Chapter 4

Analysis of HTTP Performance

In this chapter, we present a performance analysis of the Hypertext Transfer Protocol (HTTP), the application-level protocol underlying the Web. We analyze HTTP/1.0, the predominant version of the protocol in use today[1]. Given our interest in identifying performance problems, we analyze the interaction between HTTP and the TCP's algorithms for connection establishment, congestion control, and loss recovery.

We start with a description in Section 4.1 of the two ways in which HTTP/1.0 uses TCP connections: sequentially or concurrently. In Section 4.2, we present a performance analysis based on a simple mathematical model of HTTP as well as live experiments over the Internet. In Section 4.3, we analyze TCP connection traces obtained from IBM's 1996 Olympics Web server. We conclude with a summary in Section 4.4.

4.1 HTTP/1.0 Basics

As discussed in Section 2.2, a Web page download involves the transfer of multiple components such as the HTML file and the inline images. HTTP/1.0 uses a separate TCP connection for each

1. The new version, HTTP/1.1, is derived in part from our P-HTTP work. We discuss these in Chapter 5.

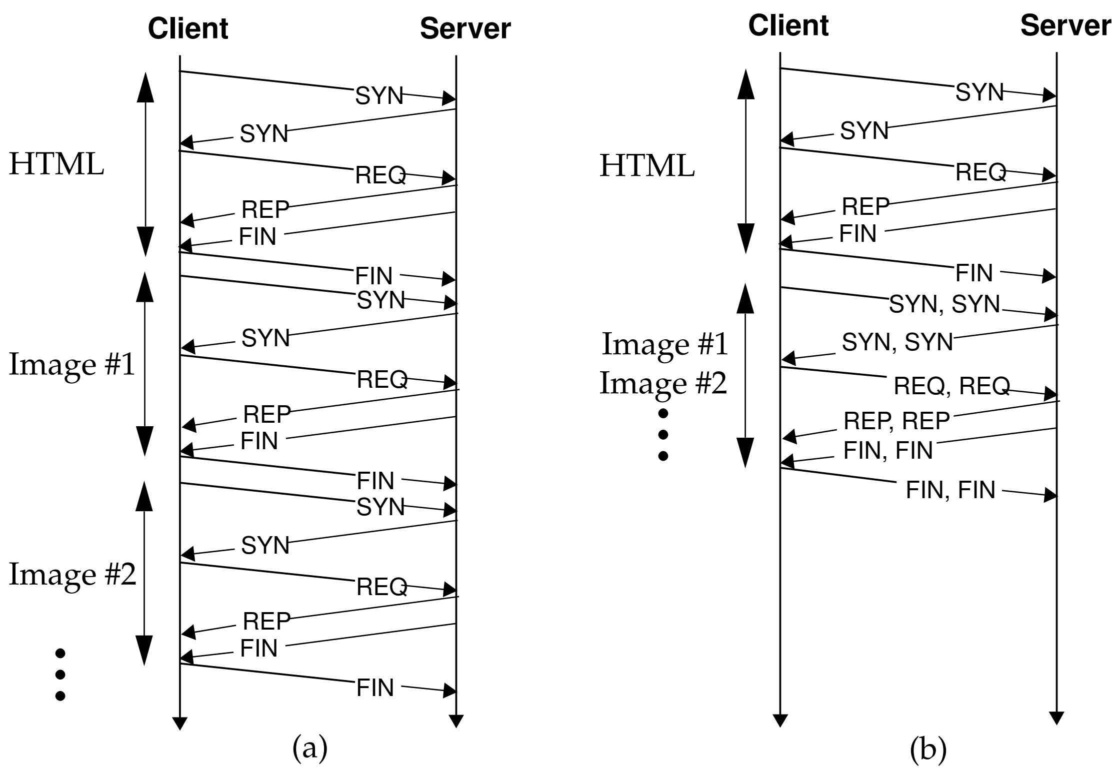

Figure 4.1 Timeline of a Web page download with (a) sequential connections, and (b) concurrent connections. Note that although we show the concurrent connections in (b) as simultaneous, in practice there would be a time gap between the packets of the individual connections.

component, in large part because TCP in itself does not provide a mechanism for demarcating multiple messages within a single TCP connection. The only mechanism that TCP provides for a sender to mark the end of a message is closing the corresponding TCP connection.

The individual components of a Web page can be retrieved either sequentially or concurrently. We discuss each possibility in turn.

4.1.1 Sequential Retrieval

The transmission of the components of a Web page from a server to a client is ordered sequentially. The client establishes a new connection and sends the request for a component only after it has received the previous component in its entirety. For a page with several inline images, the order in which the components are retrieved is usually the HTML file first, followed by image #1, then image #2, and so on. This is illustrated in Figure 4.1(a). Retrieving the HTML file and the inline images takes at least 2 RTTs each. As we will discuss in Section 4.2.1, it can take longer in practice, in part because of the overhead of TCP slow start.

