

A Proposal for Supporting Lots of Unicast Multiplexed Sessions (SLUMS)¹

*Venkata N. Padmanabhan
Microsoft Research
padmanab@microsoft.com*

Abstract

In this paper, we present a proposal for supporting lots of unicast multiplexed sessions (SLUMS). We take the position that TCP provides a good basis for SLUMS. Applications are given the flexibility to open as many TCP connections as they wish. Each connection provides an ordered byte-stream abstraction independent of the others, so by mapping application data units (ADUs) onto the connections appropriately, applications can implement various flavors of application-level framing (ALF). We discuss the performance challenges that arise, and propose solutions drawn both from our previous work and that of others.

1 Introduction

In this paper, we outline a proposal for SLUMS that meets some, though not all, of the goals listed in [Pax99]. The goals of SLUMS, as stated in [Pax99], are:

1. Quick establishment/activation, which is in tension with security considerations (authentication, flooding - not holding state)
2. **Support for application level framing:**
 - control over reliability (e.g., setting time limits on how long to try to deliver a message)
 - **the ability to supercede previous application messages**
3. **Minimize state requirements;** think of servers with 1e6 connections
4. **Muxing:**
 - **PDU's muxed**, delivered ASAP
 - Want **ACK aggregation** across the different communication streams
 - **isolated flow-control**
 - **QoS consciousness between the streams**
5. **Congestion control across multiple communication streams**
6. **Reducing slow start delays** for multiple communication streams
7. Failover: transport connection can survive across change in IP address
 - on connection attempt, SYN timeout are viewed as expensive
 - mid-stream, need to switch to backup interfaces
8. Ability for application to indicate "a reply is coming" versus "no more coming now, go ahead and ack, don't delay"

The highlighted goals are addressed in this document. Much of the presentation here is based on [Pad98].

¹ The views expressed in this paper are those of the author and do not necessarily represent the views of Microsoft Corporation or any other organization.

2 Shortcomings of TCP in the Context of SLUMS

Before discussing our proposal in detail, we enumerate shortcomings of TCP in the context of SLUMS. As will become clear in our discussion, at least some of these shortcomings arise from the way TCP is *implemented* and the way it is *used* by applications rather than due to the protocol itself.

1. **Connection setup/teardown is expensive:** Connection setup incurs a latency of at least one RTT for the SYN handshake.
2. **Large packet count:** The operation of a TCP connection involves the transmission of several packets in addition to those that carry application data. These extra packets include the SYN/FIN packets for connection setup/teardown and the ACK packets (which often do not carry any application data), and may constitute a significant overhead.
3. **Strict ordering imposed within a connection is too restrictive:** While an application may want reliable data delivery, it may be willing to accept reordering of logically-distinct application data units (ADUs) [CT90] for performance reasons. But application-level multiplexing techniques, such as P-HTTP [PM94], may unnecessarily impose ordering across logically-distinct ADUs. Certain applications, such as streaming multimedia, may not even want reliable data delivery.
4. **Concurrent connections compete:** One can get around the strict ordering imposed by a TCP connection by establishing multiple connections simultaneously between a pair of hosts. However, current implementations of TCP cause the concurrent connections to operate independently of each another, which can have several deleterious effects:
 - The connections compete with each other, often resulting in an unfair and arbitrary sharing of bandwidth. A connection may be starved of bandwidth regardless of its importance to the application.
 - The connections do not share indications of congestion along the shared path between the sender and the receiver. Therefore, each connection independently pushes the network to the point where packet losses happen. This aggravates congestion in the network as a whole, consequently driving up the packet loss rate.
5. **Short connections perform poorly:** Short connections tend to perform poorly for a couple of reasons:
 - The sender may not have sufficient opportunity to ramp up its window, so the connection may not utilize the available bandwidth effectively.
 - Timeouts rather than fast retransmission becomes the dominant mode of loss recovery.

3 Proposed Solution

We now present our solution to address many of the goals of SLUMS. We present an overview before getting into the details.

3.1 Overview

Our proposal is to enable applications to map application data units (ADUs) onto TCP connections in a flexible way. Ideally, applications will be able to establish and tear down TCP connections as they please (for example, a separate connection may be used for each ADU to be transferred) without incurring a significant performance penalty. We take the position that a TCP connection, in itself, provides a very useful service abstraction. The reliable data delivery that it provides relieves applications from having to worry about packet loss and other vagaries of the network. The freedom to establish connections at will means that applications can have multiple ADUs in transmission concurrently with no ordering imposed across the ADUs. In short, applications can take advantage of TCP's easy-to-use interface while at the same time avoid the ordering constraints imposed by a *single* TCP connection.

