Chapter 6

TCP Session

In Chapter 4 and Chapter 5, we analyzed the performance of HTTP/1.0 and P-HTTP and identified their strengths and limitations. In this chapter, we present our new solution, which we call *TCP session*. We first provide the motivation for developing TCP session and then describe its design in detail. Next, we discuss implementation details, including the our implementation in the BSD/OS TCP/IP stack. We then present detailed performance results, both from simulation and from experiments using our implementation.Finally, we summarize the benefits and the limitations of TCP session. The latter motivates our work on TCP fast start presented in Chapter 8.

6.1 Motivation

In previous chapters we have discussed the problem of Web data transport in the context of three different schemes: HTTP/1.0 with sequential connections, HTTP/1.0 with concurrent connections, and P-HTTP. Table 6.1 summarizes the key advantages and disadvantages of each approach.

We note that each of these schemes has its share of pros and cons. Closer examination reveals that HTTP/1.0 with concurrent connections and P-HTTP complement each other's strengths and weaknesses. Our goal, therefore, is to design a solution that combines the strengths of both but is not

	Pros	Cons
HTTP/1.0 with sequential connections	• Easy to write application using standard sockets API	• Sequential ordering imposed on logical data streams • Overhead of repeated slow start • Poor loss recovery
HTTP/1.0 with concurrent connections	• No coupling between logical data streams • Easy to write application using standard sockets API	• Less responsive to congestion • Overhead of repeated slow start • Poor loss recovery • Promotes unfairness
P-HTTP	• Good congestion control and loss recovery properties • Overhead of slow start amortized over multiple transfers	• Couples together logical data streams causing problems such as head-of-line blocking • Requires multiplexing support in application/socket library

Table 6.1 Summary of the pros and cons of various approaches to Web data transport.

encumbered by the weaknesses of either. Specifically, we desire a solution with the following properties:

- Applications should be free to use as many logical data streams as they wish to communicate between hosts. These data streams should not be temporally ordered.
- The use of multiple data streams should not have an adverse effect on the performance of either the application in question or the network as a whole. In particular, it should not impact the efficacy of congestion control and/or loss recovery mechanisms.
- The solution should conform to the principle of wide applicability discussed in Section 1.8. By implication, it should not be tied to a specific application or application-level protocol (such as HTTP). Also, it should require little or no support from applications, and should have little or

no dependence on their structure in terms of their organization into processes (for instance, the organization of a Web client into a browser process and helper processes for individual MIME types).

- In accordance with the principle of incremental deployment discussed in Section 1.8, it should be possible to deploy the solution without any major changes to the network. A partial deployment should be robust and should also yield some of the performance benefits of a complete deployment.

The solution we develop enables applications to map individual data streams onto separate TCP connections. The aggregate of data streams is tied into what we call a *TCP session*[1]. This solution satisfies most of the requirements listed above. In the sections that follows, we quantify its performance benefits and also point out its limitations. The also discuss how the latter is addressed in later chapters.

6.2 TCP Session Overview

We observe that many of the difficulties in the context of Web data transport arise due to the coupling of two distinct elements of TCP functionality into the entity that is a TCP connection:

1. Reliable, ordered byte-stream service with flow control.
2. Congestion control and loss recovery.

Only the former is visible at the application level. An application knows that each byte of data it sends over a TCP connection will be delivered reliably to the receiver without any reordering. A receiving application can effect flow control by controlling how fast it reads in data from a TCP connection. The latter part of TCP functionality happens under the covers, hidden from the application. However, it is critical both for supporting the reliable, ordered byte-stream service abstraction (because loss recovery is needed for reliable data delivery) and for good performance.

Mapping several logical data stream onto a single TCP connection (as in P-HTTP) causes ordering to be imposed on data that belongs to different data streams. As a result, flow control actions on

1. For ease of exposition, we use the term “TCP session” both to refer specifically to such an aggregate of connections and also to refer to the technique as a whole.

one data stream also impact the others, which is clearly undesirable. Mapping the data streams onto separate TCP connections avoids this problem, but introduces a new one — congestion control and loss recovery happen independently for each connection, which decreases their efficacy (as discussed in Section 4.3.2).

Our solution to these problems is to separate TCP functionality into two parts:

1. Reliable, ordered byte-stream service with flow control is provided on a per-TCP connection basis.

2. Congestion control and loss recovery are done in an integrated fashion across the set of TCP connections between a pair of hosts (such as a Web server and a client). We refer to this set of connections as a *TCP session*.

The rationale for integrating congestion control and loss recovery across the set of connections in a TCP session is that all of the connections are likely to traverse the same path through the network and experience similar levels of congestion. The choice of how many logical data streams (and hence TCP connections[2]) to use is purely an application-level decision, so it should not have any impact on the functioning of congestion control and loss recovery.

The TCP session abstraction enables applications to use as many TCP connections as they wish without impacting performance, either adversely or favorably[3]. So, for instance, an application may choose to break up an image file into well-defined sub-component that can be rendered independently of each other. By transferring and rendering the sub-components concurrently, the application can improve the performance perceived by a human user, although the performance of the underlying data transport may remain unchanged.

By default, the TCP session abstraction is transparent to applications. Just as before, applications open and use TCP connections using the standard sockets API. There is no need for applications to be modified or even re-linked. In addition, however, TCP session gives applications the opportunity to control the scheduling of connections within a TCP session. In this chapter, we confine our-

2. From the viewpoint of an application, a TCP connection is synonymous with a logical data stream.

3. A positive correlation between the number of TCP connections used and the performance achieved (as is the case with standard TCP in certain situations) could encourage applications to be greedy and launch several connections, to the detriment of the network as a whole.

selves to the default operation of TCP session, and defer discussion of application-controlled scheduling to Chapter 7.

The aggregation of connections into a TCP session can happen at several different levels of granularity, such as individual applications, all applications launched by an individual user, etc. However, since, in this chapter, we are focussing on the transparent operation of TCP sessions, we only discuss integration at the granularity of host-pairs (such as a client-server pairs). Many of the ideas discussed in this context carry over to other levels of granularity of integration, which we motivate and discuss in Chapter 7.

Finally, the operation of TCP session is confined to the sending side of data. So each of the communicating peers can independently choose whether or not to implement it. In practice, it would probably be more important to incorporate the algorithm in Web server hosts than in Web client hosts because the former is primarily a sender and the latter primarily a receiver of data.

6.3 Design and Operation of TCP Session

We now discuss the design and operation of TCP session in detail. We discuss four components of TCP session in turn:

1. Session Control Block
2. Integrated Congestion Control
3. Connection Scheduling
4. Integrated Loss Recovery

6.3.1 Session Control Block

The separation of TCP functionality that TCP session entails a corresponding separation of TCP state into two parts — that needed to support the reliable, ordered byte-stream abstraction and flow control, and that needed for congestion control and loss recovery. The latter is encapsulated into what we term as a *session control block* (SCB) maintained in the sender's protocol stack.

The SCB contains state variables that are shared among the connections in a TCP session. These variables, which are needed for congestion control and loss recovery, are no longer part of the TCP control block (TCB) of each individual connection. Only variables needed for the reliable, ordered byte-stream service and flow control provided by each connection are retained in the corresponding TCB.

The contents of the per-session SCB include:

1. Session congestion window (*session_cwnd*).
2. Session slow start threshold (*session_ssthesh*).
3. Amount of outstanding (i.e., unacknowledged) data across all the connections in the session (*session_ownd*).
4. Smoothed estimates for the round trip time and its variance (*srtt*, *rttvar*).
5. List of connections within the session (*connlist*)
6. List of unacknowledged segments sent on connections in the session, sorted by their time of transmission (*seglist*).

The contents of the per-connection TCB include:

1. Highest sequence number of data transmitted (*snd_max*).
2. Sequence number of the first unacknowledged byte of data (*snd_una*).
3. Receiver advertised flow control window (*rcv_adv*).

Note that we have only listed the subset state variables in the SCB and the TCB that are relevant to the discussion that follows.

A TCP session and the corresponding SCB come into existence automatically when the first connection between a pair of hosts is opened. These remain in existence so long as there is at least one connection open between the pair of hosts. When there are no open connections left, the TCP session is terminated and the SCB is de-allocated. However, in certain situations it may be desirable to retain the SCB even when there are no active connections. For instance, the SCB could hold persistent state for use by TCP fast start, which we present in Chapter 8.

Next, we discuss integrated congestion control.

6.3.2 Integrated Congestion Control

The key observation that motivates integrated congestion control is that it is *how much* data rather than *what* data that matters as far as the load and the state of congestion in the network are concerned. In other words, it is irrelevant from the viewpoint of congestion control whether the data being transferred between a pair of hosts is spread across multiple TCP connections or concentrated in a single connection. It only matters how much data is outstanding in the network. Accordingly, congestion control should not be impacted in any way by the number, duration, or other characteristics of concurrent connections launched by applications.

