

Dealing with Short TCP Flows: A Survey of Mice in Elephant Shoes

Janardhan R. Iyengar, Armando L. Caro Jr., Paul D. Amer

Protocol Engineering Lab

Computer and Information Sciences

University of Delaware

{iyengar, acar, amer}@cis.udel.edu

Abstract

Short web transfers that dominate the Internet suffer from TCP's inadequate provisioning for short flows. This paper surveys ten proposals that attempt to solve one or more of the problems of short flows, and suggests general criteria to evaluate them. The proposals range from T/TCP in 1992 to RIO-PS in 2001. We classify the proposals into three categories: (1) those that reduce connection setup/teardown overheads, (2) those that use different network state sharing mechanisms, and (3) those that improve performance during slow start.

1 Introduction

Much of the traffic on the Internet is carried by a small number of large flows (*elephants*), while most of the flows are short in duration and carry a small amount of traffic (*mice*). This phenomenon is attributed to the domination of Internet flows by web data transfers, which are characterized as mice. According to [18], the average web page is 26-32 KB, and a median client "think time" of 15 seconds occurs between initiating successive web page downloads. Such workloads are known to interact poorly with TCP, the transport protocol used for reliable web transfers.

TCP was originally designed for elephants, where a sender probes for the end-to-end available bandwidth in a "slow start" phase, and then spends most of its data transfer operation in a "congestion avoidance" phase. This design causes the short, but dominating mice to suffer in the presence of elephants. A closer look at TCP mechanisms and features brings to light some reasons for such biased behavior:

- TCP employs a three-way handshake to initiate a connection, and a four-way handshake to terminate a connection. These phases amount to a significant portion of a short flow's lifetime, while causing only minor overhead for a long flow.
- TCP probes for the end-to-end available bandwidth by increasing the congestion window (cwnd) in the slow start phase. Most TCP implementations start with a conservative initial cwnd of at most two segments. The lifetime of a short flow may be dominated or contained within the slow start phase [2], thus resulting in suboptimal bandwidth consumption by the flow.

- Each new TCP flow independently probes for the end-to-end available bandwidth between the communicating endpoints, even though other flows may concurrently exist between the same endpoints. Consequently, each new flow goes through the slow start phase unnecessarily due to lack of information sharing with the other concurrent flows, thus each suffering the slow start overhead.
- As a reliable transport protocol, TCP couples congestion control with loss detection and recovery. TCP uses retransmission timeouts and fast retransmits to detect loss. Currently a sender must receive at least three duplicate acks before it triggers a fast retransmit. Short flows in the slow start phase often do not have sufficient traffic to generate three duplicate acks, making timeouts the only loss detection mechanism available to a TCP sender. Balakrishnan et al. [5] report recovery from almost 50% of losses in web flows via timeout. Such timeouts can significantly increase the latency overhead for short flows.
- TCP timeout values are based on round-trip time estimations from a flow's data-ack samples. When a connection is initiated, TCP uses a conservative timeout value due to lack of such samples from the connection. These timeout values are large in practice (as large as three seconds). Loss among the connection establishment control packets (SYN, SYN-ACK) can cause significant increase in the latency overhead for short flows.

In trying to fit a square peg (short flows) in a round hole (TCP), researchers have proposed several techniques – some to change the peg, some the hole. In this paper, we survey the work that has been done in addressing these issues to enhance the performance of TCP mice,¹ and improve the fairness of the system (the Internet) towards them. One contribution of this paper is the following taxonomy that broadly classifies the proposed solutions into three categories:

- Reduce Connection Overhead (Section 2): These techniques reduce or eliminate redundant connection setup/teardown overheads for multiple short TCP connections. This section includes:

Section 2.1 - TCP for Transactions (T/TCP)

Section 2.2 - Persistent HTTP (P-HTTP)

- Sharing Network State Information (Section 3): These techniques use different kinds of network state sharing mechanisms to learn network state from other recent or concurrent connections between the endpoints, and use the information to reduce or eliminate slow start and/or timeout overheads. This section includes:

Section 3.1 - Control Block Interdependence (CBI)

Section 3.2 - TCP Session and TCP Fast Start

Section 3.3 - Congestion Manager (CM)

Section 3.4 - Stream Control Transmission Protocol (SCTP)

Section 3.5 - TCP/SPAND

- Improve Performance During Slow Start (Section 4): These techniques either improve slow start behavior at the endpoints to accommodate short flows, or use network support to preferentially treat short flows. This section includes:

¹Although the proposals are applicable to both “web traffic” and “short flows (mice)”, we use one or the other phrase, depending on the proposal being discussed.

Proposal	When Proposed	TCP Issues Addressed			Deployment Efforts
		3-way handshake	Slow Start BW Probing	Timeouts in Slow Start	

Reduce Connection Overhead

T/TCP	1992	yes	mentions	no	RFC1644. Major OSES (mid-1990's). Security hole. Removed in 1998.
P-HTTP	1994	yes	yes	no	In HTTP/1.1 since 1997

Sharing Network State Information

CBI	1997	no	yes	affects	TCP-INT in 1998. E-TCP in 1999.
TCP Session + TCP Fast Start	1998	no	yes	yes	–
CM	1999	no	yes	affects	RFC3124 (Proposed Standard)
SCTP	2000	yes	yes	affects	RFC2960 (Proposed Standard). Major OSES (26 implementations).
TCP/SPAND	2000	no	yes	affects	–

Improve Performance During Slow Start

TCP w/ Larger Initial Window	1998	no	yes	yes	RFC2414
TCP-SF	2001	no	no	yes	First IETF Draft [20]
RIO-PS	2001	no	affects	yes	–

Table 1: Summary of proposals surveyed

Section 4.1 - TCP with Larger Initial Window

Section 4.2 - Smart Framing for TCP (TCP-SF)

Section 4.3 - RIO with Preferential Treatment to Short Flows (RIO-PS)

Table 1 summarizes the surveyed research according to our taxonomy. We identify three TCP issues that need to be addressed by a proposed solution: (1) does the solution modify TCP's 3-way connection establishment handshake, (2) does the solution modify TCP's bandwidth probing that occurs during slow start, and (3) does the solution modify how TCP uses timeouts as an indication of packet loss. Some proposals may only affect one of these three TCP issues as a side effect, and not address the issue explicitly. In that case, the table denotes "affects". The table also uses "mentions" to denote solutions that mention without details how to address the indicated TCP issue. The table identifies the year each solution was first proposed, which of the three TCP issues each solution addresses, and current deployment efforts.

In conclusion, Section 5 gives an overall comparison of the different proposals from a broad perspective. We also suggest some general criteria to be used while evaluating these proposals as solutions.

2 Reduce connection overhead

This section presents two solutions that reduce or eliminate redundant connection setup/teardown overheads for multiple short TCP connections.

2.1 T/TCP

HTTP is a transaction-oriented (i.e., request-response based) application layer protocol that uses TCP as its underlying reliable transport protocol. Using TCP for such traffic introduces a latency overhead with its three-way handshake connection setup mechanism. Other Internet user programs, application-layer protocols (e.g., remote procedure calls and SIP), and network control and management protocols (e.g., DNS and SNMP) that are transaction-oriented commonly use UDP as their transport protocol. Using UDP requires that those transport layer services not provided by UDP (such as reliability) be built into the application. Such applications demand a transport protocol that provides the reliability of TCP without the latency of its connection establishment.