In addition, we use *connection scheduling* [Pad98] to enable applications to dynamically control the relative importance (priorities) of the individual TCP connections. So in some sense, each connection can be assigned a different QoS level.

We believe that such flexible mapping of ADUs onto TCP connections is suitable for many applications. One example is *Web browsing*, the most dominant application in the Internet today. Each inline image (or each layer in a multi-resolution image) could constitute an ADU and be mapped onto a separate TCP connection. The receiver (browser) could potentially render each ADU independently of the others. The scheduling priorities of the individual ADUs could be controlled based on application-level information. Another example application is a *unicast whiteboard* where each object pasted on to the whiteboard (e.g., text, drawing, image, etc.) could be transmitted to the peer over a separate connection.

3.2 Solution Components

Section 2 makes it clear that there exist many challenges to enabling the flexible use of TCP connections as outlined in Section 3.1. Our proposal is to address these challenges by combining solution components drawn from our previous work as well as that of others. By constructing solutions out of various subsets of these components, we are able to address various subsets of the goals of SLUMS. We strive to introduce little or no change to TCP or its API.

1. **Transaction TCP (T/TCP)** [RFC1379, Ste96]: This obviates the need for a SYN exchange in case of communication with a host whose *connection count* (*CC*) has been cached, thereby saving an RTT and reducing packet count during connection establishment. But as discussed in [RUTS98], it has security implications. In principle it is possible to overcome this problem by extending the CC cache to also hold security key information. However, we do not discuss this issue further in this paper.
2. **TCP Session** [Pad98, BPS+98]: This avoids competition and enables orderly sharing of bandwidth between the set of concurrent TCP connections between a pair of hosts (termed as a TCP *session*). For such a set of connections, any or all of the following may be performed:
 - *Integrated congestion control*: congestion control is applied on a session-wide rather than per-connection basis.
 - *Connection scheduling*: the sending application can dynamically control the scheduling priority of individual connections within a session. For instance, the application can assign high priority to connections carrying important data.
 - *Integrated loss recovery*: improves the effectiveness of data-driven loss recovery by pooling together information from all the connections in a session.

TCP Control Block Interdependence [RFC2140] provides a simpler alternative to TCP Session but with reduced functionality. While we focus of TCP Session in this document, depending on the exact context, TCP Control Block Interdependence may be sufficient for SLUMS.

3. **Ack aggregation** [Pad98]: In [Pad98, Section 7.3], we proposed a new TCP option to aggregate ack information across concurrent TCP connections between a pair of hosts. Our motivation there was to increase the robustness of integrated loss recovery in a TCP session. But this technique may be used regardless to reduce the number of ack packets transmitted for the set of concurrent connections as a whole.
4. **Efficient state management**: One of the “costs” of TCP is the maintenance and management of the TCB protocol control block (TCB) state for each connection. Several optimizations are possible to reduce this cost.
 - With TCP Session, only one copy of the TCB state variables pertaining to congestion control and loss recovery is maintained per session. This reduces memory consumption.
 - Only a single retransmission timer is maintained per session. This reduces the periodic timer processing overhead.
 - The demultiplexing of received packets can be made more efficient by using techniques such as hashing [Mog95].
 - The state management overhead can be reduced by separating out inactive connections (e.g., those in TIME-WAIT state) from the active ones [Mog95].

In the following sub-sections, we discuss these solution components in more detail.

3.3 TCP Session

TCP Session decouples two components of TCP functionality: (a) the reliable, ordered byte-stream *service abstraction*; and (b) congestion control and loss recovery *algorithms* needed to support (a). With TCP Session, the former is performed on a per-connection basis while the latter is integrated across the set of connections in a session, i.e., the set of connections between a pair of hosts.

Since the service abstraction provided by a TCP Connection remains unchanged, applications can establish multiple connections concurrently between a pair of hosts and essentially have multiple byte-streams with independent flow control. The delivery of data to the receiver on one connection is not tied to that one the other connections.

The motivation for integrating congestion control and loss recovery is that in today's best-effort Internet, packets belonging to all of the connections in a session are likely to traverse the same path through the network and receive similar service at the routers. From the viewpoint of the network then, the load imposed by the set of connections in a session is just the sum total of that offered by each connection in the session. It does not matter how many connections are there and what the load offered by each connection individually is. Indeed, as discussed above and in Section 3.1, our model is to give applications complete freedom to open as many connections as they wish. This application-level decision should have no impact on the functioning of the network stack-level congestion control and loss recovery algorithms.

