Speeding Up Short Data Transfers: Theory, Architectural Support, and Simulation Results

Yin Zhang
yzhang@cs.cornell.edu
Cornell University

Lili Qiu
lqiu@cs.cornell.edu
Cornell University

Srinivasan Keshav
keshav@ensim.com
Ensim Corporation

ABSTRACT

Today's Internet traffic is dominated by short Web data transfers. Such a workload is well known to interact poorly with the TCP protocol. TCP uses the slow start procedure to probe the network for bandwidth both at connection start up and upon restart after an idle period. This usually requires several roundtrips and is inefficient when the duration of a transfer is short.

In this paper, we propose a new technique, which we call TCP/SPAND, to speed up short data transfers. In TCP/SPAND, network performance information is shared among many co-located hosts to estimate each connection's fair share of the network resources. Based on such estimation and the transfer size, the TCP sender determines the *optimal* initial congestion window size. Instead of doing slow start, it uses a pacing scheme to smoothly send out the packets in its initial congestion window. We use extensive simulations to evaluate the performance of the resulting system. Our results show that TCP/SPAND significantly reduces latency for short transfers even in presence of multiple heavily congested bottlenecks. Meanwhile, the performance benefit does not come at the expense of degrading the performance of connections using the standard TCP. That is, TCP/SPAND is TCP friendly.

1. INTRODUCTION

Numerous measurements show that Internet traffic is now dominated by short and bursty Web data transfers [13, 19, 21]. Such a workload interacts poorly with TCP, the dominant transport protocol in today's Internet. TCP uses the slow start procedure [14] to probe the available network capacity both at connection start up and upon restart after an idle period. This usually requires several roundtrips. For short Web data transfers, which typically span only a few roundtrips, spending several roundtrips in probing the network state can be very inefficient. Several techniques like P-HTTP have been proposed in the past to speed up short Web transfers. These techniques help, but are not complete solutions.

In this paper, we propose a new technique, which we call TCP/SPAND, to effectively eliminate the slow start penalty for short data transfers. In TCP/SPAND, network performance information is shared among many co-located hosts to estimate each connection's fair share of the network resources. Based on such estimation and the file transfer size, the TCP sender determines the *optimal* initial congestion window size at connection start up and upon restart after an idle period. Instead of doing slow start, it uses a pacing scheme to smoothly send out the packets in its initial window. Once all the packets in the initial window have been paced out, the sender switches back to the behavior of the standard TCP.

We implement TCP/SPAND in the `ns` simulator [24] and use extensive simulations to evaluate its performance. Our results show that TCP/SPAND significantly reduces latency for short transfers even in presence of multiple heavily congested bottlenecks. Meanwhile, TCP/SPAND is TCP-friendly. The significant performance improvement is not at the expense of degrading the performance of the connections using the standard TCP [14]. Therefore, deploying TCP/SPAND at Web servers can provide great performance benefit. Furthermore, our system is designed to be incrementally deployable in today's Internet. It only involves modifications at the server side. All the client side applications can be left untouched.

The rest of the paper is organized as follows. § 2 gives some background about TCP. § 3 overviews the previous work. In § 4, we analytically derive the optimal initial congestion window as a function of certain network path characteristics and the transfer size. § 5 presents the design and implementation of TCP/SPAND, an incrementally deployable architecture that allows us to apply our analytical results to today's Internet. § 6 presents simulation results to evaluate the effectiveness of our approach. We end with concluding remarks and future work in § 7.

2. BACKGROUND

TCP is currently the dominant transport protocol in the Internet and forms the foundation of applications such as Web browsing, E-mail, file transfer, and news distribution. It is a closed-loop flow control scheme, in which a source dynamically adjusts its flow control window in response to implicit signals of network overload. Specifically, a source uses the slow start algorithm to probe the available network capacity by gradually growing its congestion window until congestion is detected or its window size reaches the receiver's advertised window. The slow start is also terminated if the congestion window grows beyond a threshold. In this case, it uses the congestion avoidance to further open up the window until a loss occurs. It responds to the loss by adjusting its congestion window and other parameters.

The success of TCP relies on the feedback mechanism, where it uses a packet loss as the indication to adapt itself through adjusting a variety of parameters, such as the congestion window size ($cwnd$), the slow start threshold ($ssthresh$), the smoothed roundtrip time ($srtt$) and its variance ($rttvar$). However, for a feedback mechanism to work, a connection should last long enough. When transfers are small compared with the bandwidth-delay product of the link, such as file transfers of several hundred kilobytes over satellite links or typical-sized web pages over terrestrial links, it is very likely that we have little or no feedback – loss indications. In this case, the connection's performance is primarily governed by the choice of initial parameters.

In the original TCP [14], the initial values of such parameters are chosen to accommodate a wide range of network conditions. They may not be optimal for a specific network scenario.

3. PREVIOUS WORK

There have been a number of proposals on improving the start up and/or restart performance of TCP connections.

Two typical examples of application level approaches are launching multiple concurrent TCP connections and P-HTTP (persistent HTTP) [29]. Using multiple concurrent connections makes TCP overly aggressive for many environments and can lead to congestive collapse in shared networks [10, 9]. P-HTTP reuses a single TCP connection for multiple Web transfers, thereby amortizing the connection setup overhead, but still pays the slow start penalty. Since the average Web document (including the inline Web objects) is only around 30KB [21], such penalty is significant enough to limit the performance benefit of P-HTTP.

T/TCP [5] bypasses the three-way handshaking by having the sender start transmitting data in the first segment sent (along with the SYN) [1]. In addition, T/TCP proposes temporal sharing of TCP control block (TCB) state, including maximum segment size (MSS), smoothed RTT, and RTT variance [5]. [6] also mentions the possibility of caching the *congestion avoidance threshold* without offering details.

Hoe proposes [16] to use the bandwidth-delay product to estimate the initial $ssthresh$. However, they use a packet-pair [18] like scheme to estimate the network *available bandwidth*, which does not work well for FIFO networks.

Allman *et. al.* propose [2] to increase the initial window (and optionally the restart window) to roughly 4K bytes. There have been various studies in support of [2] such as [3] and [32]. However, since they use a fixed initial window for all connections, the value has to be conservative, and the improvement is still inadequate in situations where the bandwidth delay product is much larger.

TCP control block interdependence [34] specifies temporal reusing of TCP state, including carrying over congestion window information from one connection to the next. Similar to [34], Fast start [26, 27] reuses the congestion window size ($cwnd$), the slow start threshold ($ssthresh$), the smoothed roundtrip time ($srtt$) and its variance ($rttvar$). By taking advantage of temporal locality, both approaches can lead to significant performance gain. On the other hand, as demonstrated in § 4, the optimal initial congestion window size depends on both the *network state* and the *transfer size*. Therefore, directly reusing previous parameters is not optimal. In addition, when network condition changes, such reusing can be overly aggressive. Fast Start [27] addresses the issue by resorting to router support. This is not the final solution at least for today's Internet. Moreover, in both approaches information sharing is limited to within a single host (although the possibility for inter-host sharing is mentioned in [26]).

In summary, TCP start-up performance has received considerable attention. Many proposals help, but are not complete solutions.

4. MINIMIZE LATENCY BY CHOOSING OPTIMAL INITIAL $CWND$

In this section, we show how to determine the optimal initial $cwnd$ (denoted as $cwnd_{opt}$) that minimizes the completion time for a data transfer given the transfer size and the network path characteristics. We first analytically derive $cwnd_{opt}$ for a simple scenario in which the network path characteristics remain unchanged (in § 4.1). We then extend our results to more general scenarios by introducing a technique called the *shift optimization* (in § 4.2).

4.1 Derivation of $cwnd_{opt}$ for a Simple Scenario

We consider a simple network model in which there is only one TCP connection, the network path characteristics remain unchanged for the duration of the connection, and all losses are due to network congestion.

We use the following notations in our derivations:

- Let W_{opt} = *propagation delay * bottleneck bandwidth.*
- Let $W_c = W_{opt} + B$, where B is the buffer size at the bottleneck router.