Braden [8] identified the need for a transport protocol which bridges this gap between UDP and TCP. As a result, *TCP for Transactions (T/TCP)* [11], a backwards-compatible extension to TCP, was proposed. T/TCP allows transactions to benefit from all services provided by TCP, yet complete in the same number of round-trips as UDP. Data segments are overloaded with the SYN and FIN flags, thus overlapping the three-way connection establishment handshake with the beginning of the data transfer [28, 29].

All connections opened, either actively or passively, are assigned a 32-bit connection count (CC) value from a global counter, which gets incremented by 1 each time it is used. Each segment in a T/TCP transaction uses a new TCP option, named CC, to convey the client's CC value for the connection. Hosts that receive an acceptable SYN segment maintain the initiating host's latest CC value in a per-host cache. This per-host cache helps determine the validity of subsequent initiating SYNs from the same host. For each initiating SYN, the CC option value is compared with the cached value for the client. If the received CC is greater than the cached CC, the SYN is new and any data in the segment is immediately delivered up to the serving application. However, a larger or non-existent cached CC value results in the normal TCP three-way handshake. A server responds to an initiating SYN with a SYN-ACK segment that echoes the received CC value in a new option named CCECHO. The three-way handshake overlaps data transfer unless either the client or server has rebooted. In addition, duplicate segments from previous incarnations of the same connection can be identified and rejected based on the CC value in non-SYN segments.

T/TCP also suggests increasing the default initial congestion window (cwnd). Braden [11] points out that TCP is not permitted to send a segment larger than the default size (536 bytes), unless it has received a larger value in an Maximum Segment Size (MSS) option. To avoid constraining requests to this limit, Braden suggests a default of 4096 bytes for cached hosts. If this default is too large and the client experiences losses due to congestion, the client can cache an appropriate cwnd to be used in subsequent transactions to the same server. Allman et al. [3] also consider the advantages and disadvantages of using a larger initial cwnd of 4096 bytes (see discussion Section 4.1).

Discussion

In the mid-90's, T/TCP was implemented in a variety of operating systems including SunOS 4.1.3, FreeBSD 2.0.5, BSD/OS 2.1, and Linux 2.0.32 [1]. T/TCP was being incrementally deployed throughout the Internet at various sites and had the potential to spread quickly. Not only is T/TCP backwards compatible with TCP, but it is also transparent to layers above. In other words, existing TCP applications, such as web browsers and servers, could migrate to T/TCP seamlessly without any changes. Unfortunately in April 1998, Vasim Valejev found a security hole in the protocol that exploits the trust relationship between the two hosts [32]. Nobody could solve this security problem, and the result was that T/TCP support was removed from all operating systems.

2.2 P-HTTP

Padmanabhan and Mogul conducted experiments to quantify the cost of using TCP as the transport mechanism for HTTP [24]. With each HTTP transfer being done over a separate TCP connection, they observed significant overhead for short HTTP transfers, attributed to TCP's connection setup and slow start mechanisms. To amortize these costs over multiple HTTP interactions, they proposed a connection abstraction for HTTP called *Persistent HTTP (P-HTTP)*, which has two main features: persistent connections and pipelined requests. Persistent connections aggregate more than one HTTP transfer over one long-lived TCP connection. The same TCP connection is used for multiple objects within one or more web pages from the same server. This mechanism amortizes the overhead cost of connection setup across multiple object transfers. Also, since multiple objects share a TCP connection, persistence reduces overall latency due to slow start. Figures 1 and 2 illustrate packet exchanges for a non-persistent and persistent connection, respectively.

Even with persistent connections, an HTTP request could only be sent after the previous response was completed. Pipelining requests is a mechanism which allows a client to send requests while previous responses are still in progress. For instance, one inline image can be requested while another image is being received. Mogul [21] and Nielsen et al. [14] show that pipelining can significantly enhance the throughput of web transfers.

Discussion

Though pipelining is specifically applicable to request-response style of transfers, multiple file transfers between the same endpoints can benefit from persistent connections. However, due to TCP's in-order delivery mechanism, pipelining cannot avoid the undesirable side effect of Head-of-Line (HOL) blocking among the web page components [6]. HOL blocking occurs in TCP when a sender transmits multiple TCP segments and a segment is lost. Segments arriving after the lost one must wait in the receiver's queue until the missing segment is retransmitted and arrives correctly. This blockage delays the deliver of data to the application, which increases the user's perceived latency.

P-HTTP has been implemented in HTTP/1.1 since 1997. However, the *slow start re-start* problem described in [16] negates the reduction of slow start latency in persistent connections. Some TCP implementations reinitialize the congestion window to 1 segment, forcing a slow start, whenever data is not sent for the period of one retransmission timeout (RTO). The motivation for this behavior is that after a silence period, a

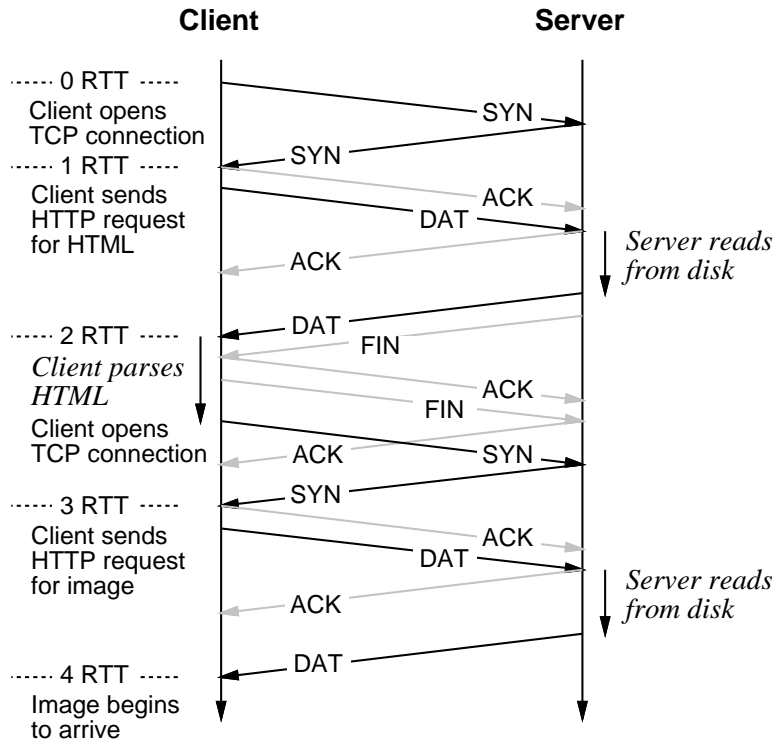


Figure 1: Packet exchanges for HTTP with non-persistence [24]

