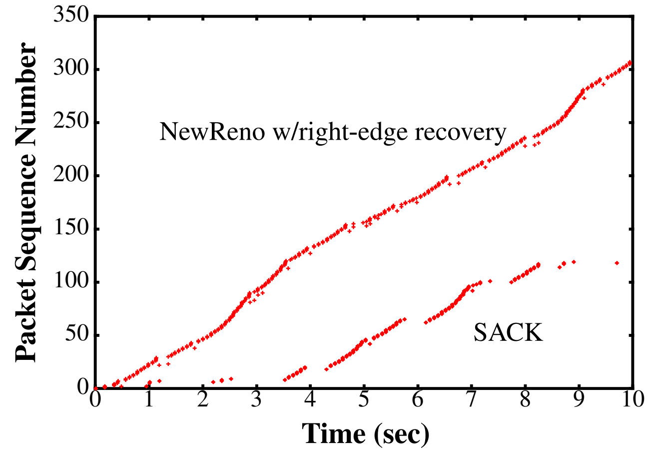

Figure 8. TCP with and without right-edge recovery.

Figure 8 shows the simulated sequence plots for TCP with SACK and our enhancement to TCP-NewReno (TCP-Reno with certain fixes described in [9]) 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 usually happen because there aren't enough duplicate acks to trigger fast retransmission. When the right-edge recovery algorithm is used, the sender transmits a new packet in response to each of these duplicate acknowledgments. These cause the receiver to generate additional duplicate acknowledgments, which trigger fast retransmission at the sender. 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

In this section, we present the design and implementation of an integrated congestion control and loss recovery scheme that enables the use of multiple parallel TCP connections without resulting in more aggressive congestion behavior. In addition, this scheme enables better loss recovery and startup performance for new connections.

5.1 Integrated Congestion Control/Loss Recovery

The motivation for 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. The latter is done in an integrated manner across the set of parallel connections. We call this modified version of TCP, *TCP-Int*.

By opening *n* separate TCP connections for a transfer, an application has *n* logically independent reliable byte-streams available for use. 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. In addition, because the congestion window is shared across connections, new connections do not have to undergo slow-start to estimate the correct congestion window. This leads to improved startup performance for additional connections.

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.

We briefly discuss how TCP-Int is different from alternative solutions proposed in the literature. Most of these involve the application multiplexing several data streams onto a single TCP connection. This is different from TCP-Int that operates within the network stack and permits the use of multiple TCP connections. We present a more detailed discussion in Section 5.2.

One notable exception among these alternative solutions is [16], which proposes the sharing of information, such as the congestion window size and the round-trip time estimate, across TCP connections. While this is similar to TCP-Int in some respects, there are many differences, including the lack of integrated loss recovery and the retention of a *per-connection* congestion window. Furthermore, it does not describe an implementation nor evaluate the potential improvement in performance.

5.2 Alternative Solution: Application-level Multiplexing

Application-level solutions avoid the use of multiple parallel TCP connections, and the resulting problems, by multiplexing several data streams onto a single TCP connection. Since TCP only provides a single, seamless byte-stream abstraction, these application-level solutions include framing schemes for demarcating the individual data streams. Examples of these include Persistent-connection HTTP (P-HTTP) [13], Session Control Protocol (SCP) [14] and the MUX protocol [8]. Significant reduction in the latency of Web accesses using P-HTTP are reported in [13].

Despite the performance benefits, application-level solutions have 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. Multiplexing over a single TCP connection introduces undesirable coupling between data transfers that are logically independent.