Clearly, W_{opt} is the number of packets the link can hold on its own, while W_c is the total number of packets that the link and the buffer together can hold.

As we know, the slow start algorithm is designed to probe the available network capacity. The throughput during slow start is very low. More specifically, without delayed ACK, no more than $2^k - 1$ packets can be sent in k RTT's during slow start; with delayed ACK, the throughput is even lower because the congestion window grows more slowly. Therefore, if the network condition is known and stable, we should try to avoid slow start, and enter congestion avoidance directly. This can be achieved by setting the initial $ssthresh$ to be no more than the initial $cwnd$. Therefore, we only need to consider the congestion avoidance phase in order to find $cwnd_{opt}$. Moreover, we only consider the *Reno-style* congestion avoidance because it is the dominant TCP flavor in today's Internet.

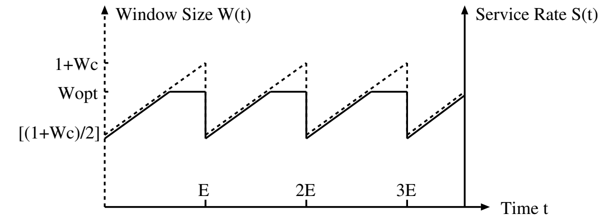

Figure 1: Evolution of service rate and congestion window during congestion avoidance for TCP/Reno. E is the duration of an epoch.

Consider a TCP/Reno connection that starts at time 0 with an initial $cwnd$ of $\lfloor \frac{W_c+1}{2} \rfloor$ and has an infinite amount of data to send. Define $S(t)$ as the service rate it receives at time t, and $W(t)$ as its congestion window size at t. Figure 1 depicts roughly how $S(t)$ and $W(t)$ evolve with time, going through periodical *epochs*. A similar figure also appeared in [15]. (For simplicity, we ignore the time required by the fast retransmission algorithm to wait for 3 duplicated ACK's.)

Let

E: the duration of an epoch
F_E: the number of packets sent during an epoch
b: the average number of packets acknowledged by an ACK

Close inspection of the congestion window evolution allows us to estimate F_E: During each epoch, the congestion window size starts from $\lfloor \frac{W_c+1}{2} \rfloor$ and increases linearly in time, with a slope of $\frac{1}{b}$ packets per roundtrip time. When the window size reaches $W_c + 1$, the sender injects $W_c + 1$ packets into the network. Since the network can hold at most W_c packets, the last packet will get dropped. After sending another W_c packets (corresponding to the W_c ACK's for those packets that are not dropped), the sender detects the loss by 3 duplicated ACK's and recovers from it through fast retransmission. Then the window drops back to $\lfloor \frac{W_c+1}{2} \rfloor$ and a new epoch begins. Therefore, the total number of packets sent during each epoch is:

$$F_E = b * (\sum_{x=\lfloor \frac{W_c+1}{2} \rfloor}^{W_c} x) + 2 * W_c + 1 \qquad (1)$$

For a given transfer with size F, choosing $cwnd_{opt}$ is essentially the problem of fitting a block with size F in the $S(t)$ curve to minimize the transfer time. More precisely, we want to find $s \in [0, E)$ to minimize T, which satisfies $\int_s^{s+T} S(t)dt = F$. (We can then choose $W(s)$ as our $cwnd_{opt}$.) For this problem, we have the following theorem:

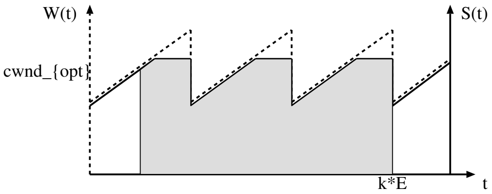

Figure 2: Minimize completion time by having the transfer end at some epoch boundary.

THEOREM 1. *As illustrated in Figure 2, we can minimize the completion time for a given transfer with size F by choosing the initial cwnd appropriately so that the transfer ends at an epoch boundary, i.e., at time $k * E$, for some integer k.*

The detailed proof for Theorem 1 can be found in [37]. Below we give some intuition behind the theorem.

Consider a transfer that starts at time $s \in [0, E)$ with a completion time of $T \leq E$. Clearly, there are only two cases: (i) $s + T \leq E$, and (ii) $s + T > E$. As illustrated in Figure 3, by comparing the area of the shaded regions, it is evident that in both cases the amount of data transfered during interval $[E - T, E]$ is no less than the data transfered in $[s, s + T]$, which is the transfer size. This means if we can have the original transfer end at time E, the completion time does not increase.

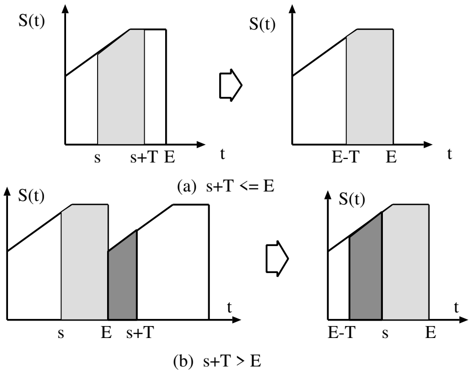

Figure 3: Fixing the width of the shaded region to $T \leq E$, we can maximize the area of the region by having the region end at E.

Based on Theorem 1 we can further derive the value of $cwnd_{opt}$. The result is summarized in the following theorem:

THEOREM 2. *The optimal initial cwnd ($cwnd_{opt}$) that minimizes the completion time of a given file transfer with size F can be determined as the largest integer m ($\leq W_c + 1$) satisfying*

$$b * \sum_{i=m}^{W_c} i + (2 * W_c + 1) \geq F \bmod F_E \qquad (2)$$

where F_E is defined in Equation 1.

The derivation of Theorem 2 is given in [37].

Note that the $cwnd_{opt}$ in Theorem 2 is derived for the case in which the transfer ends at some epoch boundary. But ending at an epoch boundary means the connection has to experience a loss during its final epoch. Although in theory, the loss can be detected and recovered very fast through fast retransmission, in reality it may still be desirable to avoid such an additional loss. This can be achieved by having the transfer end with its $cwnd$ equal W_c instead of $W_c + 1$. Accordingly, the initial $cwnd$ becomes the largest integer m ($\leq W_c$) satisfying

$$b * \sum_{i=m}^{W_c} i \geq F \bmod F_E \qquad (3)$$

4.2 Shift Optimization

We can prove that the $cwnd_{opt}$ given by Theorem 2 minimizes the *integer* number of roundtrips required for the given transfer. The *only* assumption we need to make is that W_c remains unchanged throughout the duration of the connection.

Of course, assuming a constant W_c is unrealistic in the real world. But fortunately, we can relax this assumption through a technique which we call the *shift optimization*. More specifically, for $cwnd_{opt}$ determined by Equation 3, if $b * (\sum_{i=cwnd_{opt}}^{W_c} i)$ is greater than $(F \bmod F_E)$, we can reduce the initial $cwnd$ without increasing the integer number of required roundtrips by shifting the entire transfer in Figure 2 towards left.

The exact amount of reduction in $cwnd$ can be estimated as follows: Suppose we reduce the initial $cwnd$ by Δ. Since it takes roughly $b * (W_c - cwnd_{opt} + 1)$ roundtrips for $cwnd$ to grow from $(cwnd_{opt} - \Delta)$ up to $(W_c - \Delta)$, the total amount of transfered data is reduced by around $\Delta * (b * (W_c - cwnd_{opt} + 1))$. Such reduction should be no more than $b * \sum_{i=cwnd_{opt}}^{W_c} i - F \bmod F_E$ in order to keep the same integer number of required roundtrips. This gives us

$$\Delta = \lfloor \frac{b * \sum_{i=cwnd_{opt}}^{W_c} i - F \bmod F_E}{b * (W_c - cwnd_{opt} + 1)} \rfloor \qquad (4)$$

Clearly, the shift optimization doesn't increase the integer number of required roundtrips. It does slightly increase the completion time by not more than one roundtrip time. However, we believe such marginal overhead is acceptable because in return we can have a smaller and thus *safer* initial $cwnd$. More importantly, as we will soon demonstrate in this section, the shift optimization makes our algorithm less sensitive to the exact value of W_c.