For this purpose, a sender implementing TCP session maintains a unified congestion window, *session_cwnd*, for the set of connections between a pair of hosts. This tracks the available capacity along the network path between the hosts. The sender also maintains a count of the total amount of outstanding data, *session_ownd*, across all the connections. The session congestion window places a limit on the total amount of outstanding data that the set of connections in the session can have, so the sender is entitled to send data so long as *session_ownd* is less than *session_cwnd*. Since there may be multiple connections within the TCP session, there is the question of which connection(s) should be picked when there is the opportunity to send data. We discuss this scheduling issue briefly in Section 6.3.3, and in more detail in Chapter 7.

The dynamics of *session_cwnd* are much like that of a *single* connection using standard TCP. At the time the first connection in a session is opened, *session_cwnd* is initialized to one segment. Since there is no outstanding data at this point, *session_ownd* is set to zero. As new data is sent out on any of the connections in the session, *session_ownd* is incremented. As acks indicating the successful delivery of this data are received, *session_ownd* is decremented and *session_cwnd* is grown. Initially, the sender grows *session_cwnd* at an exponential rate, in accordance with the slow start regime. When *session_cwnd* crosses the slow start threshold, *session_ssthresh*, the session switches to linear growth at the rate of one segment per RTT.

So long as there is at least one connection open in the session, the opening and closing of additional connections has no impact on the dynamics of the congestion window. This is in keeping with our goal to decouple the opening of a new connection (which is an application-level decision) from congestion control (which impacts the performance of the network as a whole). In particular, *session_cwnd* is not incremented just because a new connection has been opened. Instead, the new connection shares the existing session congestion window with other connections in the session, as we discuss in Section 6.3.3.

As in standard TCP, the loss of a segment is treated as a sign of congestion. In response, the sender backs off, except that, unlike in TCP, the entire session is backed off, not just the connection that experienced the loss. This is motivated by the observation that the loss of a segment of a connection is caused by the overall load imposed on the congested link by all the connections in the session (and also unrelated connections originating at other hosts). The "fate sharing" imposed by tail-drop gateways, that are predominant in the Internet, implies that there may be little correlation between the load that a particular connection imposes on the network and the amount of packet loss it experiences. So when it detects an implicit congestion signal in the form of a packet loss, the most that a sender can do to alleviate the congestion is to cut back the load offered by all the connections in the corresponding TCP session (which presumably all share the congested link).

Ideally, this cut back should be applied to all connections that contribute to congestion on that link, but practical difficulties make it difficult to do so:

- A sender that experiences a packet loss on a particular connection does not know which other connections of its to other hosts also share the same congested link.
- The sender does not know which other hosts have connections traversing the same congested link. Even if this knowledge were available, it would probably be too expensive (and perhaps also counter-productive because of the increase in traffic) to explicitly deliver this congestion signal to each individual host.

Therefore, our response to packet loss is confined to the corresponding session. The sender halves the congestion window of the corresponding session, provided the loss is detected via data-driven means. The session slow start threshold, *session_ssthresh*, is then set to this new, halved value of *session_cwnd*. On the other hand, if the sender is forced to time out, it sets *session_ssthresh* to half of *session_cwnd*, and resets *session_cwnd* back to one segment. These steps ensure that unlike in the case of independent TCP connections (Section 4.3.3), the congestion behavior of a TCP session is not impacted by the number of connections that applications choose to open.

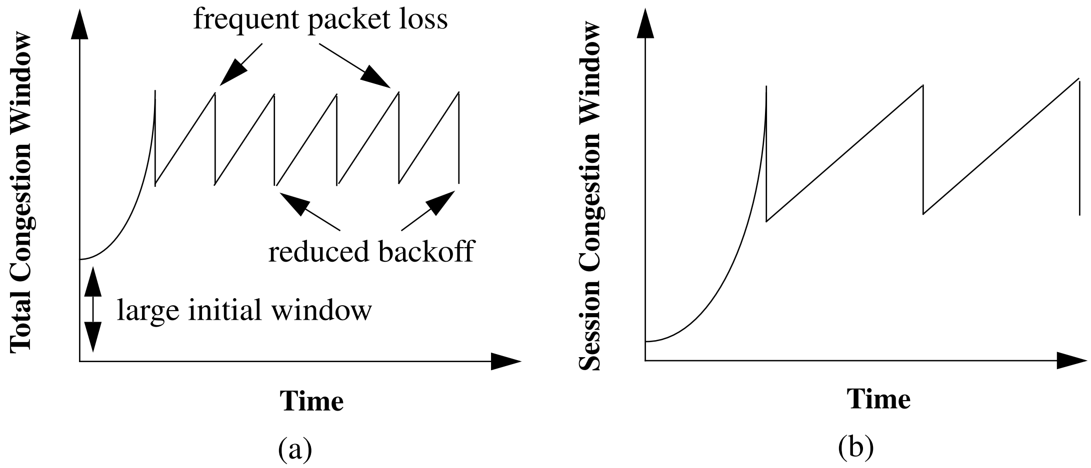


Figure 6.1 A schematic depiction of the congestion window dynamics when several connections are launched simultaneously between a pair of hosts. (a) corresponds to the case of independent TCP connections, and (b) to TCP session. The total congestion window in (a) is obtained by adding up the windows of the individual connections.

Figure 6.1 contrasts the congestion window dynamics of TCP session (Figure 6.1(b)) with that of independent TCP connections (Figure 6.1(a)). Note that these are schematics rather than actual traces. In the latter case, the total congestion window size is computed by adding up the window sizes of the individual connections. We note several significant differences that highlight the benefits of TCP session:

1. When *n* independent connections start up, the total initial congestion window size is *n* segments. In contrast, the initial window size with TCP session is one segment. The large initial window size can be advantageous if the network is able to absorb the burst of *n* packets, but, if not, it could lead to increased packet loss, and consequently degraded performance.

2. During the linear phase, the congestion window of *n* connections grows at the rate of *n* segments per RTT, compared to 1 segment per RTT for TCP session. This pushes the window size back more quickly to the point where packet loss occurs, causing more frequent packet losses.

3. With independent TCP connections, only the connection that experiences a packet loss halves its window. In contrast, with TCP session, the entire session window is halved. The reduced backoff in the former case, again, increases the frequency of packet loss.

The evolution of the congestion window in actual practice may be quite different from that depicted in Figure 6.1. This is so because of retransmission timeouts. As we shall see in Section 6.5, independent TCP connections are more prone to timeouts, especially if each of them is short in length.

Next, we turn to connection scheduling.

6.3.3 Connection Scheduling

Integrated congestion control entitles the sender to sent data so long as *session_ownd* is less than *session_cwnd*. When a new ack arrives, the sender decrements *session_ownd* to account for the data that has been acknowledged, thus clearing the way for new data to be sent. Unlike in TCP, where the sender is constrained to send new data only on the connection on which the ack arrived, the TCP session sender has the choice of possibly many connections (all within the same session) on which to send new data. In fact, it could choose to send new data on more than one connection. Picking one or more connections for this purpose is the *connection scheduling* problem.

By default, the operation of the TCP session is entirely transparent to applications. As such, all connections within the TCP session are treated equally from the viewpoint of connection scheduling. Connections are scheduled in a *round-robin* fashion. During each round, a connection is eligible to send up to one data segment (of size at most MSS). This ensures that each connection receives an equitable share of the available bandwidth in a consistent manner, i.e., each connection's share is spread out (roughly) evenly over time rather than being concentrated in bursts. The benefit of such scheduling becomes clear in Section 6.5.1 where we contrast it with independent TCP connections that compete with each other.

A connection may not be in a position to send any data when its turn comes because of two reasons — there is no more data left to send and/or the receiver-advertised flow control window (which is still maintained on a per-connection basis) does not permit sending any more data. The former could happen because the sending application is either slow in generating data or has finished sending all its data. The latter could happen because the receiving application is slow in consuming data. In either case, the connection scheduler simply skips over such connections until they are in a position to send data.

It is instructive to note that even if a subset of connections (i.e., data streams) in a TCP session stall, this has no impact on the rest of the data streams. This is in contrast to the head-of-line blocking problem that afflicts application-level multiplexing solutions such as P-HTTP.

In Chapter 7, we discuss more a sophisticated connection scheduling algorithm. We also discuss the benefits of exposing connection scheduling to applications via an appropriate API. used by applications to vary weights dynamically, in Chapter 7. Next, we discuss the final component of TCP session, integrated loss recovery.

6.3.4 Integrated Loss Recovery

We now discuss integrated loss recovery, the third component of TCP session. Integrated loss recovery is motivated by the observation in Section 4.3.2 the short length of typical Web connections makes timeouts the predominant means of loss recovery, thereby degrading performance. An important benefit of P-HTTP is that a persistent connection, which tends to be longer than individual HTTP connections, makes data-driven loss recovery more effective, thereby avoiding many of the timeouts. The goal of integrated loss recovery is to achieve a similar performance gain but in the context of TCP session, with its possibly numerous connections.