4.1.2 Concurrent Retrieval

The transmission of the components of a Web page from a server to a client happens concurrently. For a page with several inline images, the client first retrieves the HTML file, identifies the inline images that the page contains, and then requests some or all of these images concurrently, without waiting for an intervening response from the server. This is illustrated in Figure 4.1(b). When the number of inline images is *nimages*, the number of concurrent connections could vary between 1 (which degenerates to sequential retrieval) and *nimages* (which corresponds to all the images being retrieved concurrently). Even with *nimages* concurrent connections, it is at least 2 RTTs before the client requests the inline images. Although concurrent connections may potentially reduce page download time by overlapping multiple transfers, there is the possibility of adverse interaction with TCP's congestion control and loss recovery algorithms. We will discuss this in Section 4.2.3 and Section 4.3.

4.2 Performance Analysis

We analyze HTTP performance based on a simple mathematical model as well as via live experiments over the Internet.

4.2.1 Mathematical Analysis

We present a simple mathematical analysis of the performance of HTTP with sequential retrieval and with concurrent retrieval. For the purposes of this analysis, we make the following simplifying (and optimistic) assumptions:

1. Both the transmission delay and the queuing delay for a packet are negligible compared to the propagation delay.

2. No packets are dropped in the network due to congestion.

3. The processing delay (due to the operating system, disk, display, etc.) at the server and client hosts is negligible compared to the propagation delay.

Assumption #1 allows us to isolate the effect of RTT from that of limited bandwidth. This is a reasonable assumption when the available bandwidth is large and the network load is low. For example, the transmission time for a 30 KB Web page download over a lightly-loaded wide-area T3 line (45 Mbps) is about 5 ms, which is much smaller than a typical wide-area round trip time (RTT) of the order of 100 ms. But clearly this assumption does not hold for slow links including dialup lines. We consider the effect of limited bandwidth in Section 4.2.3.

Assumption #2 is predicated on assumption #1. Since we assume that the transmission delay is negligible and that little queuing occurs, it is reasonable to assume that buffers never overflow.

Assumption #3 is reasonable given that even low-end PC-based servers can handle loads of a few thousand requests per second while keeping latency under 10 ms (as demonstrated by Web server benchmarks such as SPECWeb96 [99]). Only a small number of the busiest servers in the Web today experience a load that exceeds this level. And such busy Web servers often comprise a cluster of server nodes, so latency remains low even when the load exceeds the capacity of an individual node.

With these assumptions, the time to download a Web page is dominated by the number of network round trips incurred (with each round trip consuming time equal to one RTT). We assume that the Web page transfer involves the following steps: the client requests and downloads the HTML file, then parses the HTML file to determine the URLs of embedded objects such as inline images, and finally requests and downloads each individual image.

The download of a single file involves setting up a TCP connection, exchanging a request and a response, and finally tearing down the connection. Connection setup involves the SYN handshake, which contributes one RTT to the download time. A connection can be torn down asynchronously (i.e., without having either the sending or the receiving process wait for it to complete), so it does not add to the download time. This leaves the time taken by the request-response exchange.

We define the following variables for use in our analysis:

nbytes = size of a transfer

mss = the maximum segment size (MSS) of the TCP connection used in the transfer

nsegs = the number of segments in the transfer = ceiling of *nbytes*/*mss*

rtt = network round trip time

Since we assume that the transmission time and queuing delay are negligible, the time for the download is determined by the number of RTTs consumed as the TCP connection ramps up its window. Since we also assume that there are no packet losses due to congestion, the TCP connection remains in the slow start phase throughout. In this phase, the number of segments transferred in x round trips is 2^x-1. Therefore, the number of round trips needed to transfer *nsegs* segments is[2]:

$$\lceil \log(nsegs + 1) \rceil = \left\lceil \log\left(\left\lceil \frac{nbytes}{mss} \right\rceil + 1\right) \right\rceil$$

There is, in addition, the round trip time consumed for connection setup. Therefore, the time to complete a transfer of length *nbytes* is:

$$t(nbytes) = rtt \cdot \left(1 + \left\lceil \log\left(\left\lceil \frac{nbytes}{mss} \right\rceil + 1\right) \right\rceil\right)$$

We can now compute the download time, *T*, for a Web page comprising an HTML file of size *hbytes* and *nimages* inline images each of size *ibytes*.