A simple example of the shift optimization is illustrated in Figure 4. In this example, we have $b = 1$, $W_c = 10$, and transfer size $F = 11$. From Equation 3, we obtain $cwnd_{opt} = 9$, which results in a completion time of 2 RTTs (more precisely, more than 1 but less than 2 full RTT's). After the shift optimization, we get a much smaller initial $cwnd$ of 5, while the completion time becomes 2 full RTTs, which is only slightly larger than before optimization.

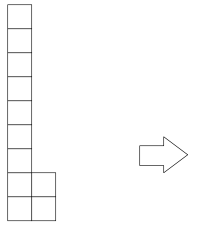

Figure 4: A simple example of the *shift optimization*. ($b = 1$. $W_c = 10$. Transfer size $F = 11$.)

To further demonstrate the effect of the shift optimization, we keep the transfer size to be 30 packets, and plot the computed initial $cwnd$ with and without the shift optimization as a function of W_c. The results are summarized in Figure 5.

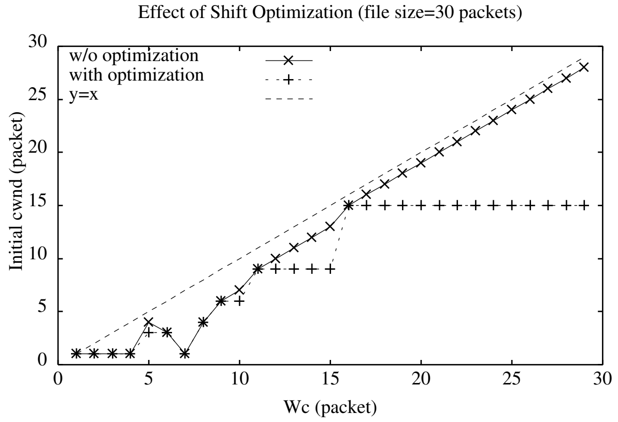

Figure 5: The effect of the *shift optimization*.

We make two observations: First, with the shift optimization, the initial $cwnd$ is more conservative than directly reusing W_c (corresponding to curve $y = x$ in the figure). Second, with the shift optimization, the computed $cwnd$ is insensitive to the value of W_c, especially when W_c is large. Multiple W_c can result in the same initial $cwnd$. Such insensitivity allows us to compute a near optimal initial $cwnd$ even if W_c varies over time or if accurate estimation for W_c is not available.

We have demonstrated that the shift optimization makes the choice of initial $cwnd$ both conservative and insensitive to the value of W_c. Meanwhile, the latency is only increased by a small amount (less than an RTT). *In the rest of this paper, we will use $cwnd_{opt}$ to refer to the optimal initial $cwnd$ after the shift optimization.*

So far we have only considered non-shared networks, we can extend our results to shared networks by redefining W_c as a connection's *share* of the available network resources, which can be estimated as one segment smaller than the congestion window size before the TCP sender detects a loss during congestion avoidance. Measurement study of Internet traces shows that the WAN performance is reasonably stable over terms of several minutes; meanwhile, *nearby* hosts experience similar or identical throughput performance within a time period measured in minutes [25, 7, 31]. Such level of stability suggests sharing performance information both temporarily and spatially (across many co-located hosts) can help to more accurately determine network performance, in particular, a connection's *fair share* of network resources (W_c).

5. TCP/SPAND: SYSTEM DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of TCP/SPAND, a system that allows applications like Web servers or FTP servers to effectively avoid the slow start penalty by applying the theory we develop in § 4. Our design only involves modifications to the server side. All the client applications can be left untouched. This makes TCP/SPAND incrementally deployable in today's Internet.

5.1 System Architecture

Our design of TCP/SPAND, similar to SPAND [31, 30], uses a *performance gateway* that monitors all traffic entering and leaving an organization's network. For each destination network, the gateway gathers network performance information from all the active TCP flows to that destination. It then aggregates such information to estimate the current network state.

The gateway needs to track two types of performance information, which are described below:

- W_c, which reflects the network resources available to a TCP flow. It is required for choosing the optimal initial $cwnd$ as shown in § 4. W_c can be estimated as one segment smaller than the current congestion window size when a TCP sender detects a loss during congestion avoidance. For a short data transfer, it is possible that entire transfer completes without experiencing any loss. In this case, its W_c can be estimated as the congestion window size when the connection terminates.
- The *roundtrip time (RTT)* information. It is used to determine the initial rate for smoothly sending out the packets in the initial congestion window, which we will discuss in detail in § 5.2.5. It is also useful for determining the initial TCP timeout value.

At connection start up or upon restart after an idle period, the application sitting on top of a TCP sender (e.g., a Web server) extracts the current estimation of W_c and RTT from the performance gateway. It then computes $cwnd_{opt}$ as described in § 4 based on such estimation, as well as the transfer size, which is locally available. It then uses a `setsockopt` system call to initialize the parameters such as the $cwnd$ and $srtt$ for the underlying TCP sender. Instead of doing slow start, the TCP sender directly enters congestion avoidance. (This can be achieved by setting $ssthresh$ to be no more than $cwnd$). Meanwhile, the TCP sender uses a pacing scheme to smoothly send out the packets in the initial window. Once all the packets in the initial window have been paced out, it switches back to the behavior of the standard TCP.

5.2 Implementation Issues

There are a number of questions we need to answer in order to actually implement TCP/SPAND:

- What is the right scope for information sharing and aggregation? That is, among which TCP flows should the performance information be shared and aggregated?
- How does the performance gateway collect performance information for active TCP flows?
- After collecting the performance information, what algorithms should the performance gateway use to aggregate the performance information and estimate the current network state?

- How can the applications sitting on top of TCP extract the current estimation of W_c and RTT from the performance gateway?
- What pacing scheme should the TCP senders use to send out packets in the initial congestion window? How can such a scheme be implemented?

In the remainder of this section, we discuss the potential solutions as well as different implementation strategies for these problems in turn.

5.2.1 Determining the Scope for Sharing and Aggregation

The first problem we discuss is how to determine the right scope for information sharing and aggregation. Ideally, we would like to share performance information among flows that share the same bottleneck router at the same time. Unfortunately, in the current Internet architecture, it is very difficult to determine which network flows share the same bottleneck, because there is no easy way to determine where a packet was dropped.

To get around such difficulty, we decide to use conservative approximations based on the destination IP address. Two possible approximations have been proposed in [30]: The *host locality*, that is, flows that share the common destination IP address; and the *network locality*, that is, flows that share the same destination network. We choose to use the latter, which allows more sharing.

Note that it is somewhat difficult to determine the network address from an IP address because the length of the network part can vary. We use a simple heuristic which assumes that IP addresses sharing the most significant 24 bits belong to the same network [30].

Below we evaluate the accuracy of the 24-bit heuristic using access logs recorded at MSNBC news site [23], one of the busiest Web sites in the Internet today. Our traces are from busy hours (from 9:00am to noon) on three consecutive weekdays (from Tuesday, August 03 1999 to Wednesday, August 05 1999). They consist of 10,688,728 HTTP requests to MSNBC, with requests for inline images excluded.

For each client IP address, we use reverse DNS lookup to resolve its host name, and take the last two segments of the host name as its domain name. We then report how often client IP addresses sharing the most significant 24 bits get resolved to different domain names. Altogether there are 656,559 IP addresses in the access logs, among which 475,803 (72.48%) can be successfully resolved to host names via reverse DNS lookup. Of those 475,803 IP addresses, there are 106,669 distinct 24-bit subnet addresses, and only 6715 (6.3%) subnet addresses contain IP addresses that are resolved to different domain names. This demonstrates the high accuracy of the 24-bit heuristic.

