Chapter 8

TCP Fast Start

Thus far we have developed two different solutions — P-HTTP and TCP session — to address the challenges of Web data transport. The *shared state* used by these techniques offers significant performance benefits compared to independent TCP connections. Furthermore, TCP session offers several advantages over P-HTTP in terms of flexibility. However, as we noted in Section 6.6, both TCP session and P-HTTP have suboptimal performance when the network has a large bandwidth-delay product.

In this chapter, we develop a new solution, *TCP fast start*, which addresses this problem. The basic idea is to use *persistent state* to enable even short transfers to fully utilize the available network capacity. TCP fast start is orthogonal to both P-HTTP and TCP session, and hence can be used in conjunction with either. The rest of this chapter is organized as follows. First, we motivate the problem that we are solving and point out the specific challenges involved in doing so. Then in Section 8.2 we present the design of TCP fast start in detail. We describe our BSD/OS implementation of fast start in Section 8.3. In Section 8.4, we detailed performance results obtained using our implementation and using simulation. We discuss various issues pertaining to fast start in Section 8.5, and present a summary in Section 8.6.

8.1 Motivation

TCP uses the slow start procedure, starting with an initial window size of one segment, to gradually probe the network for bandwidth. This procedure is invoked at the time a new connection begins and when an existing connection (such as a P-HTTP connection) resumes after being idle for longer than a threshold. These procedures are specified in [51,54], which recommend an idle period threshold of one round trip time (RTT) or one retransmission timeout value (RTO). The TCP implementation in BSD/OS sets the threshold to the latter, and so does our implementation of TCP session.

There are two reasons why slow start is invoked under these conditions:

1. Since no data has been sent (at least) during the past RTT, ack clocking would have died down. Slow start provides a convenient way to get ack clocking going again for the new transfer.
2. Since the sender has not probed the network in the recent past (during which time the load in the network could have changed unpredictably), it should be conservative in making an assumption about the available network capacity. Consequently, it starts off with a small initial congestion window size of one segment.

Such invocations of slow start interact adversely with the typical patterns of Web access, which involve periods of activity interspersed by idle periods that are usually at least several seconds long [65]. The idle periods would cause the congestion window to be reset and slow start to be invoked each time activity resumes. This happens even when the same connection is reused, as in P-HTTP. Slow start leads to poor bandwidth utilization, and hence increased latency, especially when the transfer length is short and the bandwidth-delay product is large.

A possible solution is to cache and reuse the congestion window size from the recent past, rather than invoke slow start with an initial window size of one segment. This possibility has been mentioned, for instance, in the T/TCP functional specification [13] and TCP Control Block Interdependence [103] documents. However, such a scheme runs afoul of point #1 made above: a large initial window in the absence of ack clocking could lead to a large burst of packets and, consequently, heavy packet loss. Neither [13] nor [103] presents an implementation or a performance analysis of this problem.

Rate-based Pacing [105], developed contemporaneously with our work, attempts to resolve this problem by using the TCP Vegas estimate of a connection's rate to pace out packets smoothly. While this avoids bursts of back-to-back packets, it could still run into problems if the network load has increased significantly in the interim. In this case, even pacing packets out based on the old rate leads to congestion, and hence packet loss. As we show in Section 8.4.2.2, this leads to poor performance for the connection itself as well as other competing connections.

Next, we discuss our solution, *TCP fast start*, and point out how it addresses the problems mentioned above, while achieving very good performance.

8.2 Design of TCP Fast Start

Like the solutions mentioned above, TCP fast start exploits *persistent* state. The sender caches and reuses information gathered about the network conditions in the recent past, rather than repeating the slow start discovery procedure each time. The cached information[1] includes the congestion window size (*cwnd*), the slow start threshold (*ssthresh*), and the smoothed round-trip (*srtt*) time and its variance (*rttvar*).

Before getting into the details, we mention two important objectives in accordance with the robustness principle of Section 1.8.

1. If the cached information is still valid, fast start should help improve performance. However, if this information is stale (for instance, because of a sudden surge in network load), fast start should not lead to worse performance than if standard slow start had been used in the first place. In other words, slow start places a lower bound on the actual performance.
2. The performance gains of fast start should not be at the expense of other connections. It is okay for the other connections to suffer to the extent that the fast start connection tries to use its share of the bottleneck bandwidth, but not because the fast start connection is being too aggressive.

It is probably quite difficult to meet these goals exactly without introducing a lot of complexity in the network (such as per-connection state in the routers). TCP fast start tries to do its best with

1. Throughout this chapter, the phrase "cached information" refers to cached protocol variables (such as *cwnd*), and not cached Web page contents. Also, we use the phrases "cached information" and "persistent state" interchangeably.

only a simple and easy-to-implement *priority dropping* mechanism in the routers. Our experimental results (Section 8.4) show that fast start is quite successful in meeting these goals.

While fast start is equally applicable to a standard TCP connection as well as to a TCP session, for ease of exposition we present our discussion in terms of a single TCP connection.

8.2.1 Basic Idea

Various studies (e.g., [7], [88]) have shown that the available network bandwidth tends to be stable for periods ranging from a few minutes to tens of minutes. Since the length of a typical pause during an active Web browsing session tends to be much shorter (for instance, [65] reports a median of 15 seconds), it may be reasonable to assume that in the common case the network conditions would not have changed much during such a pause. Consequently, a sender employing fast start caches information about the network conditions and reuses it to expedite start up when the connection resumes activity. To avoid sending out a large burst of packets (due to the absence of ack clocking), the sender uses the cached values of *cwnd* and *srtt* to smooth out the burst.

However, even the smoothed out rate may be too much in (the presumably less common but still important) case of an increase in the level of network load that renders the cached information invalid. This could cause heavy packet loss, and hence degraded performance, for the connection attempting fast start as well as for other connections. This is undesirable given our robustness objectives. To avoid such degradation, packets sent during the fast start phase are accorded a high *drop* priority[2]. When faced with congestion, routers in the network preferentially drop fast start packets. This limits the adverse impact that an over-aggressive fast start attempt could have on other connections.

The heavy packet loss suffered by a connection during fast start could result in worse performance than standard slow start. To avoid this, we augment the sender with new loss recovery algorithms that quickly detect the failure of a fast start attempt and fall back to standard slow start.

Thus, TCP fast start has two components — the sender algorithm and the router algorithm. We discuss each in turn.

2. In the rest of this chapter, we use the term “low priority” to refer to “high *drop* priority”.

8.2.2 Sender Algorithm

Incorporating fast start at the end-hosts involves adding new algorithms to the TCP sender but none to the receiver. The sender algorithm has four components:

1. Initiation and termination of the fast start phase.
2. Initialization of TCP state variables.
3. Use of timers to clock out packets during the first RTT.
4. Quick detection of and recovery from a failed fast start attempt.

8.2.2.1 Initiation and Termination of Fast Start

The sender initiates TCP fast start whenever a connection resumes activity after being idle for longer than an RTO but shorter than a maximum threshold (set to 5 minutes by default). We defer the question of having an adaptive maximum threshold or aging the cached information gradually to future research.