With sequential retrieval, the HTML file and the inline images are transferred sequentially, so the download time is:

$$T_{seq} = t(hbytes) + n\mathrm{imag}es \cdot t(ibytes)$$

With concurrent retrieval, the HTML file is transferred first, then all the images are transferred concurrently. So the total time taken is:

$$T_{conc} = t(hbytes) + t(ibytes)$$

2. Note that the logarithm is taken to base 2.

Fundamentally, there are only two constraints a Web page download: first, the HTML file has to be transferred before the client can determine which inline images to request[3], and second, the transfer of a file requires at least one RTT to send the request and receive the response. Thus the minimum possible completion time for the Web page download in question, with sequential and concurrent retrieval, respectively, is:

$$T_{seq}^{min} = rtt \cdot (1 + nimages)$$

$$T_{conc}^{min} = rtt \cdot (1 + 1) = 2 \cdot rtt$$

Figure 4.2 illustrates the download time for the various cases graphically. The size of the HTML file is assumed to be 5 KB and the number of inline images to be 4. The download time is expressed as a multiple of the RTT and is computed using the formulae listed above. We make several important observations based on the figure:

1. Sequential retrieval of the inline images is far more time-consuming than concurrent retrieval.

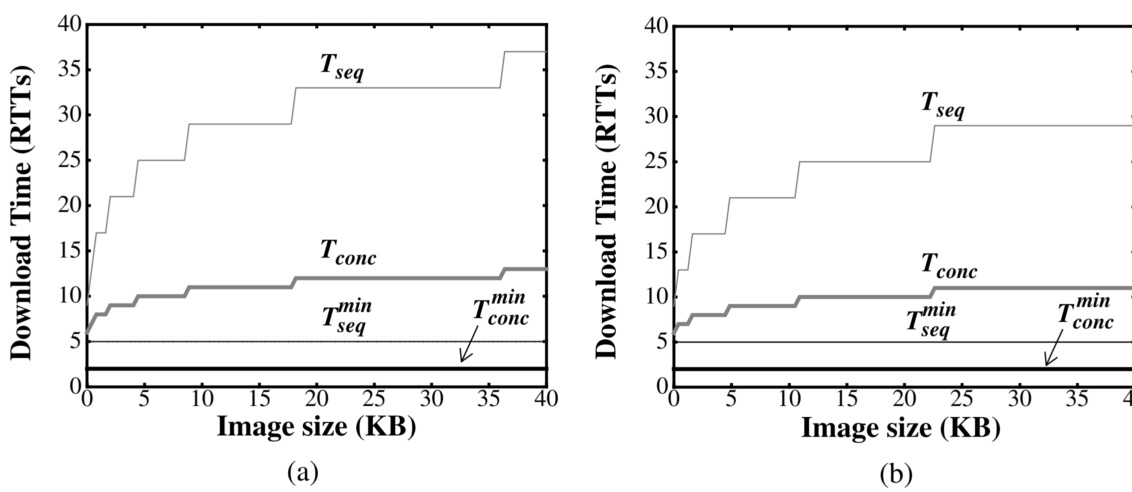

Figure 4.2 The download time for a Web page comprising a 5 KB HTML file and 4 inline images, as a function of the image size. The download time is expressed in multiples of the RTT. The MSS of the TCP connection is set either to (a) 576 bytes (a widely-used default setting for the MSS) or (b) 1500 bytes (the Ethernet MTU).

3. For now, we ignore speculative techniques such as prefetching.

2. Both with sequential and with concurrent retrieval, much of the difference between the actual download time and the minimum possible is due to the TCP slow start procedure which is invoked each time a new connection starts.

3. The smaller the MSS is, the larger the download time is, owing to the larger number of RTTs consumed during slow start.

4. When transfers are short in length (for instance, when the inline images are small in size), the RTT for connection setup could constitute a significant fraction of the total transfer time.

There is a fundamental lower bound on the RTT due to the finite speed of light. Therefore, as technological advances bring forth faster computers and higher-bandwidth networks, it will become increasingly more important to minimize the number of round trips consumed.

4.2.2 Impact of Connection Length on Throughput

The TCP slow start algorithm also has an indirect impact on the throughput that a connection can achieve. This is because the throughput is bounded by the ratio of the TCP window size to the round trip time. Therefore, when the window size is small, as it tends to be during slow start, the throughput is also low.

Figure 4.3 shows the throughput for wide-area transfers of various length between a host in Berkeley, CA, USA and one in Germantown, MD, USA. The round trip time was approximately 71 ms. Figure 4.3(a) shows a standard plot while Figure 4.3(b) uses a log-scale for the connection length. We observe that the throughput rises, sharply at first and then more slowly, as the connection gets longer. However, the throughput of even a moderate-length transfer falls far short of that of a long transfer. For instance, a transfer of length 100 KB (which is longer than most Web transfers) achieves less than one-third the throughput of a 4 MB long transfer. This significant underutilization of bandwidth causes a corresponding increase in latency.