The key to effective data-driven loss recovery is having a long enough data stream that the sender has the opportunity to detect packet loss by observing the sequence of acks received in response to the (several) data segments that it sends out. For instance, consider the case of the TCP fast retransmission algorithm illustrated in Figure 6.2. It is only because the sender sends more segments beyond the one that happens gets lost that it receives multiple *duplicate acks* for the lost segment, and so is able to reliably infer and recover from the loss. The TCP sender waits for a threshold number (typically 3) of duplicate acks to provide a certain level of protection against packet reordering.

When a connection that is short in length suffers a packet loss, it often does not receive a sufficient number of duplicate acks. However, it is still possible for the sender to detect the packet loss by observing acks for data segments sent on other connections in the same session. This is the key to integrated loss recovery. Analogous to duplicate acks, we define the notion of a *later ack* as follows:

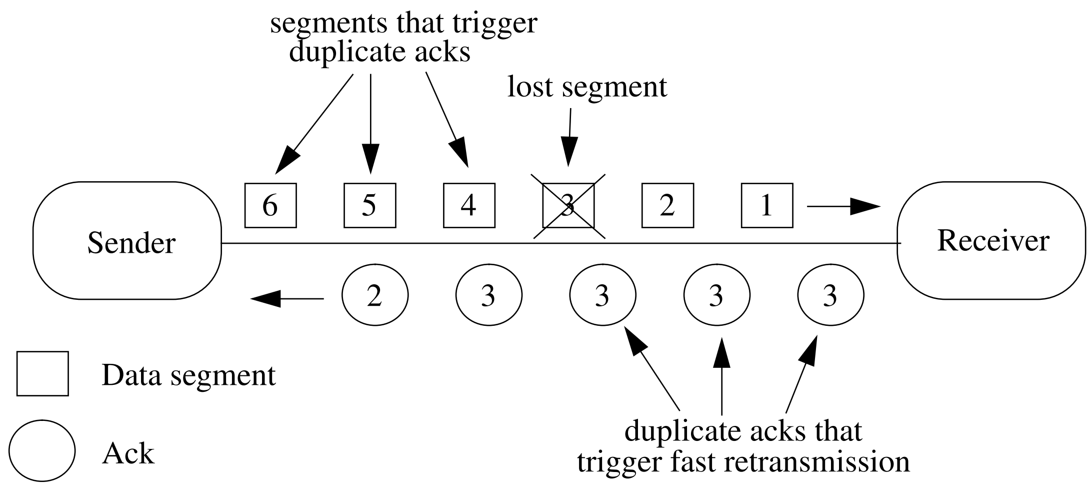


Figure 6.2 An illustration of TCP fast retransmission. If data segments 4, 5 and 6 had not been sent, the duplicate acks would not have been received and fast retransmission would not have been triggered. Note that TCP acks specify the sequence number of the next segment expected, so in our illustration an ack has a sequence number that is one greater than that of the data segment that triggered it.

Definition: An ack is termed as a *later ack* for an unacknowledged segment, *S*, belonging to a connection, *C*, if it acknowledges one or more segments sent after *S* on one of the other connections in the same TCP session as *C*.

Figure 6.3 illustrates the notion of later acks for the case of two connections in a TCP session. Acks numbered 2, 3 and 4 (shown shaded) are later acks for (the lost) segment 2 (shown in white). Later acks for an unacknowledged segment can be used in a way similar to duplicate acks to reliably infer packet loss while at the same time ruling out packet reordering as the underlying cause. This is because the segment(s) acknowledged by a later ack would presumably have (successfully) traversed the same path as the unacknowledged one. If a sufficient number of such *later* segments reach the receiver, chances are that the unacknowledged segment, which was sent prior to these later segments, was lost. The threshold for the number of later acks (plus duplicate acks) needed to reliably infer packet loss could be set to be the same as the TCP fast retransmission threshold, i.e., 3, to provide a similar level of protection against packet reordering[4].

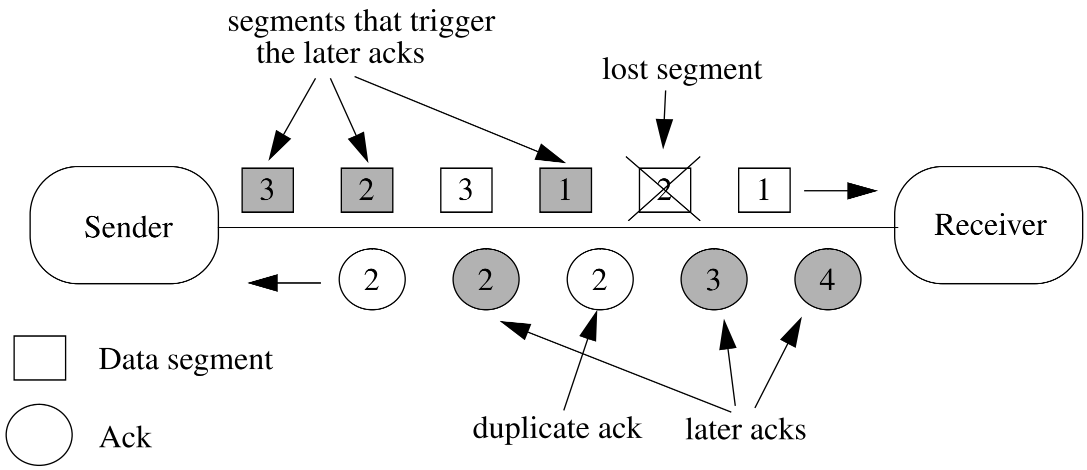


Figure 6.3 An illustration of later acks in the context of two TCP connection (the white one and the shaded one) in a TCP session.

There are two potential problems with using later acks:

1. TCP receivers often employ the delayed ack algorithm [101] where they only send an ack for every other data segment received. So the lack of an acknowledgement for a segment may simply be because the receiver is waiting for a second segment to arrive on the same connection before acknowledging them both. In the meantime, however, later segments on other connections may arrive in sufficient numbers that the receiver sends back acks for them. In such a situation, the sender may mistakenly assume that the missing segment is lost. We address this problem when we develop the loss recovery rules in Section 6.3.4.1.

2. The ack for a segment may get lost. When the sender subsequently receives later acks for this segment, it may mistakenly assume that the segment itself had been lost. This situation arises fundamentally because an ack on a connection does not explicitly convey any information about segments that have been successfully delivered on other connections. We defer the dis-

4. Links composed of multiple physical channels bundled together (e.g., a dual-channel ISDN line) may make reordering a more common occurrence. This may require revising the TCP fast retransmission threshold. We view this issue as orthogonal to integrated loss recovery and TCP session. A solution developed in the context of TCP can be translated to a corresponding one in the context of TCP session.

cussion of this issue to Chapter 7, where we evaluate the potential for false retransmissions, both via simulation and via experiments on the Internet, and propose a simple modification to the TCP receiver algorithm to alleviate this problem.

Next, we discuss the specifics of the integrated loss recovery algorithm, with the explicit goal of addressing the delayed ack problem.

6.3.4.1 Loss Recovery Rules

We now discuss in detail how the sender uses duplicate ack and later ack information to infer packet loss. For each unacknowledged segment in the session, the sender maintains an up-to-date count of the number of duplicate acks (*dupacks*) and later acks (*lateracks*). We discuss implemention details in Appendix B.

We specify a set of three rules that the server uses to infer packet loss. The first rule is the same as standard TCP fast retransmission. We denote the duplicate ack threshold by *thresh* (with a default value of 3).

Rule #1: *If a segment is the unacknowledged segment with the smallest sequence number on its connection (i.e., the left-most one in the window) AND it has at least thresh duplicate acks AND it has not already been retransmitted, then it is eligible for retransmission.*

Note that the check for the segment being the left-most one in the window is superfluous in this case because only such a segment can possibly have a duplicate ack for it. However, we retain this clause for consistency across the three rules.