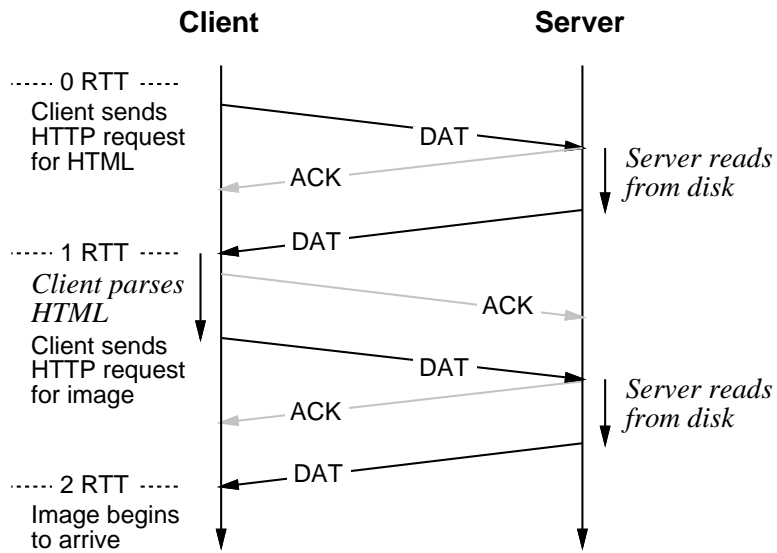


Figure 2: Packet exchanges for HTTP with persistent connections [24]

connection's available bandwidth may have changed and needs to be re-probed. Without re-probing, a large cwnd may allow a burst of packets that disturbs network equilibrium, resulting in congestion and/or packet loss.

Two possible modifications to TCP stacks are mentioned without details in [16]. One suggestion is to decay the cwnd over time, rather than reset it to one. The other suggestion is to use a pacing mechanism after a silence period to control the rate at which packets are introduced into the network. *TCP Fast Start* (discussed in section 3.2) and Visweswaraiah et al. [33] attempt in part to address the same issue.

3 Sharing Network State Information

This section presents five solutions that use different kinds of network state sharing mechanisms to learn the network state from other recent or concurrent connections at the endpoints, and use the information to reduce or eliminate slow start and/or timeout overheads.

3.1 Control Block Interdependence

For each connection, TCP maintains a Transmission Control Block (TCB), which contains local process information and connection state, such as current round-trip time estimates and congestion control information. Although TCB parameters are maintained per connection, some of the parameters carry information which is really host-pair specific. Hence, Touch argues that the host-pair specific state be shared across similar connection instances and/or similar concurrent connections [31]. Touch further argues that such sharing schemes can improve transient performance connections, benefiting short-lived flows.

State sharing or TCB interdependence can be of two kinds: *temporal sharing* and *ensemble sharing*. When cached parameters from an earlier connection are used to initialize parameters of a new connection between the same host-pair, temporal sharing is effected. Ensemble sharing occurs when some parameters are used from an existing connection to initialize a new connection between the same host-pair.

Ensemble sharing of cwnd is more complicated than it is with other shared parameters. The suggested mechanism is to maintain a constant sum of the congestion window sizes as an aggregate cwnd even after the introduction of new connections. Each new connection is given its *fair share* of the current aggregate cwnd by "extracting" cwnd space uniformly from the existing connections. After a new connection's cwnd is initialized, the aggregate cwnd of the ensemble evolves as that of a single TCP connection. The entire set of concurrent TCP connections is as aggressive as a single TCP connection. Hence, in an existing ensemble of N connections with an aggregate cwnd of $aggr_cwnd$, a new connection sets its initial cwnd to $aggr_cwnd/(N + 1)$, and the cwnd of each existing concurrent connections is reduced by $aggr_cwnd * N/(N + 1)$. A cache carries a recent sum of cwnds of all concurrent connections. This mechanism not only helps a new flow adapt rapidly to the congestion state in the network, it also avoids concurrent connections from adversely affecting each other [31].

Discussion

Control Block Interdependence introduces generic transport layer mechanisms, eliminating the need for application layer multiplexing over TCP connections as done in P-HTTP, and improves performance of short TCP transfers. Ensemble sharing avoids a new connection probing for a known bandwidth-delay estimate. This proposal makes the important point that some of the TCB parameters are really host-pair specific, not connection specific, and gives mechanisms to maintain those parameters per host-pair. The issues that need to be addressed in implementing temporal sharing include cwnd evolution during silence periods (between similar connection instances). The work described in Sections 3.2-3.5 uses the ideas of temporal and ensemble sharing.

T/TCP considers the temporal sharing case through cached TCB data, in particular, the initialization of a new connection's cwnd (see Section 2.1). Balakrishnan's *TCP-INT* was the first implementation of ensemble sharing, but did not implement temporal sharing [5]. *Ensemble-TCP (E-TCP)* implements temporal sharing, ensemble sharing, and the 3-way handshake avoidance provided by T/TCP [13]. Eggert et al. use ns simulation to compare the performances of E-TCP versus TCP Reno for web traffic, and show significant performance improvement with E-TCP [13].

3.2 TCP Session and TCP Fast Start

TCP Session and *TCP Fast Start* are proposed as a comprehensive two-component solution [22]. *TCP Session* decouples TCP's ordered byte-stream service from its congestion control and loss recovery mechanisms. Hypothesizing that traffic between two endpoints will probably take the same path through the network, *TCP Session* integrates congestion control and loss recovery mechanisms across the set of concurrent connections between a host-pair. While retaining flexibility and advantages of multiple data streams between hosts, *TCP Session* obtains the performance benefits of a shared connection.

A session is an aggregation of TCP flows at different levels of granularity: individual applications, all applications launched by an individual user, each host-pair, etc. *TCP Session* defines a Session Control Block (SCB) which is constituted by state variables shared across the TCBs of connections belonging to a session. The per-session SCB includes: (1) a session congestion window, (2) a session slow start threshold, (3) outstanding bytes across connections in the session, (4) a list of connections within the session, (5) an estimate of session smoothed round-trip time and variance, and (6) a list of unacked segments sent on the connections in the session.

Using these shared state variables, Padmanabhan describes techniques for integrated loss recovery and integrated congestion window evolution [22]. Such techniques are shown to be capable of faster loss recovery, and more responsive to network congestion than multiple TCP connections. The motivation behind integrated loss recovery is to reduce the number of timeouts for short flows. Longer flows have the advantage that losses are more likely to be detected by duplicate acks due to the larger number of acks that flow back to the sender. Duplicate acks convey information that the sender can use to infer loss. With short flows, not enough data may be sent to generate the three duplicate acks needed to trigger fast retransmit. Hence, short flows can be assisted by being provided information across concurrent flows between the same host-pair. The feedback information can be integrated to effect faster loss recovery within flows.

To assist a flow to infer loss, Padmanabhan defines *later acks* as follows (see Figure 3). "An ack is termed

as a later ack for an unacked segment, S , belonging to a connection, C , if it acks one or more segments sent after S on one of the other connections in the same TCP session as C ." Later acks can be used by short flows in the absence of sufficient duplicate acks to trigger fast retransmits. Mechanisms to infer loss with the help of later acks and duplicate acks together are described in [22].

