

Addressing the Challenges of Web Data Transport

Venkata N. Padmanabhan¹

Microsoft Research
padmanab@microsoft.com

Randy H. Katz

University of California at Berkeley
randy@cs.berkeley.edu

Abstract

The rapid growth of the World Wide Web has resulted in several new challenges for data transport. These include the need to efficiently support: (1) short and bursty transfers, (2) multiple, concurrent, logically-separate transfers, and (3) transfers over asymmetric-bandwidth access networks.

In this paper, we discuss two techniques — *TCP Sessions* and *TCP fast start* — that address these challenges. A TCP session integrates multiple TCP connections between a pair of hosts, such as a Web server and a client. This is accomplished by decoupling TCP's ordered byte-stream *service* abstraction from the underlying congestion control and loss recovery *algorithms*. The service is provided on a per-connection basis whereas the algorithms operate on a session-wide basis. TCP fast start speeds up short and bursty transfers by reusing congestion information learned in the recent past to avoid the slow-start penalty. But such information may be stale, consequently fast start may be inappropriately aggressive. To shield other network traffic from the consequent ill-effects (such as increased packet losses), packets sent during the fast start phase are assigned a lower drop priority compared to other packets.

When used together, these techniques enable applications to launch as many TCP connections as logically-separate data streams they need, without adversely affecting either their own performance or global performance. In fact, performance improves significantly in many situations, including in asymmetric-bandwidth networks. We quantify these benefits using simulations of bursty Web-like traffic. We describe implementations of both TCP fast start and TCP sessions in the BSD/OS 3.0 kernel with changes confined to the sender. We also discuss how TCP sessions enables new functionality, specifically allowing applications to explicitly control the allocation of bandwidth across connections within a session.

1. Introduction

The rapid growth of the World Wide Web has resulted in several new challenges for data transport. Since Web browsing is an interactive application, it is highly desirable to have low latency. At the same time, it is important to ensure that an attempt to cut down latency does not adversely impact the robustness of the network.

The characteristics of Web traffic and those of emerging network access technologies pose several challenges to achieving these goals:

1. The data transfer between a Web server and a client tends to be bursty. Data transfer happens in response to requests initiated by a human user. Successive requests are separated by idle periods corresponding to user think times. Each burst of data is often relatively short in length [20], but not so short that congestion control can be ignored. Initiating slow start [18] for each burst consumes several networks RTTs and results in increased latency.
2. The downloading of a Web page by a client involves multiple, logically-independent data transfers, for example, one for each inlined image. Since TCP does not provide a way of demarcating data within a connection, applications have two choices: either use a separate TCP connection for each transfer or multiplex all the transfers onto a single TCP connection. As we shall show, the former approach increases congestion in the network. The latter introduces undesirable coupling between the logically-independent data transfers.
3. Web browsing results in a traffic pattern that is largely asymmetric. This has led to the popularity of asymmetric-bandwidth networks which are able to provide bandwidth in the desired direction in a cost-effective way. However, even if there is no congestion in the direction of data transfer, congestion in the opposite direction (for instance, due to bidirectional traffic) can lead to poor performance because of adverse interaction with TCP's ack clocking mechanism. Some of these asymmetric networks, such as satellite-based networks, also have long delays, which increases the latency of Web page downloads.

Much of data transport research, including TCP research, in the literature focuses on long bulk transfers, where the steady state behavior dominates and throughput is the metric of interest. This does not address the case of Web transfers where bursts are often too short to allow steady state behavior to set in and latency is the metric of interest.

In this paper, we discuss two techniques that address the needs of short, bursty connections and multiple, concurrent connections. The first technique, which we call *TCP fast start*, enables TCP connections to avoid the penalty of slow start when restarting after an idle period. This is done by reusing the values of the congestion window and RTT from the recent past to smoothly inject packets into the network

1. This paper is based on Padmanabhan's PhD dissertation research [XXX] done at UC Berkeley.

until ack clocking sets in. This can result in a savings of several RTTs which slow start would entail. The benefit is greatest when the RTT is large, either because of an inherently large delay (as in satellite-based networks) or because of bidirectional traffic in an asymmetric bandwidth network. However, it is possible that the cached information from the recent past is stale and so the fast start attempt is over-aggressive, leading to packet loss. To address this possibility, fast start includes two safeguards: (a) fast start packets are assigned a higher drop priority² than other packets, so any congestion they cause does not impact other traffic much, and (b) during the fast start phase, a connection uses special heuristics to quickly detect packet loss and fall back to standard slow start, if needed.

The second idea, which we call *TCP session*, enables the aggregation of the set of concurrent connections between a pair of hosts. We separate TCP functionality into two parts: (1) providing an ordered byte-stream abstraction, and (2) congestion control and loss recovery. The former is done on a per-connection basis just as in standard TCP. But the latter is integrated across the set of concurrent TCP connections in the TCP session. We have presented the basic algorithm for integrated congestion control and loss recovery in a previous paper [2]. In this paper, we analyze the performance benefits in detail for bursty Web-like traffic. We also discuss the issues of scheduling connections within a TCP session and using sessions in conjunction with TCP fast start, and describe our implementation in BSD/OS.

When used together, enable applications to launch as many TCP connections as logically-separate data streams they need, without adversely affecting either their own performance or global performance. In fact, there is significant improvement in performance in many cases. TCP fast start cuts down latency for short bursts by up to 50-65% (a factor of 2-3) compared to standard slow start. TCP session reduces latency by 60-68% compared to a set of concurrent connections that operate independently of each other. It also cuts down the packet loss rate significantly. Finally, the end host implementation is confined to the sender side (Web servers), so the vast number of client hosts can be left untouched.

The rest of this paper is organized as follows. In Section 2, we survey related work. In Section 3, we discuss the details of TCP fast start. We present detailed simulation results in Section 4. In Section 5, we show how TCP fast start helps improve performance in asymmetric bandwidth networks. In Section 6, we discuss TCP sessions and present simulation results. The implementations of TCP fast start and TCP session in BSD/OS 3.0 are detailed in Section 7. In Section 8, we discuss some issues pertaining to TCP fast start and TCP sessions. Finally, we present our conclusions in Section 9 and on-going work in Section 10.

2. In the remainder of the paper, we use the term “low priority” to mean “high drop priority”.

2. Related Work

Modern TCP implementations are largely based on the algorithms presented in [18]. The key algorithm is *slow start*, which enables a TCP connection to discover the available network bandwidth by slowly growing its window from an initial window size of 1 segment. This procedure, which is performed both when a connection starts up and when it resumes activity after an idle period [19], works well for long transfers. For short transfers, however, the RTTs consumed by the slow start procedure can be a significant cost.

One proposal to alleviate this problem is to increase the initial window size for slow start to 2-4 segments (*4K-slow start* [12]). While this will help cut down the cost of slow start to some extent, the 2-4 segment window size could still be inadequate in situations where the bandwidth-delay product is much larger (e.g., satellite-based networks).

Transaction TCP (T/TCP [4]) is an adaptation of TCP for transaction-oriented applications. In T/TCP, hosts cache sequence number information from connections in the recent past. This enables the *TCP accelerated open* procedure that saves an RTT by eliminating the 3-way handshake that precedes actual data transfer in standard TCP. T/TCP also details the algorithm for caching RTT estimates. However, the issue of caching and reusing the congestion window size is not addressed. The latter is the main focus of TCP fast start, so we view our work as complementary to T/TCP.

