And those who were seen dancing were thought to be insane by those who could not hear the music.

– Friedrich Nietzsche

University of Alberta

Structured Message Transport

by

Shayan Pooya

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2012
Edmonton, Alberta

To my family.

Abstract

In this thesis, we present *Structured Message Transport* (SMT). SMT is a transport protocol coordinator designed to alleviate the head-of-line blocking problem of existing transport layer protocols including the most widely used, transmission control protocol (TCP).

SMT uses explicit dependency tracking instead of assuming total ordering between messages of a communication. Therefore, SMT can avoid head-of-line blocking that is caused by stream-based transport layer protocols.

Moreover, explicit dependency tracking creates opportunities for some optimizations. The first opportunity is using multiple paths. SMT can distribute the messages into more than one path. However, unlike the stream-based Multi-Path-TCP, SMT is not limited to a single stream of messages. Relaxing the ordering constraints between the messages makes it possible to deliver the received messages to the application layer if they do not have any unmet dependencies.

Another opportunity is traffic redundancy elimination. SMT employs a traffic redundancy elimination (TRE) module to remove repeated data segments. The operation of TRE creates some dependencies between the messages. SMT can track these dependencies efficiently without introducing any artificial dependencies.

We have designed and implemented a prototype of SMT to test our ideas. By experimental evaluation, we show that SMT can achieve higher throughput and lower latency than other communication mechanisms. Moreover, we have integrated the ideas from SMT into a proprietary software system and we show that the SMT version works better than the base version of this software system.

Acknowledgements

Although this thesis is published under my name, I would like to thank the people who mentored me throughout its development. First and foremost, I would like to thank my supervisors Dr. Mike MacGregor and Dr. Paul Lu for their guidance, patience and constructive feedback.

As well, I would like to thank Brian Lake and Dale Hagglund, my mentors at EMC® Corporation of Canada, who were a constant encouragement from the conception of my thesis to its conclusion. Also, I would like to thank my colleagues: Gary, Ron, Brad, Ying, Qinghua, Oguzhan and all of the other people who taught me a lot during my internship at EMC® Corporation of Canada.

Finally, I would like to thank EMC® Corporation of Canada for partially funding my project and providing me with the required resources for finishing my thesis.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IOPS	Input-Output Operation per Second
MP-TCP	Multi-Path TCP
SCTP	Stream Control Transmission Protocol
SMT	Structured Message Transport
SST	Structured Stream Transport
TCP	Transmission Control Protocol
TRE	Traffic Redundancy Elimination

Chapter 1

Introduction

We have designed and implemented Structured Message Transport (SMT), which is a transport layer connection coordinator. SMT sits on top of transport layer protocols and tries to alleviate the head-of-line blocking problem they suffer from. Moreover, SMT exploits some opportunities for increasing the performance of communication between the two end-hosts, including the throughput and latency of the communications.

When two end-hosts are communicating with each other via SMT, we say they have an *association* with each other. We avoid using the term *connection* to distinguish SMT associations from transport layer connections like TCP connections. The term association has been borrowed from stream control transmission protocol (SCTP) [2].

The two end-hosts of an association may wish to send messages with a set of dependencies. The dependencies of a message are the messages that must be delivered to the application layer at the receiver side before the message. These dependencies result in a set of ordering constraints – if message A is a dependency of message B, then A should be delivered before B.

The core idea behind SMT is explicit dependency tracking between messages. Each message maintains a list of all of its dependencies. This way, SMT tries to relax a lot of artificial and unneeded ordering constraints between messages.

For example, in the Hypertext Transfer Protocol (HTTP) application, the browser might need to retrieve a couple of images to show them to the user (see Fig. 1.1). In this case, there is no inherent ordering between the images. An image can be shown to the user as soon as it is received. However, transport layer protocols like TCP create a strict total ordering between images. These ordering constraints are artificial, and as we see in the next sections, result in inefficient communications.

The units of data transmission in SMT are messages. A message is an application-defined chunk of data that applications can send to or receive from SMT. With SMT, the application can specify some ordering constraints between the messages at the sender side. The SMT receiver makes sure that all of these ordering constraints are met before delivering the messages to the application.

By tracking dependencies, SMT no longer relies on the transport layer protocol nor expects it to provide any ordering between messages. However, it is still expected that a transport layer protocol will be able to deliver the exact same message that has been given to it. Therefore, the packets within a message cannot be reordered.

SMT's expectations from a transport layer protocol are:

1. congestion control
2. flow control
3. reliability

In the remaining sections of this chapter, we go through the potential use cases of SMT, the problems that SMT tries to solve, the opportunities SMT exploits and the contributions of SMT.

1.1 Use Case: Hypertext Transfer Protocol

As briefly mentioned before, Hypertext Transfer Protocol (HTTP) is one of the most widely used application layer protocols [15]. HTTP consists of two kinds of messages: requests from client (browser) to server and responses from server to the client [6]. Browsing a single Web page may result in multiple HTTP requests and responses (see Fig. 1.1).

For example, as shown in Fig. 1.1, a page may refer to some pictures and the browser may request those pictures after the HyperText Markup Language (HTML) page has been downloaded, resulting in one request-response pair for each image.

Although the only assumption of HTTP about the transport layer is reliability [6], HTTP is almost always deployed on top of TCP [15] and it maps the request-response messages to TCP connections. Inherently, TCP provides strict total ordering between HTTP messages. Meanwhile, HTTP can benefit from relaxed artificial ordering constraints between messages. There are different approaches for mapping HTTP messages to TCP connections.

One approach is to map each message to one TCP connection, which translates into one connection per each of message-pairs 1 and 2, 3 and 4, 5 and 6, and 7 and 8 in Fig. 1.1. Browsers using HTTP/1.0 can open several concurrent TCP connections and obtain data over those connections concurrently. However, opening each TCP connection has an overhead and results in high overheads for a single Web page.

Moreover, a page may contain many objects such as images, which increases the overhead of fetching all of the objects simultaneously, each one over a dedicated connection. The overhead of fetching pages is also important for servers that are supposed to be able to serve many clients simultaneously. Therefore, the browser must wait until some of the responses complete and then begin fetching remaining objects. Therefore, the time the user should wait for the whole page to load increases.

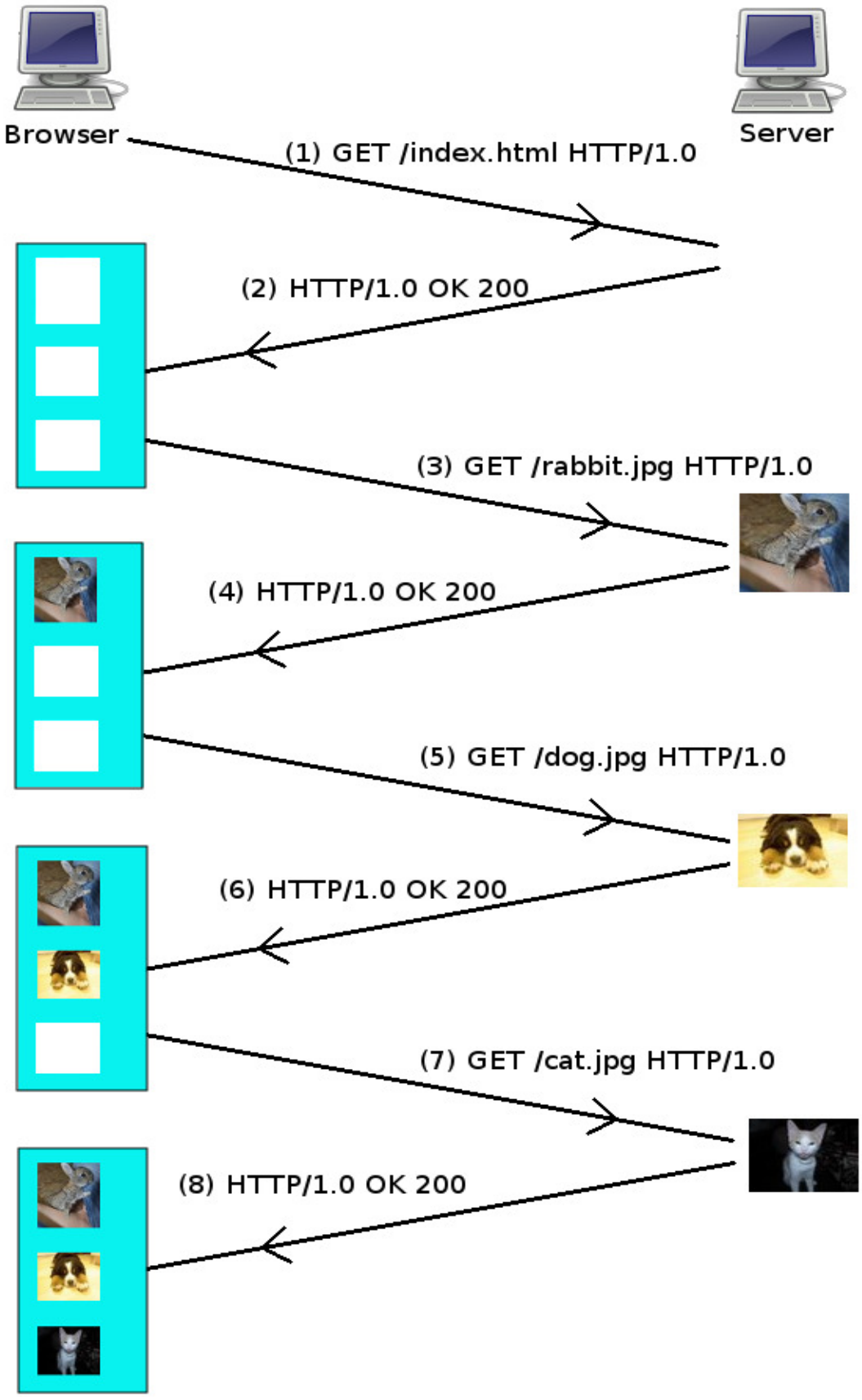


Figure 1.1: HTTP Communication

Another approach is to reuse TCP connections across HTTP messages. In our example in Fig. 1.1, reusing connections translates into one connection for all of the messages. HTTP/1.1 defines and recommends persistent connections and lets the client reuse TCP connections for multiple HTTP requests [7]. These requests can be either serialized or pipelined. In either case, they are multiplexed into a single connection, creating some artificial ordering constraints and suffering from potential head-of-line blocking (see Section 1.2).