Packets sent during the fast start phase are marked as low priority. These include all packets sent in the initial window after the connection resumes, except for the first one (which would have been sent in any case by standard slow start). Packets beyond the initial window, i.e., those sent after ack clocking sets in, do not belong to the fast start phase and hence are not marked.

Fast start could terminate prematurely if the sender detects multiple packet losses during fast start (Section 8.2.2.4).

8.2.2.2 Initialization of State Variables

When fast start is initiated, TCP state variables are initialized using their most recent values. The variables of interest are *cwnd*, *ssthresh*, *srtt* and *rttvar*.

Cwnd is set to the most recent congestion window size at which an entire window of data was transmitted successfully, i.e., without any packet loss. This window size depends on whether the connection was in the slow start phase or in the congestion avoidance phase just before the pause. In the former case, *cwnd* is set to half its old value. In the latter case, it is set to its old value minus one segment.

The cached values of *ssthresh* (which is a “safe” estimate of the size of the network data pipe), and *srtt* and *rttvar* (which together determine the retransmission timeout value) are left unchanged[3].

8.2.2.3 Clocking Out Data During Fast Start

The potentially large congestion window at the time fast start is initiated can result in a large burst, which is clearly undesirable. Since it will take at least one RTT for ack clocking to set in, we need an interim method of clocking out packets during fast start.

Our solution is to have the sender use a fine-grained timer to clock out data until ack clocking sets in. The sender has a parameter, *maxburst*, that can be configured to limit the size of any burst it sends out. The sender spaces apart such *maxburst*-sized bursts in time by *maxburst*srtt/cwnd*. This ensures that the bursts are spaced apart uniformly over an RTT, which is ideally how long the fast start phase should last. We set *maxburst* to 4 segments in our experiments[4].

There is evidence to suggest that the overhead of software timers is not likely to be significant in modern computer systems with fast processors. For instance, [29] reports an overhead on the order of a few microseconds. Furthermore, the timer overhead is unlikely to be a significant addition to the cost of taking interrupts and processing acks that goes with ack clocking. Finally, timers are required only during the fast start phase, which is about one RTT in duration.

8.2.2.4 Quick Detection/Recovery From Failed Fast Start Attempt

The information cached by the sender could be stale. A particularly dangerous situation is when several new connections become active between the time when the network parameters are cached and when the sender tries to reuse them. In this case, a fast start attempt could be too aggressive. The priority dropping mechanism shields other (non-fast start) connections in such a situation. However, the connection that is attempting fast start could itself suffer heavy packet loss, yielding worse performance than if it had used standard slow start. We augment the TCP loss recovery procedure as follows (to avoid this undesirable situation):

3. It is possible to assign a larger weight to fresh RTT samples during fast start to enable quick adaptation in case of a significant change. However, we have not experimented with this.
4. Note that even standard TCP with delayed acks can burst out 3 segments in a row during slow start.

- ***Fine-grained reset timer:*** The TCP timestamp option is used to obtain accurate RTT samples, from which accurate estimates of *srtt* and *rttvar* are computed. These are used to set a fine-grained timer[5] to detect the loss of fast start packets. We believe that the additional overhead of fine-grained timers is acceptable because it is only incurred for fast start packets. Such packets have a higher likelihood of being dropped because of their low priority.

- ***Slow start without penalty:*** If the fine-grained reset timer expires and the earliest unacknowledged packet is a fast start packet other than the first one (which, as discussed in Section 8.2.2.1, does *not* have the drop priority bit set), the TCP sender cuts *cwnd* down to one segment and initiates slow start. However, it does not cut down *ssthresh*, back off RTO, or discard selective ack (SACK [66]) information (if available). The idea is to make the behavior as close as possible to the case where the connection just did standard slow start in the first place (and did not attempt fast start at all).

  Unlike the loss of a regular packet, the loss of a fast start packet is an indication that the (optimistic) assumption made about network conditions is invalid rather than an indication of congestion.The low priority assigned to the original fast start packets is of critical importance. This makes it okay for the sender to fall back to the default behavior (i.e., slow start), but not incur the penalty associated with a standard retransmission timeout. If the network is truly congested, the connection will discover it during the ensuing slow start.

- ***Recovery from multiple losses:*** When there is packet loss during fast start, the sender (as usual) attempts to recover from it without resorting to a timeout. However, in some cases such a loss recovery procedure could be slow and could lead to significantly worse performance than if fast start had not been used at all. For instance, TCP NewReno recovers at the rate of one loss per RTT, which is equivalent to operating with a window size of one segment until all the packets lost in the burst have been retransmitted (this is assuming there is no more new data to send, as is often the case with short Web transfers).

---

5. The timeout value is still computed as $srtt + 4*rttvar$, with fine-grained estimates of *srtt* and *rttvar* instead of coarse-grained ones.

We use the following heuristic to limit such performance degradation. If selective ack information (SACK) is available, the sender uses it to recover from multiple losses within one RTT. If not and if the sender receives a partial new ack (indicating the loss of multiple packets [45]) during fast recovery, it cuts *cwnd* down to one segment and initiates slow start right away, without waiting for a timeout (and without imposing any other penalty, as discussed earlier).

- ***Capitalizing on successful transmission during a failed fast start attempt:*** Although a fast start attempt may have failed because of multiple packet drops, some packets may actually have been delivered successfully. Therefore, the sender uses cumulative ack and selective ack information, both from the fast start phase and from the subsequent slow start phase, to avoid retransmitting such packets. In contrast, the default TCP behavior is to discard selective ack information obtained prior to a timeout [66].

As our results in Section 8.4.2.2 show, these techniques are of critical importance in limiting performance degradation when the cached information is stale.

Next, we briefly discussed the router mechanism needed to support fast start.

8.2.3 Router Mechanism

The router mechanism we use is simple, packet-level priority dropping. It distinguishes between packets based on a one-bit priority field. When its buffer fills up and it needs to drop a packet, it first picks a low-priority packet, if available. Since fast start packets are assigned a low priority, this algorithm ensures that an over-aggressive fast start does not cause (non-fast start) packets of other connections to be dropped[6]. This mechanism does not require routers to do per-connection processing or maintain per-connection state. The drop priority mechanism applies equally well to drop-tail and RED routers [33].

We retain standard FIFO scheduling for all packets. This keeps the operation of the router simple when there is no need to drop a packet (presumably, the common case). Of course, FIFO scheduling means that fast start packets could increase the queuing delay for other connections, but typically queuing delay has a minor impact on TCP performance compared to packet drops.

6. As we discuss in Section 8.5, such a mechanism when used in conjunction with FIFO scheduling does *not* fully shield the high-priority (non-fast start) packets.

The notion of packet drop priority is also of interest in other contexts. The cell loss priority mechanism (CLP) in ATM provides the same functionality at the granularity of cells. There is a growing interest in using the IP type-of-service (TOS) mechanism to support differentiated services (including packet drop priority) in the Internet [25]. This effort is being supported by the major router vendors. For our purposes it is sufficient if drop priority is supported just by the bottleneck routers.

In spite of this, the lack of widespread support for the drop priority mechanism in today's Internet would impede the deployment of fast start on a large scale. However, as we discuss in Section 8.5, even a partial deployment could yield significant performance benefits.

Next, we briefly describe our implementation of fast start in BSD/OS.