The second rule pertains to the case where a combination of duplicate and later acks have been received. As mentioned in Section 6.3.4, later acks by themselves do not conclusively point to a packet loss because of the possibility of the delayed ack algorithm being employed by the receiver. However, if at least one duplicate ack has been received, then this rules out the delayed ack algorithm as the reason why no ack has yet been received for the unacknowledged segment. Therefore, the second rule is:

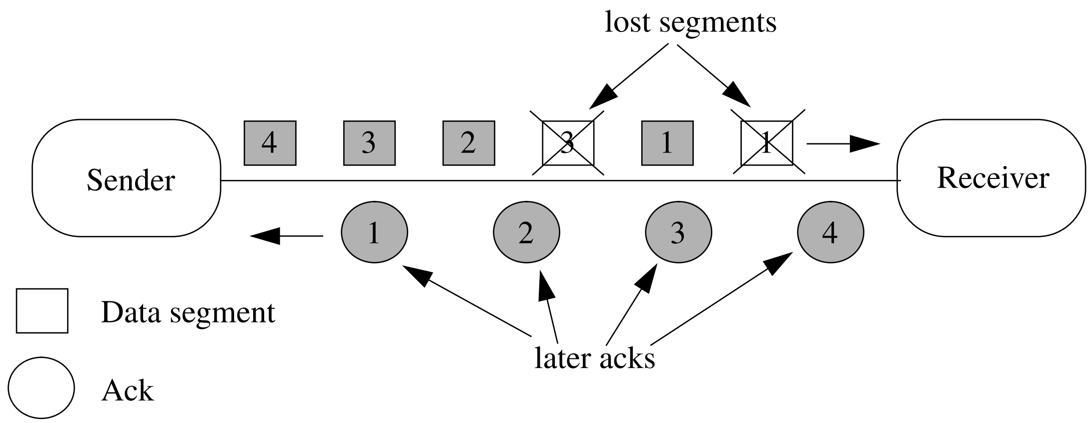


Figure 6.4 An illustration of loss recovery rule #3. Based on the sequence of later acks received, the sender can reliably infer the loss of segment 1 of the white connection (which is the unacknowledged segment with the smallest. sequence number on that connection).

Rule #2: *If a segment is the unacknowledged segment with the smallest sequence number on its connection AND it has at least thresh duplicate and/or later acks combined AND it has at least one duplicate ack AND it has not already been retransmitted, then it is eligible for retransmission.*

This rule is illustrated in Figure 6.3, where segment 2 of the white connection receives one duplicate ack and three later acks (one more than necessary assuming thresh is set to 3), and so is eligible for retransmission.

The third rule pertains to the case where only later acks (and no duplicate acks) have been received. The key observation is that even if a receiver employs the delayed ack algorithm, it is required to at least send an ack for every other segment received. So if the sender receives the threshold number of later acks (or perhaps more) for two or more segments in a connection, it can safely conclude the oldest among them must have been lost. Therefore, the third rule is:

Rule #3: *If a segment is the unacknowledged segment with the smallest sequence number on its connection AND each of two or more segments on that connection have at least thresh duplicate and/or later acks combined AND it has not already been retransmitted, then it is eligible for retransmission.*

Rules for Integrated Loss Recovery:

A segment, *seg*, belonging to connection, *conn*, is a candidate for retransmission if and only if at least one of the following conditions is true:

1. *seg->seq == conn->snd_una && seg->dupacks >= thresh && NOT_REXMT(seg)*

2. *seg->seq == conn->snd_una && (seg->dupacks + seg->lateracks >= thresh) && (seg->dupacks >= 1) && NOT_REXMT(seg)*

3. *seg->seq == conn->snd_una && (seg->dupacks + seg->lateracks >= thresh) && (seg2->dupacks + seg2->lateracks >= thresh) && NOT_REXMT(seg)*

Note:

a. *conn->snd_una* denotes the highest ack on the connection *conn*.

b. *seg2* is a second segment belonging to the same connection as *seg*.

Figure 6.5 Rules for integrated loss recovery.

Note that a particular ack may be a later ack for more than one segment on a connection. For instance, in Figure 6.4, ack 1 on the shaded connection is a later ack for just segment 1 of the white connection, but acks 2, 3 and 4 are later acks from both segments 1 and 2.

The three rules for integrated loss recovery are summarized Figure 6.5. If the sender is unable to reliably detect a loss using these rules, it falls back to a retransmission timeout.

We briefly discuss the implementation of TCP session in BSD/OS next. Further details with regard to integrated loss recovery are presented in Appendix B.

6.4 Implementation in BSD/OS

We have implemented TCP session both in the *ns* network simulator [81] and the BSD/OS 3.0 TCP/IP stack [16]. The implementation details outlined above form the basis for both implementations. Here, we briefly discuss our BSD/OS implementation from the viewpoint of how it interfaces with the existing TCP/IP code.

Our implementation of TCP session operates at the granularity of host pairs. It is implemented as a set of functions that are invoked from a small number of points in the TCP code. The set of functions includes those to create a new session, add a new connection to a session, process incoming acks, schedule the sending of data, and destroy a session.

When a TCP connection transitions to the *ESTABLISHED* state, *tcp_input()* calls a new routine, *session_addconn()*, to add the connection to the session between the two hosts. We need to wait until the *ESTABLISHED* state because only then are the identities of the two end hosts known. *Session_addconn()* adds a pointer to the connection's TCB in the *session PCB* (SCB), if it already exists. If not, it first calls *session_create()* to allocate and initialize a new SCB. Although *session_cwnd*, *session_ssthresh*, *srtt* and *rttvar* are maintained on a session-wide basis, we do not remove the corresponding variables from the TCBs of the individual connections. This is to minimize the amount of change needed to the existing TCP code. We simply ignore these variables in the individual TCBs.

When *tcp_output()* is called, it in turn invokes *session_output()*. *Session_output()* first makes sure that the *session_cwnd* does permit the sending of a new segment. Then, among the connections in the session that have data to send, it picks the one that should send a segment next according to the weighted hierarchical round-robin schedule. Note that this connection could, in general, be different from the one that invoked *tcp_output()* to begin with. However, an exception is made if the original connection wanted to send out a segment with a control flag, such as *FIN* or *RST,* turned on. When a segment is to be sent out on a particular TCP connection, *tcp_output()* is invoked with a pointer to the appropriate TCB passed as an argument. A special flag is used to indicate that *tcp_output()* is being invoked from *session_output()*, so it should *not* turn back and call *session_output()* again.

When an ack is received, *tcp_input(),* as usual, updates the highest ack information and clears out acknowledged data from its buffer. However, it does not do duplicate ack processing. Instead, it defers to *session_input()* which detects packet reordering by looking for both duplicate acks and later acks. If necessary, *session_input()* invokes *tcp_output()* to retransmit the appropriate segment.

The TCP timer function, *tcp_slowtimo()*, is invoked periodically to scan through the list of TCBs to determine whether a timeout is due for any connection. We augment this code in two ways. First, in addition to the list of TCBs, a separate list of SCBs is also scanned. For any session with a timeout pending, *session_timeout()* is invoked. Second, a retransmission timeout event for a connection (TCB) that is part of a session is simply ignored.

Finally, when the last connection in a TCP session is closed, *session_destroy()* is called to clean up state and free up the SCB. However, when TCP fast start (Chapter 8) is combined with TCP session, the SCB is retained to preserve state for future use.

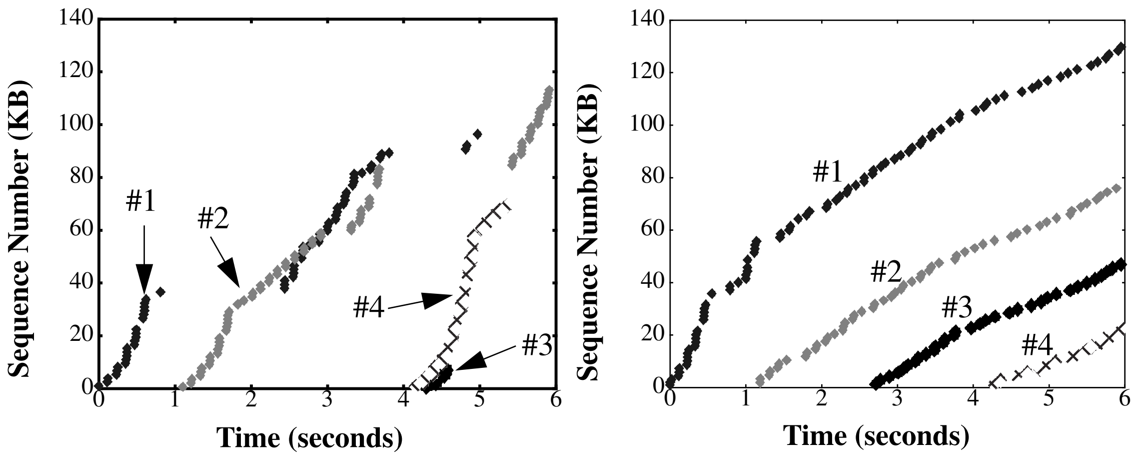
6.5 Performance Results

We conducted several experiments, using both our implementation in the *ns* simulator and that in BSD/OS, to evaluate the performance of TCP session under a variety of conditions. Much of our emphasis here is on the former because it enables us to conduct experiments on a larger scale and measure the various aspects of performance is greater detail. However, we begin with a simple experiment conducted using our implementation in BSD/OS.

6.5.1 Competition versus Sharing between Concurrent Connections

This experiment involves 4 TCP connections between a pair of hosts, launched about 1.5 seconds apart. This emulates a Web page download with a separate TCP connection for each of 4 inline images. We used our link emulator module (Chapter 3) to dial in a bottleneck link bandwidth of 1.5 Mbps (T1 speed), an RTT of 100 ms and a buffer size of 4 packets.