We recently became aware of a scheme called *rate-based pacing* [26] (based on TCP Vegas [5]) that uses Vegas’s estimate of the connection’s rate to inject packets into the network until ack clocking set is. However, a drawback of such an open-loop scheme is that it can potentially aggravate congestion in the network and cause packet losses for other traffic in the network, especially if the old estimate of rate is stale. While TCP fast start is also an open-loop scheme, it incorporates priority drops to minimize the chances of aggravating congestion. TCP fast start also includes special techniques (in the context of TCP NewReno[16]³) to quickly detect and recover from a failed fast start attempt. Our experimental results demonstrate the usefulness of these techniques (Section 4).

Several researchers have analyzed wide-area network performance and concluded that the available bandwidth is often quite stable over short periods of time [23, 3]. This suggests that in the common case TCP fast start would indeed help improve performance.

TCP control block interdependence [25] is a proposal to use shared state to *initialize* congestion control parameters for concurrent TCP connections. While this has some similarities to TCP sessions, there are significant differences. First, a TCP session manages the progress of its constituent con-

3. TCP NewReno is a variant of TCP Reno that uses partial new ack information to recover from multiple packet losses in a window. The rate of loss recovery is one per RTT.

nections throughout their duration rather than just at the time of initialization. This permits explicit scheduling of connections within a session, possibly based on application input (Section 6.2). Second, a TCP session integrates both congestion control and loss recovery, thereby decreasing both packet loss rate and latency (Section 6.3). Finally, we not only have a design for TCP sessions but also an actual implementation.

Application-level approaches have also been proposed and implemented to speed up Web data transfers. A commonly used approach is to launch multiple concurrent connections, one for each logical piece of data. In [2], we analyzed traffic traces at a busy Web server and showed that multiple concurrent connections lead to more aggressive congestion control behavior than a single connection. In this paper, we show that this (greedy) approach can potentially lead to a higher packet loss rate and degraded performance (Section 6.3) for short, bursty transfers.

Another application-level approach is to multiplex several logically-separate transfers onto a single, persistent TCP connection, as proposed in P-HTTP [22] and recommended by HTTP/1.1 [11]. While this helps improve performance, it has limitations. Despite multiplexing, each burst of data transfer could still be quite short (for instance, [20] reports the average length of an entire Web page to be 26-32 KB). This coupled with the idle time between successive bursts (requests) can lead to a significant slow start overhead. Also, multiplexing several logically-separate transfers onto a single TCP connection results in undesirable coupling between them. For instance, the loss of a packet will hold up the delivery of packets from other transfers because TCP enforces in-order delivery across all the transfers. Finally, a solution tied to a specific application or application-level protocol does not help other applications or protocols that exist or may be developed in the future. We discuss these issues further in Section 6 and Section 8.

3. TCP Fast Start

3.1 Motivation

TCP is the de facto standard protocol for reliable unicast data transport in the Internet. An important reason for its widespread use is its robust congestion control and loss recovery mechanisms that work well under a variety of network and traffic conditions. One of the key algorithms employed by TCP is *slow start*, which enables a TCP connection to discover the available network bandwidth by slowly growing its window from an initial window size of 1 segment. Slow start also makes the connection self-clocked, thereby eliminating the need for timers to clock out data.

The reason why starting with a small initial window is important is that a new connection presumably does not know the available network bandwidth to the destination host. If a new connection starts off with a large window, it could overload the network and lead to heavy congestion loss, not just for itself but also for other connections.

These arguments apply equally well to an existing connection that is resuming activity after a significant idle time. After all, the network conditions could have changed unpredictably during the intervening time. Several modern implementations of TCP, including those derived from 4.4BSD, initiate slow start if the idle time is larger than a threshold, typically the retransmit timeout value (~ a few seconds).

These algorithms have worked well in the context of traditional applications such as FTP and Telnet. In the case of FTP, the cost of slow start is amortized over a relatively long transfer. In the case of Telnet, the amount of data to be sent in each burst is small (typically 1 packet [6]), so a small TCP window does not hurt. However, the cost of slow start can be significant in the context of Web access, which tends to be bursty like Telnet but has larger bursts (typically several kilobytes to a few tens of kilobytes). The goal of fast start is to address this problem, thereby making techniques such as P-HTTP more effective.

3.2 Goals of Fast Start

The goal of TCP fast start is to enable TCP connections, especially ones that transfer small amounts of data between pauses, to reuse information from the recent past rather than be forced to repeat the discovery process via slow start each time they resume activity. The cached information includes the congestion window size (*cwnd*), the slow start threshold (*ssthresh*), and the smoothed round-trip (*srtt*) time and its variance (*rttvar*). There are two important objectives:

1. The performance of a connection should not degrade significantly because of fast start. If the cached information from the recent past is still valid, fast start should in fact help improve performance. However, if this information is stale (for instance, because there has been a sudden surge in network load), fast start should not result in worse performance than if standard slow start had been used in the first place.
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 is trying to use its share of the bottleneck bandwidth. But it is not okay for them to suffer because the fast start connection is being over-aggressive (because it is using stale information).

It is 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 meet these goals with only a simple priority drop mechanism in the routers. Simulation results show that fast start is quite successful in its goal (Section 4).

3.3 Router algorithm

The router implements a simple packet drop priority algorithm. It distinguishes between packets based on a 1-bit priority field. When its buffer fills up and it needs to drop a packet, the router first checks to see if there are any low-pri-

ority packets in its buffer. If there are, it picks one of them for dropping. If not, it picks one of the other packets. 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. Note that this algorithm does not require routers to maintain any per-connection state.

Priority drop applies equally well to drop-tail and RED routers. A RED router detects impending congestion and notifies senders by marking their packets with an Explicit Congestion Notification (ECN) [14] bit. Since the goal of ECN is to ask connections that are using a large share of the bottleneck bandwidth to slow down, packet drop priority does not impact this in any way. An individual fast start packet is just as likely to get marked with ECN as any other packet. If a fast start connection is in fact using a large fraction of the link bandwidth, then it is correspondingly more likely that one of its packets will get marked.

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 can increase the queuing delay for other connections, but typically queuing delay has a minor impact on TCP performance compared to packet drops.

The notion of packet drop priority is of interest in other contexts, too. The cell loss priority mechanism (CLP) in ATM provides the same functionality at the granularity of cells. There is a growing effort to use IP's type-of-service (TOS) mechanism to support differentiated services (including packet drop priority) in the Internet [9]. This effort is being supported by the major router vendors. Note that for our purposes it is sufficient if drop priority is supported just by the bottleneck routers.

3.4 End-host 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 in the absence of ack clocking.
4. Quick detection and recovery from a failed fast start.

We discuss each of these in more detail in the following subsections.

3.4.1 Initiation and Termination of Fast Start

The rationale for fast start is that several studies [3,23] have shown that network conditions that determine the available bandwidth tend to remain unchanged for periods lasting tens of minutes. This length of time is much longer than the typical pause (i.e., user think time) during the course of an

interactive Web session. Therefore, it is reasonable to initiate fast start after such a pause. We defer the question of having an upper bound on the length of the idle period to future research.

Packets sent during the fast start phase are marked as low priority. The marked packets 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 can terminate prematurely if the sender detects multiple packet losses during fast start (Section 3.4.4).

3.4.2 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`.

`Ssthresh` tracks the size of the data pipe available to a TCP connection. It sets the threshold up to which the connection aggressively grows its window using slow start. Since our premise is that the network conditions are not likely to have changed much during the short pause, we leave `ssthresh` unchanged when initiating fast start.