5.2.2 Collecting Performance Information

The second important problem is how the performance gateway collects performance information from active TCP flows. There are several possible implementation strategies for this problem.

First, the TCP senders can record the performance information as socket state variables. The application sitting on top of TCP can periodically get such information using the `getsockopt` system call and then report to the performance gateway by sending a special *performance report* packet. This is similar to the approach used in the original SPAND system.

Alternatively, it is possible to modify TCP so that a TCP sender piggybacks the performance information in its normal outbound data packets by introducing a new TCP option. When the performance gateway captures such packets, it can extract the performance information it needs. The TCP receivers can simply ignore this option. The bandwidth and processing overhead for the new TCP option is unlikely to cause a performance concern because the performance information doesn't have to be reported very frequently. However, the TCP senders may need to negotiate the option in advance in order to interoperate with existing implementation of TCP receivers [17]. A similar approach for piggybacking the performance information is to steal bits from normal IP headers [33]. This approach doesn't require any TCP option negotiation and can be implemented with only sender side modifications.

As a third alternative, the performance gateway itself can infer the performance information by passively monitoring all the traffic entering and leaving the organization's network and then reconstructing the TCP protocol state. This is similar to [22]. Compared to the first two approaches, the passive approach has the advantage that it doesn't consume any extra bandwidth and doesn't require any modification to the sender's protocol stack.

5.2.3 Information Aggregation

After collecting the performance information, the performance gateway needs to aggregate such information to accurately estimate the current network state.

Currently, we use a very simple sliding window averaging algorithm to aggregate the performance information. More specifically, the performance gateway keeps a sliding window of S minutes in duration. It uses the average of all values in the past S minutes as the estimation for current W_c and RTT.

In case there is not enough performance information in the sliding window, the performance gateway simply informs the TCP senders to do slow start. As part of our future research, we are interested in exploring the possibility of exponentially decaying the estimation for W_c in this case.

The only control parameter in our algorithm is S, the size of the sliding window. The choice of S involves some tradeoffs: On one hand, we want S to be as large as possible in order to maximize sharing; on the other hand, a large S means the performance gateway needs to keep a large amount of state; in addition, the choice of S needs to match the level of stability reported in the literature.

Currently, we set S to 5 minutes. It clearly matches the level of stability reported in [25, 7, 31], which is a few minutes. Below we use MSNBC traces to demonstrate that a 5-minute sliding window can achieve significant sharing while only requiring moderate amount of state kept by the performance gateway.

Figure 6 shows the cumulative distribution of the time between two consecutive requests from the same client network. (We use the 24-bit heuristic described earlier to determine the client network address.) As we can see, around 90% of the time, the time elapse between two consecutive requests from the same client network is below 5 minutes. This suggests that with a 5-minute sliding window, most Web transfers are able to benefit from the congestion information accumulated by the previous transfers.

We also assess the amount of state the performance gateway needs to keep. Since the performance gateway keeps state for each destina-

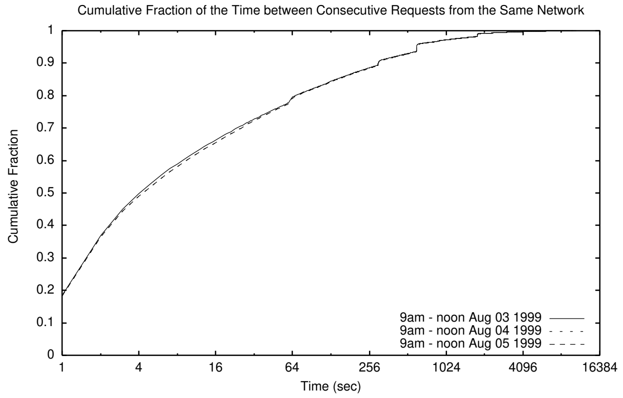

Figure 6: Cumulative distribution of the time between two consecutive requests from the same network. The 24-bit heuristic is used to determine the network address from an IP address.

tion network, the total amount of required state is proportional to the number of different destination networks showing up in a 5-minute interval. For each request in the MSNBC traces, we count the number of different client networks appeared in the next 5 minutes. We then plot the cumulative distribution for all these numbers in Figure 7. From the figure it is evident that the performance gateway only needs to keep state for 15,000 to 25,000 different destination networks even during busiest periods, which can be easily handled by modern computers.

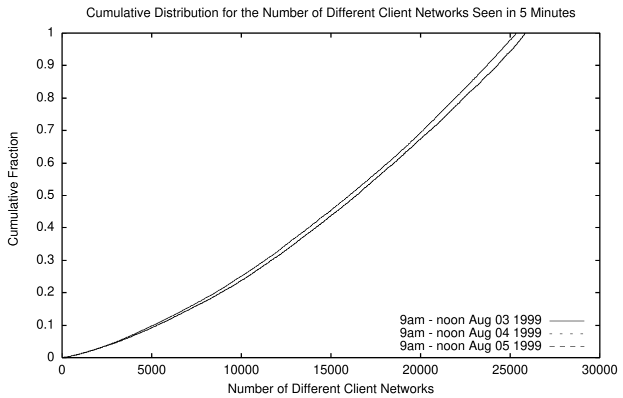

Figure 7: Cumulative distribution for the number of different client networks seen in 5 minutes.

5.2.4 Retrieving Current Estimation of Network State

There are at least two possible ways for the application to retrieve the current estimation of W_c and RTT:

First, similar to SPAND, the application can explicitly query for such information by sending a special *performance inquiry* packet to the performance gateway.

Alternatively, when the performance gateway infers that some application needs the current estimation of W_c and RTT, it can actively *push* such information to the host on which the application runs.

There are several simple heuristics that can be used by the performance gateway to determine if an application needs such estimation. For example, when the performance gateway captures an incoming file transfer request (e.g. an HTTP GET request), or even simpler, when it captures an inbound packet for a new TCP connection, or for a connection that has been idle for a while, it can safely infer that the corresponding TCP flow needs the estimation. These heuristics can be implemented very easily, especially when the gateway already uses the Windmill approach [22] to collect performance information through passive monitoring and protocol reconstruction.

5.2.5 Pacing

Lastly, we discuss pacing. In TCP/SPAND, the TCP senders don't go through the slow start procedure after initializing its $cwnd$. However, the $cwnd_{opt}$ derived in § 4 can be potentially large. Sending out all the packets in the initial congestion window back to back is clearly unacceptable because it can result in a large burst overflowing the network bottleneck buffer.

A natural solution to this problem is to have the TCP sender use a pacing scheme to smoothly send out all the packets in the initial window. Once all packets in the initial window have been sent, the sender can switch back to the behavior of the standard TCP.

There are several pacing schemes proposed in the literature [8, 27, 35]. Here we introduce an alternative scheme based on leaky-bucket. In this scheme, a TCP sender uses a leaky bucket (more specifically, a token bucket) to shape its outgoing traffic. When the sender has a packet to send, it first checks the token bucket. If there are sufficient tokens, the packet is sent immediately; otherwise, it is delayed using a fine-grained timer until there are enough tokens. The depth of the token bucket can be configured to limit the maximum burstiness of the outgoing traffic. It is set to 4 segments in our simulations. The token filling rate is set to $\frac{cwnd}{srtt}$ so that the average sending rate is no more than $cwnd$ packets per roundtrip.

In order to implement the leady-bucket based pacing scheme, we need a fine-grained timer to reschedule a segment for later transmission in case there are no sufficient tokens. Meanwhile, since the token filling rate is inversely proportional to $srtt$, we need a relatively accurate estimation of RTT, which can not be achieved with the 200 ms or 500 ms timer granularity in the standard TCP. In our simulations, we use a 50 ms timer. If the TCP $timestamp$ option is available, it can be used to further improve the accuracy of RTT estimation.

There is evidence to suggest that the overhead of software timers is not likely to be significant with modern processor technologies. ([12] reports an overhead of the order of a few microseconds.) Moreover, the timer overhead is unlikely to be a significant addition to the cost of taking interrupts and processing ACK's that goes with ACK clocking [27].