With SMT, we can reduce the overhead by using a single association instead of all of these connections. This avoids head-of-line blocking and increases the chances of using multiple parallel paths effectively, due to the relaxed ordering constraints.

We should note that the ordering requirements of HTTP messages are a trivial special case of general ordering between messages – there are usually no ordering constraints between them.

SMT, unlike TCP, introduces no artificial ordering constraints. Therefore, SMT provides exactly what is needed, without introducing artificial ordering constraints and their attendant performance impacts. The results in the following chapters show that avoiding artificial ordering constraints can improve the performance of the communication.

Moreover, using SMT creates some opportunities for improving the communication between end-hosts. For example, SMT has a traffic redundancy elimination (TRE) module that tries to decrease the amount of traffic that is transmitted between the two end-hosts. For example, our HTTP communication scenario consists of a set of messages that were independent. With TRE, the payload of some of these messages might be modified to remove the redundant segments and replace them with pointers to the same data in a previous message. Therefore, the TRE operation creates some dependencies between previously independent messages. These dependencies can be efficiently encoded in sender side of SMT and be transferred to the receiver side of SMT.

We should note that the way the previously known transport layer protocols satisfy these dependencies is by creating artificial dependencies between all of the messages [1] in order to make sure the small number of needed ordering constraints are satisfied.

1.2 The Head-Of-Line Blocking Problem

The most widely used transport layer protocol in today's Internet is Transmission Control Protocol (TCP). TCP lets the application send a totally ordered stream of data. TCP provides reliability, congestion control, and flow control for data transmission.

However, TCP does not preserve the boundaries of the segments it receives from the application layer. TCP assumes that each time the application sends a data segment, it is the continuation of the previously sent data segments. In other words, TCP works on a stream of data.

This design decision of TCP simplifies the job of many applications that are actually

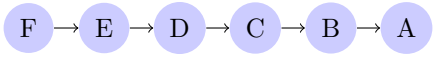


Figure 1.2: Head-Of-Line Blocking

using TCP to transfer streams of data. However, TCP is not the best choice for applications that work with individual messages.

Applications in this class including HTTP, have to parse the stream of data and find the boundaries of messages transmitted back-to-back in a TCP stream. Thus, one drawback of the streaming behavior of TCP is that it makes the applications more complicated and error-prone.

Moreover, message-oriented applications often multiplex a set of potentially independent messages into one stream of data. Therefore, if a message at the head of the stream is not completely received yet, or is delayed for any reason, none of the following messages can be delivered to the application layer. This problem is called head-of-line blocking [7, 10, 2].

For example, in Fig. 1.2, assume that there are some artificial ordering constraints between messages caused by using a totally ordered transport layer protocol like TCP. Head-of-line blocking occurs when the first message in the stream (A in this case) is delayed. In this case, none of the following messages can be delivered. Delivering these messages could have reduced the average latency of the messages, opened some room for future messages and reduced the memory needs of the communication.

The key observation is that some applications have different needs than the others. Treating all applications the same and providing the same totally ordered delivery for all of them, although simplifying, creates head-of-line blocking artifact for some of them.

1.3 Multiple Path Opportunity

With the advent of multi-homed servers and new redundant architectures for data centers, there are many hosts that are connected to each other by more than one path. TCP can use at most one of these paths, which is not the best use of the available resources.

Multi-Path-TCP (MP-TCP) is an active research project that tries to modify TCP to be able to use more than one path [11, 18, 19, 26]. MP-TCP uses a set of TCP connections and sends data segments over all of them. Therefore, in most cases, MP-TCP can achieve throughput as high as the sum of the throughput of all of the connections.

MP-TCP uses a large buffer on top of all of these TCP connections and serializes all of the segments of different connections in this buffer (see Fig. 1.3). This way, MP-TCP can provide the same behavior applications expect from a TCP connection.

However, there is a problem with this approach. Head-of-line blocking may now occur in a new form when more than one connection is being used for communication between two hosts. Various paths may have different bandwidths and latencies, resulting in different

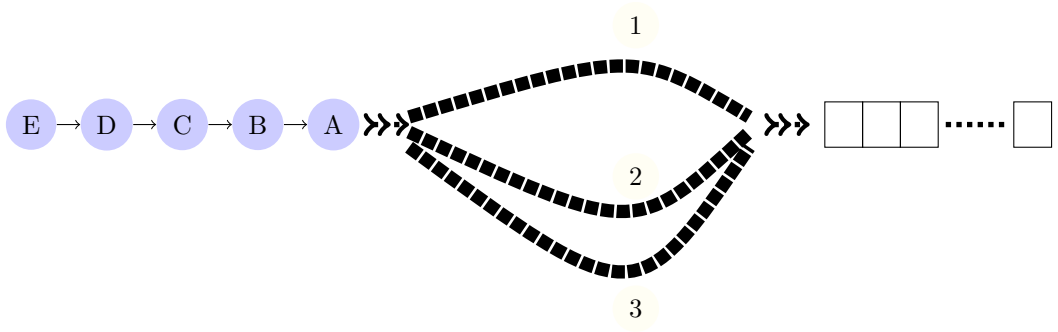


Figure 1.3: A stream of messages sent over a couple of connections.

delivery speeds over different connections. In MP-TCP, the average latency of messages is always bounded by the latency of the worst connection [26]. Although there have been some efforts to improve the performance of MP-TCP over asymmetric paths [18], these solutions are not general.

For example, in Fig. 1.3 a totally ordered stream of segments of data is sent over a set of connections. Assume that the round trip time (RTT) of connection 1 is the smallest, and the RTT of connection 3 is the largest. The first message to be delivered to the application is message A and if it is delayed for any reason, none of the following messages can be delivered. Therefore, if A is sent over connection 3, no matter what connection is used for the following messages, none of them can be delivered until A is received. Therefore, the latency of all of the messages would be bounded by the latency of connection 3 and would increase in order to maintain the total order between the messages.

Moreover, although MP-TCP can achieve high throughput for typical paths, in some cases the aggregate throughput of all of the paths can be smaller than the throughput of just using the best path. The main reason behind this phenomenon is the limited buffer size at the receiver side of MP-TCP. For example, in Fig. 1.3 assume that message A is sent over the worst connection and is received after all of the following messages. If the receiver had a buffer large enough to keep all of these messages and maximize throughput, then the only problem would be the high latency of the messages. However, if the buffer is not big enough, the receiver has to drop some messages in order to make room for message A. Therefore, it is possible for MP-TCP to end up with lower throughput than if just the best path was used. One might suggest using a large buffer at the receiver side. However, we note that the larger the buffer, the higher the average latency of the messages.

There have been suggestions for overcoming this issue in more recent MP-TCP publications [18], but we propose a more general solution to this problem by relaxing the artificial ordering constraints. If the artificial ordering between the messages were relaxed, the receiver could send some of the messages to the application layer and remove the need for extra large buffers, while also reducing latencies of those messages.

1.4 Contributions

The primary contributions of this work are:

1. proposing an architecture and a mechanism for explicit dependency tracking in communication.

2. designing a set of optimizations for our mechanism that can benefit from the relaxation of ordering constraints between messages.

3. designing a work-conserving[1] sender and receiver for SMT and implementing a prototype of the SMT protocol for experimental evaluation and performing an experimental evaluation.

1.5 Concluding Remarks

In this chapter we briefly introduced SMT and discussed the problems and opportunities that drove us to design it. We reviewed head-of-line blocking problem in transport layer protocols and showed how adding artificial ordering constraints to transport layer protocols can hurt their performance. In the next chapter, we talk about some previous work that has been done to solve head-of-line blocking. We also discuss the literature on top of which SMT is built.

[1] In TCP and TCP-like protocols, the sender is work-conserving (if there are some messages ready to be sent, one of them would be sent). However, the receiver is not work-conserving. The receiver blocks all following packets if one packet is not received (head-of-line blocking).

Chapter 2

Background and Related Work

Structured Message Transport (SMT) is built on top of some ideas from previous work. SMT is mainly about explicit dependency tracking and handling partial ordering in communication. Moreover, SMT is capable of using multiple paths simultaneously, which builds on top of other work in the field of multiple connection management. SMT also includes an optional traffic redundancy elimination (TRE) module which uses the approaches introduced in previous work. In this chapter we compare and relate SMT to these previous studies.

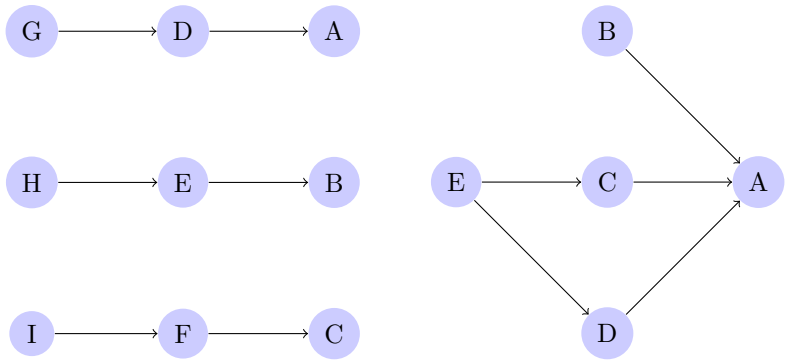
2.1 Ordering Constraints in Transport Layers

Partial ordering is not a new concept in transport layer protocols. In this section, we describe the previous work that has been done to embed the ordering constraints of messages in the communication. The goal of some of these previous studies is not specifically supporting dependency tracking. However, they all provide different approaches to avoid some of the artificial dependencies between messages.