8.3 Implementation in BSD/OS

We have implemented fast start both in the *ns* simulator and in the BSD/OS TCP/IP stack. We describe the latter here.

The fast start phase is initiated by the *tcp_output()* function. When it is called to send out data, *tcp_output()* checks to see if the connection has been idle for longer than the retransmit timeout period (RTO). If it has and if fast start has been enabled for the connection, a new value of the congestion window (*cwnd*) is computed. This is either half the old *cwnd* (if old *cwnd* < *ssthresh*) or *cwnd* minus *maxseg* (if old *cwnd* > *ssthresh*). Here *maxseg* is the maximum segment size for the connection. Thus, *cwnd* is set to the most recent congestion window size at which an entire window of data was transmitted successfully. The values of *ssthresh*, *srtt*, and *rttvar* are left unchanged.

The quantities *fs_startseq* (= *snd_nxt* + *maxseg*) and *fs_endseq* (= *snd_nxt* + *cwnd*) are computed to mark the start and end sequence numbers of the fast start phase. *Snd_nxt* is the next packet in sequence to be sent, and *maxseg* accounts for the segment that the TCP sender would have sent in any case when invoking slow start after an idle period. The current value of *ssthresh* is also saved for possible use later should the fast start attempt fail.

All segments that contain data bytes in the range (*fs_startseq, fs_endseq*) have the fast start bit set[7]. This identifies them to routers as low priority packets. However, the use of this bit is not standardized, so only routers with our *linkemu* module (Appendix A) installed will recognize it.

To avoid bursting out data during the fast start phase, *tcp_output()* breaks up potentially large bursts into smaller ones, each at most *maxburst* (by default, 4) segments in size, and spaces them apart according to the rate of the connection (*cwnd/srtt_exact*). The quantity *srtt_exact* is a smoothed and fine-grained RTT estimate computed using the TCP timestamp option with timestamps of millisecond granularity. The default timestamp resolution of 500 ms is too coarse-grained to be useful here.

Having computed the rate, *tcp_output()* computes the length of time (*delta*) over which *maxburst* segments should be spread out. It also records the times at which the last *maxburst* packets were transmitted. Using these, it ensures that no more than *maxburst* data packets are sent out in a time interval of length *delta*. If more packets are waiting to be sent, a software timer is used to schedule *tcp_output()* at a later time. Note that the software timer needs to be invoked only when bursts larger than *maxburst* segments are generated. This is likely to happen only during fast start phase (which lasts for about one RTT).

While in the fast start phase, the TCP sender enables the fast start *reset timer* instead of the standard retransmit timer. The timeout value for the reset timer is computed as *srtt_exact* + *4*rttvar_exact*, where the "exact" quantities are computed as discussed above. Rather than using a separate software timer, the firing of the reset timer is tied to the fast TCP timer (200 ms period, typically). This is in contrast to the slow TCP timer (500 ms period, typically) that the retransmit timer is tied to.

If the reset timer expires while fast start packets are still outstanding[8], the sender immediately *resets* its state to what it would have been had fast start not been attempted. *Cwnd* is reset to one segment, *ssthresh* is restored using the value saved at the time fast start was initiated, and slow

7. It is possible to use a generic "drop priority" bit in the IP header (such as one of the IP type-of-service bits), but we decided to pick an *unused* bit just to be safe. We could not find any in the IP header, so we picked one in the TCP header.
8. Note that if even the first packet sent without the fast start bit set has not yet been acknowledged, the sender treats it as a standard retransmission timeout.

start is initiated. However, none of the other penalties of a retransmit timeout (such as timer backoff) is imposed.

As explained in Section 8.2.2.4, fast start is also aborted if the sender detects the loss of multiple packets in a window during fast start phase. This detection is done in *tcp_input()* by looking for a partial new ack during TCP fast recovery, just as in TCP NewReno [45].

We now present detailed performance results.

8.4 Performance Results

We first present performance results based on our BSD/OS implementation of TCP fast start. A limitation of these experiments is the lack of priority dropping in the Internet routers today. For this reason, we also present detailed performance results based on simulation.

The protocol configurations we evaluated include:

1. *tcp*: TCP NewReno with standard slow start.
2. *fs*: TCP fast start without priority dropping.
3. *fs-pdrop*: TCP fast start with priority dropping.
4. *fs-pdrop-noflr*: TCP fast start, but without any of the fast loss recovery techniques from Section 8.2.2.4.

Each of these was used in conjunction with a scheme such as P-HTTP or TCP session.

8.4.1 BSD/OS Measurements

In this experiment, we have a client download Web pages of different sizes from a server. The client is the one we used for our experiments with P-HTTP (Section 5.5). The server is Apache version 1.3b5 using either the *tcp* or *fs* configurations. In each run of the experiment, the client first downloads a 100 KB Web page. This enables the server to obtain and cache information about the network path between it and the client. Then after an idle period of several seconds (much longer than the RTO), the client downloads another page from the server. In the *tcp* configuration, the sec-

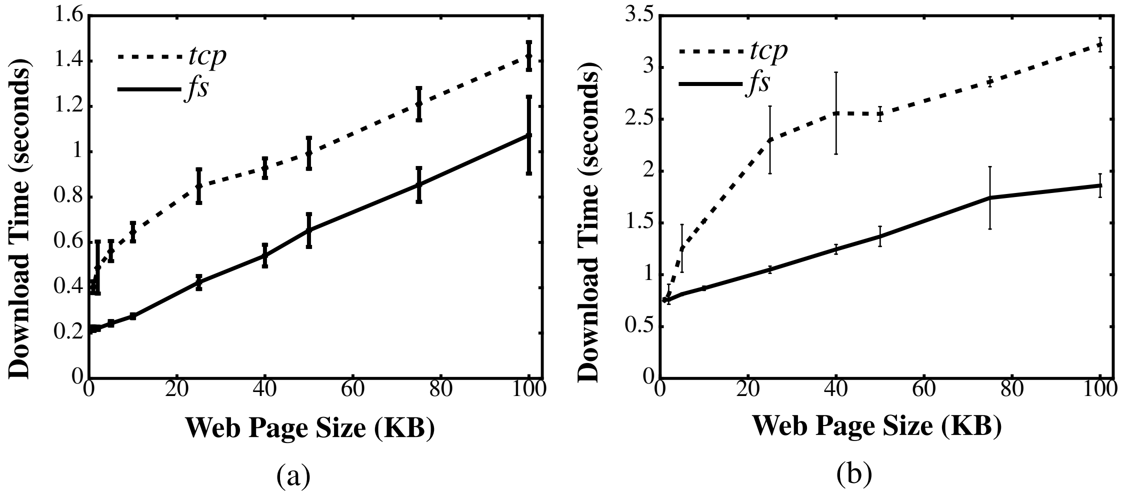


Figure 8.1 Download time versus Web page size for the (a) terrestrial WAN network, and (b) DirecPC network.

ond download would start off in slow start, but in the *fs* configuration it invokes fast start based on the information cached during the first download.

We conducted the experiment over two network paths — a terrestrial wide-area network and one including the DirecPC satellite network. These are the same as the ones used in our P-HTTP experiments (Section 5.5).