We now briefly discuss each of the components of TCP Session. Refer to [Pad98] for more details.

3.3.1 Integrated Congestion Control

Integrated congestion control regulates the amount of outstanding data that a session as a whole has in the network. It is purely an algorithmic change at the sender network stack with no impact on the API/applications, the on-the-wire protocol, or the TCP receivers.

The sender maintains a session-wide congestion window, *session_cwnd*, and a session-wide count of the amount of outstanding data, *session_ownd*. The dynamics of *session_cwnd* mimic that of *cwnd* in a standard TCP connection. *Session_cwnd* is initialized to one segment the first time a connection is established in the session. It grows exponentially during slow start and linearly during the congestion avoidance phase. It is halved when a packet loss is detected via data-driven means and is reset to one segment upon a timeout.

The sender is entitled to send new data so long as *session_ownd* is less than *session_cwnd*. At any point, the choice of which connection to send data on is entirely the sender's (modulo per-connection flow control imposed by the TCP receiver). For instance, when *session_ownd* decreases because of an ack received on one connection, the sender is entitled to send the corresponding amount of new data. However, unlike in the case of standard TCP, the sender could choose to send data on a different connection (within the same session) from the one the ack arrived on.

The net result of integrated congestion control is that: (a) the set of concurrent connections becomes as responsive to network congestion as a single TCP connection, and (b) the connections share the total bandwidth available to them in an orderly manner rather than competing haphazardly.

3.3.2 Connection Scheduling

By default, each connection within a session is apportioned an equal share of the available bandwidth. This is a reasonable default and, moreover, requires no changes to the API/applications. However, as discussed in [Pad98, Section 7.1], there are situations where it may be desirable to assign differential priorities to connections. For instance, the headline image on a news Web page may be more important than the background bitmap, so it may be desirable to transmit the former at a higher priority than the latter. The goal of connection scheduling is to enable this.

In [Pad98], we presented a design based on hierarchical round-robin scheduling (HRR) [KKK90] of connections within a session. Each connection is assigned a weight and is apportioned a share of the bandwidth that is proportional to its weight. Furthermore, HRR ensures that packets belonging to the different connections are

interleaved as finely as their weights will permit. This avoids clumping together of the packets of each connection. The weight assigned to a connection can be varied dynamically.

HRR, or even any general proportional sharing scheme, is by no means the only way to do connection scheduling. For instance, strict priority scheduling is one alternative.

No matter how connection scheduling is done, exposing it to the applications requires a change to the API (but no change to the protocol or to the receiver). In [Pad98], we proposed augmenting the API with two new functions: *setwt* and *resetwt*. *Setwt* sets the weight for a connection to the specified value. *Resetwt* resets the weight for all connections in a session to one (i.e., all connections are assigned equal weight). These simple functions, which can be implemented using socket options, give applications a lot of flexibility in controlling the (relative) allocation of bandwidth across connections in a session. However, there is ample scope for further research in developing better APIs.

Our discussion of connection scheduling is confined to the sending host. However, with the emergence of integrated and differentiated services [RFC1633, RFC2475], routers in the network interior may be capable of providing differential QoS. In such cases, the priorities expressed by the application could be used in conjunction with a signaling protocol (e.g., RSVP[RFC2205]) or *diffserv* bit marking to exploit the router capabilities.

3.3.3 Integrated Loss Recovery

When a short TCP connection experiences a packet loss, the effectiveness of data-driven loss recovery (fast retransmission) is hampered by the fact that there may not be enough data in the pipe to elicit loss feedback, i.e., duplicate acks. Consequently, the sender often has to fall back on the expensive retransmission timeout mechanism to recover from the loss (as is evident from the measurements reported in [BPS+98]).

Integrated loss recovery is based on the observation that packets on all connections in a session are likely to traverse the same network path, so it should be possible to pool together information across connections to make loss recovery more effective. The principle of fast retransmission is that substantial reordering of packets in the network is unlikely², so if more than a threshold number (typically 3) of duplicate acks are received, it is likely that a packet loss rather reordering has occurred. Extending this principle to a TCP session, the sender flags a packet as being lost whenever more than a threshold number of *later packets* have reached the receiver and have been acknowledged, where a *later packet* is one that was transmitted after the packet that has not yet been acknowledged. A later packet can belong to any of the connections within the same session. We term the ack for a later packet a *later ack*. While a standard duplicate ack can be considered as an implicit later ack, in our discussion here we choose to make a distinction between the two: duplicate acks correspond to the same connection as the missing packet whereas later acks correspond to other connections in the same session.