`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 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 1 segment.

Finally, the cached values of `srtt` and `rttvar` (which together determine the retransmission timeout (RTO) value) are used without change. It is likely that packets of the connection traverse the same path through the network. However, the queuing delay in the network could have changed significantly. A solution may be to use a larger weighting factor for fresh RTT samples during fast start to enable quick adaptation in case of a significant change. However, we have not experimented with such a scheme.

3.4.3 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 start, we need another way 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 kicks in. The sender has a configurable parameter, `maxburst`, which indicates the maximum size of a burst that it can send out. The sender spaces apart such `maxburst`-sized bursts in time by $\text{maxburst} * \text{srtt} / \text{cwnd}$. 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 experiments⁴.

The overhead of software timers is not likely to be significant in modern computer systems with fast processors. For example, [10] reports an overhead of 6-7 microseconds for setting/handling timers on a 133 MHz PowerPC running AIX 4.2, which uses a naive data structure for implementing timers. Faster processors and better data structures should do even better. Also, 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 used to clock out data only during the initial fast start phase, which is about one RTT in duration.

3.4.4 Quick Detection and Recovery From Failed Fast Start

TCP fast start attempts to speed up short transfers while minimizing the impact on other connections. However, it is possible that the cached information used by fast start is no longer valid because of significant changes in network conditions. A situation that is especially dangerous is one where the network load increases significantly (because of several new connections becoming active) during the time a particular connection was idle. When this connection becomes active again and initiates fast start, it may end up being too aggressive. The priority drop algorithm discussed in Section 3.3 shields other (non-fast start) connections in such a situation. However, the connection that is attempting fast start could itself suffer heavy packet loss. With standard TCP loss-recovery algorithms, it could take a long time for the recovery process to complete. As a result, the performance with fast start could be much worse than if standard slow start had been used instead. As pointed out in Section 3.2, this is undesirable.

We discuss a set of techniques to avoid significant performance degradation when a fast start attempt fails.

- *Fine-grained reset timer:* The TCP timestamp option is used to obtain accurate RTT samples, from which the accurate estimates of `srtt` and `rttvar` are computed. These are used to set a fine-grained timer⁵ to detect the loss of fast start packets. We believe that the additional overhead of fine-grained timers is acceptable in this case because it is only incurred for fast start packets, and such packets have a higher likelihood of being dropped because of the low priority accorded to them by routers.
- *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 3.4.1 does *not* have the priority-drop bit set),

4. Note that even standard TCP with delayed acks can burst out 3 segments in a row during slow start.

5. The timeout value is still computed as $srtt + 4 * rttvar$ with the fine-grained estimates of `srtt` and `rttvar` used in place of the coarse-grained ones.

the TCP sender cuts down `cwnd` to 1 and initiates slow start. However, it does not cut down `ssthresh`, back off its RTO, or discard selective ack (SACK) 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 (and no fast start).

It is important to note the distinction between a timeout due to the loss of a fast start packet and that due to the loss of a regular packet. The latter is an indication of congestion, so the sender should back off to alleviate the congestion. The former is an indication that the (optimistic) fast start attempt was too aggressive, so the sender should fall back to default behavior (slow start). If the network is truly congested, the connection will discover that during slow start. The fact that the original fast start packets were assigned a low priority is of critical importance here. It ensures that fast start does not aggravate congestion in the network. It also makes fast start packets more susceptible to loss than other packets. So the loss of a fast start packet is not equivalent to the loss of any other packet as an indicator of congestion.

- *Recovery from multiple losses:* When there is packet loss during fast start, TCP (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 result in significantly worse performance than if fast start had not been used at all. For example, TCP NewReno recovers at the rate of one loss per RTT, which is equivalent to operating with a window size of 1 segment until all the packets lost in the burst have been retransmitted.

We use the following heuristic to limit such performance degradation. If selective ack information (SACK) is available, TCP 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 more than one packet [16]) during fast recovery, it cuts `cwnd` down to 1 and initiates slow start (without penalty, as described above) right away, without waiting for a timeout.

- *Capitalizing on successful transmission during a failed fast start:* Although a fast start attempt may have failed because of multiple packet drops, some packets may have actually been delivered successfully. Therefore, the sender uses cumulative ack and selective ack information, both from the fast start phase and from the following slow start, to avoid retransmitting such packets.

We demonstrate the benefits of these techniques via simulation.

4. Simulation Results

We evaluated the performance of TCP fast start via experiments in the ns network simulator [21]. Our implementation of fast start in ns works in conjunction with several flavors

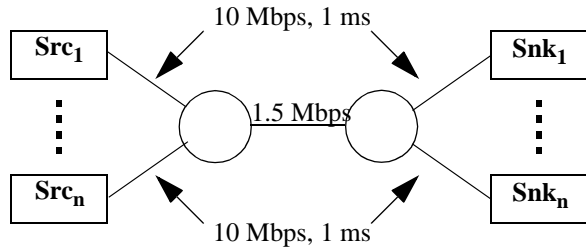


Figure 1. The network topology used for the simulation experiments. The delay of the 1.5 Mbps link is set to either 50ms or 200ms.

of TCP. However, for the purposes of this paper, we confine ourselves to TCP NewReno.

The topology used for the simulation experiments is shown in Figure 1. One or more bursty connections are established between a subset of the sources (servers) on the left and sinks (clients) on the right. Cross-traffic in the form of TCP bulk transfers is introduced between other sources and sinks to create contention for the 1.5 Mbps (T1 speed) bottleneck link. The link delay is set to either 50 ms (like terrestrial WAN links) or 200 ms (like geostationary satellite links). The maximum window of the bulk transfer TCP connections is set to either 8 KB or 32 KB corresponding to link delays of 50ms and 200ms, respectively. This implies that a single such connection is able to use approximately 40% of the link bandwidth, and a few of them together lead to congestion. The bottleneck link router uses FIFO scheduling and drop-tail buffer management unless otherwise specified. The TCP segment size is set to 1 KB.

We consider the following protocol combinations for the bursty connection:

1. Standard TCP NewReno with slow start (newreno). This is representative of a P-HTTP connection.
2. NewReno with fast start (fs)
3. NewReno with fast start and priority drops (fs-pdrop)

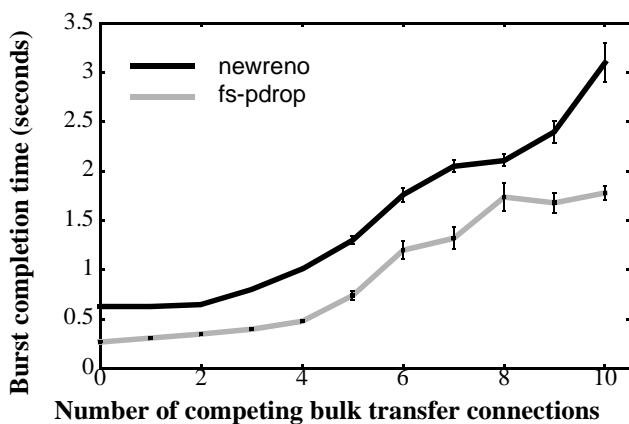


Figure 2. The completion time for the second 30 KB burst under “constant load” conditions. The bottleneck link delay is 50ms and the buffer size is 50 packets.