We conducted the experiment using two configurations — (a) independent TCP connections, and (b) TCP session. The sequence number plots for the two cases are shown in Figure 6.6. We observe a significant differences in the two cases. The performance of each connection in the case of independent connections is highly variable. There are times during which a subset of the connections makes progress while the rest stagnate. For instance, connection #3, launched at around 3 seconds, starts making progress only after connection #4 starts up. In a real setting, it may be that



(a) (b)

Figure 6.6 The dynamics of four connections sharing an emulated 1.5 Mbps bottleneck link with (a) independent TCP connection, and (b) TCP session.

connection #3 is the most important one from an application's viewpoint. But there is little that the application can do to prevent the performance degradation that this connection suffers.

In contrast, the performance with TCP session is much more consistent. Round-robin connection scheduling ensures that each connection gets an equal share of the available bandwidth. When a new connection starts up, the connections that are already active are slowed down a little to accommodate it.

In general, TCP session has three advantages:

1. Integrated congestion control eliminates competition between connections in the same TCP session, thereby reducing the total amount of competition in the network. This helps cut down the packet loss rate.
2. Explicit connection scheduling, perhaps with input from the application, removes uncertainty in how the bandwidth available to a TCP session is shared among its constituent connections. However, uncertainty in the share of the total bandwidth that a session is able to get remains, albeit at a reduced level because of 1 above.

3. Integrated loss recovery improves the chances of data-driven loss recovery, especially when connections are short in length. This decreases the occurrence of retransmission timeouts that lead to long periods of inactivity and poor performance (as evident for connection #1 in Figure 6.6(a)).

It may be argued that the first point listed above only amounts to scaling down the amount of competition in the network by a small factor. It does nothing about the competition between hosts. While this is true, we make two observations. First, there are many situations where the bottleneck link is not shared by multiple hosts. A dialup modem line connecting a client is a common example. In such situations, scaling down the competition by even a small factor helps, as our results in Section 6.5.2 show. Second, even when there are several hosts sharing the bottleneck link, the other two points listed above still stand. Our experimental results highlight this.

6.5.2 Simulation Results

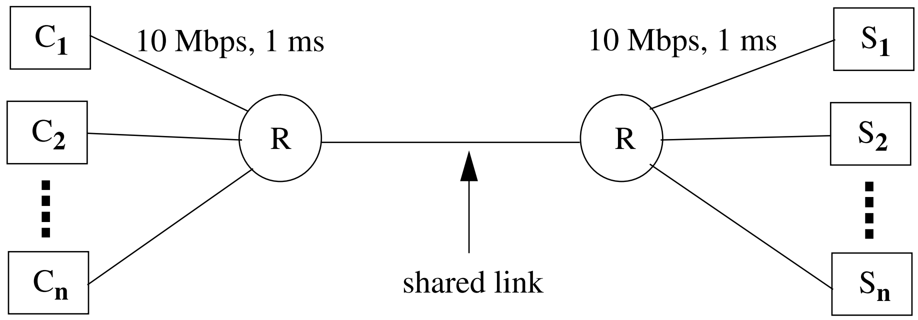
We now present detailed results from simulations experiments. The topology used for these experiments is shown in Figure 6.7. It consists of a large number of client-server pairs that share a common link. Depending on the speed of this link and the number of client-server pairs that are active simultaneously, this link may or may not be a bottleneck.

Traffic is generated using a simulated Web benchmark, which is designed to be representative of a "typical" Web transaction. It consists of four steps: the client sends a request for the HTML, the server responds with the HTML, the client then sends requests for each inline image, and finally the server sends back the images. In some experiments, background traffic in the form of bulk transfers is also introduced.

Figure 6.7 also summarizes the settings (including the default one) for the various simulation parameters.

We evaluated five different protocol configurations:

1. *httpseq*: HTTP/1.0 with sequential connections of the TCP NewReno flavor.
2. *httpconc*: HTTP/1.0 with concurrent connections of the TCP NewReno flavor. There are as many concurrent connections as inline images in the page being downloaded.



Link parameters

Bandwidth: 1.5 Mbps, 28.8 Kbps, 45 Mbps
Delay: 50 ms, 200 ms
Buffer size: 20 packets

TCP parameters

Max segment size (MSS): 1 KB
Max window size: 32 KB

Web benchmark parameters

HTML request size: 0.3 KB
HTML response size: 1 KB
Image request size: 0.3 KB
Image response size: 1-100 KB (10 KB by default)
inline images: 1-10 (4 by default)
simultaneous downloads: 1-30

Figure 6.7 The simulation topology and parameter values. For parameters whose values are varied, the default setting is underlined.

3. *phttp*: P-HTTP with a persistent connection of the TCP NewReno flavor.
4. *session*: TCP session
5. *session-noilr*: TCP session without integrated loss recovery (i.e., with only per-connection loss recovery as in standard TCP).

Of these, *httpconc* and *session* are of particular interest to us and are analyzed in the greatest detail. The former is representative of popular browsers such as Netscape Navigator that launch multiple concurrent connections with the hope of speeding up the Web transaction. As we shall see, this can often be highly counterproductive. The *httpseq* configuration is included to show that in spite of the drawbacks mentioned in Chapter 4, it could perform much better than *httpconc* in congested environments. The *phttp* configuration is included to see how close session is able to approach it in terms of performance, while at the same time avoiding the drawbacks of P-HTTP mentioned in Chapter 5. Many of these drawbacks (for example, the undesirable coupling of logically-separate data streams) are difficult to quantify and only manifest themselves in an actual implementation. They do not have an impact on the performance metrics listed below, so the graphs for *phttp* and *session* are very similar in all the cases. Finally, a comparison between *session* and *session-noilr* helps evaluate the benefits of integrated loss recovery.

We use several metrics to quantify performance. These include:

1. The *total download time*, averaged over several (up to 30) clients and also multiple runs.
2. *Fairness index* computed over the download time of each individual client and averaged over multiple runs.
3. The number of *packet losses* and the number of *retransmission timeouts*, both averaged over multiple runs.

Error bars represent 95% confidence intervals for the mean.

6.5.2.1 Impact of the Number of Simultaneous Web Downloads

In this experiment, we evaluate the impact of the number of simultaneous downloads on performance. Each download is between a distinct client-server pair, so the i^{th} download is between client C_i and server S_i. All the downloads are initiated within a short, 100 ms interval. The bottleneck link bandwidth is 1.5 Mbps and delay 50 ms.

Figure 6.8(a) shows the average download time per client versus the number of simultaneous downloads for three protocol configurations — *httpseq*, *httpconc*, and *session*. As we would expect, the download time increases as the number of simultaneous downloads increases. However, the increase is the greatest for *httpconc* and the least for *session*. There is a speedup of up to 60% (i.e., 2.5X) in the download time with *session* compared to *httpconc*. The speedup over *httpseq* is smaller but still significant — up to 30%. The performance with *phttp* is quite similar to that with *session*, so we only show the curve for the latter.

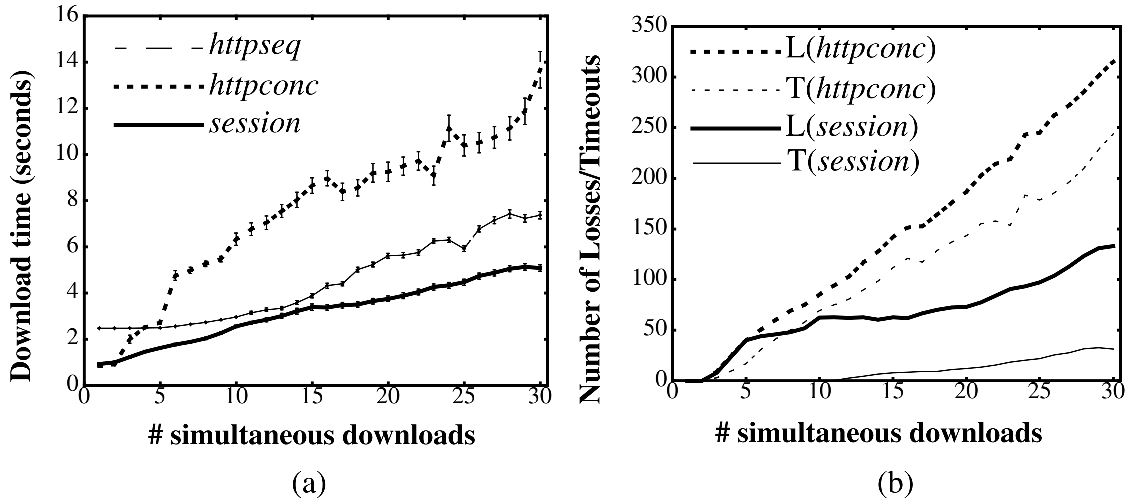


Figure 6.8 Impact of the number of simultaneous downloads on (a) the average download time per client, and (b) the total number of packet losses (L) and retransmission timeout (T).