One difficulty, in practice, is that TCP receivers often employ the delayed ack algorithm. So the lack of an ack for a segment may simply be because the receiver is waiting for another segment to arrive on the connection before acknowledging them both. However, in the meantime, later packets on other connections within the session may have been delivered to and acknowledged by the receiver. So the generation of a later ack may not necessarily be due to packet loss or reordering.

With these considerations in mind, we deem a packet to have been lost if it is at the left edge of the unacknowledged window for that connection, and:

1. At least one duplicate ack AND at least a threshold number (typically 3, as in TCP) number of duplicate or later acks (in some combination) have been received, OR
2. Two or more segments on the same connection each have at least a threshold number of duplicate or later acks (in some combination).

For a more detailed explanation, refer to [Pad98, Section 6.3.4 & Appendix B].

A retransmission timeout is still available as an option of last resort. However, there is only a single retransmission timer for a session, and a timeout will happen only when the ack streams on *all* of the connections in the session stall. So long as at least one connection is making progress, data-driven loss recovery

² The validity of this assertion in the today's Internet is an open research question.

is still possible. This makes timeouts for short connections (e.g., Web connections) far less likely than is the case with standard TCP. The experimental results presented in [Pad98] confirm this.

3.4 Ack Aggregation

A potential problem with integrated loss recovery is that the loss of a few acks can cause other acks to be treated as later acks, and consequently lead to unnecessary packet retransmission, clearly an undesirable outcome. To address this problem, in [Pad98, Section 7.3] we proposed a new TCP option to enable acks on one connection to carry acknowledgement information for other concurrent connections as well. The format of the option is as follows: 1 byte each for the kind and the length fields, 2 bytes each for the source and destination port numbers, and 4 bytes for the ack sequence number. Thus one ack packet can carry acknowledgement information for up to 4 additional connections (depending on the other options that are in use). If the number of connections in a session is large, the receiver can choose to cycle through the connections in much the same manner as a TCP receiver would cycle through SACK blocks [RFC2018]. This reduces the chances of an ack loss causing spurious retransmissions.

The receiver can choose to send such aggregate acks either in place of regular acks or in addition to regular acks. When there is no packet loss on any of the connections, the receiver may choose to do the former, thereby cutting down on the number of ack packets (and related overhead such as CPU interrupts at the sender). When packet loss occurs, the receiver could send the aggregate acks in addition to ones that carry SACK information.

3.5 Efficient State Management

Much of the state management overhead of TCP is an artifact of its implementation rather than being fundamental to the protocol. Several optimizations that leave the protocol unchanged have been proposed. We briefly survey a few here.

3.5.1 Sharing of State across Connections

With the partitioning of TCP functionality as discussed in Section 3.3, a significant portion of the TCP protocol control block (TCB) becomes part of the session control block (SCB). The variables pertaining to congestion control, RTT estimation, and the retransmission timer are stored in the SCB rather than duplicated in each individual TCB. This saves memory space and also makes certain periodic timer processing more efficient (since only the SCBs rather than all TCBs need be scanned).

3.5.2 Reduced State for Inactive connections

The size of the TCB and allied state (200-300 bytes without the optimization discussed above) is likely to be quite small compared to the socket buffer size. So we argue that the storage overhead of TCBs is insignificant for active connections. However, the overhead may be significant for inactive connections. But for such connections, the amount of state needed can be reduced substantially. For instance, consider the TIME-WAIT state, which we would classify as inactive. One of the purposes of the TIME-WAIT state is to protect against old duplicates. So a new incarnation of a connection is not permitted until $2*MSL$ time has elapsed since the TIME-WAIT state was entered. But ensuring this only requires knowledge of the 4-tuple defining the connection rather than the entire TCB.

3.5.3 Efficient State Lookup

The demultiplexing of an incoming packet requires the TCB of the corresponding connection to be looked up. Many implementations conduct a linear search through the list of PCBs. It would be far more efficient to use a technique such as hashing. Furthermore, it would help to separate out the TCBs for active connections from those for inactive connections [Mog95].

4 Issues Not Addressed Here

There are some issues that we have not addressed here, either because they have been specified as being out of the scope of SLUMS [Pax99] or because they do not fit in with our model of using TCP with little or no changes. We briefly discuss a couple of issues here.

4.1 Resumption of Activity after an Idle Period

When a session resumes activity after an idle period, it would have to incur the overhead of slow start once again (this is essentially the slow-start restart problem [Hei97]). Despite the aggregation of multiple connections within a session, this overhead can be quite significant when the total amount of data transferred across all the connections in a session is small (as is often the case with Web access because of the relatively small median size of Web pages). While we have couched it in terms of TCP, the underlying problem is quite fundamental: there is a tension between probing the network before use to be safe and the overhead of probing.