Figure 8.1 shows the download time as a function of the Web page size. Since the HTML file is only 1 KB in size, the Web page size is determined primarily by the sum of the sizes of the individual inline images. The mean and a 95% confidence interval on the mean are computed over 15 runs of the experiment.

We observe that *fs* performs significantly better than *tcp* in both the WAN and the DirecPC networks. The performance improvement, both in relative and absolute terms, tends to be larger in the latter than in the former. This is because the RTT of the DirecPC network is much larger (400 ms versus 70 ms). This means that each RTT saved with *fs* is more valuable in the DirecPC case. Also, a larger RTT implies a larger bandwidth-delay product, so *fs* can potentially cache and reuse a larger initial window size.

We also observe that the benefit of *fs*, in relative terms, tends to diminish as the Web page gets larger. For instance, in the WAN case, it drops from over 50% (2X) for pages under 10 KB in size

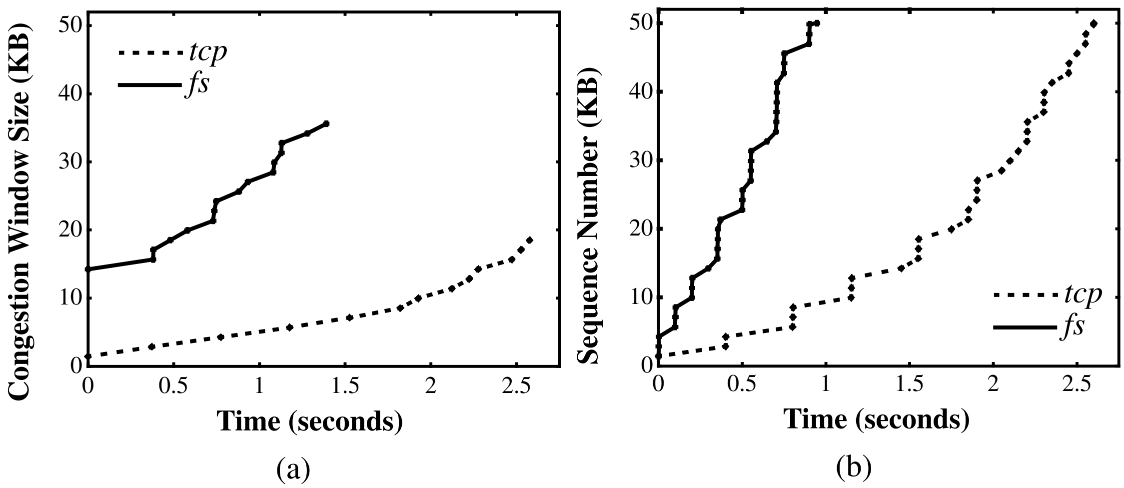


Figure 8.2 (a) The congestion window plot, and (b) the sequence number plot for the second of two 50 KB Web page downloads.

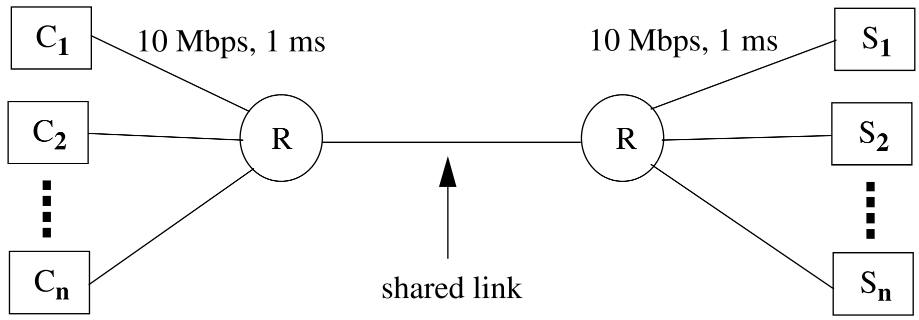
to about 25% for 100 KB pages. This is because the longer a transfer, the less important the slow start phase at start up time. However, Web page transfers tend to be short in length (for instance, [65] reports an average of around 30 KB), so fast start is quite valuable in practice.

Figure 8.2 shows the operation of fast start in greater detail for the second of two 50 KB downloads. Figure 8.2(a) depicts the growth of the congestion window over time. We observe that *fs* starts off with an initial congestion window size of about 15 KB, derived from the value cached during the first download. This translates into a much quicker completion time (about 2.75X) for the second download with *fs* compared to *tcp*, as shown in Figure 8.2(b).

A shortcoming of experiments over the Internet is the lack of support for priority dropping in the routers. To address this limitation, we now turn to simulation experiments.

8.4.2 Simulation Results

We present performance results using our implementation of TCP fast start in the *ns* network simulator. The topology used for the simulation experiments is shown in Figure 8.3. Depending on the experiment, one or more of the client-server pairs initiate Web page downloads across the shared link[9]. For each such pair, two downloads are done, with an intervening idle period. The first download, 100 KB in size, gives the server the opportunity to cache information about the network path to the client. We measure and report the performance of the second download.



Simulation parameters

Maximum TCP window size: 100 KB
Router buffer size: 20 packets

Web download parameters

Request size: 300 bytes
HTML size: 1 KB
Total size of inline images: 40 KB

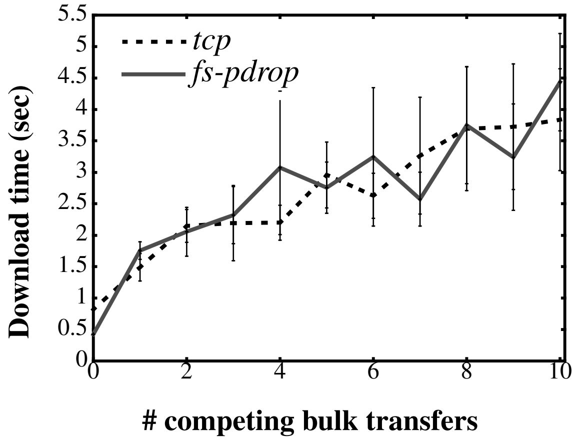
Figure 8.3 The network topology and parameter settings for the simulation experiments.

Again depending on the experiment, cross-traffic in the form of TCP bulk transfers may be introduced between other servers and clients contending for the shared link. We consider several settings for the bandwidth and the delay of for the shared link:

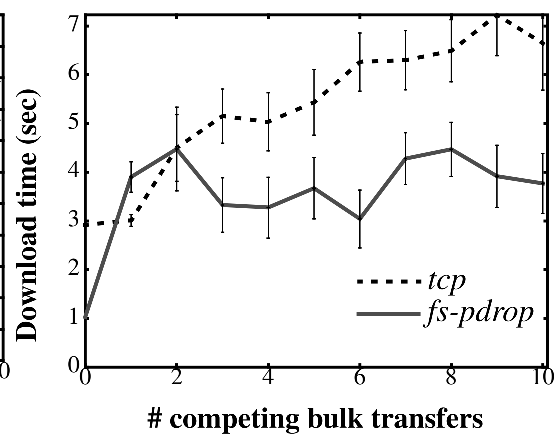
1. 28.8 Kbps, 50 ms (i.e., an RTT of 100 ms)
2. 1.5 Mbps, 50 ms
3. 1.5 Mbps, 200 ms (i.e., an RTT of 400 ms)
4. 45 Mbps, 200 ms

The protocol combinations we consider are *tcp*, *fs*, *fs-pdrop*, and *fs-pdrop-noflr*. Note that the implementation of these in *ns* does not use the three-way handshake procedure at connection setup time. In effect, each is using T/TCP-style accelerated open that avoids consuming an RTT during connection setup.