4. NewReno with fast start and priority drops but without the fast loss recovery techniques discussed in Section 3.4.4 (fs-pdrop-noflr)

The TCP modules we used in the ns simulator do not include the 3-way handshake at connection setup time. Therefore, all the four protocol combinations in our experiments assume T/TCP-style accelerated open that avoids the RTT for connection setup.

We conducted experiments under different conditions to evaluate various aspects of fast start. For each configuration, we conducted 10 runs of the experiment and report the mean. We also report the standard error as error bars.

4.1 Single bursty connection with constant load

In the constant-load experiment, bulk transfers constituting the cross traffic are left running throughout the duration of the experiment to provide a level of background load that is roughly constant. 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 between experiments, we were able to evaluate fast start under different levels of cross-traffic load.

After the cross-traffic has reached its steady state level, a single bursty connection is initiated between Src₁ and Snk₁. This connection sends 30 KB of data in a burst before becoming idle. The maximum TCP window size for this connection was set to 32 KB. After an idle time of 20 seconds (the exact value is not significant here), the bursty connection wakes up and transfers another 30 KB of data. We report the time for the second 30-KB burst to complete. The bursty connection mimics a P-HTTP connection used for two Web page downloads, 20 seconds apart. We picked 30 KB for the size of each transfer because it matches the average Web page size reported in [20].

Figure 2 and Figure 3 show the results for two different settings of the bottleneck link latency: 50 ms and 200 ms. We make two observations. First, there is a significant improve-

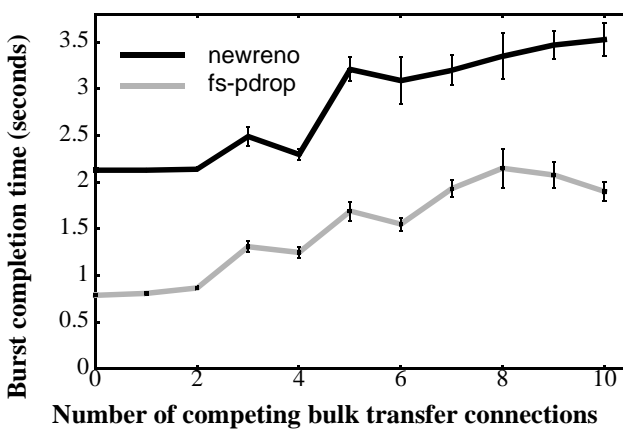


Figure 3. The completion time for the second 30 KB burst under “constant load” conditions. The bottleneck link delay is 200ms and the buffer size is 50 packets.

ment in completion time due to fast start. In percentage terms, the improvement is largest (50-65%) when the cross-traffic load is low. In absolute terms, the improvement increases with link delay. These trends are as we would expect. Under conditions of low load, the bursty connection can build up a large window size during its first burst and then successfully reuse it during fast start in the second burst. Also, since fast start saves RTTs, the absolute improvement is larger as when the RTT is larger.

Second, the difference in performance between fs and fs-pdrop is insignificant (which is why we have only plotted the results for fs-pdrop). This is because under conditions of constant load, the cached window size used by fast start remains fairly accurate. So fast start is not over-aggressive, and results in few packet drops. As a result, the use of priority drop does not make much of a difference in this case.

In summary, under conditions of constant load, fast start can result in a significant improvement in performance, especially when the load is low or RTT is large. If network conditions tend to be stable over periods of tens of minutes [3,23] and the length of the idle period is shorter than this, then the constant-load assumption is reasonable.

4.2 Single bursty connection with changed load

This experiment is similar to the previous one except for one significant difference: the cross-traffic is absent at the time the bursty connection transfers its first burst, but is introduced into the network during the idle time between the two bursts. Therefore, the network conditions will have changed for the worse between when the bursty connection caches various network parameters (at the time of the first burst) and when it tries to reuse them (at the time of the second burst). Figure 4 shows the results with a 50-ms bottleneck link delay.

There is still a significant decrease in completion time due to fast start (50-60%) when the load is low (though this is difficult to see due of the scale of the graph). This is because there are few dropped packets under these conditions. This also explains why there is not much difference in performance between the various flavors of fast start (fs, fs-pdrop and fs-pdrop-noflr).

However, the situation is quite different under conditions of high load. There is a sharp upswing in all the curves beyond 5 bulk transfer connections because now the 50-packet buffer often fills up and causes packet drops. In particular, when the bursty connection initiates fast start using the parameters it had cached (but which are now stale because of the changed network load), it pushes the already loaded network to the point where several packets get dropped.

We make several important observations in the high-load case. Fast start without priority drop (fs) still results in a better completion time than newreno. This is because in the absence of priority drop, packet drops are spread across both the bursty connection as well as the bulk transfers. So the bursty connection, whose fast start attempt is primarily

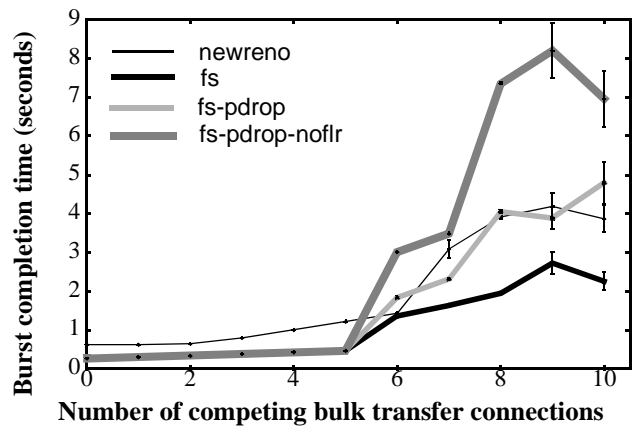


Figure 4. The completion time for the second 30 KB burst under “changed load” conditions. The bottleneck link delay is 50ms and the buffer size is 50 packets.

responsible for the packet drops, does not suffer much. On the other hand, with fs-pdrop the bursty connection bears the brunt of the packet drops. As a consequence, its performance under conditions of high load is similar to that of newreno, or marginally worse in some cases because of the penalty of the failed fast start attempt. Further, fs-pdrop-noflr performs significantly worse than newreno under these conditions. This clearly demonstrates the importance of the special loss recovery techniques described in Section 3.4.4. A fast start attempt with only standard TCP loss recovery mechanisms could significantly degrade performance.

Fast start without priority drop (fs) achieves better performance at the cost of the ongoing bulk transfer connections. This is clear from Figure 5 which shows the sequence number trace of a bulk transfer connection in the immediate aftermath of a fast start attempt by 10 bursty connections. Due to the absence of priority drop, fs causes several packet drops for the bulk transfer, which as a result stalls 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.

In summary, fast start is resilient even when the network

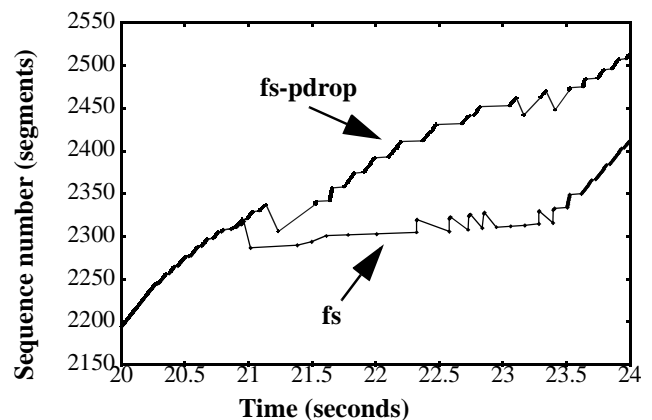


Figure 5. Effect of fast start by bursty connections on an ongoing bulk transfer connection. The link delay is 50 ms and buffer size is 50 packets.