Figure 6.8(b) goes into the reasons for the significant performance gains of *session* in more detail. It shows the total number of packet loss and retransmission timeout events for each of *httpconc* and *session*. Comparing the total number of losses, we observe that it increases much more sharply for *httpconc* than for *session*. Towards the right edge of the graph, it is over 2.3X larger for *httpconc* than for *session*. As discussed in Section 6.3.2 (and depicted in Figure 6.1), the independent connections (4 per download, in this case) in *httpconc* do not share congestion information with each other and push the network beyond the brink (thereby causing packet loss) much more frequently.

There is an even more dramatic difference when we compare the number of timeouts in the two cases. This is often more than 7X greater for *httpconc* than for *session*. Another way of viewing this is that whereas over 75% of packet losses result in a retransmission timeout with *httpconc*, under 25% do so with *session*. The short length of each connection (10 KB for each inline image) often forces it to wait for a timeout to recover from a packet loss. In contrast, integrated loss recovery enables data-driven loss recovery to succeed much more often in the case of *session*.

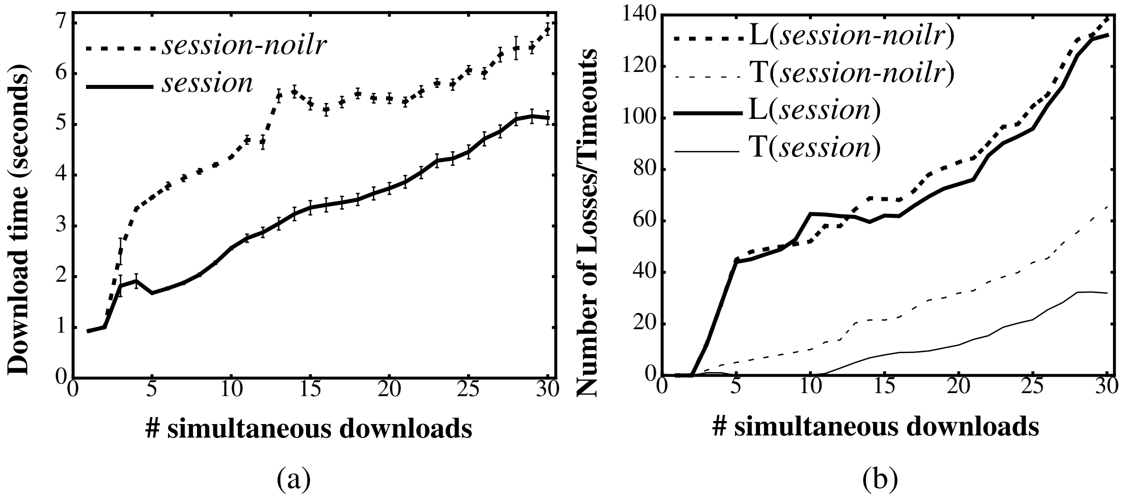


Figure 6.9 Impact of integrated loss recovery on (a) the average download time per client, and (b) the total number of packet losses (L) and retransmission timeouts (T).

Integrated loss recovery is also the reason why *session* performs better than *httpseq*. Although *session* suffers about 30% more losses than *httpseq* (on account of the latter operating with a very small effective window), it only experiences about half as many retransmission timeouts.

In summary, launching multiple connections concurrently can result in significant performance degradation. Part of this is because of increased packet losses. The rest is because of the predominance of timeouts. The latter also afflicts separate connections that are launched sequentially (*httpseq*). TCP session achieves significant performance gains compared to these because of integrated congestion control and integrated loss recovery. We analyze the latter in more detail next.

6.5.2.2 Impact of Integrated Loss Recovery

To isolate the role of integrated loss recovery in the performance gains of *session*, we create a version of *session* minus integrated loss recovery. We call this *session-noilr*. Basically, *session-noilr* is identical to *session* except that it does not use later ack information for loss recovery. This is implemented by setting the *lateracks* count to zero for the purpose of loss recovery rules #2 and #3 in Figure 6.5.

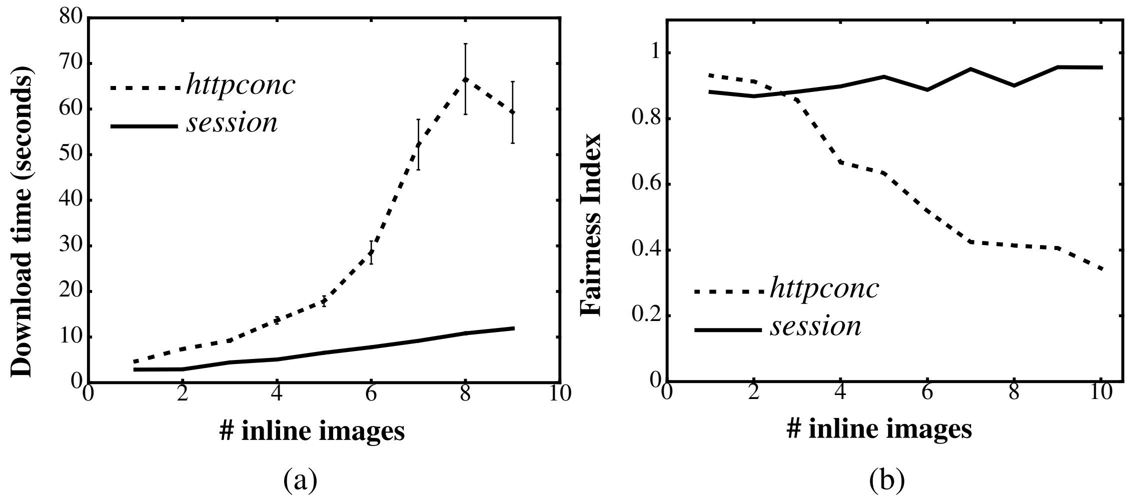


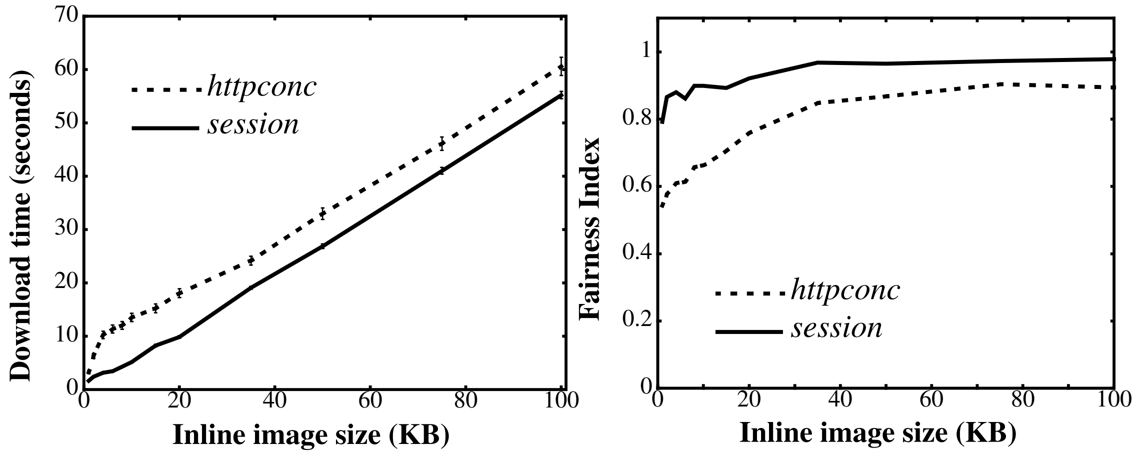
Figure 6.10 Impact of the number of inline images per page on (a) the average download time per client, and (b) the fairness index computed over the download time for each individual client. The inline image size was set to 10 KB and the number of simultaneous downloads to 30.

We conduct the same experiment as in Section 6.5.2.1. Figure 6.9(a) shows that the average download time degrades by amounts ranging from 35-100% (i.e., 1.35-2X) when integrated loss recovery is disabled. This is corroborated by Figure 6.9(b) which shows that although the total number of losses is about the same in both cases (because both employ the same integrated congestion control algorithm), the number of timeouts with *session* is only about half as much as with *session-noilr.*

Our results underscore the importance of integrated loss recovery. It is this that enables TCP session to perform data-driven loss recovery very effectively even when connections are short in length.

6.5.2.3 Impact of the Number of Inline Images

In this experiment, we investigate how the number of inline images in the page being downloaded impacts performance. We compare *httpconc* and *session*, both of which launch as many concurrent connections as there are inline images. Figure 6.10(a) plots the average download time per client as a function of the number of inline images. We observe that while the performance in the two



(a) (b)

Figure 6.11 Impact of inline image size on (a) the average download time per client, and (b) the fairness index computed over the download time for each individual client. The number of inline images per page is set to 4 and the number of simultaneous downloads to 30.

cases is similar when there is only one inline image per page, there is a growing difference as the number of inline images increases, ending up with a difference of more than 6X. This trend arises because the benefit of integrated congestion control and loss recovery is greater when the integration happens over a larger number of (concurrent) connections.