9. We prefer to use the term "*shared* link" rather than "*bottleneck* link" because there are situations where this link is fast enough that it is not a bottleneck. However, we do use the term "bottleneck link" in a few instances where it is appropriate.



(a)



(b)

Figure 8.4 The completion time for the second of two Web downloads under conditions of constant load. The bottleneck link bandwidth is 1.5 Mbps and the delay is (a) 50 ms, and (b) 200 ms.

First, we consider the performance of a single Web download in the presence of cross-traffic in the form of bulk transfers.

8.4.2.1 Single Web Download with Constant Load

In the constant-load experiment, the bulk transfers constituting the cross-traffic are left running throughout the experiment to provide a roughly constant background load. Under such conditions, it is likely that cached values of TCP parameters remain valid after an idle period. By varying the number of bulk transfer connections from one experiment to the next, we can evaluate fast start under different levels of cross-traffic load.

After the cross-traffic has reached its steady-state level, a single Web download, 100 KB in length[10], is initiated between client C_1 and server S_1. After a long pause, a second download, 40 KB in length, is initiated between the same pair of hosts. This mimics a Web browsing session in which a user downloads multiple pages from the same server, with intervening idle periods. We report the time taken for the second download (which is in a position to exploit cached information, if fast start is used).

10.We pick a larger size, 100 KB, for the first download to emulate the combined effect of multiple downloads that may have happened in the past.

Figure 8.4(a) and Figure 8.4(b) show the results for a 1.5 Mbps bottleneck link with two different settings of delay — 50 ms and 200 ms. We make several observations. In the 50 ms case, the performance of both *tcp* and *fs-pdrop* is similar. Furthermore, *fs* (not shown) performs only about 15% better than *fs-pdrop*. The reason for these is that the bandwidth-delay product of the network is rather small (1.5 Mbps*100 ms = 18.75 KB). And since this is shared among multiple connections, the share of each is even smaller. Under these conditions, the Web connection does not build up a large window at the time of the first download. Consequently, the initial window size during the subsequent fast start is not much larger than the default (i.e., one segment), so fast start only has a limited impact. The relatively small performance difference between *fs* and *fs-pdrop* arises because in the latter case, priority dropping makes the Web connection more prone to losing packets.

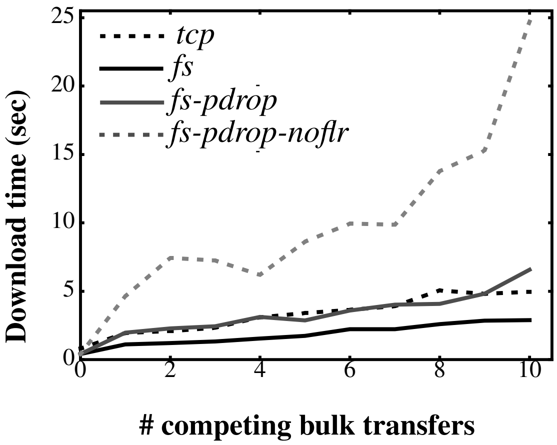
When the delay is 200 ms, the bandwidth-delay product is four times as large, about 75 KB. Consequently, fast start has a far more significant performance impact. As Figure 8.4(b) shows, *fs-pdrop* improves download time by 30-50% (1.43-2X) compared to *tcp*. Also, *fs* (not shown) performs about 5% better than *fs-pdrop*.

When the bandwidth is set to 45 Mbps and delay to 200 ms, the bandwidth-delay product is very large, about 2.25 MB. This makes fast start very beneficial. Regardless of the level of cross-traffic (in the range that we considered), the download time with *tcp* is about 2.85 seconds and that with *fs* or *fs-pdrop* is about 0.96 seconds, a 3X improvement.

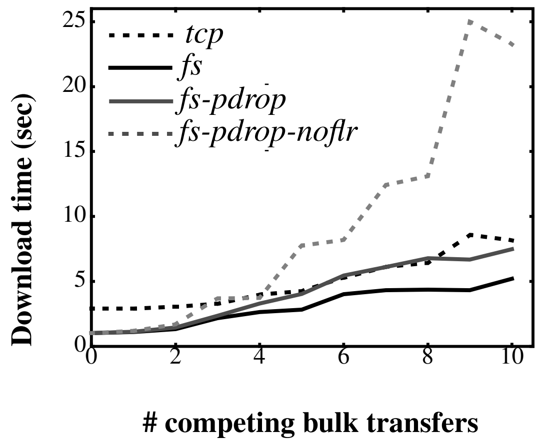
In summary, we make two observations. First, the benefits of fast start are greater when the bandwidth-delay product is larger. Second, under conditions of constant load, there is not much of a difference in performance between *fs* and *fs-pdrop.* This is because the cached information used by fast start remains fairly accurate, so fast start causes few packet drops. As a result, the use of priority dropping does not make much of a difference in this case.

8.4.2.2 Single Web Download with Changed Load

This experiment is similar to the previous one except for one significant difference — the cross-traffic is absent at the time of the first Web download, but is introduced into the network during the idle period between the two downloads. Therefore, the network conditions will have changed for



(a)



(b)

Figure 8.5 The completion time for the second of two Web downloads under conditions of changed load. The bottleneck link bandwidth is 1.5 Mbps and the delay is (a) 50 ms, and (b) 200 ms.

the worse between when the Web connection caches various network parameters (at the time of the first download) and when it tries to reuse them (at the time of the second download). Figure 8.5(a) and Figure 8.5(b) show the results for a 1.5 Mbps bottleneck link with a delay of 50 ms and 200 ms, respectively. For the sake of clarity, we do not show the error bars.

In the 50 ms case, the performance of *fs-pdrop* is similar to that of *tcp*. When fast start is initiated using stale information, it causes the already loaded network to drop several packets. In spite of this, *fs-pdrop* suffers little performance degradation because the augmented loss recovery procedure (Section 8.2.2.4) quickly detects and recovers from the failed fast start attempt. The value of the augmented procedure is further underscored by the performance of *fs-pdrop-noflr*, which only uses the standard TCP loss recovery procedure. The heavy loss of packets results in a download time that is up to 5X worse than that in the case of *tcp*.

When *fs-pdrop* is used with a coarse-grained reset timer instead of one that is fine-grained, the performance degradation relative to *tcp* is only 1.35X or less (not shown in the figure). Much of the benefit of the augmented loss recovery procedure is due to the three other techniques discussed in Section 8.2.2.4 rather than fine-grained timer.

The performance of *fs* is 40-50% (1.67-2X) better than that of *fs-pdrop* and *tcp*. In the absence of priority dropping, packet drops tend to be spread across the Web connection and the bulk transfers.