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

5.2.1 HTTP/1.1 with Persistent Connections

The HTTP/1.1 protocol [7], 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 helper applications). 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 solution works regardless of the application-level protocol.

```
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 9. New structures necessary for shared congestion windows and shared error recovery.

The third drawback mentioned in Section 5.2 also affects persistent connections in HTTP/1.1. As an example, consider the simultaneous transfer of several images over a persistent connection. Because TCP provides an *ordered* byte-stream abstraction, the loss of a data packet of one image can stall the delivery of data of the other images to the receiving application such as a Web browser. Clearly, this is undesirable.

5.3 TCP-INT Implementation

We now describe an 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 9) 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 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. We use round-robin scheduling across the connections though it is not an essential requirement.

New recv ack routine: When a new ack arrives, the sender increases the cwnd variable as appropriate. Also upon the arrival of a new ack, the sender removes any packets from the pkts[] list that have reached

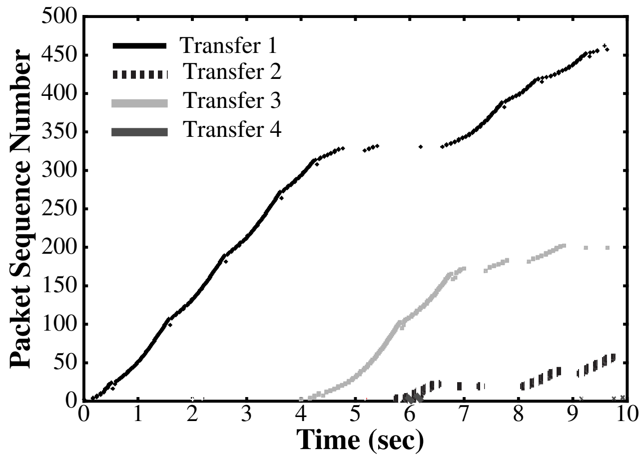

Figure 10. Four TCP-SACK transfers through a router with buffer size 3. Transfers start at 0, 2, 4 and 6 seconds.

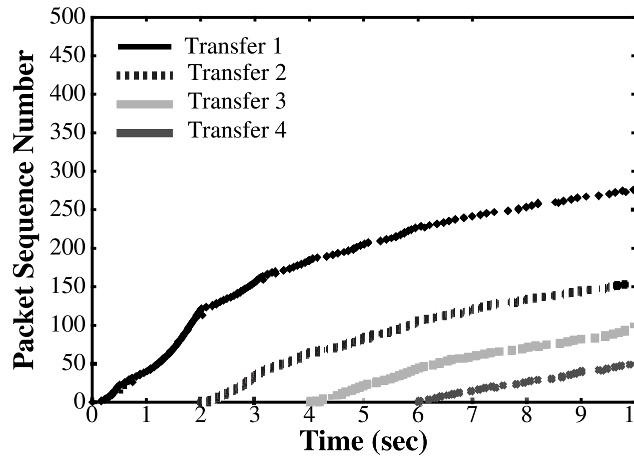

Figure 11. Four TCP-INT transfers through a router with buffer size 3. Transfers start at 0, 2, 4 and 6 seconds.

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 for sure 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 7. 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 staggered in time.

Figure 10 shows the sequence plot for the test using a SACK-based sender. 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). 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. Such unfairness and unpredictable performance (due to coarse timeouts) are undesirable from an application's viewpoint because connections carrying critical data could get slowed down while others carrying less important data do better.

Figure 11 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 per-

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.

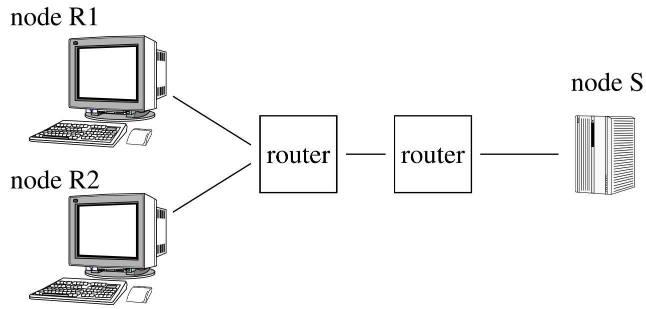

Figure 12. Topology for simulation tests.

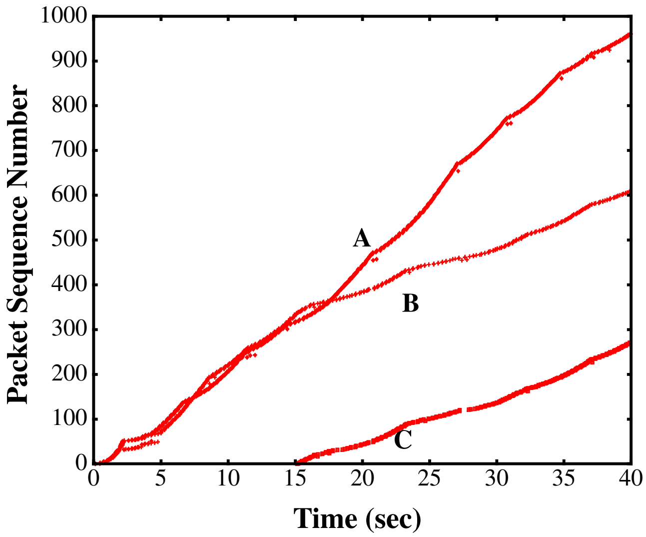

Figure 13. Three TCP-INT transfers from two hosts through a single bottleneck router. Connection A originates from the first host and starts at time 0. Connections B & C originate at the second host and start at time 0 and 15 sec respectively.

formance of the transfers is much more consistent and predictable. Also, new connections can build on the slow-start window growth already achieved by existing connections and cut down the timed needed for completion.

Next, we discuss our second test that involved competing connections from more than one host.

5.5 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 used the network topology shown in Figure 12. At time 0, a single TCP transfer starts from node S to each of nodes R1 and R2. Some time later (in this case, fifteen seconds), a second transfer starts between node S and R2. In addition, a significant amount of additional cross traffic was generated across the shared bottleneck link. This was intended to make the simulation more realistic.