One natural approach as outlined in RFC-1693 [4] is to group the packets and capture and encode all the ordering constraints between the packets in one group, creating an ordering graph of messages (see Section 3.1). The adjacency matrix of the resulting ordering graph is then sent to the receiver before any of the data packets. The adjacency matrix is then used to enforce the ordering constraints [4]. This approach is impractical for two reasons.

The first problem with this approach is that the graph size, say n, must be determined and packets can be sent in groups of n packets only. After that the adjacency matrix can be sent to the receiver. The graph size, n, should not be too small because the overhead of sending the adjacency matrix should be amortized over n packets. Nor can n be too big because the size of the adjacency matrix grows with n^2. Thus, finding the best value for n is a challenging task.

The second problem with the adjacency matrix approach is that after finding n, packets should be grouped at the sender by delaying them until n packets are accumulated. Then the matrix and the packets can be transferred to the receiver. However, none of these packets can be delivered to the application layer at the receiver until the packet containing the adjacency



(a) An ordering graph that can be encoded with SCTP

(b) An ordering graph that *cannot* be encoded with SCTP

Figure 2.1: Encoding the ordering constraints with SCTP

matrix of the ordering graph is received. Due to these problems and complexities, RFC-1693 has no public implementation and has been deprecated [5].

Another approach is employed by Stream Control Transmission Protocol (SCTP) [25] which was designed to solve some of the shortcomings of TCP. SCTP provides partial ordering between packets and overcomes head-of-line blocking [2]. In SCTP, instead of working with one totally ordered stream of data, the application sends streams of messages to the transport layer. These streams are transmitted to the receiver in parallel.

However, not all inter-message ordering constraints can be encoded with SCTP. In fact, SCTP only supports a set of totally ordered streams of messages. There is no structure for encoding the ordering constraints between these streams. For example, in Fig. 2.1b, the only way to capture the ordering of the messages is to put all of them in a totally ordered stream. Also, we should note that in addition to introducing head-of-line blocking, putting two unrelated messages in one stream makes it hard to treat them based on their priorities or applications. Thus, the partial ordering shown in Fig. 2.1b cannot be encoded with SCTP, unless it is augmented with some meta-data about inter-message ordering constraints. Ordering graphs like Fig. 2.1b are needed in some situations (e.g. see Section 2.3).

Intentional Networking [12] is another approach to avoid head-of-line blocking. The main purpose of intentional networking is labeling the traffic as high or low priority in the application layer and using these labels for that traffic to find the best available path.

An intentional network gets active and passive feedback from the connections and sends traffic over the most appropriate connection based on the labels of the traffic. The use-case that intentional networking has been designed for is a smart phone where the device has concurrent access to WiFi and 3G networks (Fig. 2.2). As shown in Chapter 1, maintaining a total order between the messages sent over these two paths would result in high latencies and probably lower throughput, because the latency and bandwidth of these two paths vary significantly. Hence, intentional networking makes use of partial ordering to avoid these

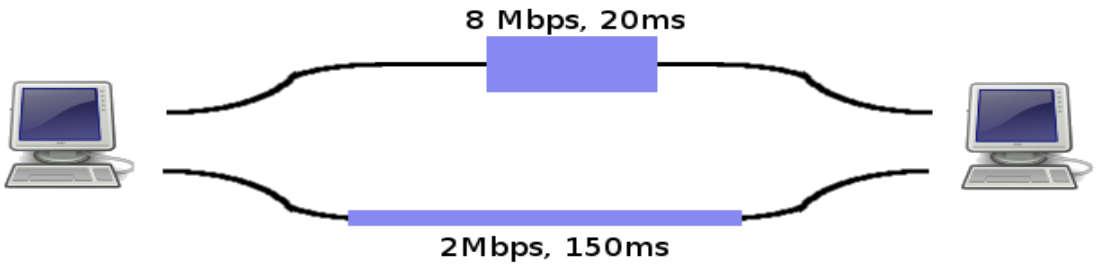


Figure 2.2: Two hosts that have two different paths (based on Costin Raiciu et al. [18])

problems. It maintains adjacency lists for all of the messages which include all of their dependencies. Adjacency lists are then transferred to the other side of the communication and used for enforcing the ordering constraints.

SMT, like intentional networking, uses the adjacency list of the messages. However, SMT focuses on ordering constraints and the optimizations that can be made with explicit dependency tracking rather than the priorities and labels of the traffic. Moreover, intentional networking is built on top of TCP and therefore uses the total ordering of TCP. In contrast, SMT does not depend on total ordering of the transport layer.

Moreover, SMT tries to be efficient in dependency tracking by employing novel data structures at the receiver side (see Chapter 3). SMT does not assume anything about the number of the connections nor does it need to label the connections as suitable for low priority or high priority traffic.

SCTP [25] and UDT [9] also have another approach for partial ordering. They provide two operation modes for the applications. The first mode, in which messages are totally ordered, is vulnerable to head-of-line blocking. In the second mode, a message can be sent out-of-order, providing the simplicity of streams of messages and the efficiency of unordered messages for special cases.

Also, there have been some efforts to add out-of-order messages to TCP. The *urgent* fields of TCP have been designed for the same reason [13]. There are some inconsistencies between the specification and implementation of the TCP *urgent* field in different operating systems. Therefore, TCP *urgent* cannot be reliably used in today's Internet for messages larger than one byte [8].

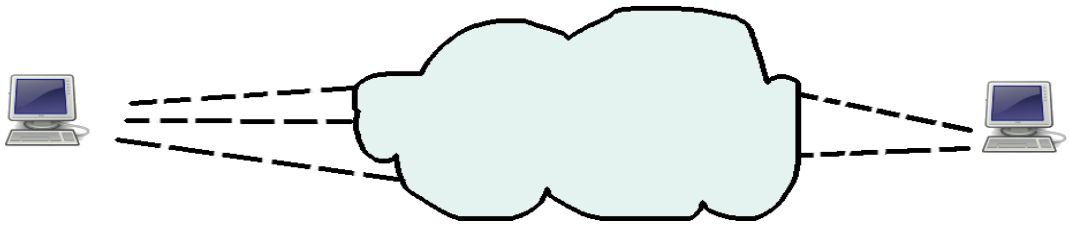


Figure 2.3: Two end-hosts with a couple of independent paths

Moreover, there are more recent research projects like Minion [17] that try to add out-of-order messages to TCP. Although out-of-order delivery might be useful for some applications, sending out-of-order messages is not a general solution for head-of-line blocking problem. For example, the ordering graphs of Fig. 2.1a or Fig. 2.1b cannot be sent without augmenting the messages with explicit dependencies.

SMT works best on top of transport layer protocols that support out-of-order messages (e.g. SCTP and UDT) because they do not incur any head-of-line blocking by themselves. That is one of the reasons that SMT was initially built on top of UDT.

Structured Stream Transport (SST) [7] is another research project that tries to avoid head-of-line blocking by creating cheap, short-lived streams of data. Streams are cheap, because they do not need an RTT for TCP's handshake [7]. Streams are also short-lived, which means for an application like HTTP, a SST stream does not span more than one HTTP request-response. SST makes the cost of creation and termination of streams negligible so that applications can use a new stream for each new message. Therefore, like SCTP, SST provides a method to send parallel streams and removes head-of-line blocking.

SST has been designed for HTTP and makes the assumption that there are no dependencies between different streams. This assumption is almost always true for HTTP. Therefore, SST works fine for its use case. However, there are other applications that need finer-grained control over the dependencies of messages.

SMT tries to solve the same problem as SST, but with a different approach. SMT works with messages rather than streams. A message can have any number of dependencies. Moreover, SST is a transport layer protocol and SMT has been designed to work on top

of transport layer protocols. Therefore, SMT can rely on the underlying transport layer protocol for reliability, congestion control and flow control. In contrast, SST must provide all of these functions itself. In summary, SMT, unlike SST, separates the ordering concerns from other transport layer protocol concerns and tries to provide the most general way for expressing the ordering constraints.

2.2 Multiple Connection Management

As discussed in Chapter 1, many hosts have access to more than one path for communication (Fig. 2.3). One of the key motivations for SMT was making it easy and efficient to use a set of connections. MP-TCP had the greatest influence on SMT in this area. In this section, we describe the achievements and problems of MP-TCP and the ways in which it has influenced SMT.

The goal of the MP-TCP project is to add support for more than one path to TCP [11, 19, 26]. Therefore, redundant paths would be actually used to transfer some traffic instead of just being used as fail-over resources.

MP-TCP tries to be transparent to applications, thus, it provides the same TCP interface to applications. TCP provides total ordering between segments of data so MP-TCP was obliged to provide the same semantic. This means, at the sender side, MP-TCP receives some segments of data, multiplexes them over multiple TCP connections and serializes them again at the receiver side so that they can be delivered to the application layer in-order.

MP-TCP employs a sequence number on top of the regular TCP sequence number. MP-TCP's sequence number is used at the receiver side for making sure the segments are in-order.

In most cases, the paths that MP-TCP operates on are similar. In other words they have approximately the same latency and bandwidth. However, this is not something that can be guaranteed. The latency and available bandwidth of any path is dependent on the competing traffic crossing the path. If MP-TCP is actually using different paths for connections, it will likely face different bandwidths and latencies on these connections (see Fig. 2.2).

As we discussed in the previous chapter, having different connections with different latencies can result in lower throughput because of limited receiver buffer size. Also, the average latency of the messages increases as we increase the size of the receiver buffer.

SMT uses the ideas from MP-TCP and tries to solve this problem. SMT embraces the different paths and tries to use the full capacity of each one. SMT operates on an ordering graph (Section 3.1) and maps an ordering graph to a set of connections. Thus, it has more potential for smarter connection selection. Chapter 3 documents the methods that SMT uses for solving this problem.