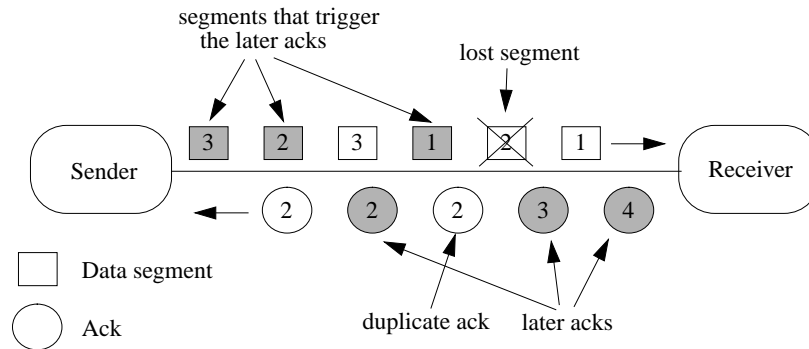


Figure 3: Illustration of *later acks* within two TCP connections (white & shaded) of a TCP session [22]

TCP Fast Start [22, 23] is a technique which, as the name suggests, tries to alleviate TCP's slow start overhead. Fast Start implements the idea of temporal sharing (see Section 3.1). A sender caches network parameters of the current connection so that subsequent transfers to the same host need not suffer the slow start overhead. Fast Start provides mechanisms to improve start-up performance, while ensuring that the network equilibrium is not adversely affected.

For each connection, a sender caches the congestion window size ($cwnd$), the slow start threshold ($ssthresh$), the smoothed round-trip time ($srtt$), and its variance ($rttvar$). The cached values are used after the connection has been idle, or when starting a new connection. In either case, the cached and used $cwnd$ is the most recent window size of data successfully transmitted. Just before an idle time or connection termination, the $cwnd$ would have grown to twice (in the slow start phase) or one MSS more than (in the congestion avoidance phase) the last successfully transmitted window of data. Hence, the cached value of the $cwnd$ is set to half or one MSS less than the last $cwnd$ to appropriately reflect the last successful window size. $Ssthresh$, $srtt$, and $rttvar$ are cached unaltered.

Cached $cwnd$ values can potentially be fairly large, which could result in a large burst during the Fast Start phase. Generally, ack clocking is used to avoid such large bursts, but Fast Start uses a pacing mechanism to avoid waiting an RTT for ack clocking to set in. The sender paces out $maxburst$ size bursts uniformly over an RTT, by spacing the bursts apart in time by $maxburst * srtt / cwnd$. The burst size, $maxburst$, is a configurable parameter which Padmanabhan set to 4 segments in his experiments.

Since cached network information at the sender can become stale, Fast Start could disturb the network equilibrium. To address this issue, a sender marks all packets except the first sent in the Fast Start phase as low priority. The first packet is not marked because it would be sent by slow start anyway. The priority marking mechanism proposed is similar to the cell loss priority (CLP) mechanism in ATM. If a bottleneck router needs to drop a packet, the router should drop a low priority packet. Fast Start thus attempts to avoid causing congestion in the network due to stale network information.

A connection attempting Fast Start could itself suffer heavy packet loss, and end up with worse perfor-

mance than if it had used standard slow start. To avoid this side effect, the TCP loss recovery procedure is augmented in several ways, as discussed in [22].

Discussion

The dependence of Fast Start on router support (for *drop priority*) is a significant hurdle in its deployment. Currently, no such mechanism exists in the Internet, but drop priority could be implemented with the deployment of differentiated services.

3.3 Congestion Manager

Congestion Manager (CM) [6] provides a framework for aggregating and managing network congestion from an end-to-end perspective using the ideas of both temporal and ensemble sharing. Work such as TCP-INT [5], E-TCP [13], and TCP Session [22] share state and promote learning across TCP connections between the same host-pair. CM is a more generic framework, which takes a step back and promotes learning across all fws (independent of transport protocol used) between the same host-pair. Such learning reflects the fact that network state between endpoints is not specific only to TCP. New fws can rapidly adjust to network conditions via the sharing implemented by the CM.

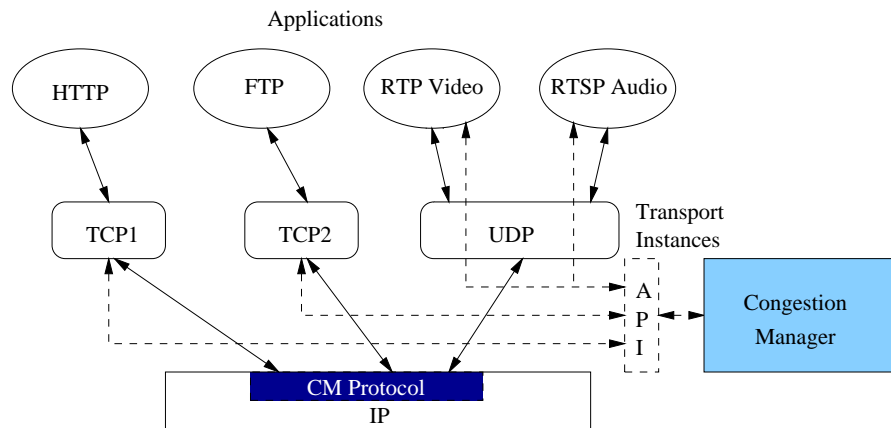


Figure 4: Sender protocol stack with CM [6]

The architecture and interactions between the various components of the protocol stack are shown in Figure 4. Transport protocols that are responsive to congestion, such as TCP, are modified to interact with the CM. Applications sensitive to delay (which typically use UDP) can also interact with the CM directly, and adapt efficiently to congestion through the CM’s callback API.

The CM maintains network statistics gathered from the different fws, and uses them to govern the rate of transmission of the fw aggregate. The CM also incorporates a pluggable scheduler to apportion bandwidth to the various fws. Thus, short fws may not have to suffer the latency due to slow start, if the CM already has an estimated available bandwidth (cwnd) for a host-pair. In such cases, a new fw’s fair share is apportioned from the current cwnd. The CM uses an Additive Increase Multiplicative Decrease (AIMD) algorithm with two kinds of feedback to probe for available bandwidth. One kind is where the application

or transport itself elicits feedback from the receiving application or transport, and informs the sender’s CM about successful transmissions and losses through the callback API. In such cases, the CM is not required at the receiver. The second kind is for application flows without feedback, such as most streaming applications using UDP. In these cases, the CM implements a loss resilient protocol for eliciting feedback from the receiver’s CM. When feedback is infrequent, the CM ensures conservative sending behavior by exponentially aging the sending window.