5.3 Implementation Status

Currently, we have implemented TCP/SPAND in the `ns` network simulator [24]. Our implementation is based on TCP NewReno [16, 11], a variant of TCP that uses partial new ACK information to recover from multiple packet losses in a window at the rate of one per RTT. As described above, the performance gateway uses a 5-minute sliding window to aggregate the performance information. For simplicity, we assume that the communication between the performance gateway and the other hosts in the organization's network is instantaneous. This is reasonable, because compared to the large delay for a WAN connection, the communication latency in a LAN environment is negligible.

6. SIMULATION RESULTS

In this section, we use extensive simulations in the `ns` network simulator to study the performance of TCP/SPAND. We first discuss our simulation topology and experiment methodology, and then present detailed results for various simulation scenarios.

6.1 Simulation Topology

In our simulations, we use single-bottleneck topologies to uncover and illuminate the important issues, and more realistic multiple-bottleneck topologies to evaluate the performance of TCP/SPAND in *real-world* scenarios.

The single-bottleneck topology is shown in Figure 8. One or more bursty connections are established between a subset of the sources on the left and sinks on the right. The bottleneck buffer is 10 KB. The bottleneck router uses FIFO scheduling and drop-tail buffer management. All non-bottleneck links have 10 Mbps capacity and 1 ms one-way propagation delay. We consider three scenarios shown in Table 1.

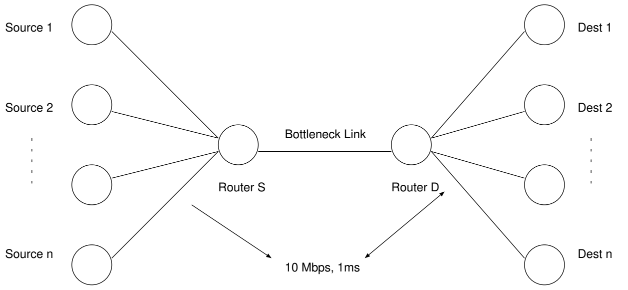

Figure 8: Single-bottleneck topology. Bottleneck buffer is 10 KB. The settings for bottleneck link are summarized in Table 1.

The multiple-bottleneck topology is illustrated in Figure 9. In this topology, a set of M user flows traverse a congested network path that consists of K hops. Cross traffic is generated at each intermediate router R_i ($i = 1, 2, \dots, K$) from C cross-traffic sources. Each router has 10 KB buffer and uses FIFO scheduling and drop-tail buffer management. All links other than those between adjacent routers have 10 Mbps capacity and 1 ms one-way propagation delay. As for the links between adjacent routers, we consider two scenarios summarized in Table 2, which roughly correspond to Scenario 1 and 2 for the single-bottleneck topology. (Note that the aggregated buffer size is larger in the multiple-bottleneck scenario.)

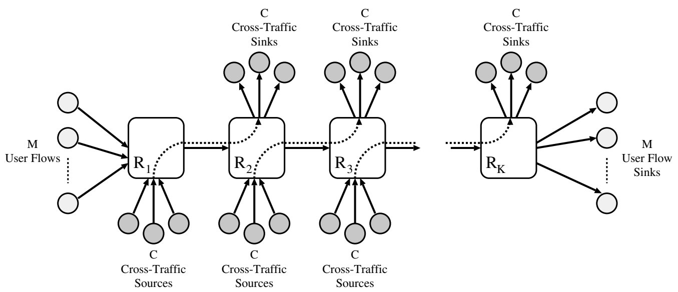

Figure 9: The multiple-bottleneck topology. Bottleneck buffer is 10 KB. The settings for bottleneck link are summarized in Table 2.

For the interest of brevity, we only include the results for Scenario 1 and Scenario 4 in this paper. Interested readers can refer to [37] for results of the other scenarios. In general, the performance gain using TCP/SPAND is higher when each connection share of W_c is large. In other words, TCP/SPAND yields even higher performance benefit for the scenarios omitted here.

6.2 Experiment Methodology

We set up our experiments in the following way to mimic the behavior of Web transfers: Each experiment consists of 40 rounds. In each round every sender transfers one file with start time uniformly distributed around a central point by a time interval (denoted as *jitter*). Between each round there is 120 seconds of idle time.

We use the average completion time of all file transfers in an experiment as our performance metric. For each simulation configuration, we report the mean of 10 runs of an experiment. We also looked at the variation, but since it is very small compared to the mean, we don't report it here.

We compare the performance of TCP/SPAND with the following four variants of TCP's:

- Reno with slow start restart (reno-ssr): TCP/Reno which enforces slow start when restarting data flow after an idle period.
- Reno without slow start restart (reno-nssr): TCP/Reno which reuses the prior congestion window upon restarting after an idle period (This is the scheme used in SunOS.).
- NewReno with slow start restart (newreno-ssr): TCP/NewReno with restart behavior similar to reno-ssr.
- NewReno without slow start restart (newreno-nssr): TCP/NewReno with restart behavior similar to reno-nssr.

The maximum window size of all TCP connections in our simulations is set to 100 KB. The TCP segment size is set to 1 KB. Moreover, in order to remove the performance difference due to different timer granularities, all TCP flavors use the same timer granularity of 50 ms unless otherwise specified. Finally, just like in [26, 27], all the TCP protocols in our experiment use one-way connections instead of two-way connections. That is, TCP classes derived from `TcpAgent` are used instead of those derived from `FullTcpAgent` in the `ns` simulator. Consequently, there is no overhead of 3-way handshaking at connection setup. We believe removing such connection setup overhead is necessary for us to better understand the performance impact of the slow start procedure, which is the major focus of TCP/SPAND. Avoiding connection setup overhead is orthogonal to avoiding slow start penalty and has already been well studied in the literature. For example, P-HTTP [29] can effectively amortize such overhead across multiple transfers.

6.3 Performance Evaluation on the Single-Bottleneck Topology

In this section, we evaluate the performance of TCP/SPAND on the single-bottleneck topology illustrated in Figure 8. We examine the multiple-bottleneck topologies in the following section.

6.3.1 Performance evaluation with many concurrent web transfers

6.3.1.1 Varying the number of competing connections

First, we compare performance as the number of competing TCP connections varies from 1 to 30 while transfer size is kept at 30 KB, which is the average Web transfer size [21]. Five telnet sessions compete with the main flows to help avoid deterministic behavior. The inter-arrival times for telnet sessions are drawn from the "tcplib" distribution as implemented in `ns`.

Figure 10 shows the results for Scenario 1 in Table 1. As shown in the figure, TCP/SPAND leads to significant reduction in average completion time. Compared with reno-ssr and newreno-ssr, TCP/SPAND reduces latency by more than 50% in most cases, which means a 100% improvement in average data rate. Compared to reno-nssr and newreno-nssr, the completion time reduction is smaller but still significant, over 20% in most cases. Notice that reno-nssr and newreno-nssr are well-known to be overly aggressive but still perform considerably worse than TCP/SPAND. There are two major reasons for

Scenario	Bandwidth	Link Delay	Descriptions
1	1.6 Mbps	50 ms	typical terrestrial WAN links with close to T1 speed
2	1.6 Mbps	200 ms	typical geostationary satellite links with close to T1 speed
3	45 Mbps	200 ms	typical geostationary satellite links with T3 speed

Table 1: Different simulation scenarios for the single-bottleneck topology

Scenario	Bandwidth	Link Delay	Descriptions
4	1.6 Mbps	50/$(K-1)$ ms	roundtrip time (RTT) similar to Scenario 1
5	1.6 Mbps	200/$(K-1)$ ms	roundtrip time (RTT) similar to Scenario 2

Table 2: Different simulation scenarios for the multiple-bottleneck topology. $K \geq 2$ is the number of hops.

this: first, directly reusing previous $cwnd$ is not optimal; second, sending all the packets in initial window at once is usually too bursty and can cause more losses than TCP/SPAND.