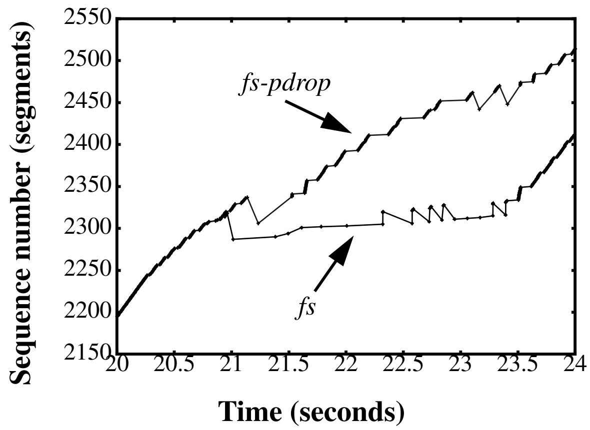


Figure 8.6 A section of the sequence number trace of a bulk transfer connection in the immediate aftermath of simultaneous fast start attempts by ten Web connections. The bottleneck link bandwidth is 1.5 Mbps and the delay is 50 ms.

The Web connection, whose fast start attempt is primarily responsible for the packet drops, does not suffer much. On the other hand, with *fs-pdrop* the Web connection bears the brunt of the packet drops.

Fast start without priority dropping (*fs*) achieves better performance at the cost of the ongoing bulk transfer connections. This is clear from Figure 8.6, which shows the sequence number trace of a bulk transfer connection in the immediate aftermath of a fast start attempt by ten Web connections. Due to the absence of priority dropping, *fs* causes the bulk transfer to lose several packets, and consequently, stall for several seconds. On the other hand, *fs-pdrop* has a much smaller impact on the bulk transfer. Its impact is primarily in the form of increased queuing delay.

When the bottleneck link delay is 200 ms, the performance trends are similar to the 50 ms case. However, the larger bandwidth-delay product places the network in a better position to absorb the burst of packets due to fast start. This leads to fewer packet losses. Therefore, the impact of both the augmented loss recovery procedure and that of priority dropping is smaller than in the 50 ms case.

In summary, fast start coupled with priority dropping is resilient under adverse conditions, i.e., when fast start is attempted based on stale information. The augmented loss recovery procedure

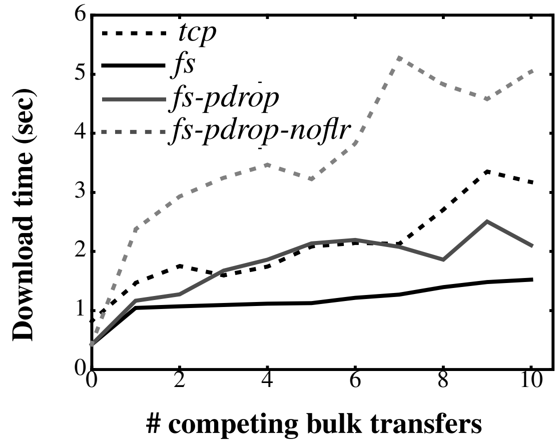


Figure 8.7 The completion time for the second of two Web downloads under conditions of changed load. The bottleneck link bandwidth is 1.5 Mbps and the delay is 50 ms. The bottleneck router uses RED buffer management with a buffer size of 20 packets, a minimum marking threshold of 3 packets, a maximum marking threshold of 12 packets, and a weight of 0.02 for computing an exponentially smoothed estimate of the average queue length.

limits performance degradation for the connection attempting fast start while priority dropping protects other traffic.

8.4.2.3 Impact of RED Buffer Management

So far, we have only considered drop-tail buffer management. We now briefly consider the impact of Random Early Detection[11] (RED) buffer management [33]. RED enables high link utilization while at the same time maintaining free buffer space to absorb bursts. So there are fewer packet losses due to fast start, even when it uses stale information. Consequently, the download time improves compared to the drop-tail case. The across-the-board improvement in performance is evident when one compares Figure 8.7 to Figure 8.5(a).

Packets sent during the fast start phase are just as likely to get marked as other packets. Also, if a fast start packet gets marked but the corresponding (short) transfer terminates before the sender invokes congestion control action, the sender preserves this notification by caching the new

11.We consider the variant of RED that marks packets with an ECN notification rather than drop them.

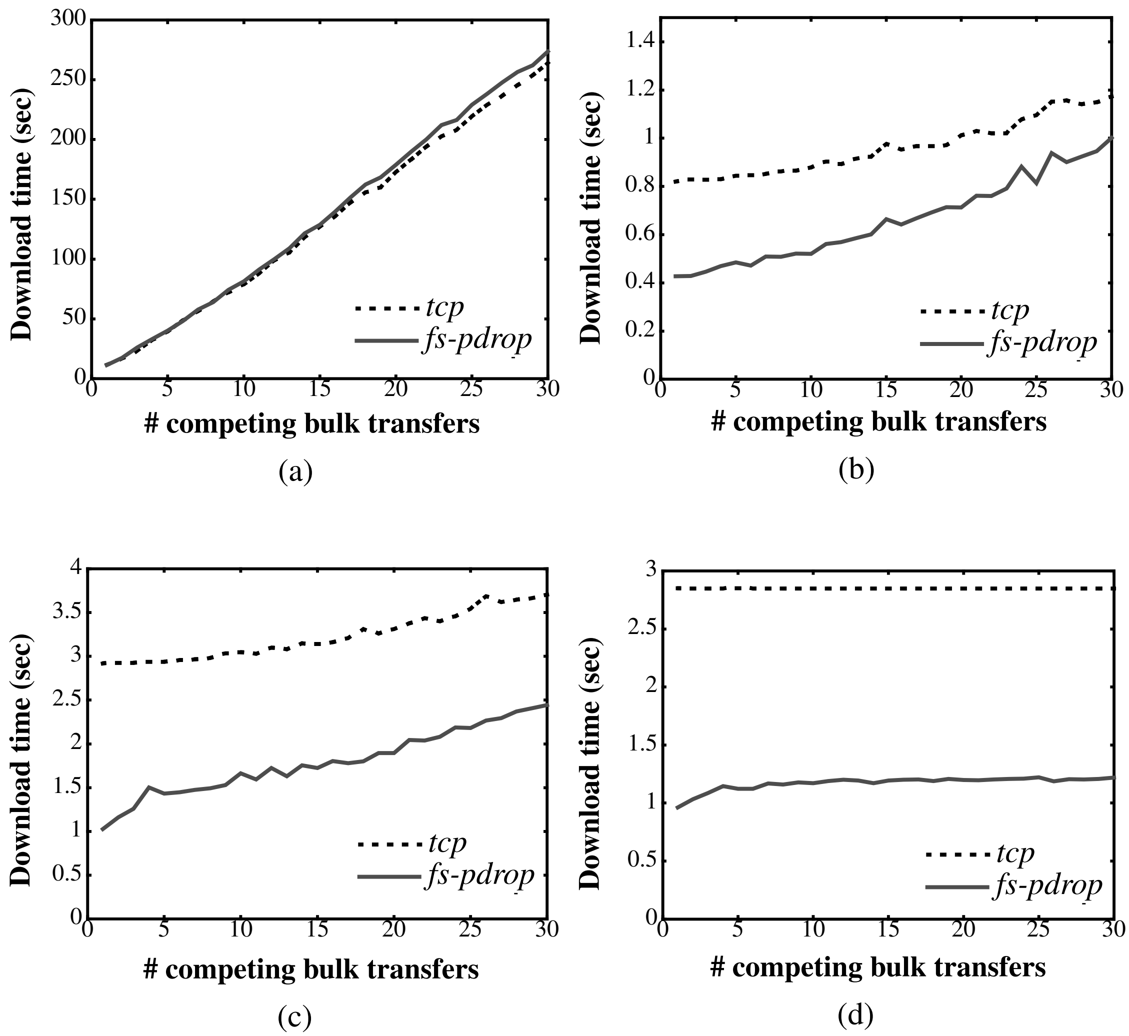


Figure 8.8 The average download time for competing Web downloads under conditions of constant load. The bandwidth and delay of the shared link are (a) 28.8 Kbps/50 ms, (b) 1.5 Mbps/50 ms, (c) 1.5 Mbps/200 ms, and (d) 45 Mbps/200 ms.