Balakrishnan et al. [6] modify TCP to run over the CM (TCP/CM), and describe the interactions among the various modules in the protocol stack. When an application uses TCP/CM to send data to a client, TCP/CM opens a channel with the local CM. To allow the CM to handle the rate of transmission, TCP/CM does not send data directly to the client. Instead, as the application has data to send, TCP/CM requests the CM to schedule transmission of data for this connection. When the CM is able to handle TCP/CM’s request, the CM performs a callback to TCP/CM allowing up to 1 MTU of data to be transmitted. The request is granted depending on the current cwnd and the receiver’s advertised window. On sending this data, IP notifies the CM to update its estimate of the number of outstanding bytes. TCP/CM does not perform its own congestion control, but notifies the CM about successful delivery of data upon receipt of acks. TCP/CM does perform its own loss detection and recovery, and also notifies the CM about losses. The CM uses this information to evolve the cwnd. Thus the CM decouples TCP’s loss recovery and congestion control mechanisms - in TCP/CM, loss recovery is provided by TCP while congestion control is done by the CM.

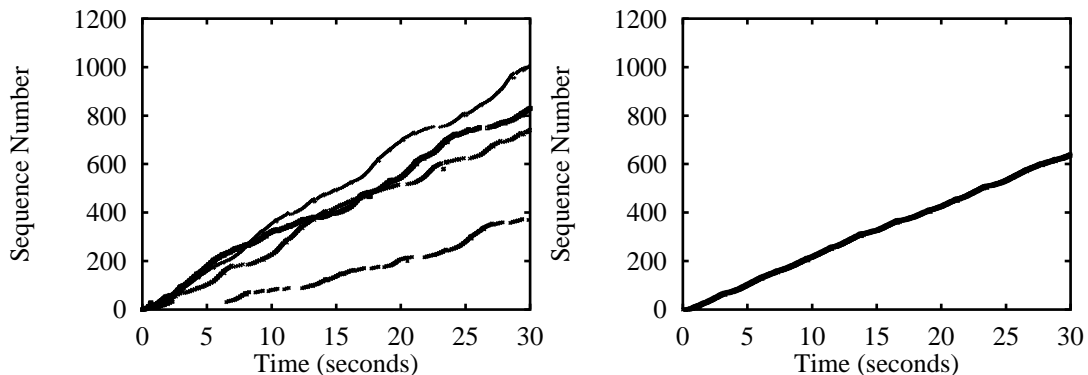


Figure 5: Sequence number vs time for a web-like workload using 4 concurrent connections: (left) TCP NewReno and (right) TCP/CM [6]

Simulation results for web transfers using TCP/CM against TCP NewReno are shown in Figure 5. The aggregate throughput obtained by the TCP NewReno connections (≈ 785 Kbps) is higher than the throughput obtained by the TCP/CM connection (≈ 680 Kbps). However, the performance with TCP NewReno is highly variable and inconsistent, while TCP/CM provides consistent and predictable performance. Balakrishnan et al. argue that the lower aggregate throughput, while unfortunate, is a result of “correct” congestion control. TCP applications also benefit from the CM in obtaining improved performance consistency and predictability.

Discussion

The CM project was a significant motivation behind the formation of the Endpoint Congestion Management (ECM) working group of the IETF. Since implementing the CM at the sender alone provides benefit, in-

cremental deployment is more easily facilitated than if the receiver also needed modification. The CM API provides for applications to adapt their behavior to prevalent network conditions. Through its mechanisms, the CM imposes congestion control on unruly applications. The scheduler, being pluggable, allows for further work on mechanisms so that applications can share the congestion window space in a fair and efficient manner.

A couple of issues with regard to TCP mice continue to exist with the CM. In particular, the significant overhead of TCP's 3-way handshake in short TCP transfers is not addressed in the CM. Since loss recovery is also left to TCP, timeout recoveries, which are probably more expensive among TCP mice than elephants, persist. It may be worthwhile to consider integrated loss detection and recovery mechanisms, such as those in TCP Session (see Section 3.2), to reduce the number of timeouts.

3.4 Stream Control Transmission Protocol (SCTP)

The *Stream Control Transmission Protocol (SCTP)* [30] is an alternative general purpose transport protocol that uses *multistreaming* to aggregate related application flows into a single connection, or *association* in SCTP terminology. Each application flow does not create a separate transport layer connection. All streams within a single association share network state, and AIMD congestion control is performed for the entire association. Thus, the shared state information affects a single slow start across all streams.

An SCTP stream is a unidirectional logical data flow within an SCTP association. Streams are specified during association setup, and exist for the life of an association. Each stream is allocated independent send and receive buffers. For example, in Figure 6, Hosts *A* and *B* have a multistreamed association, where during association setup, Host *A* requested three streams to Host *B* (numbered 0 to 2), and Host *B* requested one stream to Host *A* (numbered 0).

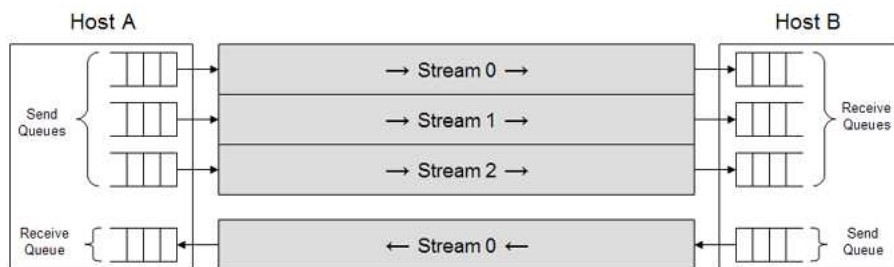


Figure 6: Example multistreamed association

Within streams, data order and reliability are preserved with the use of Stream Sequence Numbers (SSNs) for each data unit. SCTP prevents head-of-line blocking by eliminating ordering constraints between streams. A missing data unit on one stream does not block data of other streams from being delivered to the application. Also, acks acknowledge data for all streams, thereby allowing a sender to detect missing data on one stream from successful acks on other streams. This interstream assistance with loss recovery is similar to *later acks* from TCP Session (discussed in Section 3.2).

Discussion

SCTP proponents argue that management of multiple streams that are logically or semantically associated (i.e., part of the same end-to-end application) is inherently a transport layer responsibility, and should not be a burden placed on an application (as is the case when the application maintains a separate TCP connection for each stream) nor as a separate sublayer (as done in CM). Practicality has prevented multistreaming from being incorporated into TCP; with TCP's global deployment, even minor changes are difficult to make. But with SCTP being a new transport protocol, the opportunity to provide multistreaming transport service is both practical and timely.

SCTP could also be modified to work with the CM. SCTP would handoff its to the CM, but retain loss recovery control. SCTP would still benefit applications which require independent streams, because those streams will assist each other with loss recovery. With the CM, the streams would be incorporated into the larger aggregation of flows to provide the same benefits that CM provides to TCP/CM and UDP.

SCTP also includes other useful features, such as unordered data delivery. Since different pipelined HTTP GET requests can be treated independently by a web server, the unordered delivery feature of SCTP could help by allowing later GET requests to be processed independently in the face of loss. At the Protocol Engineering Lab at the University of Delaware, we are investigating the performance benefits of using SCTP for FTP [17] and web traffic.