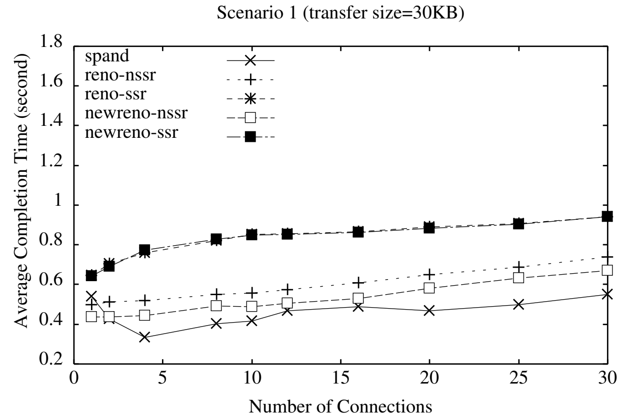

Figure 10: Performance comparison for different number of connections. Bottleneck link has setting of Scenario 1 (defined in Table 1). In each round, file transfers start within 10 seconds (i.e. jitter=10 sec). 5 telnet sessions are used to avoid deterministic behavior.

6.3.1.2 Varying the transfer size

Now we compare performance as the transfer size varies. As shown in Figure 11, TCP/SPAND reduces the completion time over a wide range of transfer sizes. In percentage terms, the improvement decreases as the transfer size increases. This is what we would expect, because avoiding the slow start penalty has a much bigger impact on small transfers than on larger ones.

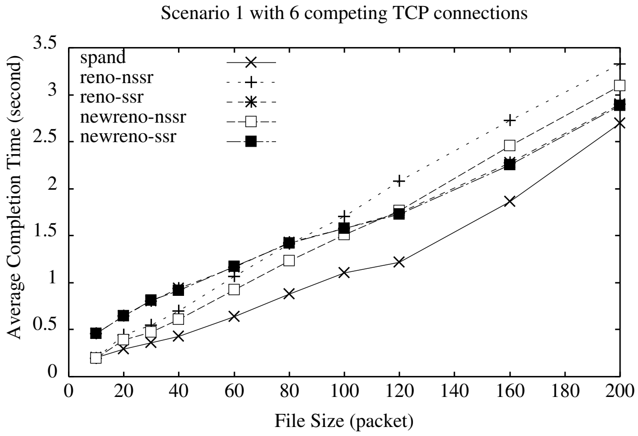

Figure 11: Performance comparison for different transfer sizes. 6 TCP connections competes for the bottleneck link with setting of Scenario 1 (defined in Table 1). In each round, file transfers start within 10 seconds (i.e. jitter=10 sec). 5 telnet sessions are used to avoid deterministic behavior.

6.3.2 Performance evaluation with ON/OFF UDP flows as cross traffic

It has been reported in [28] that WWW-related traffic tends to be self-similar in nature. In [36], it is shown that self-similar traffic may be created by using several ON/OFF UDP sources whose ON/OFF times are drawn from heavy-tailed distributions such as the Pareto distribution. So in this section, we evaluate the performance of TCP/SPAND with ON/OFF UDP flows as cross traffic.

As before (in § 6.3.1), we first evaluate performance as the number of competing connections varies while the transfer size is still kept at 30KB. The simulation results are illustrated in Figure 12. TCP/SPAND reduces latency by 35% to 65% compared with reno-ssr and newreno-ssr, which means a 60% to 200% improvement in average data rate. Compared with reno-nssr and newreno-nssr, the latency reduction is around 25% to 50%.

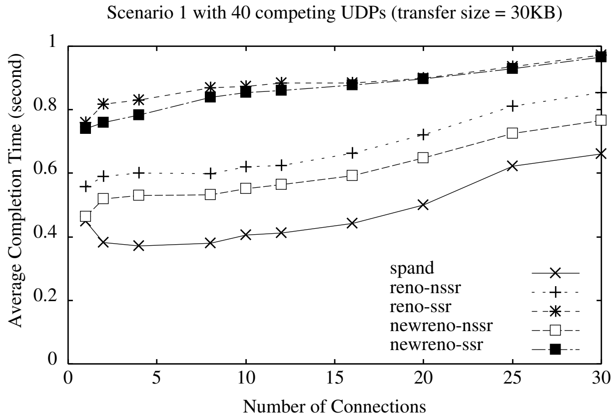

Figure 12: Performance comparison for different number of connections with 40 ON/OFF UDP flows as cross traffic. The ON/OFF times of the UDP sources are drawn from Pareto distributions with the "shape" parameters set to 1.2. The mean ON time is 1 second and the mean OFF time is 2 seconds. During ON times, the sources transmit with a rate of 12 Kbps. In both (a) and (b), the jitter for transfer start time in each round is 10 seconds.

Figure 13 shows the result of varying the transfer size and keeping the number of connections constant. Again, TCP/SPAND reduces completion time over a wide range of transfer sizes.

6.4 Performance Evaluation on the Multiple-Bottleneck Topology

In this section, we evaluate the performance of TCP/SPAND when the underlying network path is heavily congested and has multiple bottlenecks.

We use the multiple-bottleneck topology illustrated in Figure 9. The number of hops (K) is fixed to 5. At each intermediate router R_i

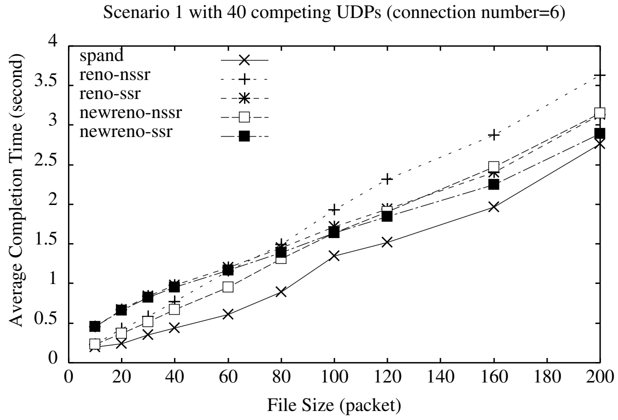

Figure 13: Performance comparison for different transfer sizes with UDP as cross traffic. Settings are the same as in Figure 12.

($i = 1, 2, 3, 4$), the cross traffic is generated from $C = 40$ ON/OFF UDP connections. Just like in § 6.3.2, the ON/OFF time of the UDP sources are drawn from Pareto distributions with the "shape" parameters set to 1.2. The mean ON time is 1 second and the mean OFF time is 2 seconds.

6.4.1 Performance evaluation with 12 Kbps UDP Sources

We first evaluate performance when each ON/OFF UDP source generates cross traffic at a rate of 12 Kbps during ON time, which is the same as for the single-bottleneck case in § 6.3.2.

As before, we first keep the transfer size at 30 KB and vary the number of competing connections. As illustrated in Figure 14, TCP/SPAND leads to significant reduction in completion time: 30% to 65% over reno-ssr and newreno-ssr, 25% to 55% over reno-nssr and newreno-nssr. Such performance improvement is comparable to the single-bottleneck case shown in Figure 12.

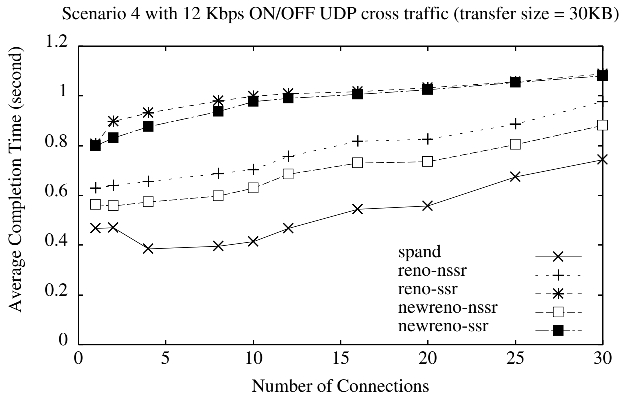

Figure 14: Performance comparison for different number of connections with 12 Kbps ON/OFF UDP flows as cross traffic. The jitter for transfer start time in each round is 10 seconds.