In the case where node S uses standard TCP, congestion control is performed on a *per-connection* basis and so 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. Figure 13 shows the same test using TCP-INT. After the second connection from node R2 starts, each of the transfers from R2 receive approximately half the bandwidth of the transfer on node R1. This is because congestion

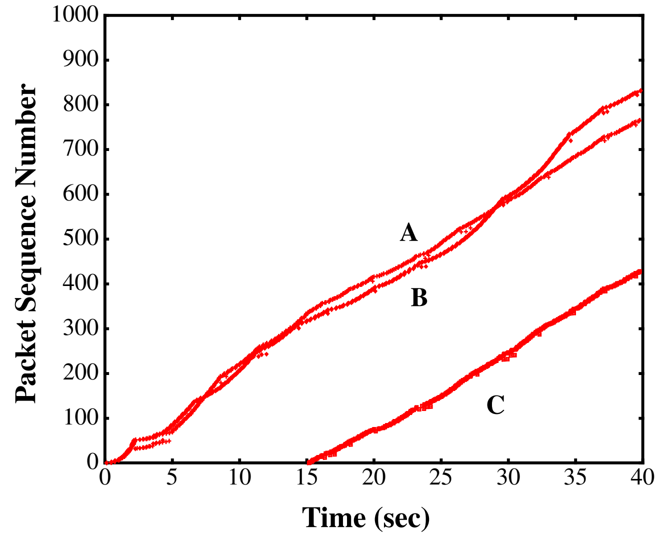

Figure 14. Three TCP-INT transfers using the congestion control algorithm for proxies from two hosts through a single bottleneck router. Connection A originates from the first host and starts at time 0. Connections B & C originate at the second host and start at time 0 and 15 sec respectively.

control is performed on a *per-host* basis. Therefore, each host receives an equal share of the bottleneck link bandwidth.

However, there are situations where an unequal distribution of bandwidth may be desirable. For example, a proxy host that launches connections of behalf of several end-clients should probably receive a larger share of the bandwidth than another individual client host that is communicating (directly) with the same server. We are able to achieve this using TCP-INT by modifying the window growth/shrinkage process, while still performing integrated loss recovery just as before. The basic idea is to manage the overall congestion window as though the set of TCP connections to the proxy host were operating independent of each other. We discuss this in more detail in [3].

Figure 14 shows the result for the same test as before with the congestion control policy on node R2 modified so that each of its connections (B and C) receives the same share of the bottleneck bandwidth as connection A of node R1. Connections B and C continue to benefit from integrated loss recovery.

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.* Almost 50% of all losses required a coarse timeout to recover. Fast retransmissions recovered from less than 45% of all losses. The remainder of losses were 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.
- *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 presented 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* 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.

We are currently implementing our enhanced loss recovery and integrated connection techniques and examining 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 [4].

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, V. N. Padmanabhan, S. Seshan, M. Stemm, and R.H. Katz. TCP Behavior of a Busy Web Server: Analysis and Improvements. Technical Report UCB/CSD-97-966, University of California at Berkeley, July 1997.

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

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

[6] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and Sack TCP. *Computer Communications Review*, July 1996.

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

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

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

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

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

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

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

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

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

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

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

[18] 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.