In Proc. IEEE Globecom '98 Internet Mini-Conference, Sydney, Australia, November 1998.

TCP Fast Start: A Technique For Speeding Up Web Transfers

Venkata N. Padmanabhan and Randy H. Katz
{padmanab,randy}@cs.berkeley.edu

Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, Berkeley, CA 94720-1776.

Abstract

Web browsing is characterized by short and bursty data transfers interspersed by idle periods. The TCP protocol yields poor performance for such a workload because the TCP slow start procedure, which is initiated both at connection start up and upon restart after an idle period, usually requires several round trips to probe the network for bandwidth. When a transfer is short in length, this leads to poor bandwidth utilization and increased latency, which limit the performance benefits of techniques such as P-HTTP.

In this paper, we present a new technique, which we call *TCP fast start*, to speed up short Web transfers. The basic idea is that the sender caches network parameters to avoid paying the slow start penalty for each page download. However, there is the risk of performance degradation if the cached information is stale. The two key contributions of our work are in addressing this problem. First, to shield the network as a whole from the ill-effects of stale information, packets sent during the fast start phase are assigned a higher drop priority than other packets. Second, to prevent the connection attempting fast start from experiencing degraded performance under such adverse conditions, the standard TCP loss recovery procedure is augmented with new algorithms. Performance results, obtained from both simulation and experiments using our BSD/OS implementation, show a reduction in latency by up to a factor of 2-3.

Keywords: **TCP, Web, Congestion Control, Latency**

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 activity, it is highly desirable to have low latency. Web transfers tend to be short and bursty. The average Web page is quite small (26-32 KB according to [18]). There is often a pause (user think time) between successive downloads initiated by a client ([18] reports a median think time of 15 seconds).

These characteristics of Web transfers interact poorly with TCP. The basic problem is that the TCP slow start procedure [16], which is initiated both at connection start up and upon restart after an idle period, may require several round trips to probe the network for bandwidth and ramp up its window. When transfers are short in length, this probing leads to poor utilization of the available network bandwidth, and consequently, increased latency. The challenge, therefore, is to reduce latency for short Web transfers, but at the same time ensure that the well-being of the network as a whole is not impacted in the process.

In this paper, we discuss a new technique, which we call *TCP fast start*, to speed up short transfers such as Web page downloads. TCP fast start helps reduce the slow start penalty by reusing the values of the congestion window and RTT from the recent past to smoothly inject packets into the network until ack clocking sets in. This can result in a savings of several RTTs that 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 temporary traffic patterns (such as bidirectional traffic in an asymmetric-bandwidth network) (Section 5).

It is possible, however, that the cached network parameters are stale and so fast start ends up being over-aggressive, leading to heavy packet loss. The two key contributions of our work address this problem. First, fast start packet are assigned a higher drop priority[1] than other packets, so other traffic in the network is largely insulated from any congestion caused by fast start. Second, during the fast start phase, the sender augments the standard TCP loss recovery procedure with new algorithms that enable it to detect packet loss quickly and fall back to standard slow start, if necessary.

We have implemented TCP fast start both in the ns network simulator [20] and in the BSD/OS 3.0 [8] TCP/IP stack. Our results show that in many situations fast start reduces latency for short transfers by up to 50-65% (a factor of 2-3) compared to standard slow start. The reduction in latency leads to faster Web page downloads and better resource utilization at servers. Fast start coupled with priority drop is also resilient to staleness of the cached network parameters. There is little or no performance degradation in such a case.

Our end host implementation of fast start is confined to the sender side (Web servers), so the vast number of client hosts are left untouched. This is convenient from the viewpoint of incremental deployment. The drop priority mechanism is not universally supported by IP routers today. But this situation is changing fast. Drop priority is among the mechanisms being standardized as part of the differentiated services effort [9]. Even if TCP fast start is not deployed universally, significant performance benefits can be obtained (in certain situations) with a partial deployment (for instance, at a Web proxy). This provides a convenient path for incremental deployment of fast start.