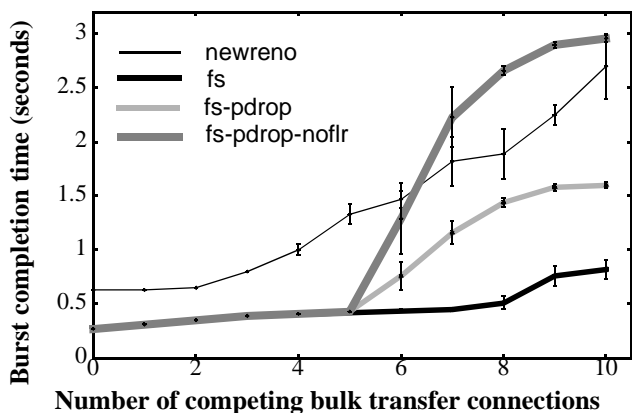


Figure 6. The completion time for the second 30 KB burst under “changed load” conditions. The bottleneck link delay is 50ms, buffer size is 50 packets and buffer management algorithm is RED.

load changes significantly and invalidates the cached state used by fast start. The special techniques used to detect and recover from a failed fast start help avoid a significant degradation in performance. The use of priority drop ensures that an over-aggressive fast start attempt does not have a significant adverse effect on other traffic. This is in keeping with our goal of having fast start help speed up bursty transfers when there is bandwidth available, but not at the expense of other traffic (Section 3.2).

4.3 Single bursty connection with changed load and RED buffer management

We repeat the experiment discussed in Section 4.2 with the bottleneck router configured with RED buffer management [14] instead of drop-tail. RED enables high link utilization but at the same time maintains free space in the buffer to absorb bursts. So fast start packets will typically not encounter full buffers even when the load is high. We configure the RED gateway to mark (ECN notification) rather than drop packets. The weighting factor for computing the average queue length is set to 0.02. The minimum and maximum queue length thresholds for probabilistic marking are set to 15% and 60%, respectively, of the buffer size (7.5 and 30 packets, respectively, for the 50-packet buffer that we used). When the *average* queue length exceeds the maximum threshold, all packets are marked.

As explained in Section 3.3, both fast start packets and other packets are treated equally by the ECN algorithm. If a connection doing fast start receives an ECN notification, it cuts down its window just as any other connection would. This reduced window size is reflected in a smaller initial window size for the next fast start attempt, if any.

Figure 6 is the analog of Figure 4 for this experiment. Comparing the two figures, we see that the RED algorithm helps improve performance significantly, especially under conditions of high load. This is because RED is usually able to maintain some free space in the buffer, which helps cut down packet loss during fast start. This enables fs-pdrop to provide a 25-50% reduction in completion time compared to

newreno even under conditions of high load. As before, fs performs even better, but at the expense of the bulk transfer connections.

In summary, RED buffer management has a significant beneficial effect on the performance of fast start. Like priority drop, RED does not require the router to maintain per-connection state. We believe that this is a desirable feature to enable efficient implementation.

4.4 Multiple bursty connections

The goal of this experiment is to evaluate the behavior of fast start in an environment where some bursty connections use fast start while others do not. We have 10 connections in each group (i.e., 10 of them do fast start while the other 10 do not). Each of the 20 connections becomes active at some point in a 10-second interval and transfers a burst of 30 KB. After a long idle time, each of the 20 connections becomes active again and transfers another 30 KB. But this time, all the transfers are initiated within a 1-second interval. This mimics the “hot spot” phenomenon in the Web where under normal conditions, client requests to a server are spread out in time, but when a hot story breaks out, many client requests are concentrated over a short period of time. Table 1 reports the completion time for the second burst, averaged separately for connections in each group.

We observe that the connections in group #1, which use fast

	Group #1 (FS)	Group #2 (no FS)
newreno	2.41 (0.05)	2.34 (0.07)
fs	1.90 (0.03)	2.88 (0.06)
fs-pdrop	1.80 (0.07)	2.38 (0.05)

Table 1. The average completion time (in seconds) of a 30 KB burst with a mix of connections that do fast start (group #1) and those that do no (group #2). The standard error is given within parentheses. Note that neither group of connections attempts fast start in the newreno case. The link delay is 50 ms and buffer size is 50 packets.

start for the second burst, obtain a 20-25% improvement in performance. But the more important point to note is that when fast start is used without priority drop, the gain comes at the expense of connections in group #2 that do not attempt fast start, which is unacceptable. The use of priority drop eliminates this problem, reinforcing the point we made in Section 4.2 about the importance of having such a safeguard mechanism.

4.5 Summary of results

In summary, TCP fast start cuts down the completion time for short bursts significantly under a variety of conditions. Fast start is most beneficial when the network is not congested, the bottleneck router maintains free buffer space to absorb bursts and/or the RTT is large. Under adverse conditions, fast start avoids any performance degradation com-

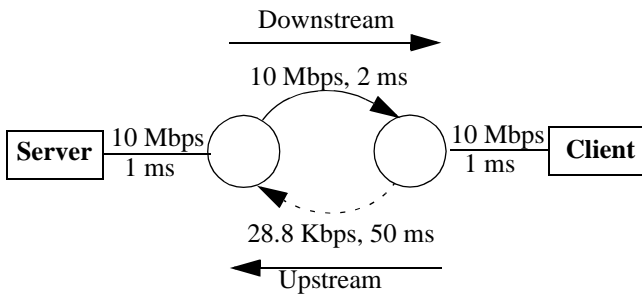


Figure 7. The simulation topology used for the asymmetric-bandwidth network experiment. The buffer size at all routers was set to 30 packets.

pared to standard slow start.

5. Asymmetric-bandwidth networks

Our simulation results indicate that the reduction in latency due to fast start in absolute terms is larger when the RTT is larger. One instance of a network with a large RTT is a satellite-based one. But as we shall explain, even high-bandwidth terrestrial networks can temporarily have a large RTT because of bandwidth asymmetry.

An asymmetric-bandwidth network provides high bandwidth towards a client (human user) but much lower bandwidth in the opposite direction. The raw difference in bandwidth in the two directions could range from a factor of 10 to 1000 depending on the network. Examples include cable modem downlink with dialup uplink, and asymmetric digital subscriber line (ADSL). The deployment of such networks is being driven by the relatively low cost of providing high downstream bandwidth, which is well matched to the asymmetric bandwidth demands of popular applications such as Web access.

For our simulation experiments, we use a network topology (Figure 7) modeled after the wireless cable modem network from Hybrid, Inc. [17] deployed at our location. This network provides a unidirectional 10 Mbps channel and uses a 28.8 Kbps dialup link for connectivity in the reverse direction.

Although the RTT between the server and the client in Figure 7 is inherently not large, it can become quite large in the presence of bidirectional traffic. The queuing of the (large) data packets of an upstream transfer at the bandwidth-constrained upstream link could increase the RTT significantly for a concurrent downstream transfer.

The main problem with the large RTT caused by bidirectional traffic is that slow start for a new downstream transfer progresses very slowly. This is despite the fact that there may be little or no congestion in the forward direction. If the new downstream transfer was in fact just restarting after an idle time, it could initiate fast start using the cached window size instead of going through slow start. An example scenario where this may be useful is a user who takes a short break from a Web browsing session to send out a large file via e-mail (upstream transfer) and then resumes browsing

(downstream transfer) immediately⁶.