Another difference between SMT and MP-TCP is that SMT, being a user-space proto-

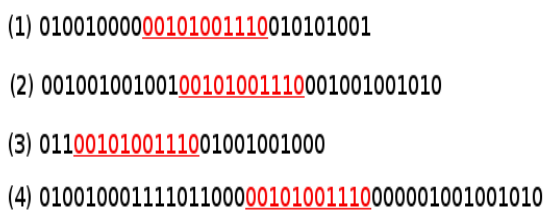


Figure 2.4: Packets with redundant data

col, can simultaneously operate on top of a heterogeneous set of transport layer protocols. In other words SMT can manage a set of TCP, UDT, SCTP and other transport layer protocol connections and decide on which connection to use based on the characteristics of the connections. In this case, SMT is more similar to Minion [17].

We should note however, that unlike MP-TCP, SMT does not have a coupled congestion control [26]. SMT assumes that the connections it is managing do not share any links. SMT is not fair to competing traffics if any two of its connections have a link in common. SMT can be told to use exactly one connection out of a set of connections. This way, SMT can be instructed to create multiple connections over a path and decide which one to use based on each connection's actual performance. Implementing an integrated congestion manager [3] for SMT remains a future work.

2.3 Traffic Redundancy Elimination

SMT has an optional traffic redundancy elimination (TRE) module. If enabled, it can reduce the amount of data that is transmitted by SMT. In this section, we introduce the concept of redundancy elimination in the end hosts.

As alluded to in the previous chapter, TRE's job is removing repeated data and replacing it with meta-data that can be decoded at the receiver side. For example, in Fig. 2.4, we see the content of four messages. As shown in this figure, there is a data segment repeated in all of these messages. Therefore, the task of TRE is to remove the repeated content from messages 2, 3, and 4.

TRE can be done at different network layers and at different locations between two communicating end-hosts. EndRE [1] was one of the first proposals that introduced TRE for end-hosts; it sits on top of TCP streams.

The operation of EndRE is simple. At the sender side, EndRE finds the redundant segments of data and replaces them with pointers to the original data segment. At the receiver side, EndRE reconstructs the messages using the meta-data in the pointer messages and the EndRE cache.

The reference messages of a pointer message should be received first at the receiver side

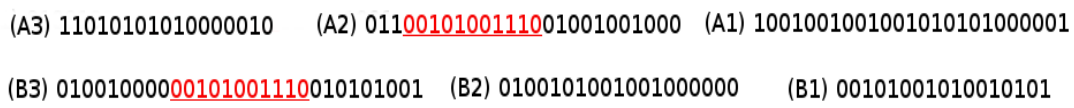


Figure 2.5: Packets from different connections and with redundant data

of EndRE because they have the actual data that should be used in pointer messages. Thus, EndRE creates ordering constraints between messages.

For example, in Fig. 2.5, assume EndRE is removing the redundancies of two parallel connections A and B, and there is a repeated data segment in messages A2 and B3. Assume A2 contains the original data and B3 is a pointer message referring to A2. Therefore, EndRE would need message A2 for decoding message B3, creating an ordering constraint between these two messages.

EndRE gives each message an increasing sequence number and uses the sequence number at the receiver side to create a total ordering between all of the messages, no matter whether they have any ordering constraints or not [1].

SMT's TRE module builds on top of EndRE's ideas, but also uses SMT's built-in dependency tracking capability and just adds another dependency to the pointer message for each reference message. Therefore, SMT's TRE module does not enforce a total ordering between possibly unrelated messages. For example, in Fig. 2.5, SMT's TRE module would add message A2 as one of the dependencies of message B3, as well as all of the ordering constraints between the messages within a stream. No other artificial dependencies would be added.

2.4 Concluding Remarks

In this chapter, we presented the background on top of which SMT is built. First we described the concept of partial ordering and dependency tracking. Then we talked about multiple connection management and the way MP-TCP approaches the problem. Finally, we talked about TRE and the ways we can reduce the amount of transmitted traffic.

In the next chapter we present the design of SMT. SMT is designed based on many ideas from previous work, briefly mentioned in this chapter, to solve the problems introduced in Chapter 1.

Chapter 3

Architecture and Design

In this chapter we explain Structured Message Transport (SMT) and describe the methods it employs. We begin with introducing the ordering graph which is a way for encoding the dependencies and tracking them. Then we describe the way SMT encodes the dependencies of messages and SMT's over-the-wire protocol. Then we move on to describing the sender side and the receiver side and their architecture and data structures. After that we present the optimizations that were possible and have been designed and implemented on top of SMT core. Finally, we briefly describe our prototype implementation.

3.1 Ordering Graph

RFC-1693 introduced the concept of an *ordering graph* [4]. An ordering graph is a graph in which the nodes are messages and the edges are dependencies. An edge from B to A in the ordering graph, shown $B \rightarrow A$ and read "B depends on A", means A is a dependency of B[1].

Fig. 2.1 shows two sample ordering graphs. The ordering constraints in Fig. 2.1b are: $B \rightarrow A$, $C \rightarrow A$, $D \rightarrow A$, $E \rightarrow C$, and $E \rightarrow D$.

The problem of encoding the ordering constraints translates into encoding this graph and sending it to the receiver. As described in Chapter 2, RFC-1693 [4] suggests using the adjacency matrix of this graph and sending it to the receiver before transmitting any of the messages. The adjacency matrix of the ordering constraints in Fig. 2.1b is:

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

We know that messages do not have forward dependencies. Thus the adjacency matrix of the ordering graph is lower-triangular, in other words all elements above the main diagonal are zero. Therefore, the lower half of the matrix is enough to encode and decode all of the ordering constraints.

[1]The meaning of $B \rightarrow A$ in this document is the reverse of its meaning in the RFC. In this document, $B \rightarrow A$ means B depends on A. However, in the RFC, $B \rightarrow A$ means B must be delivered before A.

3.2 Encoding Ordering Constraints

In SMT, the ordering constraints of messages are checked when they are being sent and received. SMT guarantees that messages leave the sender with a *valid ordering*, which means that if message B has dependency A, then A would be sent before B.

Messages are not necessarily delivered to the SMT layer at the receiver side in the same order in which they were sent. Thus, to ensure order validity of the messages, we must check the ordering of the messages at the receiver again. Therefore, the receiver needs a way to find out if an ordering is valid.

SMT uses two mechanisms to encode message ordering constraints: explicit dependencies encoded in SMT message headers, and *round numbers*. We describe the round number later in this section.

As alluded to in the previous sections, in SMT, each message maintains a list of its dependencies. To save space and reduce the overhead of communication, SMT sends the difference (delta) of the dependency message number and the current message's message number instead of the absolute message numbers. These differences are used instead of message numbers in order to save space in the SMT message header and fit more dependencies in the limited field size of 30 bits (see Section 3.3).

For example, if the current message number is 101,000 and one of its dependency's message numbers is 100,000, instead of putting 100,000 as the dependency, SMT puts the delta of the two message numbers, which is 1000, in the message header. The receiver is aware of this mechanism and recalculates the message number of the dependency as 100,000.

SMT always tries to encode all of the ordering constraints with explicit dependencies in the message header. However, if this is not feasible, it falls back to round numbers. There are three fallback situations:

1. The message has more than three dependencies in the current round. There is no room in SMT message header to accommodate more than three dependencies.

2. At least one of the dependencies is from the same round and the difference computed for it cannot fit in the available space in the message header.

3. The message number overflows. Handling message number overflow makes the code complicated. For simplicity, SMT just increases the round number in case of message number overflow. The message number field is 32 bits. Therefore, this case rarely happens and its effect is negligible.

At the beginning of a new round, SMT resets the message numbers and sends at least one *reset message*. Reset messages are just like regular messages, but they do not have any dependencies in the current round. Reset messages should signal the receiver not to wait

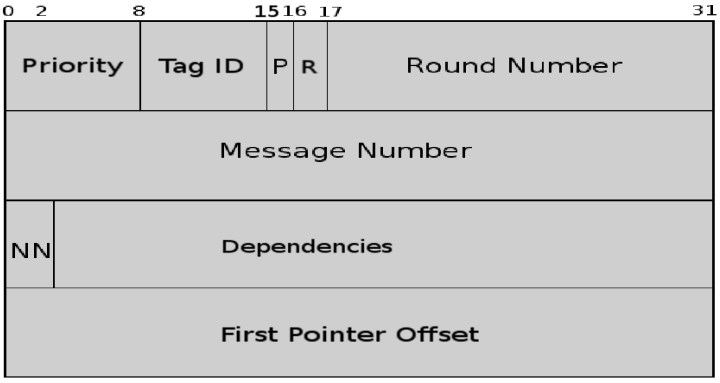


Figure 3.1: SMT Protocol Header

for any more messages from the previous round. Therefore, they contain the last message number of the previous round, called the *Termination Number*. On reception of a reset message, the receiver uses the termination number to determine whether all messages of the previous round have been received; if so, SMT's receiver begins processing the messages of the new round. Otherwise, if there are messages from the previous round that have not been received, SMT does not increase the round number.

Increasing the round number has the potential of creating artificial dependencies and head-of-line blocking. If the number of inherent dependencies of messages is high and the in-degree of messages in the ordering graph is higher than three in an application, then we can increase the size of SMT header to accommodate more than three dependencies. The applications that are currently using SMT do not need more than three dependencies per message.

We have designed another mechanism for identifying separate directed acyclic graphs (DAGs) of an ordering graph. To support a dynamic number of dependencies, SMT message headers can be augmented with DAG-IDs. The messages with different DAG-IDs would have independent round number and message number spaces. As mentioned earlier, the current applications do not need more than three dependencies per message. Therefore, we have not implemented this feature in our SMT prototype.

3.3 SMT Protocol

In this section, we describe SMT's over-the-wire protocol. As discussed in Chapter 1, SMT can be deployed on top of various transport layer protocols. The SMT sender still needs a header to communicate the SMT-specific information to the receiver.