The rest of this paper[2] is organized as follows. In Section 2, we survey related work. In Section 3, we discuss the design of TCP fast start. We present detailed simulation and implementation-based results in Section 4 and Section 5. A discussion of some general issues pertaining to TCP fast start appears in Section 6. Finally, we present our conclusions in Section 7, and give pointers to future work in Section 8.

2. Related Work

Modern TCP implementations are based largely on the algorithms presented in [16]. A key algorithm is *slow start*, which enables a TCP sender to discover the available network bandwidth by slowly growing its window from an initial size of 1 segment. This procedure, performed both at connection start up and when it resumes activity after an idle period [17], works well for long transfers, but proves expensive for short ones.

---

1. In the remainder of this paper, we use the term "low priority" to mean "high *drop* priority".

2. This paper has been abbreviated to conform to the Globecom page count limit. A more complete version is available on-line [22]. Padmanabhan's Ph.D. dissertation, with more details on this and related work, is also available on-line [21].

One proposal to alleviate this problem is to increase the initial window size for slow start to 2-4 segments (*4K-slow start* [1]). While this will help 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 [5]) is a TCP extension that enables fast connection establishment by having hosts cache sequence number information from connections in the recent past. T/TCP also mentions the possibility of caching the "congestion avoidance threshold" without offering details [6].

TCP control block interdependence [25] specifies temporal sharing of TCP state, including carrying over congestion window information from one connection to the next. A drawback, though, is that a new connection could send out a large burst of packets, both to its detriment and that of the rest of the network.

Rate-based pacing [26] addresses this problem in the context of a connection that restarts after an idle period but does not invoke slow start. An estimate of the connection's rate (obtained using the TCP Vegas algorithm [7]) is used to smooth out the burst of packets. However, such an open-loop scheme still has the drawback that it can potentially aggravate congestion in the network and cause heavy packet loss, both for itself and for other traffic in the network, especially if the old estimate of the rate is stale. While TCP fast start is also an open-loop scheme, it uses drop priority and an augmented loss recovery procedure to guard against this problem. Our experimental results demonstrate the usefulness of these techniques (Section 4 and Section 5).

Several researchers have analyzed wide-area network performance and concluded that the available bandwidth is often stable over periods lasting several minutes [4,24]. This suggests that in the common case fast start would indeed be successful.

Application-level approaches have also been proposed and implemented to speed up Web transfers. One common approach is to launch multiple concurrent TCP connections. However, this has the pitfall that the application's decision to launch a certain number of connections is not tied to the conditions in the network, and so can aggravate congestion [3].

Another application-level approach is *persistent-connection HTTP* (P-HTTP) [23], which re-uses a single TCP connection for multiple Web transfers, thereby amortizing the connection setup overhead. However, the slow start overhead is still incurred upon restart after an idle period [13]. As we shall see, P-HTTP coupled with fast start avoids this problem.

3. TCP Fast Start

In this section, we discuss the design of TCP fast start in detail.

3.1 Basic Idea

Fast start enables TCP connections, especially ones that transfer small amounts of data between pauses, to reuse network information from the recent past rather than be forced to repeat the slow start discovery procedure each time. 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. 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 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 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 only a simple drop priority mechanism in the routers. Our experimental results show that it is quite successful in its goal.

3.2 Router Mechanism

The router implements a simple packet drop priority mechanism. It distinguishes between packets based on a 1-bit priority field. When its buffer fills up and it needs to drop a packet, it picks a low-priority packet, if available, first. 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. This mechanism does not require routers to maintain any per-connection state. The drop priority mechanism applies equally well to drop-tail and RED routers [12].

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.

The notion of packet drop priority is of interest in other contexts as well. The cell loss priority mechanism (CLP) in ATM provides the same functionality at the granularity of cells. There is a growing effort to use the IP 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.