Figure 6.10(b) plots the fairness index in the two cases. This index is computed over the download time for each client, and is a measure of how equitably the clients are treated. The figure shows the index starts off with a high value (around 0.9) in both cases, indicating that all clients receive similar performance. However, as the number of inline images increases, the fairness index for *httpconc* plunges, ending up at around 0.4 when there are 10 inline images per page. The reason for this is that as the number of inline images increases, the amount of packet loss in the case of *httpconc* tends to increase. (Part of this increase is because of the large initial burst mentioned in Section 6.3.2). In turn, this leads to an increase in the occurrence of timeouts. Since timeouts have a very significant impact on performance, there is a great disparity in the download time for clients whose connections experience a large number of timeouts, compared to those that experience fewer timeouts or even none. This causes the decline in the fairness index.

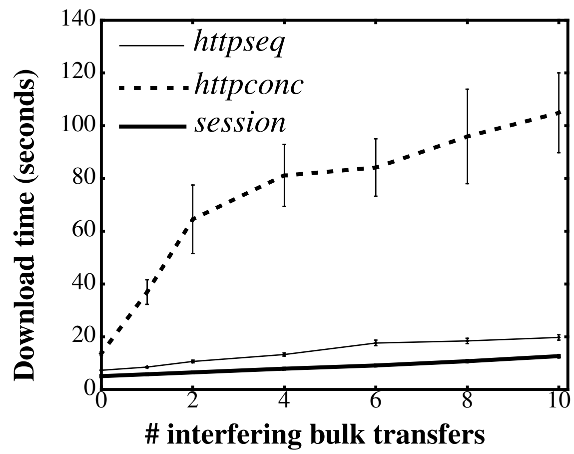
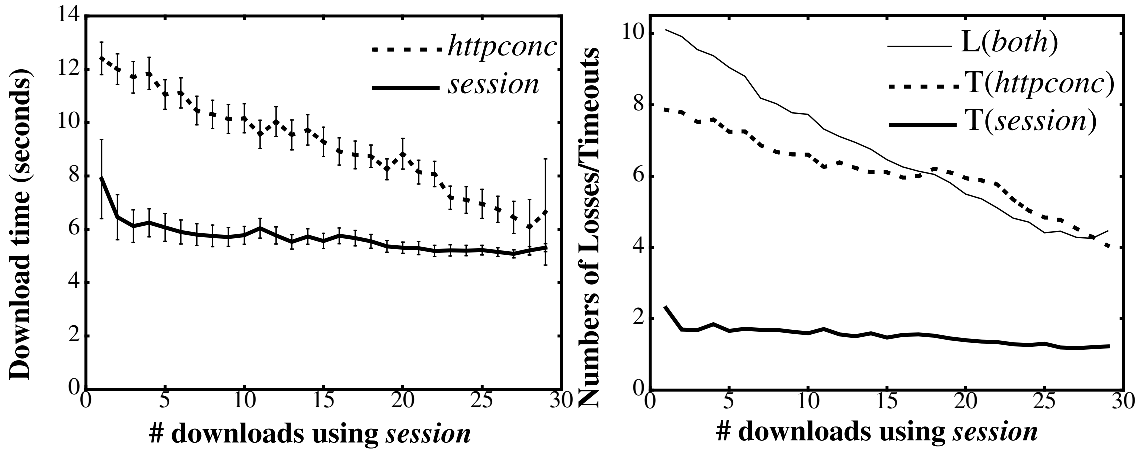


Figure 6.12 Impact of interfering bulk transfers on the average download time per client. The number of inline images per page is set to 4 and the image size to 10 KB.

On the other hand, *session* is able to avoid such degradation in the level of fairness by reducing the occurrence of timeouts.

6.5.2.4 Impact of the Inline Image Size

In this experiment, we study the impact of the size of each inline image on performance. Recall that the size of an inline image determines the length of the connection used to transfer the image. From Figure 6.11(a), we see that there is a steady increase in the average download time as the inline image size increases. This is as we would expect. Furthermore, the benefit of *session* over *httpconc* diminishes, in percentage terms, from over 50% (i.e., over a factor of 2) for 1 KB images to under 10% for 100 KB images. The reason for this decrease in relative benefit is that as the inline image size grows, each individual connection gets longer. So data-driven loss recovery becomes increasingly more feasible, even in the absence of integrated loss recovery. Given this, the trend in the fairness index (Figure 6.11(b)) is also as we would expect. As the inline image size grows and timeouts become less frequent, the degree of fairness also improves, more so for *http-conc*.



(a) (b)

Figure 6.13 Impact of a heterogeneous traffic mix, where some downloads use the *httpconc* configuration and others *session*, on (a) the average download time per client, and (b) the average number of packet losses (combined for both configurations) and timeouts (separate for each configuration) per download. The number of downloads of each type is varied, while holding the total constant at 30. The number of inline images per page is set to 4 and the image size to 10 KB.

6.5.2.5 Impact of Interfering Bulk Transfer Traffic

So far in our experiments, we have only considered Web-like traffic between clients and servers. We now introduce cross-traffic in the form of bulk transfers, in addition. These share the bottleneck link with the Web downloads. The number of bulk transfer connections is varied from 0 through 10. Figure 6.12 shows the impact of the bulk transfers on the Web download time. The most striking point in this graph is the wide disparity in the performance of *httpconc* and *session*. The download time with *httpconc* grows to be about 10X that with *session*. The reason for this is that the bulk transfers cause congestion in the network. When *httpconc* starts of with a large total initial window, there is heavy packet loss, leading to a large number of timeouts.

There is a smaller, but still significant, improvement compared to *httpseq*. *Session* cuts down the average download time by up to a factor of 2. This is because of its improved ability to do data-driven loss recovery.

6.5.2.6 Impact of Heterogeneous Traffic Mix

In the same spirit as the previous experiment, we now evaluate *session* in an environment where there is a mixture of Web downloads that use *session* and those that use *httpconc*. The number of downloads using session is varied from 1 through 29 while the total number of simultaneous downloads is kept constant at 30 (so by implication, the rest use *httpconc*).

Figure 6.13(a) shows how the performance of the two categories of downloads varies with the traffic mix. We observe that both categories benefit as the mix shifts towards *session*. The benefit is greater for *httpconc*. The reasons for these trends is that when the traffic mix is dominated by *httpconc*, the network is more congested. The consequent packet losses adversely impact *httpconc* more than *session* because the former is more vulnerable to timeouts. As the share of *session* in the mix increases, the level of congestion in the network subsides, so *httpconc* suffers fewer timeouts and benefits substantially. *Session* also benefits, although not quite as much because timeouts were not a significant problem for it to begin with.

The trend in the frequency of timeouts is evident from Figure 6.13(b), which shows the average number of packet losses and timeouts per download. The frequency of timeouts with *httpconc* drops from being about 4.5X worse than *session* when the mix is dominated by *httpconc* to being about 3X worse when the mix is dominated by *session*. Note that in some cases, the average number of timeouts per download for *httpconc* actually exceeds the average number of losses per download. This is not an anomaly because the latter is computed by averaging over all downloads, including those that use *session*.

The results of this experiment and the previous one highlight the resilience of *session* in environments with a heterogeneous mix of traffic. It is advantageous to use *session* even when the majority of hosts in the network do not. This is important from the viewpoint of deploying it incrementally. The increasing use of *session* reduces the level of congestion in the network, which also benefits hosts that do not use *session*.

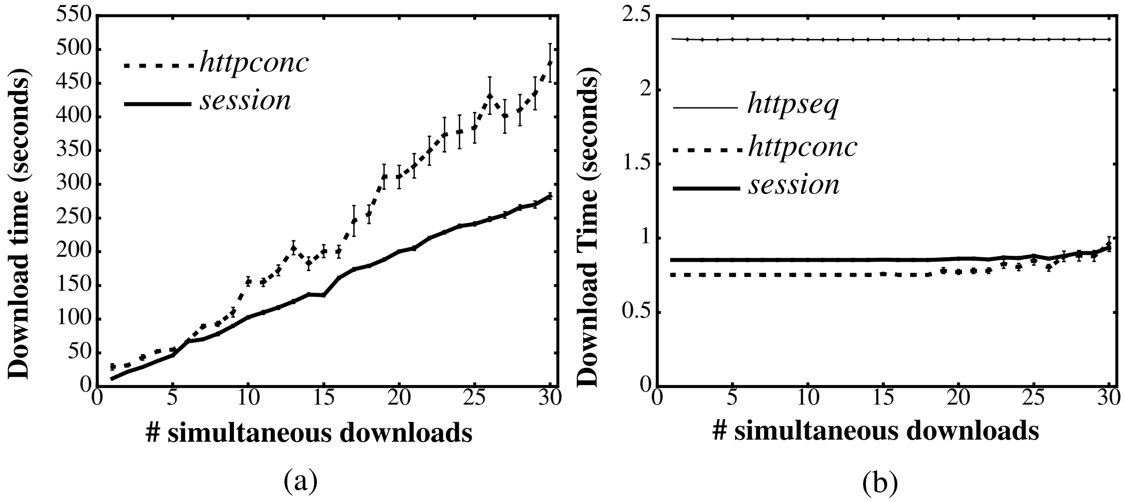


Figure 6.14 Impact of link bandwidth, (a) 28.8 Kbps and (b) 45 Mbps, on the average download time per client. The number of inline images per page is set to 4 and the image size to 10 KB.