A number of solutions have been proposed for this problem, including temporal sharing of congestion window [RFC2140], rate-based pacing [VH97], and TCP fast start [Pad98,PK98]. This problem remains an active area of research. However, it is out of the scope of SLUMS (according to [Pax99]).

4.2 Unreliable Data Delivery

Certain applications, such as streaming multimedia, may not require reliable data delivery, in part because of the time-sensitive nature of the data. However, by definition, TCP provides reliable data delivery. Despite these seemingly contradictory properties, a resolution is possible in some instances. Consider the delivery of viewgraphs as part of an online lecture. By mapping the data corresponding to each viewgraph onto a separate TCP connection, the application has the flexibility to enforce selective reliability at the granularity of individual viewgraphs by aborting connections. This may be useful, for instance, if the speaker has gone past a viewgraph that is still in transmission. (Instead of aborting such connections, the application could alternatively choose to decrease their scheduling priority using the connection scheduling mechanism discussed in Section 3.3.2.)

5 Summary

In the paper, we have made the case that TCP connections form a good basis for SLUMS in the context of many applications, including the predominant one today, namely Web browsing. In many instances, the reliable, ordered byte-stream abstraction of a TCP connection provides a useful service: applications can hand data to the network stack once and then forget about it; they do not have to be constantly engaged in the process of transmitting/retransmitting the data. We believe that this simplicity is a significant benefit of the TCP paradigm. By mapping data onto one or more TCP connections appropriately, applications can implement various flavors of ALF.

We have discussed the performance challenges that arise and possible solutions, including algorithmic improvements at the end hosts (TCP Session), new TCP options (ack aggregation), and new API functions (for connection scheduling).

6 References

- [BPS+98] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, R. H. Katz, "TCP Behavior of a Busy Web Server: Analysis and Improvements", Proc. IEEE Infocom, March 1998.
- [CT90] D. Clark, D. Tennenhouse, "Architectural Considerations for a New Generation of Protocols", Proc. ACM SIGCOMM, August 1988.
- [Hei97] J. Heidemann, "Performance Interactions between P-HTTP and TCP Implementations", ACM SIGCOMM Computer Communication Review, April 1997.
- [KKK90] C. R. Kalmanek, H. Kanakia, S. Keshav, "Rate-Controlled Servers for Very High-Speed Networks", Proc. IEEE Globecom, December 1990.

- [Mog95] J. C. Mogul, "Network Behavior of a Busy Web Server and its Clients", Research Report 95/5, Compaq Western Research Laboratory, USA, October 1995.
- [Pad98] V. N. Padmanabhan, "Addressing the Challenges of Web Data Transport", PhD Thesis, University of California at Berkeley, USA, September 1998. <http://www.cs.berkeley.edu/~padmanab/phd-thesis.html>
- [PK98] V. N. Padmanabhan, R. H. Katz, "TCP Fast Start: A Technique for Speeding Up Web Transfers", Proc. Globecom Internet Mini-Conference, November 1998.
- [PM94] V. N. Padmanabhan, J. C. Mogul, "Improving HTTP Latency", Proc. 2nd International WWW Conference, October 1994.
- [Pax99] V. Paxson, "SLUMS BOF for Addressing RUTS Requirements", Mail to RUTS mailing list, February 1999.
- [RFC1379] R. T. Braden, "Extending TCP for Transactions – Concepts", RFC-1379, IETF, November 1992.
- [RFC1633] R. Braden, D. Clark, S. Shenker, "Integrated Services in the Internet Architecture: An Overview, RFC-1633, IETF, July 1994.
- [RFC2018] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgement Options", RFC-2018, IETF, April 1996.
- [RFC2140] J. Touch, "TCP Control Block Interdependence", RFC-2140, IETF, April 1997.
- [RFC2205] R. Braden et al., "Resource Reservation Protocol (RSVP) Version 1 Functional Specification", RFC-2205, IETF, September 1997.
- [RFC2475] S. Blake et al., "An Architecture for Differentiated Services", RFC-2475, IETF, December 1998.
- [RUTS98] "Requirements for Unicast Transport/Sessions (RUTS)", Meeting Report, IETF, December 1998.
- [Ste96] W. R. Stevens, "TCP/IP Illustrated, Volume 3", Addison-Wesley Publishers, 1996.
- [VH97] V. Visweswaraiiah, J. Heidemann, "Improving Restart of Idle TCP Connections", Technical Report 97-661, University of Southern California, USA, November 1997.