3.3 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.
4. Quick recovery from a failed fast start.

3.3.1 Initiation and Termination of Fast Start

The rationale for fast start is that several studies [4,24] have shown that network conditions that determine the available bandwidth tend to remain stable for periods ranging from a few minutes to 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. 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 3.3.4).

3.3.2 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 1 segment.

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

3.3.3 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 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 sets in[4]. 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[5].

There is evidence to suggest that the overhead of software timers is not likely to be significant in modern computer systems with fast processors ([11] reports an overhead of 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.

3.3.4 Quick Recovery From Failed Fast Start

It is possible that the cached information used by fast start is no longer valid because of significant changes in network conditions (such as several new connections becoming active). As a result, a fast start attempt could turn out being too aggressive. The priority drop algorithm (Section 3.2) shields other (non-fast start) connections in such a situation. However, the connection that is attempting fast start could itself suffer heavy packet loss, and end up with worse performance than if it had used standard slow start instead. We augment the TCP loss recovery procedure in several ways to avoid this undesirable situation.

- *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[6] 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 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 3.3.1, does *not* have the drop priority bit set), the TCP sender cuts down cwnd to 1 and initiates slow start. However, it does not cut down ssthresh, back off its retransmission timeout value (RTO), or discard selective ack (SACK [19]) 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 that unlike the loss of a regular packet, the loss of a fast start packet is not as much an indication of congestion as an indication that the (optimistic) assumption made about network conditions is invalid. In this context, the low priority assigned to the original fast start packets is of critical importance. The appropriate response for the sender is to fall back to default behavior (slow start), where no such assumption is made. If the network is truly congested, the connection will discover it during the subsequent 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 result in significantly worse performance than if fast start had not been used at all. For example, TCP NewReno[7] 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 (this is assuming there is no more new data to send, as is often the case with short Web transfers).

  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 [14]) during fast recovery, it cuts cwnd down to 1 segment and initiates slow start right away, without waiting for a timeout (and without imposing any other penalty, as described above).
- *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 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, to avoid retransmitting such packets. In contrast, the default TCP behavior is to discard selective ack information obtained prior to a timeout [19].

4. Simulation Results

We present performance results using our implementation of TCP fast start in the ns network simulator [20]. We use the NewReno [14] flavor of TCP as the basis for comparison.

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 size 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 in conjunction cause congestion. The bottleneck link router uses

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.

4. This is similar to the technique we have used previously to combat the ack feedback problem in asymmetric networks [2].

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

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

7. TCP NewReno is a variant of TCP Reno that uses partial new ack information to recover from multiple packet losses in a window, at the rate of one per RTT.

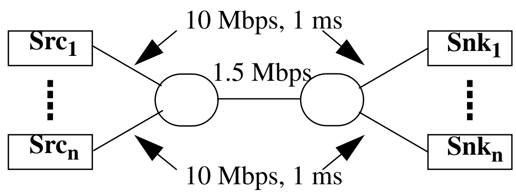

Figure 1. The network topology used for the simulation experiments. The delay of the 1.5 Mbps link is set to either 50ms or 200ms.

FIFO scheduling and drop-tail buffer management, unless specified otherwise. The TCP segment size is set to 1 KB. We consider several protocol combinations for the bursty connection:

1. Standard TCP NewReno with slow start (newreno).
2. NewReno with fast start (fs)
3. NewReno with fast start and drop priority (fs-pdrop)
4. NewReno with fast start and drop priority, but without the fast loss recovery techniques described in Section 3.3.4 (fs-pdrop-noflr)

The TCP modules we use in the ns simulator do not employ 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 report the mean of 10 runs, and the standard error (as error bars).

4.1 Constant-Load Experiment

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 are 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 Src_1 and Snk_1. This connection sends 30 KB of data (which matches the average Web page size reported in [18]) in a burst before becoming idle. The maximum TCP window size for this connection is set to 32 KB. After a long pause, 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 spaced apart in time.