We conducted a simple simulation experiment using the topology shown in Figure 7 to quantify the benefits of fast start. There is a bursty downstream connection that transfers two 30 KB bursts spaced apart by a 20-second idle period. During the idle period, an upstream bulk transfer is initiated. We measure the completion time for the second burst.

	FIFO	acks-first
newreno	24.47 (0.46)	7.33 (0.56)
fs-pdrop	11.79 (0.35)	4.74 (0.58)

Table 2. The completion time (in seconds) for the second 30 KB burst in the downstream direction.

The standard error is reported in parentheses.

We experimented with two configurations of the upstream router: FIFO scheduling of data and ack packets, or priority scheduling of the (small) ack packets over the (large) data packets (*acks-first* scheduling that we proposed in [1]).

Table 2 shows the results. Using fs-pdrop for the bursty downstream transfer results in substantially better performance than using newreno. In absolute terms, the improvement is larger when FIFO scheduling is used. But fast start also helps significantly when acks-first scheduling is used. The reason is that acks first scheduling does not help avoid the delay due to a data packet whose transmission is in progress. This transmission time can increase the RTT for the downstream transfer significantly (by up to 280 ms for a 1 KB data packet over a 28.8 Kbps dialup line).

Although upstream congestion does not directly impact the available downstream bandwidth, it could cause a significant increase in the RTT. Therefore, fast start based on an old estimate of the RTT is likely to time out⁷. But despite this, fast start is very beneficial because as the acks for the packets sent during the first RTT are received, the sender is able to quickly grow its window. With standard slow start, this procedure would take several (long) RTTs. The window size thus attained can then be used for future fast starts.

6. TCP Session

TCP fast start helps alleviate the problem of short bursty transfers common in the Web. It can also help in asymmetric-bandwidth networks. But there is still the third challenge mentioned in Section 1, which is that a Web page download typically involves multiple logically-separate transfers, say one for the HTML and one for each inlined image in the page. We outlined two alternative solutions in Section 2:

6. Another situation where bidirectional traffic could be important is a small office with multiple hosts, some of which are trying to download data while others are trying to send out data.

7. Decreasing the weightage of old RTT estimates during fast start (as has been suggested in the context of T/TCP [24]) could help. However, we have not investigated this.

using a separate TCP connection for each transfer or multiplexing all the transfers onto a single TCP connection. Here we propose a third solution, TCP session, which combines the desirable feature of each of the above solutions while leaving out the undesirable ones.

The advantage of using a separate TCP connection for each transfer is that it gives the application n independent data channels (TCP connections) for n independent data streams (text, images, etc.). This is a good match. However, the problem is that the n connections do congestion control and loss recovery independent of one another. As we shall show, this leads to performance problems.

The advantage of an application-level multiplexing solution, such as P-HTTP, is that it does not suffer from the performance inefficiencies of independent TCP connections. However, because of TCP's ordered byte-stream abstraction, the data streams multiplexed by the application are no longer independent. For instance, the loss of a packet can (temporarily) hold up data delivery for other data streams. Application-level multiplexing also requires a new on-the-wire protocol format for framing, and an agreement on this between the communicating peers. Finally, a solution like P-HTTP is specific to an application-level protocol, namely HTTP.

We chose a different approach in which applications use a separate TCP connection for each transfer. But we separate TCP functionality into two parts: (1) in-order data delivery, which is implemented (independently) by each TCP connection, and (2) congestion control and loss recovery, which is integrated across concurrent TCP connections between a pair of hosts⁸. The latter is accomplished by introducing a new abstraction — that of a TCP session — to aggregate TCP connections. This abstraction is transparent to applications, though as we discuss in Section 8, applications can choose to manage a TCP session explicitly. Implementing TCP sessions involves changes only at the sender side.

In previous work [2], we described the basic algorithms for integrated congestion control and loss recovery. We also used some simple simulations of bulk transfers to show how these techniques improve fairness. Here, we briefly recapitulate the basic algorithm (Section 6.1), and then discuss some new issues and enhancements to the basic algorithm (Section 6.2). We also present simulation results using a traffic pattern that mimics Web page download (Section 6.3).

6.1 Overview of Integrated Congestion Control and Loss Recovery

We briefly review the integrated congestion control and loss recovery algorithms from our previous work [2]. For the set of TCP connections between a pair of hosts, there is a TCP

8. In general, TCP sessions can support integration at several different granularities such as individual applications, the set of applications launched by an individual user, etc. However, in this paper, we only consider integration at the granularity of host pairs.

session protocol control block (SCB) that maintains shared state for the connections. This includes a unified congestion window (`session_cwnd`), RTT and RTT variance estimates, and state to help in loss recovery. Each of the communicating peers can independently choose whether or not to maintain an SCB. Since the TCP session algorithm only impacts the process of sending data (and not the process of acknowledging received data), in practice it would be more important to incorporate the algorithm in Web server hosts than in Web client hosts.

Integrated congestion control limits the total amount of outstanding data that the set of connections in the session can have to at most `session_cwnd`. When the first TCP connection between two hosts is established, `session_cwnd` is initialized to 1 segment. As acks indicating the successful delivery of data packets are received, `session_cwnd` is grown. This growth is exponential until a threshold and linear beyond, just as in TCP. If a packet loss is detected, `session_cwnd` is halved, thereby halving the rate of the *entire* set of connections. The rationale is that packet loss indicates congestion along the *shared* path of the entire set of connections. If a retransmission timeout happens, `session_cwnd` is reset to 1 segment.

Integrated loss recovery improves data-driven loss recovery, thereby decreasing dependence on retransmission timeouts. The data-driven loss recovery algorithm in TCP, namely fast retransmission, is based on the observation that packets rarely get reordered in the network. So once a small number of duplicate acks (usually 3) have been received, it is safe to assume that the corresponding packet is lost and to retransmit it immediately. Integrated loss recovery uses essentially the same algorithm, except that instead of counting just duplicate acks, it also keeps track of later packets of other connections that have been successfully delivered (*later acks*). Since packets of all connections between a pair of hosts are likely to follow the same path through the network, reordering among packets of concurrent connections should also be a rare event. Therefore, the algorithm counts both duplicate acks and later acks to detect and recover from packet loss.

Next we discuss some issues and enhancements to this basic algorithm that we have developed since [2].

6.2 Some Issues

The first issue is the scheduling of connections within a session. The basic algorithm is to apportion the bandwidth equally across the connections. However, an application may consider certain data streams more important than others. The integration of multiple TCP connections by a TCP session makes it trivial to implement this. It would be much more difficult to do so with TCP connections that are entirely independent. We would have to clamp down the window size of less important connections so that the more important ones can grow their windows. Clamping down the window artificially could lead to under-utilization of the *available* bandwidth at times when the more important connections have no data to send.

A related issue is how a TCP session interleaves packets belonging to different connections. It is best to interleave them as finely as possible, i.e. interleaved on a per-packet basis. This not only makes the data stream for each connection as smooth as possible but also improves loss recovery. If the network drops a burst of packets in a row, fine-grained interleaving minimizes the number of packets dropped in a single connection. Since the cumulative ack of each connection only allows the TCP session to recover from one loss per RTT per connection, spreading out the losses across connections minimizes the time to recover from all of them.

In view of the scheduling and interleaving considerations just discussed, TCP session implements a *weighted round-robin scheduler*. It is possible to vary the relative weights dynamically.