(halved) congestion window. Then the next fast start attempt will be less aggressive. Thus, the use of persistent state makes an explicit congestion notification scheme such as RED effective even when the individual transfers are short in length.

8.4.2.4 Competing Web Downloads with Constant Load

We now consider performance in the presence of competing Web downloads, but without any bulk transfer cross-traffic. Each run of the experiment involves between 1 and 30 client-server pairs initiating a download that traverses the shared link. The initiation times are spread uniformly at random over a 10-second interval. Then, after a long idle period, each client-server pair does a second download, again spread over a 10-second interval. So this experiment is similar to that discussed in Section 8.4.2.1 in that the overall level of load in the network (i.e., the number of concurrent downloads) remains roughly constant between the time of the first download and that of the second.

In Figure 8.8, we report the time taken for the second download (averaged across clients) for various configurations of the shared link. We make two observations. First, the performance benefit of fast start is larger when the bandwidth-delay product is larger. In particular, there is little performance gain when the bandwidth is as low as 28.8 Kbps. As explained in Section 8.4.2.1, a larger bandwidth-delay product provides fast start with the opportunity to cache and reuse a larger initial window size.

Second, the performance benefit of fast start, in relative terms, tends to decrease as the number of concurrent downloads increases. For instance, with a 1.5 Mbps/50 ms bottleneck link, the reduction in download time of *fs-pdrop* relative to *tcp* decreases from 50% to 15% (2X to 1.18X) as the number of concurrent downloads increases from 1 to 30. Likewise, the performance benefit drops from 65% to 35% (2.85X to 1.5X) in the case of a 1.5 Mbps/200 ms bottleneck link. The reason for these trends is that an increased level of load diminishes the opportunity for an individual connection to cache and reuse a large window. It is only in the case of a 45 Mbps/200 ms link that this trend is not evident. The levels of load that we consider (i.e., up to 30 concurrent downloads) is not large enough to impact performance, given the very large bandwidth-delay product.

In summary, a significant improvement in performance is possible even when there are multiple connections that initiate fast start concurrently. As the load in the network increases, the performance benefit decreases gradually.

8.4.2.5 Competing Web Downloads with Changed Load

This experiment is similar to the previous one expect that the second download by each client is initiated within a 0.1-second interval (as against a 10-second interval for the first download).

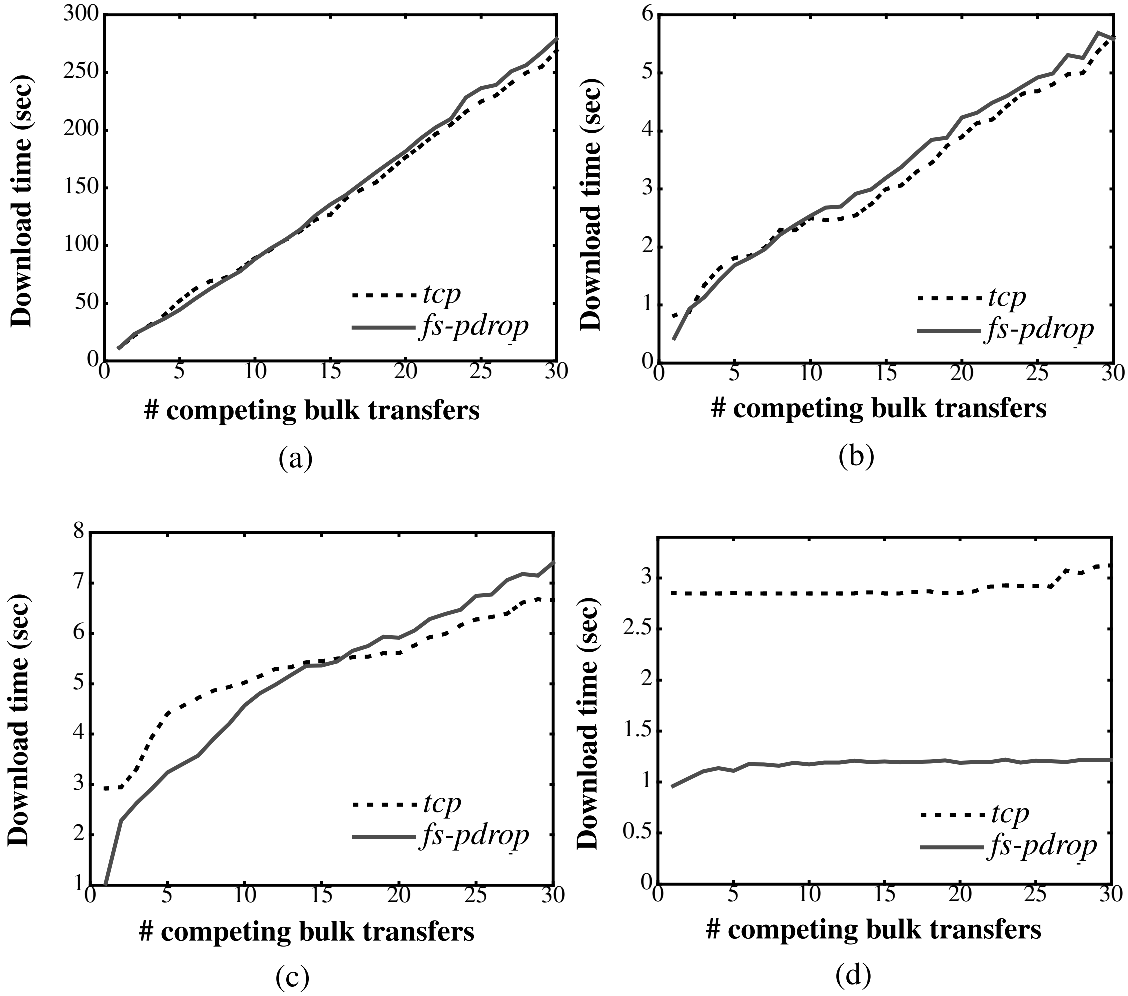


Figure 8.9 The average download time for competing Web downloads under conditions of changed load. The bandwidth and delay of the shared link are (a) 28.8 Kbps/50 ms, (b) 1.5 Mbps/50 ms, (c) 1.5 Mbps/200 ms, and (d) 45 Mbps/200 ms.