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 improvement 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 reuse it successfully for fast start during the second burst. Also, since fast start saves RTTs, the absolute improvement is larger when the RTT is larger.

Second, the difference in performance between fs and fs-pdrop is insignificant (which is why we only show 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 results in few packet drops. As a result, the use of priority drop does not make much of a difference in this case.

4.2 Changed-Load Experiment

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 for the 50-ms delay case.

There is still a significant decrease in completion time due to fast start when the load is low and few packets are dropped (50-60%, although this is difficult to see due of the scale of the graph). 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 the 50-packet buffer tends to fill up, causing frequent packet drops. In particular, when the bursty connection initiates fast start using the information it had cached (but that is now stale because of the changed network conditions), it causes the already loaded network to drop several packets.

We make several important observations in the high-load case. Fast start without priority drop (fs) still performs better than newreno. This is because, in the absence of drop priority, packet drops tend to be spread across both the bursty connection and the bulk transfers. So the bursty connection, whose fast start attempt is primarily 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, but still its performance under conditions of high load is similar to that of newreno. In contrast, fs-pdrop-noflr performs significantly worse than newreno under these conditions. This underscores the importance of the augmented loss recovery procedure described in Section 3.3.4. A fast start attempt with only the standard TCP loss recovery mechanisms could degrade performance significantly.

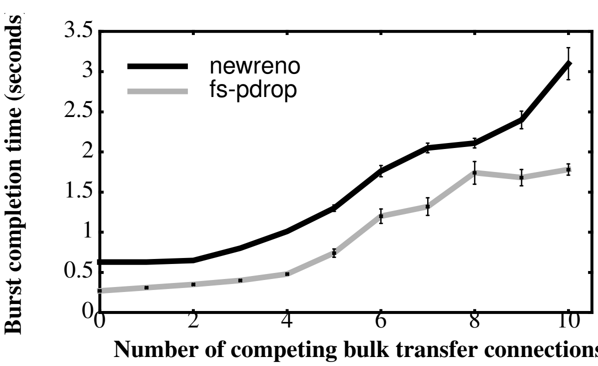

Figure 2. The completion time for the second 30 KB burst under constant-load conditions. The bottleneck link delay is 50ms and buffer size is 50 packets.

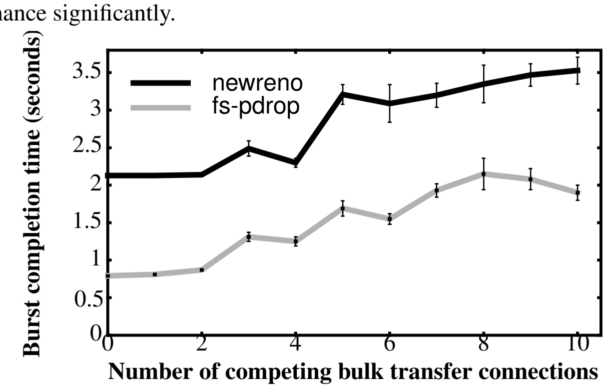

Figure 3. The completion time for the second 30 KB burst under constant-load conditions. The bottleneck link delay is 200ms and buffer size is 50 packets.

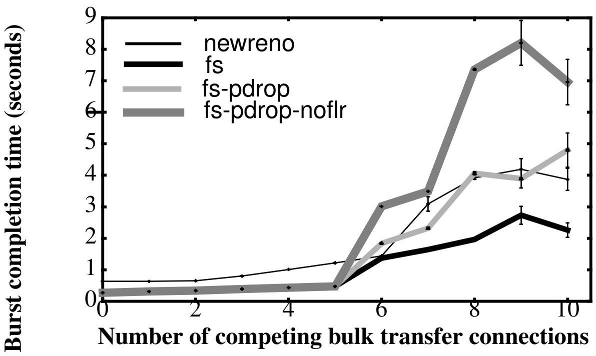

Figure 4. The completion time for the second 30 KB burst under "changed-load" conditions. The bottleneck link delay is 50ms and buffer size is 50 packets.