3.5 TCP/SPAND

TCP/SPAND [34] is based on the *Shared PAssive Network performance Discovery (SPAND)* architecture [27]. In this architecture, network resource and performance information is shared among many co-located hosts, and is used to estimate each connection's fair share of the network resources. Based on such estimation and *a priori* knowledge of the transfer size, a TCP/SPAND sender determines the optimal initial congestion window size at connection start-up, and upon restart after an idle period. To avoid slow start, TCP/SPAND uses a pacing scheme to smoothly send out the packets in the initial window, and directly enters congestion avoidance mode by setting $ssthresh = cwnd$. Once all packets in the initial window have been paced out, the sender switches back to the behavior of standard TCP in congestion avoidance mode.

To minimize transfer completion time, the authors analytically determine the largest initial $cwnd$ that allows the transfer to end at some epoch boundary, as shown in Figure 8. The solution to the optimization problem assumes that the bandwidth-delay product and the bottleneck router's buffer size are constant. This assumption, being unrealistic, is relaxed by using a technique called *shift optimization*. Shift optimization makes the selection of optimal initial $cwnd$ as conservative as possible for the given transfer size, without increasing the integral number of RTTs required for the transfer [34].

A *performance gateway* is employed in the SPAND architecture to track and provide estimated values of the bandwidth-delay product, the bottleneck router's buffer size, and the RTT per destination network [26]. This performance gateway monitors all traffic entering and leaving an organization's network, and the network performance information from all the active TCP flows to that destination network. This information allows for estimation of connection RTT to a destination network, which is used for the initial RTO and for the pacing of the packets in the initial window of a TCP/SPAND sender. Zhang et al. enumerate and attempt to address, without offering too much detail, the following implementation issues of this TCP/SPAND

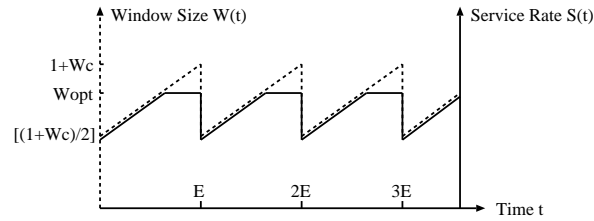


Figure 7: Evolution of service rate and congestion window during congestion avoidance for TCP Reno [34]

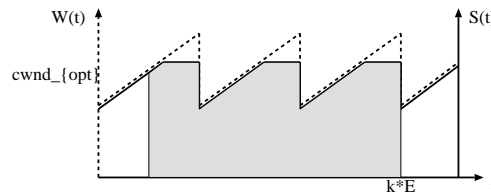


Figure 8: Minimize completion time by having the transfer end at some epoch boundary [34]

design [34]:

- What is the right scope for information sharing and aggregation?
- How does the performance gateway collect performance information for active flows?
- What algorithms should the gateway use to aggregate the gathered performance information and estimate the current network state?
- How can the applications sitting on top of TCP extract the current estimation of the bandwidth-delay product, the bottleneck router's buffer size, and the RTT from the performance gateway?
- What pacing scheme should the TCP senders use to send out packets in the initial window? And how can such a scheme be implemented?

The ns implementation of TCP/SPAND, which is based on TCP NewReno, shows TCP friendly behavior in simulation studies [34]. The studies also show that TCP/SPAND reduces latency for short transfers even in the presence of multiple heavily congested bottleneck links.

Discussion

TCP/SPAND needs a SPAND-like performance gateway architecture implemented. This architecture is incrementally deployable, since modifications are required only on the server side. Provided the performance gateway information is reliable and correct, TCP/SPAND has been shown to be TCP-Friendly.

A reliable gateway design and implementation could significantly improve the performance of short flows using TCP/SPAND. On the other hand, the entire architecture here depends on a reliable gateway design and implementation. This gateway thus becomes a single point of failure; one faulty gateway could potentially

cause significant congestion on the Internet. Further, fbws to the same host still compete against each other for bandwidth².

Even using shift optimization, the sensitivity of the optimal initial cwnd is higher for lower bandwidth-delay products and bottleneck router buffer sizes. Hence, for paths that have lesser end-to-end available bandwidth, the initial value of the cwnd for a new connection will be closer to the last cached cwnd. In other words, there would be little or no degradation of the cached cwnd value, suggesting that the network state has not changed. Such a value of cwnd could cause congestion and loss.

4 Improve Performance During Slow Start

This section presents three solutions that either improve slow start behaviour at the endpoints to accomodate short fbws, or use network support to preferentially treat short fbws.

4.1 TCP with Larger Initial Window

Allman et al. [3] propose increasing the initial window size of TCP from one to four segments. Such an increase reduces the latency of a data transfer in three ways. First, most TCP implementations implement a delayed ack algorithm [4], and only ack every second data segment. If a second data segment is not received within some timeout interval, the sender acks the single segment. An initial window of at least two segments eliminates the need for receivers to delay the first ack for the full value of the TCP ack timer (200ms in many implementations). Second, up to three RTTs are eliminated with an initial window of four segments. An initial cwnd of one segment requires a sender to receive three RTTs to grow to four segments. Third, a larger initial window provides a chance for faster loss recovery using fast retransmit. The fast retransmit algorithm requires three duplicate acks to indicate a missing segment, and trigger a retransmission. Thus, a fast retransmit is only possible when the sender's cwnd is larger than four segments; otherwise, only a timeout can recover from loss.

A survey of simulation and empirical results that explore the effect of larger initial windows is presented in [3]. In summary, TCP connections which use a larger initial window provide a significant improvement in a variety of networks. These TCP connections alone generally introduce less than 1% increased drop rate. However, in a highly congested network (where each connection's share of the bottleneck bandwidth is close to one segment), a 1% to 2% additional drop rate is introduced when all TCP connections in the network use an initial window of four segments. In addition, TCP connections spent more time waiting for the retransmit timer (RTO) to expire to resend a segment than was spent when using an initial window of one segment.

Discussion

As mentioned above, increasing a TCP sender's initial window could result in increased overall packet drops at congested links. Allman et al. argue that though dangers of congestion collapse on the Internet

²This problem has been addressed in other designs discussed in this section, such as the Congestion Manager.

today do exist (such as the increasing deployment of UDP connections without end-end congestion control), no such danger to the network is introduced by increasing TCP's initial window to about 4K bytes. A larger initial window *may* help short flows to grow and probe the network faster, but the static limit of 4 segments only slightly alleviates the latency issue with short transfers. According to [3], increasing the initial window would show significant benefits in high latency links, but it seems that such congestion state sharing mechanisms (discussed earlier) could do much better with such long pipes.

4.2 Smart Framing for TCP (TCP-SF)

TCP uses two mechanisms to recover from loss: fast retransmits and timeouts. The fast retransmit algorithm requires three duplicate acks to indicate a missing segment and trigger a retransmission. Thus, a fast retransmit is only possible when the sender's cwnd is larger than 4 MSS (Maximum Segment Size), and the flow is long enough to allow the transmission of at least four back-to-back segments. These conditions may be unlikely for short TCP flows.

Smart Framing for TCP (TCP-SF) [19] aims at enhancing TCP's behavior in the operating region where timeouts are the only way to recover losses. Smart Framing exploits the fact that the fast retransmit algorithm is based on the *number* of duplicate acks. A TCP Smart Framing sender is allowed to send an initial window of four segments, whose aggregate payload is equal to the initial cwnd. Thus, the resulting network load is, byte-wise, the same as a classic TCP connection (except for the segmentation overhead). However, with an increased number of segments, the sender has a better chance of inferring loss with a fast retransmit.