Therefore, as in the changed-load experiment (Section 8.4.2.2), a fast start attempt at the time of the second download would be based on stale information.

Such a change in load mimics the *flash crowd* phenomenon often observed in the Web, where a large number of users (clients) converge on a Web server almost simultaneously, often because of some extraordinary news event. In such a situation, information cached by the server during a (normal) period of relatively modest activity becomes stale.

Figure 8.9 shows the results for various configurations of the bottleneck link. Comparing this to Figure 8.8, we observe that the performance benefit of fast start disappears in most cases. This is because the use of stale information causes fast start to invariably fail at the time of the second download. The only exception is the 45 Mbps/200 ms case, where the large bandwidth-delay product prevents significant packet loss in spite of fast start being over-aggressive. Even in the cases where fast start fails, the augmented loss recovery procedure limits performance degradation of *fs-pdrop* to less than 10%.

In summary, this experiment again underscores the resilience of fast start in the face of adverse conditions, including those that mimic flash crowds.

8.4.2.6 Competing, Heterogeneous Web Downloads with Changed Load

In this experiment, we consider a mixture of connections, some that use fast start and others that do not. The experiment involves 20 client-server pairs doing downloads in two phases. The first phase is spread over a 10-second interval and the second over a 1-second interval. One half of the client-server pairs (i.e., 10 of them) use fast start while an equal number do not.

Protocol	Without fast start	With fast start
tcp	3.37 (0.20)	n/a
fs	4.09 (0.17)	3.32 (0.19)
fs-pdrop	3.38 (0.16)	3.20 (0.18)

Table 8.1 Completion time (in seconds) for downloads that use fast start and those that do not. A 95% confidence interval for the mean is reported in parentheses. Note that in the case of *tcp*, there are no downloads that use fast start.

Table 8.1 summarizes the results. We observe that both with *fs* and with *fs-pdrop*, the use of fast start results in little performance gain (under 5%) compared to *tcp*. This is as expected because the change in the network load renders the cached information stale, so fast start tends to fail. The more important point is that when fast start is used by a subset of the connections in conjunction with priority dropping (*fs-pdrop*), connections that do not use fast start experience little performance degradation. In contrast, when fast start is used without priority dropping (*fs*), the download time for the latter degrades by over 20% compared to the case where fast start is not used at all (*tcp*). This, again, underscores the importance of using priority dropping in conjunction with fast start.

8.4.3 Summary of Results

We now summarize our key results both simulation-based and implementation-based:

1. Fast start has the potential to cut down latency significantly (up to 3X in our experiments) under a variety of conditions. The performance gains tend to be largest when the bandwidth-delay product is large, the network load is low and when the Web page size is neither too small nor too large.

2. Even when the conditions are not favorable (e.g., high network load, or worse still, changing network state that renders the cached information stale), fast start at worst performs similar to standard TCP. The augmented loss recovery procedure is effective in quickly detecting and recovering from a failed fast start attempt.

3. The use of priority dropping in conjunction with fast start is critical in avoiding performance degradation for other connections in case of an over-aggressive fast start attempt based on stale information.

4. RED buffer management improves the performance of fast start.

Next, we turn to a discussion of some general issues pertaining to fast start.

8.5 Discussion

We discuss issues pertaining to the performance benefits of fast start, its robustness, and incremental deployment.

TCP fast start is complementary to the P-HTTP and TCP session techniques that we have developed in previous chapters. In fact, there is a synergy between those techniques and fast start. P-HTTP and TCP session make the congestion control parameters more stable and reliable compared to when there are lots of short, independent and competing connections. Therefore, fast start is in a better position to cache and successfully re-use this information.

Both Web clients and Web servers benefit from fast start. From the viewpoint of a client, fast start means faster Web page downloads. From the viewpoint of a server, it means increased capacity because resources (such as the user-level process/thread, socket buffers, etc.) associated with a particular download are freed sooner and can be used to serve other requests.

The robustness of TCP fast start comes from it use of priority dropping and its dependence on past information. If the network becomes congested, a fast start attempt is likely to fail. If the connection is only able to build up a small window during the subsequent slow start, its next fast start attempt will be much less aggressive. This self-regulating mechanism ensures that senders do not keep flooding the network with packets bound to get dropped. Of course, this assumes that hosts are cooperative, just as TCP congestion control does. To be safe, the network should have mechanisms to detect and penalize malicious behavior (e.g., [32]).

While priority dropping is a useful mechanism to exercise control on a time-scale finer than a round trip time, it may not, however, completely shield high-priority packets from losses due to low priority packets. This is because of a combination of FIFO queuing, which could delay the servicing of high-priority packets when there are low-priority packets ahead of it in the queue, and the policy of dropping packets only when the queue fills up. To address this problem, it is possible to use a more sophisticated (but still stateless) priority dropping scheme such as the one discussed in [3], which preemptively drops packets even before the queue fills up.

Support for priority dropping is not widespread in the Internet today. However, this situation is set to change with major router vendors including such mechanisms in their newer products. This is change is being driven by the Internet's *differentiated services* effort [25], which is standardizing priority dropping as one in a set of per-hop behaviors. The underlying philosophy is that packets that do not conform to a profile (*out-of-profile*) should be dropped preferentially with respect to packets that do conform (*in-profile*). Our use of drop priority in conjunction with fast start exactly matches this philosophy if we define the "profile" to be standard TCP slow start. Fast start packets are out-of-profile and therefore dropped preferentially.

In spite of this, we cannot depend on universal support for priority dropping at this time. A proxy-based approach provides a path for incremental deployment in certain situations, while still obtaining significant performance benefits. For instance, consider the DirecPC satellite network. If all client Web accesses are routed via a proxy at the earth station, much of the benefits of fast start can be obtained by having the proxy implement fast start and by deploying priority dropping in the network domain encompassing the proxy and the clients. Neither the Web servers nor the routers in the rest of the Internet need be modified. (And in any case, fast start does not require any modification to the clients.) An added advantage is that, with all communication routed via it, the proxy would tend to have better estimates of the bandwidth available to each client than any individual server would. In Chapter 9, we discuss another instance, namely an asymmetric-bandwidth access network, where such a partial deployment would be feasible and beneficial.

Finally, although the focus of this chapter has been the Web, TCP fast start could also be useful in other situations involving bursty data transfer. Examples include wide-area transactions, RPC, and sensor data collection.

8.6 Summary

In this chapter, we have presented a new technique, *TCP fast start*, to complement the P-HTTP and TCP session techniques that we developed earlier. The basic idea behind fast start is to use cached information (i.e., persistent state) to decrease the overhead of probing using TCP slow start. If fast start succeeds, it cuts down latency significantly, especially for short transfers. However, to avoid performance degradation in adverse circumstances, priority dropping and augmented loss recovery procedures are used in conjunction with fast start.

Thus far in this thesis, we have considered performance issues that arise due to bandwidth and delay limitations, in general. In the next chapter, we consider the performance issues that arise specifically due to asymmetry in network characteristics in a new class of access networks. Although it only form a small part of the end-to-end network path, an access network could have a significant impact on performance. As we shall see, TCP fast start could help improve performance significantly in such networks even though their inherent delay may not be very large.