Fig. 3.1 illustrates the header of an SMT message. As the name suggests, *Priority* is the preemptive priority of a message (see Section 3.4.1). In the current design and implementation of SMT, priority is only used at the sender side. It is used to show which message should be chosen from the priority scheduler. The value of the preemptive priority

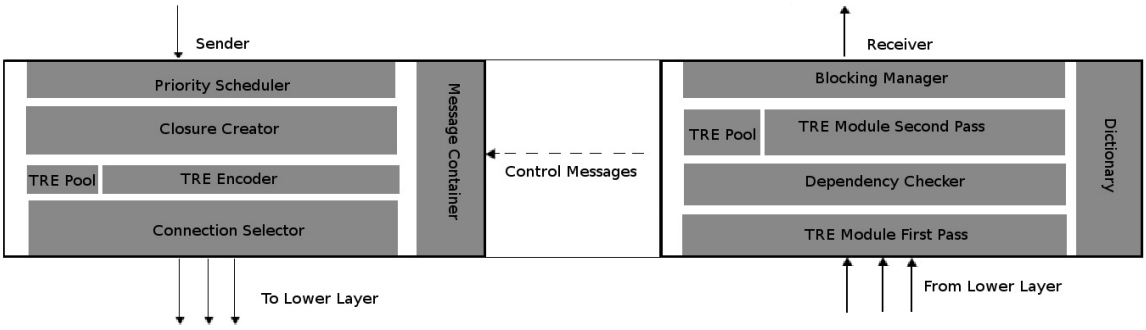


Figure 3.2: Architecture of the SMT layer

is transferred to the receiver side for future use.

Tag ID is a 7-bit number that identifies the application that generated the message and is waiting for the message at the receiver side. SMT reserves the Tag ID of zero for its control messages. Therefore, SMT can support up to 127 different applications at the same time. Within an SMT association (as defined in Chapter 1), Tag IDs uniquely identify the thread waiting for a message. The blocking manager (Section 3.5.2) then uses Tag IDs to block the threads that issue a *recvmsg* (Table 3.1) when there are no messages in the SMT's receive buffer for them.

The P bit shows whether a message is a pointer message or not. A pointer message is a message whose payload has been modified and some of its data chunks have been replaced with meta-data in SMT's traffic redundancy elimination (TRE) module. If this bit is set, the fourth double-word of the header (First Pointer Offset) points to the first pointer in the payload of the message. The fourth double-word of the header is zero otherwise.

The remaining parts of header are used for encoding the ordering constraints between the messages as described in Section 3.2. R is set for reset messages and unset for other messages. *Round Number* and *Message Number* show the round number and the message number respectively. *NN* shows the number of dependencies. And finally, *Dependencies* shows the dependencies of the message.

The use of the third double-word in the SMT message header is different for reset messages. Reset messages, by definition, do not have any dependencies in the current round. Therefore, they do not need the third double-word for their dependencies. The third double-word in reset messages is used for sending the termination number of the previous round to the receiver.

3.3.1 Control Messages

SMT has a set of control messages that are used to signal the status of the receiver of an end-host to the sender of the other end-host. SMT uses the Tag ID of zero to show control messages. The current implementation has the following control messages:

1. ACK. Although SMT relies on the transport layer protocol for reliability, it should still provide reliability in case a connection fails. Therefore, SMT should know when a message has been delivered to the application layer at the receiver side. Therefore, once in every constant interval, the SMT receiver sends an ACK to the sender side of the other hosts. This ACK contains the *smallest* of the *delivered* message number dictionary (see Section 3.5.1). Implementing a selective acknowledgment remains future work.

2. STOP. SMT tries to notify the sender side when a receiver application cannot keep up with the receiving rate and the application's receiver buffer is filling up. This control message warns the sender to slow down. Otherwise, messages may be dropped.

3. RESTART. The SMT receiver sends a RESTART control message to the sender side to show the end of STOP condition. Therefore, RESTART shows that the receiver can accept messages.

4. DROP. The SMT receiver might have to drop some messages when it does not have enough room in its receiver buffers. As discussed previously, unlike totally-ordered protocols, a large buffer does not increase the latencies of the messages in SMT. However, end-hosts usually allocate a limited amount of memory to communication. Therefore, a message might have to be dropped in the SMT layer. The SMT receiver of one side sends a DROP control message to the sender of the other side to notify it that the receiver's buffer has filled up and a message has been dropped. SMT's receiver buffers are per application. In other words, the number of messages SMT keeps for each application is limited.

DROP, STOP and RESTART messages form the control flow of the SMT layer. The difference between DROP and STOP control messages is that DROP messages signal loss of a single message while the STOP messages tell the sender to stop sending messages of the specified application.

3.4 Sender Side

At the sender side, SMT tries to find the most appropriate messages and send them over the proper connection. The SMT sender completely decouples these two steps. As shown in Fig. 3.2, the SMT sender consists of three main components plus an optional TRE component. The first component is the priority scheduler that selects the message with the highest priority. The second component is the closure creator that finds the messages that should be delivered before the message selected in the priority scheduler. The third component is the connection selector that finds the most appropriate connection and assigns the messages found in the closure creator component to it. The TRE module is an optional

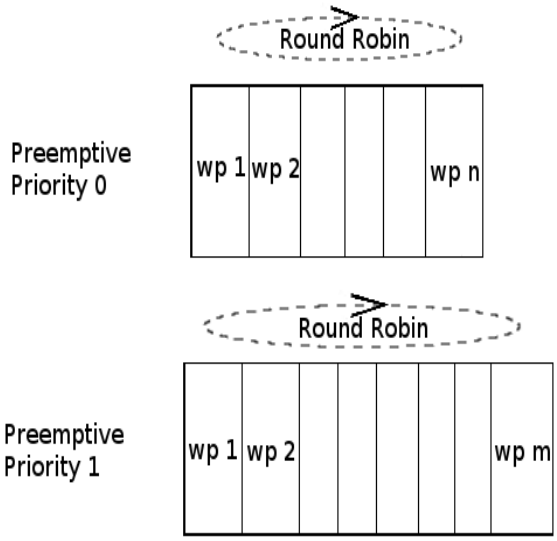


Figure 3.3: SMT priorities. WP stands for wdrr-priority and is used by the WDRR as the weight of classes.

component that tries to find and replace the repeated segments of data in the payload of the messages.

3.4.1 Priority Scheduler at Sender Side

The task of the priority scheduler is to find the highest priority message to be sent. SMT, similar to Spdy [24], SST [7], and intentional networking [12], has priority scheduling. Applications can assign priorities to messages, and SMT decides on which message to send based on the message's priorities.

SMT supports two priority types. The first type is called *preemptive priority*. If messages A and B are queued in the SMT layer and A has a higher preemptive priority than B, then A is chosen first, no matter what the status of the scheduler is.

The second type of priority is called *wdrr-priority*. If all of the messages buffered in the SMT layer are of the same preemptive priority, the SMT scheduler uses a round-robin approach to go through all of the wdrr-priorities. A share of the available bandwidth is used for each of these wdrr-priorities. SMT uses Weighted Deficit Round Robin (WDRR) [22] for scheduling messages with the same preemptive priority (Fig. 3.3).

For example, Fig. 3.3 shows a priority scheduler with two preemptive priorities. If there is any message from the higher preemptive priority, the scheduler chooses it. Otherwise, the lower preemptive priority is chosen. Next, based on the remaining share of the wdrr-priorities, a wdrr-priority is chosen within the selected preemptive priority. The final output of the priority scheduler is the first message of the selected wdrr-priority.

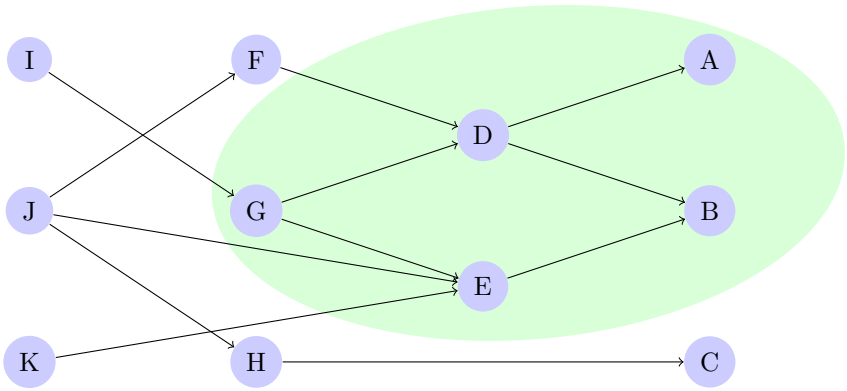


Figure 3.4: Dependency closure of message G

3.4.2 Dependency Closure Creator

The *dependency closure* of message G is the set of all messages in the ordering graph that G has a path to. In other words, the dependency closure of message G is the set of all messages that must be delivered before G can be delivered (see Fig. 3.4).

When a message (e.g. message G) is chosen based on its priorities, the dependency closure of G is constructed. The connection selector, as discussed below, then uses the dependency closure as the unit of assignment of messages to connections. In other words, all of the messages in a dependency closure are assigned to the same connection.

For example, assume message G is chosen in the priority scheduler in Fig. 3.4. The dependency closure of message G then would consist of messages: A, B, D, E, and G. Therefore, all of these messages would be sent over one connection selected in the connection selector.

3.4.3 Connection Selector

After a set of messages is chosen to be assigned to a connection, one of the connections is chosen and the messages are sent over it. By connection, we mean a transport layer connection that is managed by SMT. SMT gets feedback from the connections to find out which connection is most suitable for sending these messages.

The main criterion used for ranking the connections is the estimated latency of the set of messages if sent over that connection. SMT uses different approaches to calculate an estimate of the latency of the messages. Some of the values used in these estimates are:

1. Sending Rate in bytes per second: If a connection has been sending data with high rates, it will probably send data with the same rate in the near future.