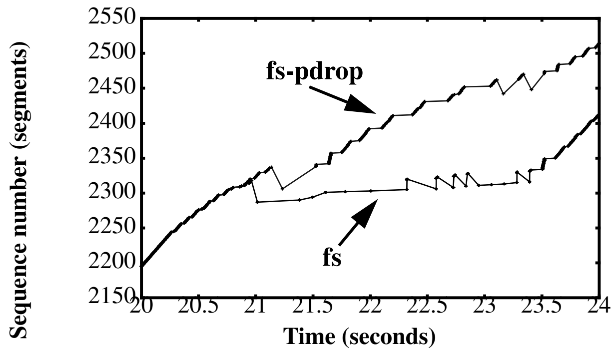

Figure 5. Effect of fast start by bursty connections on an ongoing bulk transfer connection. The link delay is 50 ms and buffer size is 50 packets.

Fast start without drop priority (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 drop priority, 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 improves performance significantly under favorable conditions, but is also resilient under adverse conditions. Both the drop priority mechanism and the augmented loss recovery procedure contribute to its robustness.

5. Other Results in Brief

In the interest of space, we describe the other results, both from simulation and our BSD/OS implementation, quite briefly. A more detailed treatment appears in [21] and [22].

5.1 RED Buffer Management

RED buffer management [12] enables high link utilization, but at the same time maintains free buffer space to absorb bursts. So fast start packets are less likely to encounter full buffers even when the load is high. This results in a significant improvement in performance. For instance, when the changed load experiment (Section 4.2) is conducted with a RED bottleneck gateway, fs-pdrop performs 25-50% better than newreno, even under conditions of high load.

5.2 Flash Crowds

Flash crowds refers to the situation where a large number of users (clients) converge on a Web server almost simultaneously, often because of some extraordinary news event. To emulate this, we configure several clients to communicate with a server, evenly spread out over a relatively long period of time. As a result, the server caches a snapshot of the network conditions in the absence of congestion. After a pause, all the clients initiate a download from the server almost simultaneously, so the cached information becomes stale. Further, we have half of the clients use fast start and the other half standard slow start. Our results indicate that fs-pdrop yields up to a 25% improvement for the first set of connections, and negligible degradation for the second set. In contrast, fs alone results in a similar improvement for the first set, but up to a 21% degradation for the second set. So fs-pdrop improves performance without penalizing other connections.

5.3 DirecPC Satellite Network

The DirecPC network [10] is an early example of satellite-based network dedicated to providing data access to end-users. The geostationary satellite link only provides connectivity in the downstream direction (12 Mbps). Upstream connectivity is provides by another means such as a dialup phone line. The RTT of a connection to a Web server via the DirecPC network is about 400 ms. This large RTT means that fast start performs quite well. For instance, with our BSD/OS implementation of fast start, the second[8] of two 50 KB transfers is sped up from 2.6 seconds down to 0.95 seconds, and similarly the second of two 175 KB Web page downloads (each with 8 inline images) takes only about 3.05 seconds instead of 7.70 seconds with newreno.

5.4 Cable Modem Network

A cable modem network, another instance of a new breed of access networks, is much like the DirecPC network in that provides highly asymmetric bandwidth. However, the delay of the cable modem link is usually quite small (about 2 ms in the system from Hybrid, Inc. [15] in our network testbed). But, in the presence of bidirectional traffic, the RTT for a downstream TCP connection can become quite large (hundreds of milliseconds). This is because acks for the downstream connection could get queued behind one or more (large) data packets of the upstream transfer. Under such conditions, fast start is again very helpful. For instance, the completion time for the second of two 175 KB Web downloads decreases from over 25 seconds to 14 seconds.