4.2.3 Impact of Limited Bandwidth

We assumed in Section 4.2.1 is that bandwidth is plentiful. In practice, this may not be the case. There are several instances where bandwidth may be limited; examples include slow speed access

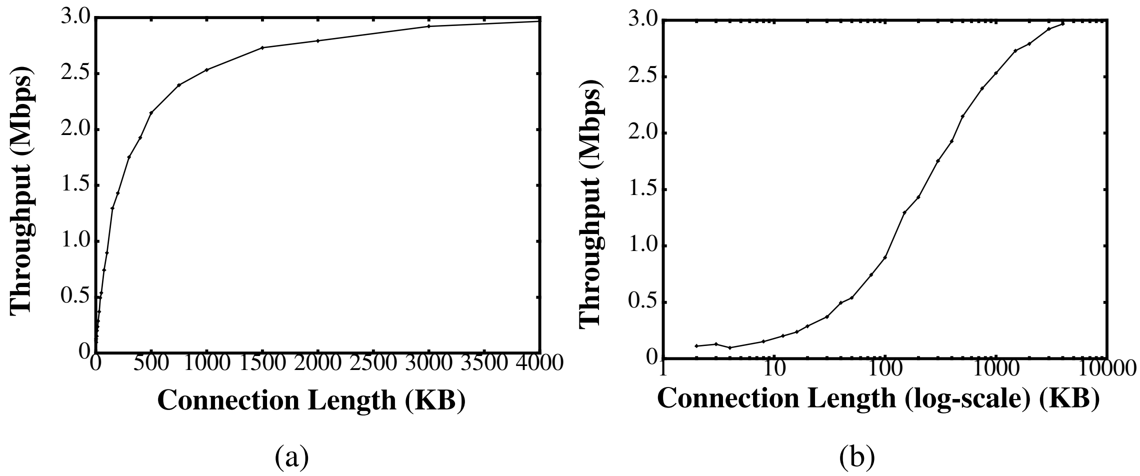

Figure 4.3 The throughput between a host in Berkeley and one in Maryland as a function of the connection length. The connection length is plotted (a) on a linear scale, and (b) on a log scale.

links such as dialup lines and heavily-loaded wide area links such as trans-continental links. The limited bandwidth could impact HTTP performance in several ways:

1. The transmission delay could become large. For instance, the transmission delay for a 1 KB packet over a 28.8 Kbps dialup line is about 280 ms.

2. The product of the bandwidth and the RTT represents the capacity of the data pipe. When the window size of a TCP connection, or the sum of the window sizes of a set of concurrent connections, exceeds the bandwidth-delay product, a queue forms at the bottleneck link. Such queuing has several adverse consequences:

   a. It increases the RTT. For example, a burst of four 1 KB segments over a 28.8 Kbps line could increase the RTT by more than a second. As a result, a TCP sender's retransmission timer might expire prematurely, causing performance degradation.

   b. The queue could grow so long as to cause a buffer overflow, forcing packets to be dropped. The delay incurred in recovering from these packet losses increases latency.

   c. While the inherent burstiness of slow start makes even a single connection susceptible to the adverse effects discussed above, the possibility is even greater with multiple concurrent connections. This is because each connection grows its window independently of the others, and does not back off until one or more of *its* packets are dropped.

Therefore, evaluating HTTP performance in general requires considering the effects of limited bandwidth in addition to the number of round trips consumed. However, it is difficult to model the effects of limited bandwidth analytically, especially due to the dominance of transient effects such as a packet loss burst during slow start. In light of this, we chose to study these effects via traffic trace analysis rather than pure mathematical analysis.

4.3 Analysis of Traces from the IBM Olympics Web Server

We analyzed traces from the official Web server for the 1996 Olympic Games. Because the server was based on HTTP/1.0, clients used a separate TCP connection to retrieve the individual components of each Web page. The components were retrieved either sequentially or concurrently depending on the client Web browser and its configuration. The Web server implemented the Reno variant of TCP.

The pre-processed traces contained a complete record of TCP events at the server, including packet transmission and reception, packet retransmission and timeouts. We analyzed the traces to answer to several questions pertaining to Web data transport performance:

1. How large is the effective window size of TCP connections used for Web transfers? How often is the effective window size constrained by the receiver-advertised window? A frequent occurrence of the latter would indicate underutilization of network bandwidth.

2. How effective are the TCP loss recovery mechanisms in the context of Web transfers? What are the potential benefits of more sophisticated algorithms such as selective acknowledgements?

3. How does the performance seen by a client correlate with the number of concurrent connections that it establishes to a server? A positive correlation would indicate that clients have an incentive to use a large number of concurrent connections. But as the answer to the next question indicates, such use of connections could have a detrimental impact on the network as a whole.

4. How does the establishment of multiple concurrent connections between a client and a server impact congestion control?

On account of the extremely large size of the traces, we base the analysis presented here only on a subset gathered during a busy 3-hour period.