2. Round Trip Time (RTT) in seconds. Transport layer protocols usually maintain their RTT to be used for congestion control. RTT is a good measure of how long a single byte would be in transit. Some protocols like UDT [9] have a mechanism for the clients

to get the RTT of the connection. However, TCP does not have a standard way to get the RTT of a connection[2].

3. Buffered Data in bytes. The amount of data that is buffered at different layers of the sender side. Most transport layer protocols use first-in-first-out service. Therefore, data buffered at these layers would be sent over-the-wire before any messages that SMT sends in the future.

SMT maintains an exponential weighted moving average (EWMA) [16] for smoothing all of these values. Implementing a smarter method like flip-flop method employed in Intentional Network [12] remains a future work. These values are used to estimate the latency of a path which will then be used to rank the connections and select the one with the smallest estimate.

Moreover, SMT can modify the way it chooses the most appropriate connection. One way to modify this behavior is by taking into account the size of the dependency closure and its priority. If a dependency closure has a low priority and a large size, it might be useful send it over the higher latency connection [12]. Adding other factors to the selection criteria, like the price of a connection per bit, remains a future work.

Algorithms 1 and 2 are used for querying the transport layer protocols and selecting a connection respectively. Query-time is the time the statistics of the connection are retrieved from the underlying layer. Selection-time is the time that the gathered statistics are used to estimate the new estimate for each connection. The connection with the smallest estimate is chosen for sending the dependency closure.

Algorithm 1 Query-Time

$queryTime \leftarrow getCurrentTime()$
$tlpOccupied \leftarrow getTlpOccupied()$
$rtt \leftarrow getTlpRtt()$
$sendRate \leftarrow getTlpSendRate()$
$plr \leftarrow getTlpPlr()$
$bandwidth \leftarrow getTlpBandwidth()$
$estimateThroughput0 \leftarrow \frac{mss}{\sqrt{plr \times rtt}}$
$estimateThroughput \leftarrow \min(\max(sendRate, estimateThroughput0), bandwidth)$

Algorithm 2 Selection Time

$timeDiff \leftarrow getCurrentTime() - queryTime$
$remainingBytes \leftarrow tlpOccupied - sendRate \times timeDiff + dataSentSinceQuery$
$estimate \leftarrow \frac{rtt}{2} + \frac{closureSize}{\frac{sendRate + estimateThroughput}{2}} + \frac{remainingBytes}{sendRate}$

[2]There is a Linux-only TCP option called TCP_INFO that can be used to get the RTT of a TCP connection. However, we found that the RTT obtained with TCP_INFO does not have enough precision to be used for estimating connection delay.

3.5 Receiver Side

The receiver side of SMT is responsible for delivering the messages to the application layer in a valid order. In other words, all of the dependencies of any message should be delivered to the application layer before the message itself.

One simple way to satisfy all of the ordering constraints is delivering messages with a total order. In other words, delivering messages based on their increasing message numbers can satisfy all of the ordering constraints. However, a totally ordered delivery defeats the purpose of dependency tracking (Chapter 1). The sender side of SMT encodes all of the ordering constraints and sends them to the receiver side of SMT. Therefore, the receiver has all information it needs about the ordering constraints.

The goal of SMT is removing all artificial dependencies and head-of-line blocking. Therefore, SMT avoids any unneeded waiting at the receiver side. SMT guarantees that a message whose dependencies are delivered to the application layer is delivered as soon as it is received.

The SMT receiver uses a set of data structures to achieve this goal. In the following we describe the data structures and mechanisms that SMT employs to ensure that the messages are delivered in a valid order.

3.5.1 Message Number Dictionary

The message number dictionary is a data structure that the SMT receiver uses for efficient lookup of message numbers. Multiple instances of the message number dictionary are used in the SMT receiver for keeping track of received messages, delivered messages, etc.. This data structure supports the following operations:

1. insert(). Inserts a message number in the message number dictionary.
2. contains(). Returns true if the argument has been inserted in the message number dictionary before.
3. isFullRange(). Checks to see if all of the message numbers smaller than the argument, have been inserted in the message number dictionary before.
4. clear(). Empties the message number dictionary.
5. getSmallest(). Returns the smallest number that has not been inserted in the message number dictionary yet.

Although a simple binary search tree could provide all of these functions, we needed a more efficient and lower memory-footprint data structure. The key observation for designing the data structure is that although there is no strict rule for the insertion of the message numbers in the message number dictionary, they are inserted almost in-order. In other words, although messages might get reordered within a connection or between a set of connections, the number of reorderings is small.

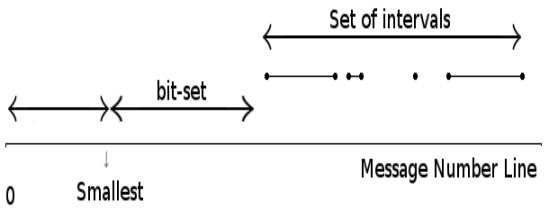


Figure 3.5: Three parts of a message number dictionary covering all message numbers.

Based on this observation, the message number dictionary consists of three parts (see Fig. 3.5):

1. Smallest absent number (or "smallest" for short) shows the first message that has not been inserted in the message number dictionary. At query-time, the argument is compared to this number and if it is smaller than smallest we are sure that it has been inserted in the message number dictionary before.

2. A sliding bit-set with a fixed size. This bit-set covers the interval beginning from the smallest. For each number in this interval, if the element in bit-set is one, then the message has been inserted before. If the element is zero, the message has not been inserted.

3. A binary search tree of intervals of inserted message numbers beginning from where the bit-set ends. If the queried message number is bigger than the biggest number the bit-set covers, it will be compared to the elements of this set. Queries referred to this tree have $O(\log n)$ time complexity where n is the number of intervals.

For example, assume that the size of the bit-set is 8 bits and the message numbers 0, 1, 2, 4, 5, 6, 7, 10, 11, 12, 13 14, 15, 20, 24, and 25 have been inserted. The smallest would be 3 which is the smallest number that has not been inserted in the message number dictionary. The bit-set starts from 4 and includes a bit for message numbers 4 to 11. The bits of the bit-set would then be 11110011, where the zeros show the absence of message numbers 8 and 9 in the message number dictionary.

The remaining values should be inserted in the interval tree (Fig. 3.5). The intervals would be [12, 15], [20, 20], and [24, 25]. These three intervals would be held in a binary search tree with the key being the beginning of the interval.

With this status, the getSmallest() method would return 3 as the smallest number. If a message number is queried, it would be compared with 3. If it is smaller, then it has been inserted before and contains() returns true. If the message number is greater than smallest, then it would be compared with 11 which is the end of the bit-set. If the queried message

number is smaller than 11, it is checked in the bit-set. Otherwise, the message number would be checked against the binary search tree.

3.5.2 Blocking Manager

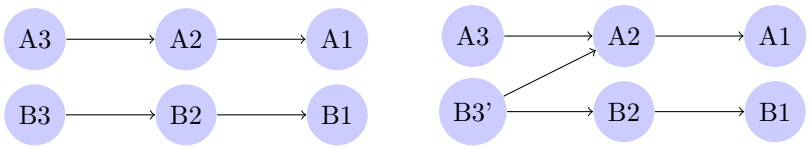
SMT's receiver is work-conserving. In other words, it delivers a message to the application layer when the application has requested a message and there is at least one message for the application with no unmet dependencies.

Whenever a new message is delivered, other messages may become eligible to be delivered. The blocking manager is a utility responsible for determining the application that can be unblocked.

Whenever a message is received and cannot be delivered because of unmet dependencies, the blocking manager sets a hook on the first unmet dependency of the message and blocks the receiving application. When the unmet dependency is delivered, the blocking manager wakes up the blocked receiver application. The dependencies are rechecked and if there is a message with no unmet dependencies, it will be delivered to the application. The benefit of this approach is that any message is delivered as soon as it has no unmet dependencies.

For example, in Fig. 3.4, assume the order in which the messages are received is: A, D, B, C, J, F, G, H, E, I, and K. The behavior of the blocking manager on reception of these messages is as follows:

1. A is received, it has no unmet dependencies. Therefore, it is delivered.
2. D is received, the blocking manager finds out that D depends on B. Therefore, the application requesting D blocks until B is received.
3. B is received and delivered. Moreover, it wakes the application blocking for D. When this application is woken up, it rechecks the dependencies of D and since all of its dependencies have been delivered, D can be delivered.
4. C is received and delivered.
5. J is received. It has three dependencies and the blocking manager sets a hook on one of them, say F.
6. F is received and delivered. The application blocking for J is woken up, rechecking the dependencies of J and finding out that H and E have not been received. Therefore, it sets a hook on another dependency of J, say E, and blocks again.
7. G is received, sets a hook on E.
8. H is received and delivered. Although, it satisfies another dependency of J, no hook has been set over H. Therefore, the application waiting for J does not have to wake up at all.



(a) Before TRE (b) After TRE

Figure 3.6: Traffic Redundancy Elimination

9. E is received and delivered. The applications waiting for G and J are unblocked and G and J are delivered.

10. I is received and delivered.

11. K is received and delivered.

Therefore, the order in which the messages are delivered is: $A, B, D, C, F, H, E, G, J, I, K$.

3.6 Optimizations

Explicit dependency tracking makes it possible to enhance the communication between two end-hosts. In this section, we describe the opportunities of dependency tracking and the ways SMT tries to exploit them.

3.6.1 Priority Inheritance

As discussed in Section 3.4.1, SMT has a priority scheduling module. The priority scheduler chooses a message based on its priorities relative to the others available and hands the chosen message to the dependency closure creator.

The unique feature of SMT is that messages can have dependencies of different priorities. For example, in Fig. 3.4, message G has the highest priority and is chosen by the priority scheduler. Messages A, B, D, E, and G form the dependency closure of message G. Therefore, G donates its priority and share (of WDRR) to the messages in its closure. This is known as priority inheritance. Priority inheritance is required to make sure that the highest priority messages are delivered as soon as possible. High priority messages should not be delayed because their low priority dependencies do not get enough bandwidth to be sent.