We then fix the number of competing connections to 6 and vary the transfer size. It is evident from Figure 15 that TCP/SPAND again achieves significant performance improvement similar to the single-bottleneck case shown in Figure 13.

6.4.2 Performance evaluation with 48 Kbps UDP Sources

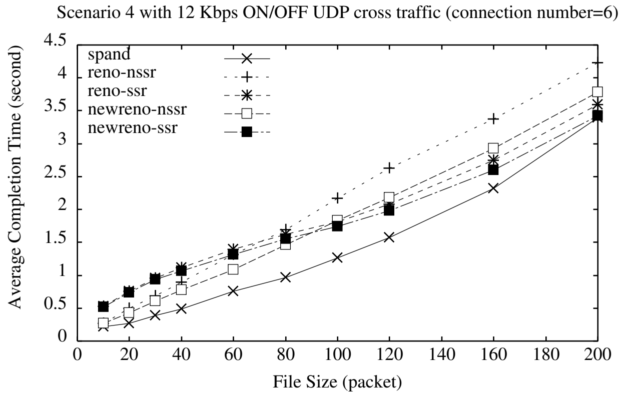

Figure 15: Performance comparison for different transfer sizes with 12 Kbps ON/OFF UDP flows as cross traffic. Settings are the same as in Figure 14.

To further investigate the performance of TCP/SPAND under heavy congestion, we increase the sending rate of the UDP sources during ON time to 48 Kbps and redo all the experiments. The simulation results are summarized in Figure 16 and Figure 17.

From the significant increase in the completion time, it is evident that the underlying network path is highly congested. But even under such heavy congestion, TCP/SPAND consistently out-performs the other TCP flavors. Of course, the performance improvement decreases. This is not surprising, because the impact of the initial congestion window becomes less significant as the network becomes highly congested.

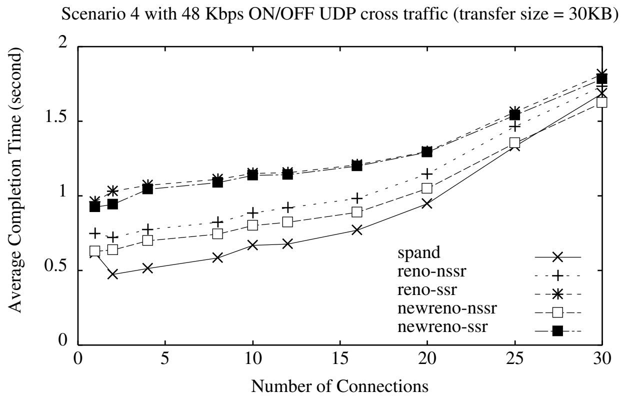

Figure 16: Performance comparison for different number of connections with 48 Kbps ON/OFF UDP flows as cross traffic. The jitter for transfer start time in each round is 10 seconds.

6.5 TCP Friendliness

In this section, we demonstrate the performance improvement of TCP/SPAND does not come at the expense of degrading the performance of the connections using the standard TCP. In other words, TCP/SPAND is TCP friendly.

We show this by considering a mixture of TCP's on the single-bottleneck topology. One half of the connections use TCP/SPAND, while an equal number use reno-ssr, one of the least aggressive TCP schemes. We then compare their performance with the case in which all connections use reno-ssr. The jitter for transfer start time during each round is set to 0.1 second to create maximum contention for the bottleneck bandwidth. Again 5 telnet sessions are introduced to avoid deterministic behavior.

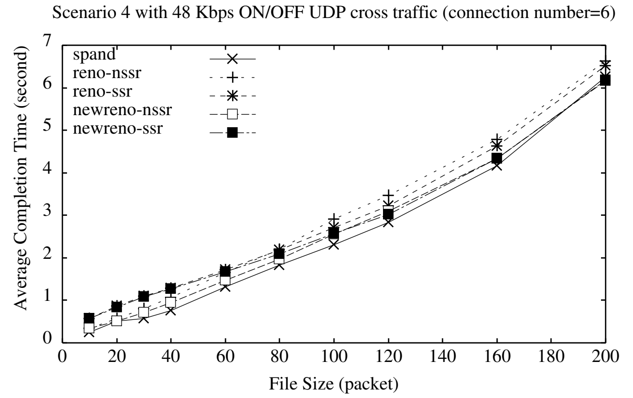

Figure 17: Performance comparison for different transfer sizes with 48 Kbps ON/OFF UDP flows as cross traffic. Settings are the same as in Figure 16.

First, we use 50 ms timer granularity for reno-ssr, which is the same as TCP/SPAND. Figure 18 summarizes the simulation results, where the bottleneck link is T1 link with latency of 50 ms (Scenario 1), and the transfer size is kept at either 30 KB.

From the figure it is evident that the performance of reno-ssr, when mixed with TCP/SPAND, is virtually the same as when all connections use reno-ssr. This demonstrates TCP/SPAND is TCP-friendly even under heavy contention.

Also worth mentioning is that TCP/SPAND performs almost the same as reno-ssr. This is because the jitter is only 0.1 second. In such case, each connection's share of W_c is very small due to heavy contention. Consequently, the optimal initial $cwnd$ is very close to 1, which makes TCP/SPAND behave almost the same as reno-ssr.

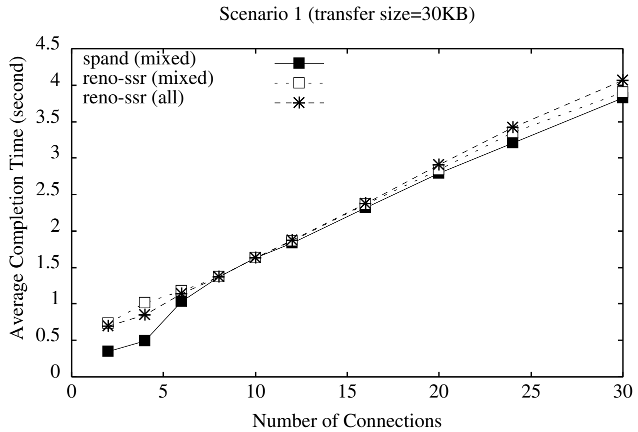

Figure 18: TCP friendliness. 5 telnet sessions are used as background traffic. Transfer start time in each round has a jitter of 0.1 sec. Transfer size is either 30 KB or 100 KB. The timer granularity for both reno-ssr and TCP/SPAND is 50 ms.

We also use 200ms timer granularity for reno-ssr to evaluate the TCP-friendliness of TCP/SPAND. The results are illustrated in Figure 19. We can see from the figure that TCP/SPAND significantly outperforms reno-ssr. Meanwhile, reno-ssr experiences almost no performance degradation in presence of TCP/SPAND.

6.6 Summary of Simulation Results

To summarize, in this section we use extensive simulations to evaluate the performance of TCP/SPAND. TCP/SPAND consistently outperforms reno-ssr, newreno-ssr, reno-nssr, and newreno-nssr in all

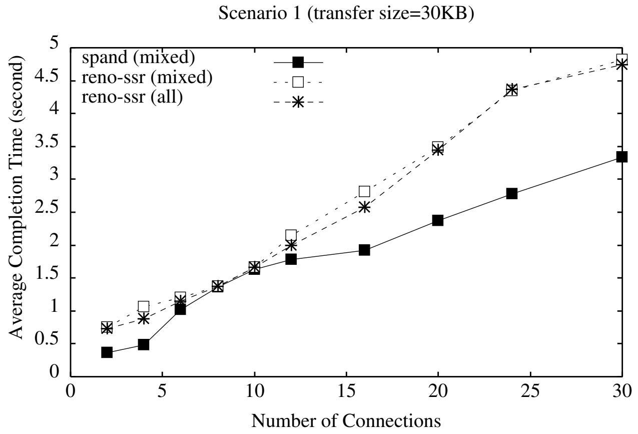

Figure 19: TCP friendliness. The configuration is the same as in Figure 18 except that reno-ssr uses 200 ms timer granularity instead of 50 ms. TCP/SPAND still uses 50 ms timer granularity.

simulation scenarios, even in presence of multiple heavily congested links. The performance benefit is greatest when each connection's share of W_c is large. In such cases, TCP/SPAND can reduce latency by 35% to 65% compared with reno-ssr and newreno-ssr; or by 20% to 50% compared with reno-nssr and newreno-nssr. Meanwhile, such significant performance improvement does not come at the cost of degrading unenhanced TCP connections. We have demonstrated that TCP/SPAND is TCP-friendly even under heavy contention.