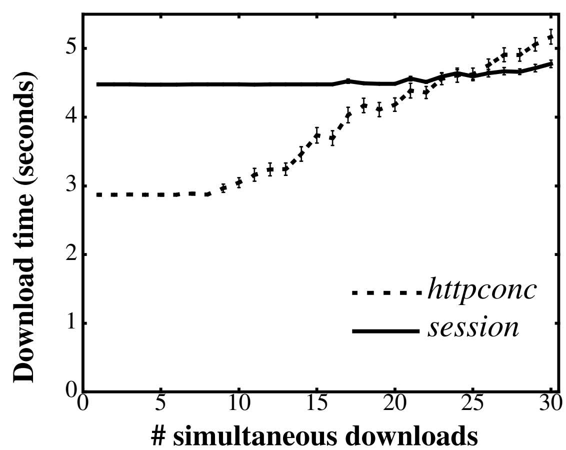


Figure 6.15 The impact of a large RTT on the average download time per client. The link bandwidth is set to 45 Mbps and the delay to 200 ms (so that the RTT is about 400 ms). The number of inline images per page is set to 10 and the image size to 10 KB.

6.5.2.7 Impact of the Link Speed and Delay

In all our experiments so far, the bottleneck link bandwidth has been held fixed at 1.5 Mbps, and its delay at 50 ms. We now vary these and repeat the experiment from Section 6.5.2.1, to study their impact on performance.

Figure 6.14(a) show the download time when the bottleneck link bandwidth is 28.8 Kbps. This extremely limited link bandwidth causes heavy congestion and packet loss. The consequent timeouts cause a sharp increase in download time as the load (i.e., number of simultaneous downloads) increases. The increase is greater for *httpconc* than for *session* because of the former's greater susceptibility to timeouts. Although the difference in performance in absolute terms is quite large (often over 100 seconds), the difference in relative terms is smaller than when the bottleneck link bandwidth was larger (Section 6.5.2.1). The reason is that with a link bandwidth of 28.8 Kbps, the transmission time for each packet is quite significant compared to the retransmission timeout (RTO) value. For instance, the transmission time for a 1KB packet over the 28.8 Kbps link is about 280 ms. In contrast, the transmission time over a 1.5 Mbps link is only about 5 ms. However, the difference in RTO in the two cases is not quite as much because it is computed at a rather coarse granularity (typically 500 ms). Consequently, the cost of a timeout (i.e., the RTO) is less significant in relative terms when other costs (e.g., transmission time) are large.

Figure 6.14(b) shows the situation at the other end of the spectrum where the link bandwidth is large (45 Mbps). We observe a markedly different behavior. The performance of *session* is slightly worse (about 13%) than that of *httpconc*. However, both of these perform much better (about 3X) than *httpseq*. These trends arise because there is only mild congestion given the high link bandwidth. Consequently, packet losses and timeouts are infrequent and only have a small impact on performance. The main issue becomes growing the congestion window to a large enough size to utilize the available network capacity. In this context, *session* and *httpconc* are in an advantageous position, the latter more so because the ensemble of concurrent connections starts of with a larger total initial window. As explained in Chapter 4, *httpseq* suffers, both because of repeated invocations of slow start and because of sequential operation.

These effects are even more pronounced in Figure 6.15, which shows the case of a 45 Mbps link with an RTT of 400 ms (akin to a geostationary satellite link). The number of inline images is set to 10 (instead of 4 in Figure 6.14). This means that *httpconc* launches 10 concurrent connections to retrieve the inline images. The consequent large initial window and faster window growth enable it to perform up to 35% better than session when the load is low. However, as the load increases, the concurrent connections launched by a large number of downloads overwhelm the network, causing more frequent packet loss. Consequently, the performance of *httpconc* begins to degrade, ending up worse than *session*.

The download time for *httpseq* under conditions of low load is around 21 seconds (not show in the figure), which is over 7X worse than for httpconc and a little under 5X worse than for session.

Thus, while *session* still maintains its performance advantage when the network bandwidth is very constrained, it performs worse than *httpconc* when capacity is plentiful and delay is large.

6.5.3 Summary of Results

Here is a summary of our key results:

1. *Session* outperforms *httpconc* under conditions of high load. In terms of download time, there is a speedup by up to a factor of 10. There are two reasons for this:

   - Integrated congestion control cuts down the packet loss rate significantly.

   - Integrated loss recovery cuts down the frequency of timeouts drastically.

   Improving performance when the network is heavily loaded is especially important because it is under these conditions that users perceive long delays, which leads to complaints.

2. The use of *session* is often advantageous even in a mixed environment where not all traffic uses *session*. This is significant because it facilitates incremental deployment. Furthermore, as the use of *session* grows, even other traffic in the network benefits because of the reduced level of congestion.

3. *Session* also performs better than *httpseq* though not quite as much. The performance difference is the greatest (up to 2X) when there is a significant amount of interfering traffic other than the Web traffic of interest. The performance benefits arise due to the ability of integrated congestion control to cut down the occurrence of timeouts.

4. When the bandwidth-delay product is large, *session* performs worse than *httpconc*. This is because, unlike the latter, it always starts off with a small initial session window size of 1 segment, so it is less able to fully utilize the available capacity. *Httpseq* performs much worse than both *session* and *httpconc*. *Phttp* performs similar to *session*.

5. Connection scheduling enables *session* to provide a consistent share of the bandwidth to each concurrent connection. In contrast, with *httpconc* the share obtained by each connection can be quite unpredictable. By comparison, neither *httpseq* nor *phttp* permits concurrent connections.

6. In spite of allowing applications to launch short and concurrent connections, *session* achieves as small a download time as *phttp*. Integrated congestion control and loss recovery blunt the adverse impact that short connections may have otherwise had. Furthermore, session eliminates the disadvantages of phttp mentioned in Chapter 5 — it enables having concurrent data streams without any unnecessary coupling between them, it is transparent to applications, and it confines changes to the sender side (typically, Web servers).

6.6 Limitations of TCP session

In spite of its advantages, TCP session has some limitations:

1. As mentioned above, the small initial session window size could lead to poor utilization of the available network capacity, given that total transfer size for a Web page may only be of the order of 20-30 KB. This is especially so when the network capacity is plentiful, i.e., the network bandwidth is large. Clearly, this is an undesirable situation.

   In Chapter 8, we present an orthogonal technique, *TCP fast start*, to alleviate this problem by reusing information about the network learned in the recent past.

2. It may be expensive to multiplex several data streams within a TCP session at a very fine granularity because of the overhead of the TCP/IP header (usually 40 bytes or more). If this is really an issue, header compression techniques similar to those used for low-speed dialup lines [52] could be used to reduce the overhead.

3. TCP session does not share any information across hosts. There are situation where this may be advantageous. For instance, two hosts on a LAN could benefit by sharing information about the network path to a common server. The SPAND system [97] points to a possible way of effecting such sharing.

6.7 Summary

In this chapter, we have presented the design and evaluation of a new technique, TCP session, which enables applications to open as many concurrent connections as they desire, without the attendant performance penalties. Connections share information rather than compete with each other. The three main components of TCP session are:

1. Integrated congestion control.

2. Connection scheduling.

3. Integrated loss recovery.

Our experimental results show that TCP session performs much better than HTTP with concurrent connections, and about the same as P-HTTP, while at the same time avoiding many of the drawbacks of the latter.

Since TCP session is transparent to applications (by default), it could be used in conjunction with an application-level multiplexing technique such as P-HTTP. If an application chooses to use P-HTTP, TCP session will essentially make no difference to performance. However, if another application on the same host chooses to launch several concurrent connections, TCP session would guard against performance degradation.

We note that although we have presented TCP session in the context of the Web, there is nothing in it that ties it to the Web. The technique would be useful in any situation where hosts exchange multiple concurrent data streams. An example is a shared whiteboard that uses TCP for communication between a pair of hosts [93]. Only pieces of data for which ordered delivery is desirable are mapped onto the same connection. Those corresponding to independent activities (for instance, writing on one end of the whiteboard while loading up an image at the other end) are mapped onto separate connections. TCP session would then be useful in ensuring that these connections do not compete with each other.

In the next chapter, we turn to some advanced issues pertaining to TCP session. Chief among these is exposing connection scheduling to the application. This enables the application to allocate a greater share of the bandwidth to certain connections based on application-level information, and even vary this allocation dynamically.