Two possible implementations are suggested:

- **Fixed-Size TCP Smart Framing (FS-TCP-SF)** - While the cwnd is less than $N * MSS$, a fixed segment size is used: $\frac{1}{N}$ of the initial cwnd. A sender uses MSS as the segment size once the cwnd reaches $N * MSS$.
- **Variable-Size TCP Smart Framing (VS-TCP-SF)** - Until the cwnd reaches $N * MSS$, a sender transmits segments of size $\frac{1}{N}$ cwnd. Once the cwnd reaches $N * MSS$, the sender begins transmitting full size segments (i.e., MSS). While the cwnd is less than or equal to $N * MSS$, the sender may not allow more than N outstanding segments.

The recommended value for N is four, since TCP needs three duplicate acks to fast retransmit. The cwnd growth algorithms during slow start and congestion avoidance are modified to consider the segment size in use. When using smaller size segments, the cwnd growth per incoming ack is proportional to the segment size. Simulation results and a *Linux* implementation show improvement in latency for short TCP transfers at the cost of increased load on the network [19].

Discussion

TCP-SF causes an increased amount of ack traffic, which could deteriorate the health of the network. The mean file size on the Internet has been reported as 32KB [18]. TCP-SF will stay in Smart Framing mode until the cwnd equals 4 MSS, which translates to 6 (or 7) packet transmissions when the initial cwnd is 2

(or 1). Considering 1460 byte packets, these packets constitute about one-third of a 32KB fbw. Thus, for about one-third of the lifetime of the fbw, ack traffic could be increased roughly 3-4 times (since TCP-SF breaks each segment into at least 4 smaller segments). Considering the number of short fbws on the Internet and also the fact that almost 50% of the packets on the Internet are 40-44 bytes in length (most of which are likely pure acks) [12], TCP-SF could result in significantly increased ack traffic that degrades overall network performance.

4.3 RIO with Preferential Treatment to Short Flows (RIO-PS)

Inspired by the Differentiated Services (diffserv) architecture, Guo and Matta propose to give differential treatment, with preference to short fbws in the bottleneck queue [15]. Such preferential treatment should allow short connections to experience lesser packet drop rate than the long fbws. The authors employ *Random Early Detection (RED) with In and Out (RIO)* queue management policy which uses different drop functions for the different classes of traffic.

In the proposed diffserv-like architecture, fbws are classified at the edge routers, and core routers need to manage per-class fbws. To classify fbws at the edges, a simple threshold based approximation is used which looks at the number of packets from a fbw seen thus far. The first L_t packets of any fbw are classified as “Short” packets, and the following packets are classified as “Long” packets. The threshold L_t can be static, or can dynamically change according to the size of fbws through the router. The router can be provided with a *Short-to-Long Ratio (SLR)* using which the router can periodically determine the value of L_t depending on the current fbws.

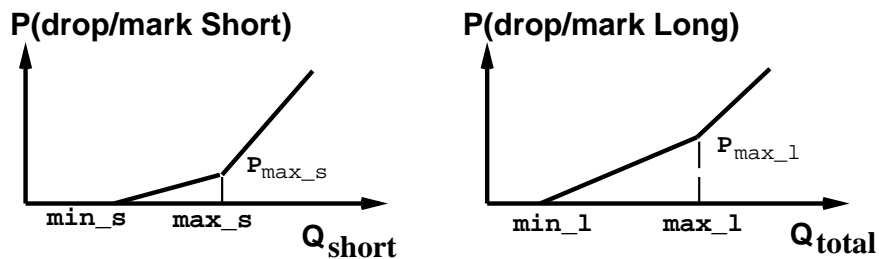


Figure 9: RIO queue with preferential treatment to short fbws [15]

Short packets are classified as *In* packets, and Long as *Out* packets. The core routers then use an RIO queue management policy for packet marking/dropping. Figure 9 shows the marking/dropping function for RIO queue management. Guo and Matta explain that the operation of the queue on In (Short) packets is unaffected by the arrival of Out (Long) packets since the dropping/marking probability for Short packets is computed based on the average backlog of Short packets (Q_{short}) only. On the contrary, for the Long packets, the total average queue size (Q_{total}) is used to detect incipient congestion and thus, fbws that carry Long packets to give up resources when persistent backlog from both classes of packets exists. Since a single FIFO queue is used for all packets, packet reordering does not happen even when packets from the same fbw are classified differently.

Simulation results, shown in Figure 10, illustrate the impact of preferential treatment by comparing link utilization of short and long fbws at the bottleneck queue using different queuing techniques. Under the RIO-PS (RIO with Preferential treatment to Short fbws) scheme, Guo and Matta argue that though short

fbws seem to temporarily take up more bandwidth than long fbws, in the long run their early completion returns an equal amount of resources (if not more, avoiding packet retransmissions) to long fbws. Guo and Matta show through simulations that even when a route is dominated by either of short or long fbws (the suggested worst-case scenario for RIO-PS), the proposed scheme reduces to traditional unclassified traffic with a RED queue management policy. To encourage incremental deployment, RIO-PS needs to be implemented only at the busy bottleneck routers, and the edge classification device can be placed in front of a busy web server.

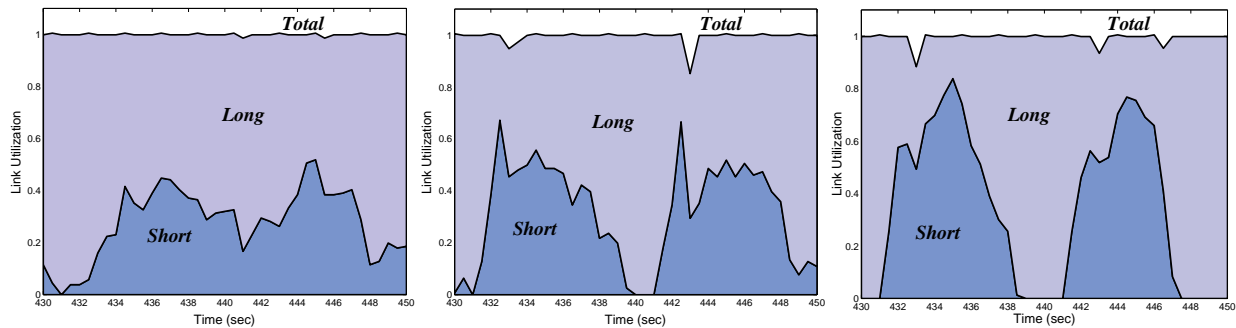


Figure 10: Link utilization: (left) DropTail, (middle) RED, and (right) RIO-PS [15]

Discussion

RIO-PS attempts not only to help short fbws, but also the first packets of any fbw, thus making a fbw resilient to SYN losses and timeouts in the early part of the fbw. An assumption which underlies this work is that the routers are aware of the nature of traffic, and that short fbws are handled with preferential treatment over long fbws. A question that needs to be answered in the spirit of the end-to-end argument [25] is whether the router mechanisms should be made aware of, and should reflect the prevalence of the types of traffic that traverse the network? Would RIO-PS still be relevant if TCP mechanisms no longer cause the short fbws to suffer? In other words, should an enabling network reflect the end-to-end mechanisms deployed?