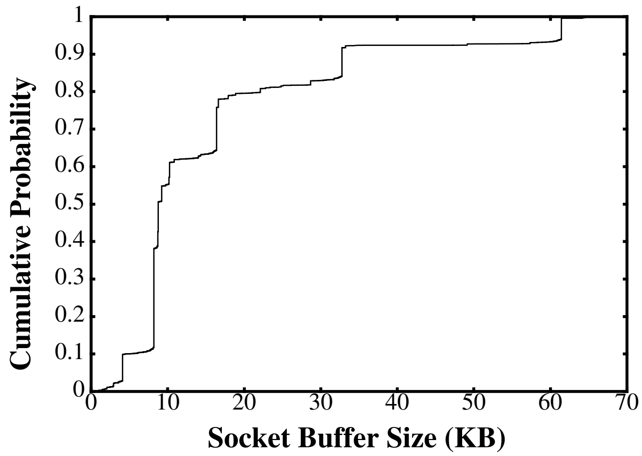

Figure 4.4 Cumulative distribution function (CDF) of the receiver advertised window size.

4.3.1 TCP Window Size Distribution

The TCP window size determines the amount of unacknowledged data that the sender can have outstanding in the network. It has a direct bearing on the throughput achieved by a connection, and consequently, the latency for completing a transfer of a certain size. In addition to the congestion control window, TCP also employs a flow control window, which indicates to the sender the amount of free buffer space that the receiver has. Since the size of this window is carried in acks transmitted by the receiver, it is often called the *receiver-advertised window*. The sender is constrained by both windows, so its *effective* window size is the smaller of its congestion window size and the receiver-advertised (flow control) window size.

In an actual implementation, the buffer space available to the receiver, and consequently the receiver-advertised window, is determined by the size of the buffer (called the *socket buffer*) allocated by the receiving application (such as a Web browser) at the time of connection establishment. Figure 4.4 shows the cumulative distribution function (CDF) of the receiver-advertised window size for the connections in the trace. There are marked upswings at commonly used socket buffer sizes such as 4 KB, 8 KB, 16 KB, etc. On the whole, the receiver-advertised window tends to be rather small. For instance, it is smaller than 8 KB in over 60% of the cases.

We also investigated how the receiver-advertised window compares with the congestion window size (as computed by the TCP emulation engine). Analysis of the traces reveals that in approximately 14% of all connections, the latter grew to be larger than the former, i.e., the receiver-advertised window was the limiting factor in performance. Of course, this also means that *in the vast majority of connections, the congestion window remains smaller than the receiver-advertised window, which in itself tends to be small* (as evident from Figure 4.4).

The reason that the receiver-advertised window tends to be small is that most operating systems tend to use a small receiver buffer by default, and applications such as Web browsers often do not explicitly request a larger buffer. On the other hand, the congestion window tends to be small because the individual connections tend to be short in length; the mean connection length is 5.5 KB and the median 3.1 KB [7]. As a result, slow start fails to build up a large window.

Trace Statistic	Value	%
Total connections	1650103	
Total packets	7821638	
During slow-start	6662050	85
During congestion avoidance	1159588	15
Total retransmissions	857142	
Fast retransmissions	375306	44
Slow-start retransmissions	59811	7
Coarse timeouts retransmissions	422025	49
Avoidable with SACKs	18713	4
At least one duplicate ack received	104287	25

Table 4.1 Summary of Analysis Results (percentages are relative to the corresponding category)

The tendency connections to be short in length also implies that slow start dominates the congestion avoidance phase. As indicated in Table 4.1, an overwhelming majority (85%) of the packets were transmitted by the server during the slow start phase of the corresponding connection. Only 15% percent were sent during the congestion avoidance phase. The dominance of slow start means that connections spend the majority of their time probing for bandwidth rather than effectively utilizing their share of the network bandwidth.

In addition to reducing throughput and increasing latency, a small window size worsens loss recovery, as we discuss next.

4.3.2 Effectiveness of TCP Loss Recovery

When a TCP sender detects that a packet loss, it attempts to recover from the loss by retransmitting the corresponding packet. A packet retransmission fall into one of three categories:

1. Fast retransmission
2. Timeout-driven retransmission
3. Slow start retransmission

Fast retransmission and timeout-driven retransmission are triggered by the receipt of a threshold number of duplicate acks and the expiration of the retransmission timer, respectively. A slow start retransmission could happen during slow start following a retransmission timeout. In some cases, it may be an unnecessary retransmission. The original transmission of the packet may have reached the receiver, but the sender does not know this because the cumulative ack still indicates a smaller sequence number because of the loss of an earlier packet.

Table 4.1 summarizes results from the analysis of the trace. Approximately 49% of packet retransmissions resulted from timeouts and a further 7% from slow start following timeouts. Only 44% of the retransmissions were triggered by the fast retransmission algorithm. *Thus timer-driven loss recovery, which entails substantial idle time waiting for the timer to expire and consequently increases latency, is the dominant mode of loss recovery.*

There are two main reasons why a TCP Reno sender could suffer a retransmission timeout:

1. *When multiple packets are lost within a window*, it is highly likely that the sender is forced to time out in order to recover from losses other than the first one [28]. This is especially so when the effective window size is small.
2. *When an insufficient number of duplicate acks are received*, either because the are not a sufficient number of outstanding packets or because most of the packets in a window are lost, the sender may be forced to time out even to recovery from the first loss. As before, a timeout is more likely to happen when the effective window size is small.