3.6.2 Traffic Redundancy Elimination

SMT has an optional traffic redundancy elimination (TRE) module that tries to reduce the amount of data transferred in the SMT association. As briefly mentioned earlier, the job of the TRE module is detecting repeated data segments and replacing them with pointers to reference data segments. The TRE pointer messages refer to previously sent messages. Therefore, a TRE pointer creates a dependency, which is taken care of in the SMT layer. This is the reason the TRE module is implemented in the SMT layer.

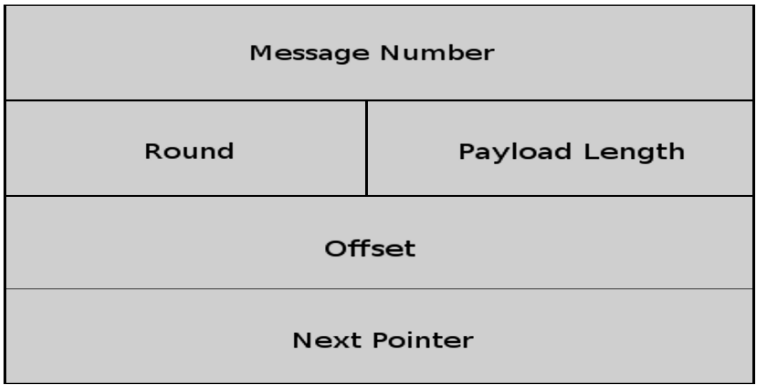


Figure 3.7: TRE Pointers

For example, Fig. 3.6 shows an abstraction of Fig. 2.5 in which messages $A2$ and $B3$ have redundant data. SMT's TRE module finds the repeated data segment and adds $A2$ as a dependency of $B3$.

At the receiver side, SMT goes through the payload of the message twice (Fig. 3.2). During the first run, it decodes all of the dependencies from the payload of the message. Then SMT's dependency checker makes sure all of the dependencies are received before the second run. When all of the dependencies have been received, the second run over the payload is executed. All of the pointers are replaced by data from the reference messages.

Fig. 3.7 shows a pointer that would replace a repeated data segment at the sender side. The round number and message number identify the reference message that contains the actual data. The offset and payload length show the offset of the data segment in the reference message and the size of the omitted data segment. The next pointer shows the offset of the next pointer, if any, in the current message. The next pointer lets us pass over all of the pointers and distinguish pointers from data segments.

3.7 Implementation

We have implemented a prototype of SMT as a proof of concept, and in order to compare its performance with other transport layer protocols. The prototype is about 6000 lines of code in C++ and it currently works on top of UDT and SCTP. The reason we chose these two transport layer protocols is that they support out-of-order delivery of messages.

We had to modify the implementation of these protocols in order to accept a scatter-gather list rather than a simple payload. The implementation of choice is SMT over UDT, which performs best in our benchmarks.

3.7.1 API

We tried to keep the interface of SMT similar to that of Berkeley sockets. However, we had to make some modifications so applications can express message dependencies and priorities. The results of these modification is shown in Table 3.1. Table 3.2 shows the SMT opaque

recvmsg(SmtSocket socket, SmtMessage message, AppId id, SmtToken token)
deleteMsg(SmtToken token)
sendmsg(SmtSocket socket, SmtMessage message, Priority priority, SmtDependencyList depList, AppId id)
createSocket()
accept(SmtSocket socket, Port port)
addPath(SmtSocket socket, IP ip, Port port)

Table 3.1: SMT API

Type	Meaning
SmtSocket	The socket used by the SMT layer. Returned by createSocket() call.
SmtMessage	The content of a message which can be a scatter-gather list.
SmtToken	An internal type of SMT which should be given back to it when SMT is done with a message.

Table 3.2: SMT Types

types, which are defined to encapsulate SMT's internal data structures.

3.8 Concluding Remarks

In this chapter we introduced the internals of SMT and how it can efficiently avoid head-of-line blocking. We introduced the ordering graph and described how SMT encodes the ordering constraints of messages. Moreover, we introduced SMT's over-the-wire protocol as well as the sender side and receiver side of SMT, and the novel data structures they employ. We talked about the optimizations that can achieved with SMT. Specifically, we described the TRE module of SMT and how it uses the dependency tracking feature of SMT to do TRE between independent streams of data.

In the next chapter we provide the results from some benchmarks to show what SMT achieves in practice. We compare the SMT prototype described in this chapter with other well-known transport layer protocols including TCP and SCTP.

Chapter 4

Experimental Evaluation

In this chapter we compare Structured Message Transport (SMT) with other communication mechanisms and investigate the effectiveness of SMT. We begin by describing the test-bed we use for running the benchmarks. Next we present our test application and its synthetic ordering constraints. Then we show that this application can achieve a higher performance with SMT.

For example, when communication is through a single path with 200 ms latency, as the packet loss ratio increases from 0 to 0.01, the average round trip time (RTT) of messages using SMT remains under 900 ms. In contrast, in the same configuration, the average RTT of the messages using UDT is higher than one second. The average RTT of the messages using other mechanisms are higher than UDT.

Furthermore, we compare SMT with a proprietary communication mechanism. We describe the architecture of this system before and after integration. We show that by employing the ideas from SMT, we can double the throughput and the number of input-output operations (messages) per second (IOPS) in the configurations where two paths connecting the end-hosts are not symmetric.

4.1 Test-bed

For our experiment, we create a test network consisting of two end-hosts and a dummynet [20] between them (Fig. 4.1). The dummynet makes it easy to compare the communication mechanisms over different configurations (e.g. different bandwidths, packet loss ratios, and latencies). Each end-host is equipped with an Intel Xeon E5620 2.4 GHz CPUs (quad core, two hyper-threads per core) and 36 GB of memory. The operating system of the end-hosts is Suse Linux Enterprise Server 11 with Linux kernel 2.6.27.

4.2 SMT Prototype and Synthetic Ordering Constraints

We implemented a simple client-server application to compare SMT with other communication mechanisms (e.g. TCP, UDT, and SCTP). The client sends messages to the server over the forward path. At the server-side, the messages are delivered to the application layer. For every received message, a response is issued, which is then transferred to the client over the return path. The client application measures the RTT of the message when the response

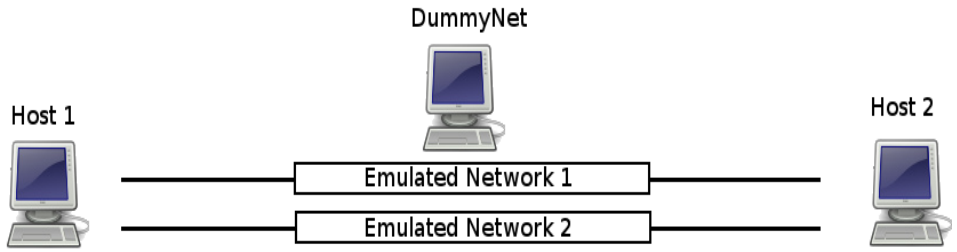


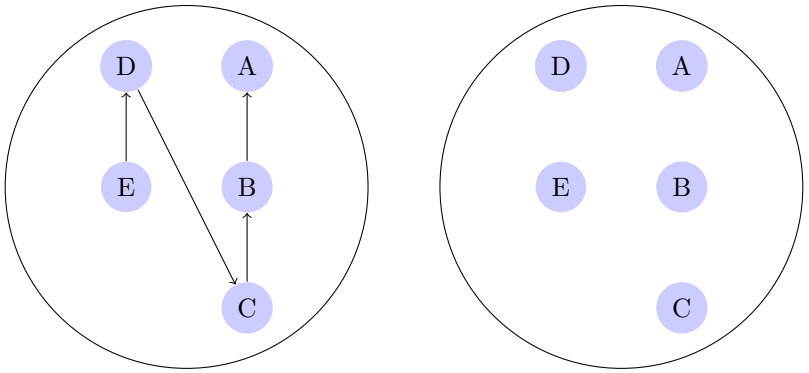
Figure 4.1: Test-bed used for running the benchmarks

is received.

The forward messages (from host-1 to host-2 in Fig. 4.1) have some ordering constraints to test the capabilities of SMT in tracking dependencies. We use an available mechanism in each of the communication mechanisms to ensure that messages are delivered to the server application in a valid order. In UDT, whenever needed, the order bit is used to indicate that all of the previous messages should be delivered before a message. For SCTP and TCP we use a single stream in an association and a connection respectively. TCP is not message-oriented. Therefore, we added a message format (header and payload) to its applications. We specify the length of the payload in the message headers so that we can decode the messages at the received side.

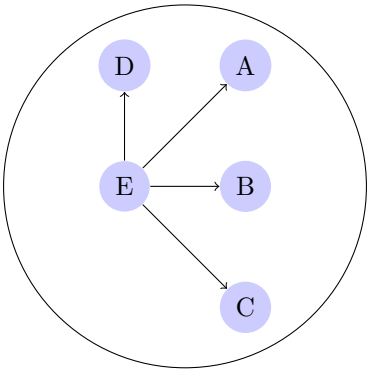
For our prototype, we use the standard TCP included in this Linux kernel, UDT version 4.10, and the latest available versions of SCTP. In case of TCP, SCTP and SMT over SCTP, we set the NO_DELAY option of the connection (association) to disable Nagle (Nagle-like) algorithm [14, 21] in order to obtain lower latencies. We only use one of the paths of our test-bed when comparing these mechanisms.