There are three major factors that contribute to the good performance of TCP/SPAND. First, by sharing and aggregating performance information among many co-located hosts, the sender can have a quite accurate estimation of current network conditions. Second, the initial $cwnd$ used by TCP/SPAND is chosen based on our theoretical analysis for optimal initial $cwnd$. We also employ the *shift optimization* (described in § 4) to make such choice conservative and insensitive to the accuracy of the estimation of current network characteristics. In this way, TCP/SPAND can utilize available bandwidth efficiently and safely when each connection's share of W_c is large. When the network is heavily loaded, TCP/SPAND tends to be conservative. Particularly, when each connection's share of W_c is less than 4K bytes, the choice of initial $cwnd$ in TCP/SPAND is more conservative than what is proposed in RFC 2414 [2]. Finally, by using a pacing scheme to send out packets in the initial congestion window, TCP/SPAND effectively reduces the burstiness of TCP upon start-up and restart.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we investigate the possibility of speeding up short data transfers by effectively avoiding the slow start penalty. By analyzing the TCP start-up dynamics, we derive the optimal initial congestion window ($cwnd_{opt}$) as a function of the transfer size and a connection's share of network resources (W_c). We then propose an incrementally deployable architecture called TCP/SPAND, which accurately estimates a connection's fair share of network resources by sharing performance information among a large number of co-located hosts. Based on such estimation and the transfer size, a TCP sender can further compute the optimal initial congestion window size. Instead of doing slow start, it directly enters congestion avoidance and uses a pacing scheme to smoothly send out the packets in its initial congestion window. We then do extensive simulations using `ns` simulator to evaluate the performance of the resulting system. Our results show that TCP/SPAND significantly reduces latency for short transfers even in presence of multiple heavily congested bottlenecks. Meanwhile, the performance benefit does not come at the expense of degrading the performance of connections using the standard TCP.

There are a number of directions that we want to further explore in the future. First of all, we plan to implement TCP/SPAND in real systems and evaluate its performance through Internet experiments. Secondly, we want to better understand the impact of pacing on TCP performance, especially for short TCP flows. We are also interested in developing effective techniques for determining which network flows share a bottleneck. Finally, we plan to investigate better algorithms for information aggregation.

8. REFERENCES

[1] M. Allman, S. Dawkins, D. Glover, J. Griner, T. Henderson, J. Heidemann, H. Kruse, S. Ostermann, K. Scott, J. Semke, J. Touch, and D. Tran, "Ongoing TCP Research Related to Satellites," Internet Draft draft-ietf-tcpsat-res-issues-05.txt, Nov. 1998.

[2] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's Initial Window," RFC-2414, Sept. 1998.

[3] M. Allman and C. Hayes, "An Evaluation of TCP with Larger Initial Windows," *ACM Computer Communication Review*, July 1998.

[4] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," *Proc. IEEE INFOCOM '2000*, Mar. 2000.

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

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

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

[8] H. Balakrishnan, V. Padmanabhan, and R. Katz, "The Effects of Asymmetry on TCP Performance," *Proc. ACM/IEEE Mobicom '97*, Sept. 1997.

[9] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements," *Proc. IEEE INFOCOM '98*, Mar. 1998.

[10] S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet," *IEEE/ACM Transactions on Networking*, Aug. 1999.

[11] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC-2582, Experimental, April 1999.

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

[13] S. Gribble and E. Brewer, "System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace," *Proc. 1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, Dec. 1997.

[14] V. Jacobson and M. Karels, "Congestion Avoidance and Control," *Proc. SIGCOMM '88*, Aug. 1988.

[15] D. Heyman, T. Lakshman, and A. Neidhardt, "A New Method for Analyzing Feedback-Based Protocols with Applications to Engineering Web Traffic over the Internet," *Proc. SIGMETRICS '97*, 1997.

[16] J. Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP," *Proc. SIGCOMM '96*, Aug. 1996.

[17] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," RFC-1323, May 1992.

[18] S. Keshav, "A Control-Theoretic Approach to Flow Control," *Proc. SIGCOMM '91*, Sept. 1991.

[19] K. Thompson, G. Miller, and R. Wilder, "Wide-Area Internet Traffic Patterns and Characteristics," *IEEE Network*, 11(6):10-23, Nov. 1997.

[20] T. Lakshman and U. Madhow, "Performance Analysis of Window-Based Flow Control using TCP/IP: the Effect of High Bandwidth-Delay Products and Random Loss," *IFIP Transactions C-26*, *High Performance Networking V*, pp. 135-150, North-Holland, 1994.

[21] B. Mah, "An Empirical Model of HTTP Network Traffic," *Proc. INFOCOM '97*, 1997.

[22] G. Malan and F. Jahanian, "An Extensible Probe Architecture for Network Protocol Performance Measurement," *Proc. SIGCOMM '98*, 1998.

[23] http://www.msnbc.com.

[24] UCB/LBNL/VINT Network Simulator - `ns` (version 2). http://www-mash.cs.berkeley.edu/ns, 1997.

[25] V. Paxson, "Measurements and Analysis of End-to-End Internet Dynamics," PhD thesis, U.C. Berkeley, May 1996.

[26] V. Padmanabhan, "Addressing the Challenges of Web Data Transport," Ph.D. Thesis, UC Berkeley, 1998.

[27] V. Padmanabhan and R. Katz, "TCP Fast Start: A Technique for Speeding Up Web Transfers," *Proc. IEEE Globecom '98 Internet Mini-Conference*, Nov. 1998.

[28] K. Park, G. Kim, and M. Crovella, "On the Relationship between File Sizes, Transport Protocols and Self-Similar Network Traffic," *Proc. ICNP '96*, 1996.

[29] V. Padmanabhan and J. Mogul, "Improving HTTP Latency," *Proc. Second International World Wide Web Conference*, Oct. 1994.

[30] S. Savage, N. Cardwell, and T. Anderson, "The Case for Informed Transport Protocols," *Proc. 7th Workshop on Hot Topics in Operating Systems (HOTOS '99)*, Mar. 1999.

[31] S. Seshan, M. Stemm, and R. Katz, "SPAND: Shared Passive Network Performance Discovery," *Proc 1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, Dec. 1997.

[32] T. Shepard and C. Partridge, "When TCP Starts Up With Four Packets Into Only Three Buffers," RFC-2416, Sept. 1998.

[33] I. Stoica and H. Zhang, "Providing Guaranteed Services Without Per Flow Management," *Proc. SIGCOMM '99*, Sept. 1999.

[34] J. Touch, "TCP Control Block Interdependence," RFC-2140, April 1997.

[35] V. Visweswaraiah and J. Heidemann, "Improving Restart of Idle TCP Connections," Technical Report 97-661, University of Southern California, Nov. 1997.

[36] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson, "Self-Similarity through High Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level," *Proc. SIGCOMM '95*, 1995.

[37] Y. Zhang, L. Qiu, and S. Keshav, "Speeding Up Short Data Transfers: Theory, Architectural Support, and Simulation Results," Technical Report 2000-1799, Department of Computer Science, Cornell University, June 2000.

9. ACKNOWLEDGMENT

Special thanks to Brad Karp, Geoffrey M. Voelker and anonymous NOSSDAV reviewers for their valuable comments. Also thanks to Venkata N. Padmanabhan for providing MSNBC web server trace data.