One proposed technique to improve the performance of TCP loss recovery is *selective acknowledgements* (*SACK*) [67]. There are two ways in which SACK could potentially help. First, it could prevent the unnecessary retransmission of packets that have reached the receiver but have not yet been acknowledged by a cumulative ack because of an earlier packet loss. Thus, slow start retransmissions (7% of all retransmissions) could potentially be avoided.

Second, SACK allows for an alternative interpretation of the 3-duplicate ack threshold that triggers fast retransmission in cases where a timeout would otherwise be necessary. Basically, the sender could use SACK information to determine if a later packet, that is at least 3 packets beyond the one that is suspected to be lost, has reached the receiver. If so, the sender could immediately invoke fast retransmission, while making the same assumption as standard TCP about the likelihood and extent of packet reordering. The important point to note is that the sender need not wait for 3 duplicate acks. *In spite of this potential benefit, our analysis of the traces shows that SACK would have avoided only about 4% of the retransmission timeouts that happened.* This is because the effective window size is often too small for SACK to provide any information about packets much beyond the one suspected to be lost. Therefore, the sender is constrained to suffer a timeout.

In summary, retransmission timeouts are the dominant form of loss recovery for TCP connections in the Web, due mainly to the small window size of individual connections. The overhead of timeouts increases latency and decreases throughput. SACK does little to rectify the situation.

4.3.3 Effectiveness of TCP Congestion Control

Having discussed in some detail the impact that short connection lengths, and hence small window sizes, have on performance, we turn to the impact that multiple concurrent connections between a Web server and a client have. The point here is that TCP congestion control operates at the granularity of individual connections. So each of the concurrent connections invokes congestion control independently of the others. We discuss how this impacts the aggregate throughput and the aggregate congestion response of the concurrent connections.

4.3.3.1 Throughput Analysis

We analyzed the traces to determine if there is any significant correlation between the throughput seen by a client host and the number of concurrent connections (n) it has open to the server. (Each connection could, for instance, be used to transfer a logically-separate component of a Web page.)

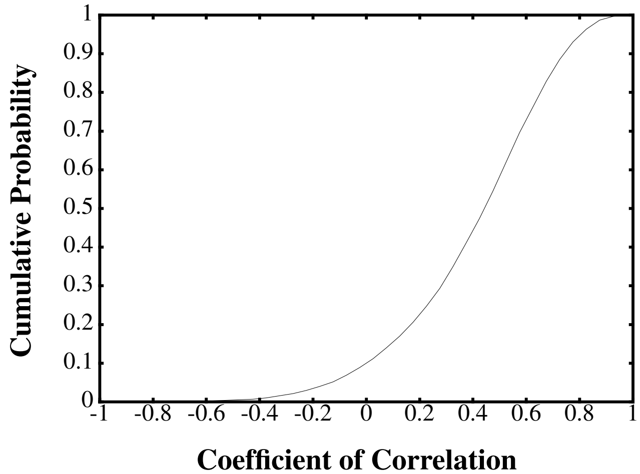

Figure 4.5 CDF of the coefficient of correlation between throughput and the number of simultaneous connections. The CDF is obtained by aggregating across all hosts.

A positive correlation would indicate that applications could appropriate an unfair share of the bottleneck link bandwidth by opening several concurrent connections to another host.

For each host, we divided the entire duration of its interactions with the server into periods during which the number of concurrent connections, 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 begins (n increases by 1). For each such period, we computed the throughput as the ratio of the total number of useful bytes transferred during the period by all the connections to the duration of the period. We computed the throughput for a period only when a significant amount of data (at least 5 KB) was transferred during that period. Finally, we computed the correlation coefficient between the throughput and the number of concurrent connections, n. We performed this computation only for hosts which had at least 10 pairs of (*throughput, n*) samples.

By aggregating the correlation coefficient for different hosts, we obtained the cumulative distribution function (CDF), which is shown in Figure 4.5. It is clear from the figure that there is a positive correlation for about 90% of the hosts. The correlation coefficient is larger than 0.5 for about 45% of the hosts. The substantial positive correlation indicates that clients are better off performance-wise if they open a larger number of concurrent connections. But as we discuss next, there is a cost in terms of increased congestion which can worsen the performance of the network as a whole.