Another issue is that the small initial window (1 segment) for the entire session could slow down progress compared to independent TCP connections. We use fast start to alleviate this problem by using a larger initial window size when past information is available. On the other hand, launching multiple independent connections in effect increases the window size but without the safeguard of priority drop. As shown in Section 4, this can hurt other traffic in the network. As we show next, applications could hurt themselves by launching multiple independent connections in parallel.

6.3 Simulation Results

In previous work [2], we experimented with long bulk transfers and demonstrated how integrated congestion control makes bandwidth sharing more fair and eliminates the incentive to (greedily) launch multiple independent connections simultaneously. Here we evaluate the performance of TCP sessions in the context of short, concurrent transfers that are characteristic of the Web.

The simulation experiments involve the same topology as in Figure 1. A TCP session integrates the set of TCP connections between a source-sink host pair (such as Src_1 and Snk_1). Each source transfers a burst to the corresponding sink. Each burst involves 4 concurrent transfers, each 10 KB long. So this experiment mimics the transfer of a Web page with 4 different components (such as inlined images). The number of source-sink pairs is varied from 1 through 12. The link delay is set to 50 ms and buffer size to 20 packets.

We evaluate the following configuration:

1. Independent TCP connections for each concurrent transfer between a source-sink pair (*indep*).
2. A single persistent TCP connection for all the transfers (*phttp*).
3. A separate TCP connection for each transfer but with a TCP session binding them together (*session*).

Figure 8 shows the “Web page” transfer time for the three configurations. The transfer time for *indep* increases sharply as the number of source-sink pairs increases. The increase is much more gradual for *session*. Beyond 6 source-sink pairs,

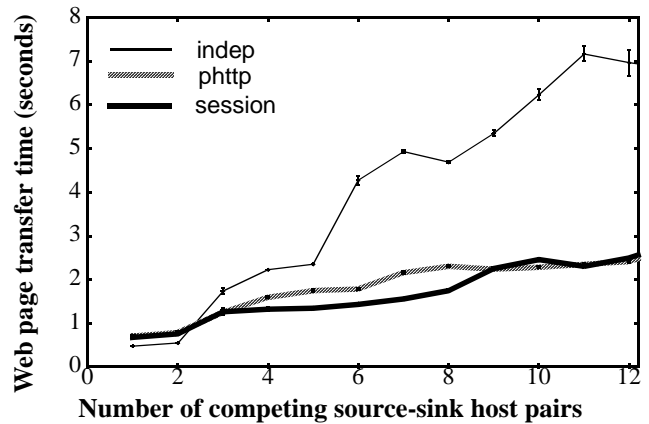


Figure 8. The transfer time for a “Web page” consisting of 4 components, each 10 KB in size. The bottleneck link delay is 50 ms and buffer size is 20 packets.

session performs between 60-68% better than *indep* (a factor of 2.5-3 reduction in transfer time). The reason for this significant performance difference is two-fold.

First, the independent TCP connections in *indep* are much more aggressive. Each TCP connection keeps increasing its window size until it (individually) suffers a packet loss. In contrast, in *session* the congestion window of the entire TCP session is cut down as soon as a (single) packet loss is detected. As a result, the packet loss rate is 30-40% higher for *indep* compared to *session*. The situation gets worse when transfers are longer, because the steady state behavior, in which each independent connection periodically pushes the network to congestion, sets in. Our experiments with bulk transfers show that the loss rate for *indep* is over 200% (i.e., more than a factor of 3) worse than for *session*.

Second, integrated loss recovery used by *session* is more effective than independent loss recovery by individual TCP connections. Since each TCP connection is short (10 KB in length), it often is the case that a packet loss cannot be recovered from via fast retransmission because not enough duplicate acks are generated by later packets of the same connection. Consequently, the connection is forced to time out. *Session*, on the other hand, does better because it also uses acks for later packets sent on concurrent connections.

The ability to use acks from concurrent connections is also the reason why *session* performs 20-25% better than *phttp* under conditions of moderate load, as shown in Figure 8. While *phttp* can recover from at most one packet loss per RTT, *session* can recover from packet loss in concurrent connections in parallel.

7. BSD/OS Implementation

We have implemented TCP fast start, priority drop and TCP session in the BSD/OS 3.0 kernel. At present, the implementations of TCP fast start and TCP session are separate, i.e., fast start operates on a per-connection basis rather than on a session-wide basis. We are in the process of integrating the two implementations.

7.1 Fast Start

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 a retransmit timeout (RTO) period. 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 `ssthresh` is the slow start threshold and `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` (smoothed RTT estimate) and `rttvar` (mean deviation of RTT) are left unchanged.

We then compute `fs_startseq` ($= \text{snd_nxt} + 2 * \text{maxseg}$) and `fs_endseq` ($= \text{snd_nxt} + \text{cwnd}$) to mark the start and end of the fast start phase. `Snd_nxt` is the next packet in sequence to be sent. The quantity $2 * \text{maxseg}$ accounts for the two segments⁹ that the BSD/OS 3.0 implementation of TCP would have sent in any case while restarting 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 (an unused bit in the TCP header¹⁰). This identifies them to routers as low priority packets.

To avoid bursting out data during the fast start phase, `tcp_output()` breaks up potentially large bursts into smaller ones, each at most `maxburst` 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 fine-grained timestamps¹¹. 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 (4 seg-

9. This is different from 4.4BSD which only sends one segment.

10. It is possible to use a generic “priority drop” 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, and couldn’t find any in the IP header.

11. We use timestamps with microsecond accuracy. Since the timestamp is a 4-byte quantity, this wraps around approximately every 36 minutes, which is quite safe. However, we could switch to timestamps with millisecond accuracy, which would increase the wraparound time thousand fold while still being accurate enough for our purposes.

ments, by default) are generated. This is likely to happen only during fast start phase (typically 1 RTT in duration).

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 $\text{srtt_exact} + 4 * \text{rttvar_exact}$, 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, the sender immediately *resets* its state to what it would have been had fast start not been attempted. `Cwnd` is reset to 2 (the default for restart after an idle period in BSD/OS 3.0), `ssthresh` is restored using the value saved at the time fast start was initiated, and slow start is initiated. However, none of the other penalties of a retransmit timeout (such as timer backoff) is imposed.

As explained in Section 3.4, 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 is done in TCP NewReno [16].

7.2 Priority Drop Algorithm

In practice, the priority drop algorithm should be implemented by the routers in the interior of the network. However, since we do not have control over the routers, we decided to implement the priority drop algorithm in conjunction with a *link emulation* module. Briefly, the link emulation module allows us to dial in the link parameters such as bandwidth, delay and queue size using the user-level `ifconfig` program. This enables emulation of a wide-area bottleneck link for the purpose of experiments.

The implementation of the priority drop algorithm itself is quite trivial. An auxiliary queue of pointers to low-priority packets is maintained. At present, the code just examines the TCP fast start bit in order to determine whether or not a packet has low priority. When a packet needs to be dropped and a low-priority packet is in the queue, the one closest to the tail of the queue is dropped.

7.3 TCP Session

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. TCP session includes functions 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 `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. The SCB includes variables such as `cwnd`, `ssthresh`, `srtt`, and `rttvar` to support integrated congestion control and loss recovery. The corresponding variables in the individual TCBS are simply ignored.