The following text from [7], which suggests that according to the end-to-end argument [25], the network should be maintained as general as possible, seems appropriate here: “The most important benefit of the end-to-end arguments is that they preserve the flexibility, generality, and openness of the Internet. They permit the introduction of new applications; they thus foster innovation, with the social and economic benefits that follow. Movement to put more functions inside the network jeopardizes that generality and flexibility as well as historic patterns of innovation.”

5 Discussions and Conclusions

The motivation for the research surveyed in this paper has been the rapid and continued growth of the web. This growth has been accompanied by performance problems that have arisen because of a mismatch between the characteristics of web applications (short fbws) and the traffic presumed by the underlying TCP

protocol. We now suggest some general guidelines to be used while comparing and evaluating the different proposals surveyed in this paper, as well as future suggestion for handling short fbws.

Wide Applicability

To accommodate new applications with minimal redundant effort, it is important that a solution apply to a wide variety of applications. All solutions discussed, except for P-HTTP, have wide applicability to short fbws. They are not application specific, and can be used generally to improve the performance of short TCP fbws. Though P-HTTP is specific to HTTP and web fbws, the ideas of persistent connections and pipelining are applicable to a broader class of request-response applications, such as multiple file transfers [17].

Incremental Deployment

Solutions that offer benefits as they are gradually deployed stand a better chance of success than those that must be fully deployed. Proposals such as TCP/SPAND, RIO-PS, TCP Fast Start, claim that they can be incrementally deployed. TCP Fast Start and RIO-PS need router support to even be considered for deployment. The incremental deployment of router support is an added overhead in such cases. TCP/SPAND is dependent on a robust and reliable performance gateway being implemented. Such gateways do not exist in the Internet today, and the ones proposed [26] have not been mass-deployed on the real Internet. These gateways have to evolve before TCP/SPAND can be considered for deployment. The dynamics of incremental deployment at the endpoints are quite different from those in the network.

Completeness

Considering the amount of effort and time it takes for mass deployment of any Internet solution, it is better to deploy a solution that is more generic and universal. For instance, TCP Smart Framing addresses only the issue of timeouts in the slow start phase. On the other hand, E-TCP attempts to address all issues faced by short fbws.

Impact on New Application Paradigms

New Internet application designs and paradigms surface time and again. The “killer app” of the Internet shifted from file transfer and telnet, to email, to the web, to peer-to-peer file sharing applications, and perhaps soon to distributed game playing. Performance gains by any of the mechanisms discussed in this paper for short fbws could have a serious impact on the performance of future Internet applications. Thus, the end-to-end argument [7, 25] which underlies the design philosophy of the Internet, should be kept in mind while considering any proposal for deployment.

References

- [1] T/TCP Home Page (TCP for Transactions). <http://www.kohala.com/start/ttcp.html>.
- [2] M. Allman. A Web Server's View of the Transport Layer. *ACM Computer Communication Review*, 30(5), June 2000.
- [3] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC3390, October 2002.
- [4] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC2581, April 1999.
- [5] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz. TCP Behavior of a Busy Internet Server: Analysis and Improvements. In *INFOCOM 1998*, March 1998.
- [6] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM 1999*, September 1999. *ACM Computer Communication Review*, 29(4), October 1999.
- [7] M. Blumenthal and D. Clark. Rethinking the Design of the Internet: End-to-End Arguments vs. The Brave New World. *ACM Transactions on Internet Technology*, 1(1):70–109, August 2001.
- [8] R. Braden. Towards a Transport Service for Transaction Processing Applications. RFC955, September 1985.
- [9] R. Braden. Extending TCP for Transactions – Concepts. RFC1379, November 1992.
- [10] R. Braden. TIME-WAIT Assassination Hazards in TCP. RFC 1337, May 1992.
- [11] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. RFC1644, July 1994.
- [12] K. Claffy, G. Miller, and K. Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. In *INET 1998*, April 1998.
- [13] L. Eggert, J. Heidemann, and J. Touch. Effects of Ensemble-TCP. *ACM Computer Communications Review*, 30(1):15–29, January 2000.
- [14] H. F. Nielsen et al. Network Performance Effects of HTTP/1.1, CSS1, and PNG, February 1997. www.w3.org/pub/WWW/Protocols/HTTP/Performance/Pipeline.html.
- [15] L. Guo and I. Matta. The War Between Mice and Elephants. Tech Report BU-CS-2001-005, CS Dept, Boston University, May 2001.
- [16] J. Heidemann. Performance Interactions Between P-HTTP and TCP Implementations. *ACM Computer Communication Review*, 27(2):64–73, April 1997.
- [17] S. Ladha and P. Amer. Improving File Transfers Using SCTP Multistreaming. Tech Report TR2003-06, CIS Dept, University of Delaware, March 2003.
- [18] B. A. Mah. An Empirical Model of HTTP Traffic. In *INFOCOM 1997*, April 1997.
- [19] M. Mellia, C. Casetti, and M. Meo. TCP Smart Framing: Using Smart Segments to Enhance the Performance of TCP. In *GLOBECOM 2001*, November 2001.
- [20] M. Mellia, M. Meo, and C. Casetti. TCP Smart Framing. draft-mellia-tsvwg-tcp-smartframing-00.txt (expired), November 2001.

- [21] J. C. Mogul. The Case For Persistent-Connection HTTP. In *SIGCOMM 1995*, August 1995. *ACM Computer Communication Review*, 25(4), October 1995.
- [22] V. Padmanabhan. Addressing the Challenges of Web Data Transport. PhD Dissertation, CS Dept, University of California at Berkeley, 1998.
- [23] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In *GLOBECOM 1998*, November 1998.
- [24] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *2nd Second International World Wide Web Conference*, October 1994.
- [25] J. Saltzer, D. Reed, and D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [26] S. Seshan, M. Stemm, and R. H. Katz. Homepage for SPAND. www.seshan.org/spand.
- [27] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. In *1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, Monterey, CA, December 1997.
- [28] W. Stevens. *TCP/IP Illustrated, Volume 1*. Prentice-Hall, 1994. pages 352-353.
- [29] W. Stevens. *TCP/IP Illustrated, Volume 3*. Prentice-Hall, 1996. pages 29-38, 62-67.
- [30] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC2960, October 2000.
- [31] J. Touch. TCP Control Block Interdependence. RFC2140, April 1997.
- [32] Vasim Valejev. T/TCP Spoofing Problem, April 1998. www.insecure.org/spl0its/ttcp.spoofing.problem.html.
- [33] V. Visweswaraiiah and J. Heidemann. Improving Restart of Idle TCP Connections. Tech Report 97-661, University of Southern California, November 1997.
- [34] Y. Zhang, L. Qiu, and S. Keshav. Speeding Up Short Data Transfers: Theory, Architectural Support and Simulation Results. In *NOSSDAV 2000*, June 2000.