4.3.3.2 Congestion Avoidance Analysis

There are several reasons why concurrent connections may improve performance for a client. These include:

1. The connection setup phase of concurrent connections could overlap in time, thereby cutting down latency by several RTTs.
2. Each of the concurrent connections has an initial window size of 1 segment. So *n* of them would have an aggregate initial window size of *n* segments. So during slow start, the aggregate congestion window size of the concurrent connections starts off at a higher point (*n* segments) on the exponential growth curve than the window for a single connection would (1 segment). This can potentially decrease the latency that the client experiences by *log(n)* RTTs.
3. During the congestion avoidance (linear) phase, each connection increases its window size by 1 segment per RTT. Therefore, *n* connections that are in the linear phase concurrently will increase the aggregate window by *n* segments per RTT. The faster window growth, again, has the potential of cutting down the latency experienced by the client.

Clearly, it is desirable to cut down or eliminate the connection establishment latency (the first point mentioned above). Opening concurrent connections is one way of doing so. The TCP accelerated open (TAO) technique employed by T/TCP (Section 2.3.4) is an alternative way. However, both a larger initial window size (the second point) and a faster window growth (the third point) are undesirable for two reasons.

First, these steps undermine TCP congestion avoidance. The aggregate initial window size is a function of the number of concurrent connections, *n*, opened by the client. The choice of *n* is made by the application without necessarily taking into account the network conditions, such as the bottleneck link bandwidth or the network load. While a large initial window size may be appropriate when the bandwidth-delay product is large (as in a geostationary satellite-based network), it could overwhelm a slower link, such as a dialup line, and cause heavy packet loss. Similarly, speeding up the pace of window growth may be appropriate for fast networks but not for slow ones.

Second, these steps contribute to unfairness. Consider two clients, C_s and C_c, that are trying to download a Web page from a server. C_s uses a single TCP connection at a time for the purpose while C_c opens multiple connections concurrently. Furthermore, assume that the two clients share the same bottleneck link. Ideally, the two clients should share capacity of the bottleneck link equally. But because of its more aggressive behavior, C_c would unfairly obtain better performance. Worse still, a client could decide to open multiple concurrent connections with the explicit purpose of exploiting this unfairness. For instance, the client could have the server split up a logical Web page component[4], such as an inline image, into multiple pieces and open a separate connection to retrieve each piece, just to improve its own performance.

4.3.3.3 Congestion Control Analysis

In addition to the three points listed in the previous section, a client that opens multiple concurrent connections also benefits because the aggregate congestion control response of the connections is more aggressive than that of a single connection. When a TCP sender detects a packet loss (an implicit signal of congestion), it slows down, either by halving its congestion window size (with data-driven loss recovery) or by resetting it to one segment (with timer-driven loss recovery). These actions are taken by individual connections based on losses suffered by their own packets. So in a set of concurrent connections, only those that experience a packet loss slow down. This is in spite of the loss on even one connection being indicative of congestion along the (shared) path of all of the connections.

To evaluate the aggregate congestion control response of concurrent connections, we analyzed the traces to determine how packet losses tend to be distributed across the set of concurrent connections between the server and each individual client. Consider a period of time when a client host has n concurrent connections open to the server. Suppose that one of the n connections experiences a packet loss. This event marks the beginning of a *loss epoch*. We recorded the amount of outstanding data on each connection at such points in time. The time when this outstanding data on all of the connections has been acknowledged marks the end of the loss epoch. For each loss epoch, we recorded the number of concurrent connections, the outstanding window size of each connection and the distribution of packet loss events across the connections. We aggregated this data over all client hosts.

---

4. The *byte-range* option in HTTP [30] allows a client to request a specific range of bytes from a file.

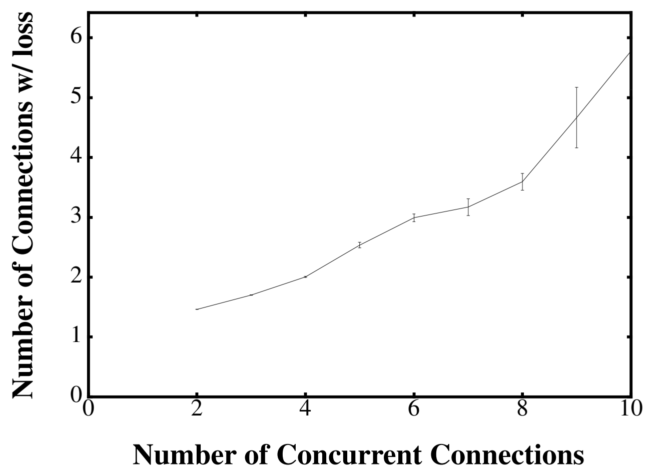

Figure 4.6 Average number of connections that experience a loss within the same window when using concurrent connections.

Of the set (of size *n*) of connections that are active during a loss epoch, only a subset (of size $m \leq n$) actually experiences a loss during that loss epoch. Only the connections in this subset cut down their individual congestion windows during the loss epoch. 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 backoff in a host's total transmission window is $(1 - m/2n)$. Figure 4.6 shows that on average half the total number of concurrent connections experience a loss during a loss epoch, i.e., *m* is approximately $n/2$. So the effective multiplicative backoff factor is $3/4$, which represents a much more aggressive behavior than halving the window.