6. Discussion

Our experimental results show that TCP fast start can help cut down Web transfer latency significantly when the cached network information is valid, which is presumably the common case, given the results in [4,24]. At the same time, it avoids performance degradation when the cached information is stale. This robustness comes from its use of packet drop priority and its dependence on past information, which ensure that senders do not keep flooding the network with packets bound to get dropped.

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

The DirecPC and the cable modem systems point to how even a partial deployment of fast start could be very beneficial. Fast start could be deployed at a Web proxy located at the satellite earth station or the cable modem head end. If all the clients route

8. The first transfer allows the sender to cache information about the network, which it then uses for fast start during the second transfer.

their Web accesses via the proxy, much of the performance benefit of fast start can be had without having to modify the thousands of Web servers in the Internet. (In any case, the clients do not require any modification.) This also provides an incremental deployment path for the drop priority mechanism, since it only needs to be incorporated in the subset of the network that lies along the path from the proxy 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.

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

7. Conclusions

In this paper we have presented a new technique called *TCP fast start* for speeding up short and bursty transfers such as Web page downloads. Fast start enables a TCP connection to reduce the overhead of slow start by reusing network parameters cached in the recent past, but at the same time avoids performance degradation when the cached information is stale. Our key results 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.
- Techniques such as P-HTTP are made more effective.

We have implemented TCP fast start in the BSD/OS 3.0 TCP/IP stack with sender-side only changes. This facilitates deployment in the context of the Web because changes would be confined just to the servers, leaving the large number of clients untouched.

8. Future Work

Two directions for future work are:

- Policies for aging the cached network parameters.
- Mechanisms for sharing cached information across hosts.

9. Acknowledgements

We would like to thank Tom Henderson and the anonymous Globecom reviewers for their comments.

10. References

[1] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. Internet Draft, IETF, May 1998. Expires Nov 1998.

[2] H. Balakrishnan, V. N. Padmanabhan, and R.H. Katz. The Effects of Asymmetry on TCP Performance. In *Proc. ACM MOBICOM '97*, September 1997.

[3] 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.

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

[5] R. T. Braden. *Extending TCP for Transactions - Concepts*. RFC, Nov 1992. RFC-1379.

[6] R. T. Braden. *T/TCP – TCP Extensions for Transactions Functional Specification*. RFC, July 1994. RFC-1644.

[7] 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.

[8] Berkeley Software Design, Inc. http://www.bsdi.com.

[9] Differential Services for the Internet. http://diffserv.lcs.mit.edu.

[10] DirecPC Web page. http://www.direcpc.com.

[11] W. Feng, D. Kandlur, D. Saha, and K. Shin. Understanding TCP Dynamics in an Integrated Services Internet. In *NOSSDAV '97*, May 1997.

[12] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[13] J. Heidemann. Performance Interactions Between P-HTTP and TCP Implementations. *Computer Communications Review*, April 1997.

[14] J. C. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, 1995.

[15] Hybrid Networks, Inc. http://www.hybrid.com.

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

[17] V. Jacobson and M. Karels. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, August 1990.

[18] B. A. Mah. An Empirical Model of HTTP Network Traffic. In *Proc. Infocom 97*, April 1997.

[19] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgments Options*. RFC, Oct 1996. RFC-2018.

[20] UCB/LBNL/VINT Network Simulator - ns (version 2). http://www-mash.cs.berkeley.edu/ns/.

[21] V. N. Padmanabhan. *Addressing the Challenges of Web Data Transport*. PhD thesis, University of California at Berkeley, September 1998. http://www.cs.berkeley.edu/~padmanab/research/phd-thesis.html.

[22] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique For Speeding Up Web Transfers. In *Proc. Globecom Internet Mini-Conference*, 1998. http://www.cs.berkeley.edu/~padmanab/papers/gi98-unabridged.ps (Unabridged version).

[23] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proc. Second International World Wide Web Conference*, October 1994.

[24] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California at Berkeley, May 1997.

[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.