When `tcp_output()` is invoked, it in turn invokes `session_output()`. `Session_output()` first makes sure that the session's `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 a weighted round-robin schedule. Note that this connection could in general be different from the one that invoked `session_output()`. However, an exception is made if the original connection wanted to send out a segment with a flag such as FIN or RST set.

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 (i.e., acks for later packets of other connections). Later acks are detected by comparing the echoed timestamp against a sorted list of timestamps of packets that were transmitted. In the common case of no packet loss and little or no reordering, ack processing will be confined to the packets at or close to the head of the list.

If there are packet(s) for which the sum of duplicate and later acks exceeds a threshold (default 3), indicating a likely loss, the session's `cwnd` is halved and such packet(s) are retransmitted. If such data-driven loss detection fails, the session's retransmission timer will fire eventually, causing the session to initiate slow start with `cwnd` set to 1 segment.

Finally, when the last connection in a TCP session is closed, `session_destroy()` is called to clean up state and free the SCB. However, when we integrate our implementations of TCP sessions with fast start, it will be desirable not to tear down the session right away because the SCB provides a convenient location to cache the information needed for fast start in the future.

8. Discussion

Our results from Section 4 and Section 6.3 show that TCP fast start and TCP sessions help improve performance significantly in several situations. Further, in other situations, there is either only a small improvement in performance or none, but there is no degradation in performance.

The robustness of TCP fast start comes from its use of packet drop priority and its dependence on past information. If the network becomes congested, a fast start attempt will likely fail. If the connection is only able to build up a small window during the subsequent slow start, its next fast start will be much less aggressive. This self-regulating mechanism ensures that connections 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., [13]).

Each of TCP fast start and TCP sessions is a useful technique in itself. But they are in fact complementary, so combining them would be useful. Fast start would help protocols such as P-HTTP (by decreasing the penalty of restarting after an idle time) when the network is not congested. Sessions would help applications that want to transfer multiple data streams (but without coupling them together as in P-HTTP) when the network load is high. The end-host implementation of both techniques is confined to the sender side. So only Web servers need to be modified while the vast number of client hosts can be left untouched.

The TCP session abstraction can be exposed to applications to enable dynamic control of the relative bandwidth allocation to connections in a session. For example, a Web server could decrease the bandwidth share of a particular component for a Web page, such as a large MPEG file, so that other components can be transferred more quickly. Such decisions can also be made dynamically based on input from the client. For instance, if the user scrolls down a large Web page, it would be desirable to speed up the transfer of images in the visible portion of the Web page.

We believe that application-level multiplexing solutions, such as P-HTTP, violate the application-level framing (ALF) principle [8]. The ordered byte-stream abstraction of a single TCP connection is too restrictive to support multiple logically-separate transfers. The use of a single connection leads to several problems. One is that of coupling when there is a packet loss (Section 6). Another is that because the sequence number streams of the logically-separate transfers are intertwined, it is not possible for a transparent Web cache (e.g., [7]) to selectively intercept and serve a subset of Web requests from a client to a server.

In contrast, the TCP session approach is more in line with the ALF principle. Applications are free to launch as many TCP connections as logically-separate data streams they need. The only potential overhead is that of connection set up and state management. Connection set up overhead can be minimized by using T/TCP-like accelerated open. State management overhead can be minimized by using hashing rather than linear search for PCB lookup (e.g., as in [15]).

Finally, although the focus of this paper has been the Web, the techniques presented here will also be useful in other situations that involve bursty data transfer. Examples include wide-area transactions, RPC, and sensor data collection.

9. Conclusions

In this paper we have presented two techniques for improving the performance of Web data transfers. The first is TCP fast start, which allows a TCP connection to reuse network parameters cached in the recent past to avoid the penalty of slow start in many cases. The main results for fast start are:

- Up to 50-65% (2-3X) latency reduction for short bursts.

- Large improvement in absolute terms for high-latency and asymmetric bandwidth networks.
- RED buffer management helps.
- Priority drop and special loss recovery techniques help.
- Makes techniques such as P-HTTP more effective.

The second technique is TCP session, which integrates congestion control and loss recovery for multiple concurrent TCP connections. The main results for TCP session are:

- Up to 60-68% (2.5-3X) latency reduction.
- Significant reduction in packet loss rate.
- Enables use of multiple connections without penalty.

We have implemented both TCP fast start and TCP sessions in the BSD/OS 3.0 kernel, with sender-side only changes.

10. Ongoing Work

We are extending this work in several ways:

- Integration of the fast start and session implementations.
- Performance evaluation of the implementation.
- Application-level control of session bandwidth allocation in the context of the Web.

We plan to have these done before the final paper is due.

11. References

- [1] H. Balakrishnan, V. N. Padmanabhan, and R.H. Katz. The Effects of Asymmetry on TCP Performance. In *Proc. ACM MOBICOM '97*, September 1997.
- [2] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R.H. Katz. TCP Behavior of a Busy Web Server: Analysis and Improvements. In *Proc. IEEE Infocom '98*, March 1998.
- [3] H. Balakrishnan, S. Seshan, M. Stemm, and R.H. Katz. Analyzing Stability in Wide-Area Network Performance. In *Proc. ACM SIGMETRICS '97*, June 1997.
- [4] R. T. Braden. *Extending TCP for Transactions - Concepts*. RFC, Nov 1992. RFC-1379.
- [5] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. ACM SIGCOMM '94*, August 1994.
- [6] R. Caceres, P. Danzig, S. Jamin, and D. Mitzel. Characteristics of Wide-Area TCP/IP Conversations. In *Proc. ACM SIGCOMM '91*, September 1991.
- [7] Cisco Cache Engine. http://cco.cisco.com/warp/public/751/cache/cds_wp.htm.
- [8] D. D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proc. ACM SIGCOMM '90*, September 1990.
- [9] Differential Services for the Internet. <http://diff-serv.lcs.mit.edu>.
- [10] W. Feng, D. Kandlur, D. Saha, and K. Shin. Understanding TCP Dynamics in an Integrated Services Internet. In *NOSSDAV '97*, May 1997.
- [11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol - HTTP/1.1*. RFC, Jan 1997. RFC-2068.
- [12] S. Floyd, M. Allman, and C. Partridge. Increasing TCP's Initial Window. Internet Draft, IETF, 1997. Expires Jan 1998.
- [13] S. Floyd and K. Fall. Router Mechanisms to Support End-to-End Congestion Control. Technical report, Lawrence Berkeley National Lab, 1997.
- [14] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397-413, August 1993.
- [15] FreeBSD, Inc. <http://www.freebsd.org>.
- [16] J. C. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, 1995.
- [17] Hybrid Networks, Inc. <http://www.hybrid.com>.
- [18] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 88*, August 1988.
- [19] V. Jacobson and M. Karels. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, August 1990.
- [20] B. A. Mah. An Empirical Model of HTTP Network Traffic. In *Proc. Infocom 97*, April 1997.
- [21] UCB/LBNL/VINT Network Simulator - ns (version 2). <http://www-mash.cs.berkeley.edu/ns/>.
- [22] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proc. Second International World Wide Web Conference*, October 1994.

- [23] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California at Berkeley, May 1997.
- [24] W. R. Stevens. *TCP/IP Illustrated, Volume 3*. Addison-Wesley, Reading, MA, Jan 1996.
- [25] J. Touch. *TCP Control Block Interdependence*. RFC, April 1997. RFC-2140.
- [26] Vikram Visweswaraiah and John Heidemann. Improving restart of idle TCP connections. Technical Report 97-661, University of Southern California, November 1997.