We complemented this back-of-the-envelope calculation with an analysis to obtain bounds on the effective multiplicative backoff factor. The analysis discussed previously told us the congestion window sizes of each connection during a loss epoch and which connection(s) experienced packet loss during that epoch. To obtain bounds on the effective multiplicative backoff factor, we made 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 assumed that each such connection does a fast retransmission and halves its window. To obtain a lower bound, we assumed that each undergoes a retransmission timeout and resets its window to one segment. For each of the two cases, the ratio of the combined congestion window after before the loss event to that before gave us a bound on

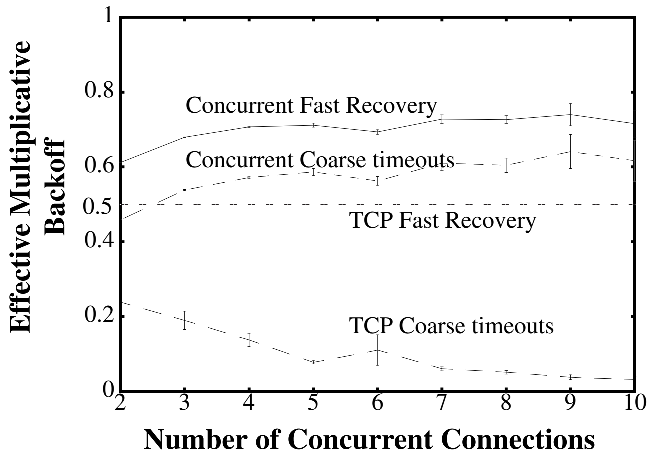

Figure 4.7 Effect of concurrent connections on the extent of backoff in response to congestion-induced packet loss.

the effective multiplicative decrease factor. These bounds are plotted as a function of the number of concurrent connections in Figure 4.7 (the curves labelled "Concurrent Fast Recovery" and "Concurrent Coarse Timeouts"). As a baseline comparison, we also compute the multiplicative backoff in congestion window that would result if the connections were treated as a single unit and (a) a fast retransmission occurred, or (b) a coarse timeout occurred. The corresponding curves are labelled as "TCP Fast Recovery" and "TCP Coarse Timeouts", respectively, in the figure. With concurrent connections, the effective multiplicative backoff factor is in the range 0.6 to 0.75, significantly more aggressive than a single TCP connection which would backs off twofold or more (i.e., a factor of 0.5 or less).

Therefore, not only do multiple concurrent connections cause aggressive growth of the congestion window compared to a single TCP connection, they also cause reduced responsiveness to indicators of congestion such as packet loss[5]. The former makes it more likely that the network gets into a congested state. The latter confounds recovery from congestion. Together, these two effects could degrade overall network performance. It is difficult to quantify the degradation using the Web server traces, so we demonstrate this via simulation experiments in Chapter 6.

5. The situation is similar with other indicators of congestion such as ECN [31] that also operate at the granularity of individual connections.

4.4 Summary

In this chapter, we have analyzed the performance impact of composing HTTP/1.0 and TCP, using both mathematical analysis and traffic trace analysis. HTTP/1.0 maps each component of a Web page onto a separate TCP connection. Such a mapping is convenient because TCP in itself does not provide any means for demarcating multiple messages within a single connection. But it causes several performance problems, both with sequential retrieval of Web page components and with concurrent retrieval.

Sequential retrieval simplifies both the server and the client processes because they contend with only one active connection at a time. Moreover, the single connection in progress reacts to congestion by backing off twofold or more. On the downside, though, delay is incurred in establishing each connection, issuing the HTTP request and waiting for the response for each component, and slow start. The short length of each connection causes the congestion window to be small. So bandwidth utilization is poor and loss recovery often depends on retransmission timeouts, which further increase latency.

Concurrent retrieval reduces the delays due to connection establishment the HTTP request-response exchange because these happen concurrently over multiple connections. A fundamental problem though is that the concurrent connections operate independently of each other. In particular, the connections do not share indications of congestion along their shared path. Consequently, the set of concurrent connections as an aggregate is more aggressive than a single connection in terms of congestion avoidance and less responsive in terms of congestion control. By opening multiple concurrent connections, an application can obtain a disproportionate share of network bandwidth, and also exacerbate network congestion, to the detriment of other connections in the network. Furthermore, since each connection is still short in length, retransmission timeouts would remain the predominant form of loss recovery.

To tackle these problems, we proposed a new connection abstraction for HTTP, which persists across multiple HTTP transactions. Our proposal, called *persistent-connection HTTP* (*P-HTTP*), decreases Web download latency without resorting to aggressive use of network resources. We discuss P-HTTP in the next chapter.