We synthesize ordering constraints using the three building blocks shown in Fig. 4.2. The first building block is a stream of sequential messages. A stream of messages is a recurring pattern of messages for applications that use a set of messages to transfer a large stream of data (like a large text file) that should be processed and consumed sequentially. The second building block is a group of parallel messages, representing the applications that do not need any ordering constraints between messages. The third building block is a set of parallel messages with a barrier message (see Fig. 4.2c). This third building block is an attempt



(a) A Stream of Messages

(b) A Set of Parallel Messages



(c) A Set of Parallel Messages with a Barrier

Figure 4.2: Building Blocks of Synthetic Ordering Constraints

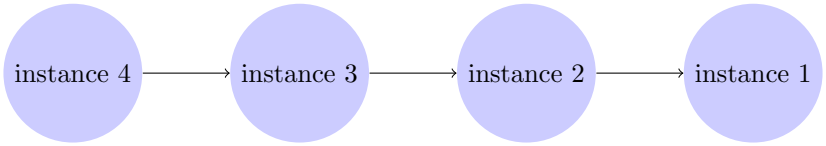


Figure 4.3: The synthetic ordering constraints of the application. Each circle is an instance of one of the three building blocks.

to emulate the applications (e.g. multimedia and remote database applications) generating messages with partial ordering constraints in the style introduced in RFC-1693 [4]. A random number between 2 and 10 is chosen as the number of dependencies of the barrier messages (like messages E in Fig. 4.2c).

All of the ordering constraints of messages of the application consist of a stream of instances of the building blocks (Fig. 4.3). The edges in this figure show that each message in a building block instance must be delivered after all of the messages of all of the previous building block instances.

We use the Harpoon message generation method for inter-message arrival times [23]. We assume that applications have two phases: message generation and computation (or sleep). In the message generation phase, the application generates a message, sleeps for a

short period of time (intra-batch sleep time) and then sends a message again. Once the application has sent a fixed (batch size) number of messages it sleeps for a longer time (inter-batch sleep time) and then the next round begins.

Fig. 4.4 shows the comparison of the five protocols. To cover multiple configurations in one graph, the x-axes of these graphs show both the path latency and the packet loss ratio. Also as there was a large difference between the results, we use log-scale for the y-axes. Each value reported in these figures is the average of seven runs.

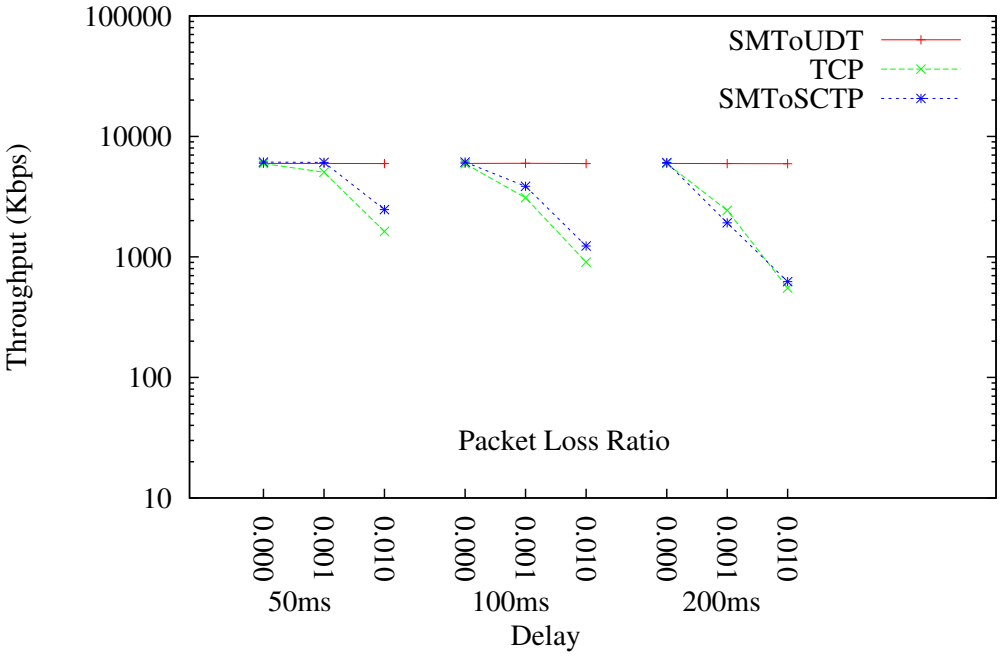
The throughput of SMT over UDT (SMToUDT in Fig. 4.4a) is the same as UDT and the throughput of SMT over SCTP (SMToSCTP in Fig. 4.4a) is the same as SCTP. Therefore, to make the figure more readable, we do not show the throughput of UDT and SCTP. The current implementation of SMT relies on the lower layers to allocate enough bandwidth. Therefore, we should not expect any difference between the throughput of SMT and the underlying transport layer protocol. Meanwhile, SMT manages to result in lower RTTs in some configurations. Furthermore, the round trip time of SCTP is almost the same as SMT over SCTP (SMToSCTP in Fig. 4.4b). Therefore, we only show the round trip time of SMT over SCTP in this figure.

In some configurations, TCP results in worse throughput and RTT than UDT and SMT over UDT. The reason is that in order to provide the required ordering, TCP must use a single stream. A single stream fails to allocate a reasonable amount of bandwidth, especially in high packet loss ratio configurations. Moreover, SCTP and SMT over SCTP perform orders of magnitudes worse than the others. These results suggest that SCTP is not yet a mature protocol.

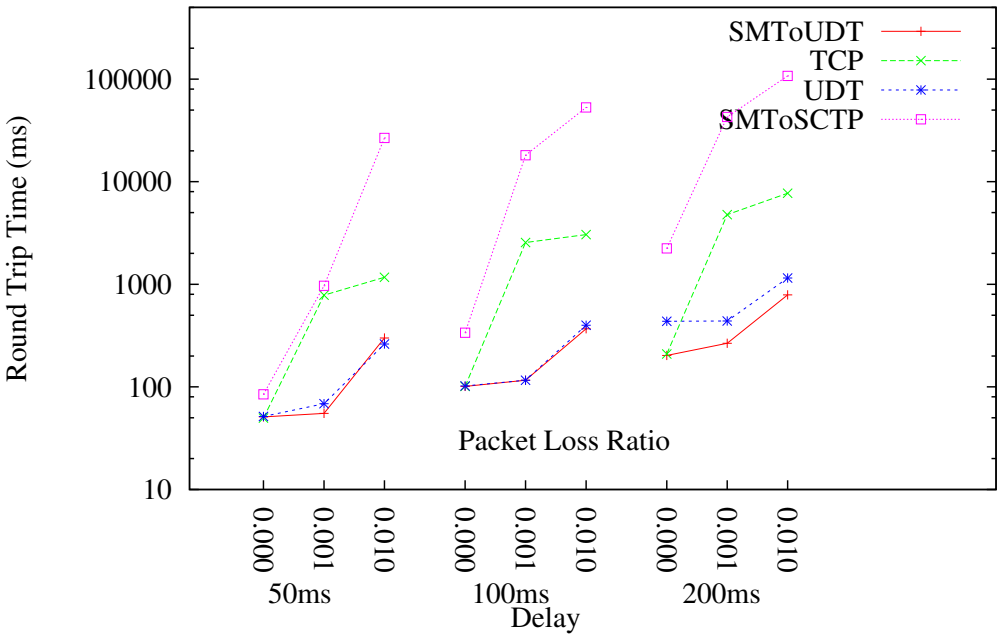
As the packet loss ratio increases, it becomes more important to avoid head-of-line blocking. The main difference between SMT and other mechanisms is visible in the configuration with a rather large RTT (200ms). In this configuration, if a packet of a message is lost, that message cannot be delivered until the lost packet is retransmitted to the receiver side. Meanwhile, the advantage of SMT is that it can deliver the queued messages that do not depend on the incomplete message to the application layer. Therefore, SMT can reduce the latency of these potential messages by one RTT of the path. Avoiding head-of-line blocking is more useful when the path has a large RTT and it is expensive to wait for an incomplete message. SMT is capable of explicit dependency tracking and can maintain a lower average latency for the messages when the packet loss ratio increases.

4.3 Integration in the proprietary product

The ideas from SMT have been integrated into the development branch of a proprietary software system. This software system was chosen because it is a real-world system accessible to us. We do not have permission to disclose details of this system. However, we describe enough details to interpret the results. In this section we describe the integration of SMT



(a) Throughput (higher is better); UDT and SCTP are missing because their average throughput is exactly the same as SMToUDT and SMToSCTP respectively.



(b) RTT (lower is better); SCTP is missing because its average RTT is exactly the same as SMToSCTP.

Figure 4.4: Comparison between SMT, UDT and TCP protocols for sending messages with partial ordering constraints.

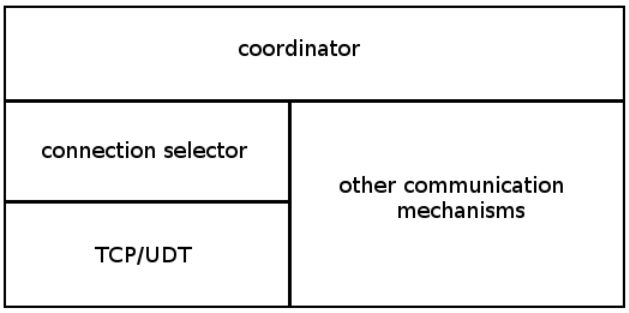


Figure 4.5: Architecture of the proprietary communication component

into the communication subsystem of the proprietary software system and compare the software system before and after integration. We show how these ideas have been used to improve the performance of communication between the two end-hosts in this software system.

4.3.1 Base Architecture

As depicted in Fig. 4.5, the architecture of the communication in the initial version consists of a set of transport layer protocols (TCP and UDT), a connection selector that decides the connection a message should be sent on and a coordinator which is responsible for maintaining the ordering constraints of the messages. The coordinator is responsible for managing a set of different communication mechanisms.

In order to limit the potential latency of all of the messages, the coordinator limits the number and total size of the messages that have not been acknowledged. Whenever the number of unacknowledged messages or the total size of the unacknowledged messages reaches a limit, the coordinator stops sending more messages. The coordinator is shared between multiple different communication mechanisms and this behavior is required in order to support various communication mechanisms.

The connection selector manages a set of connections and distributes messages over them. The transport layer connections used in Fig. 4.5 provide a totally ordered stream of messages within them. However, the coordinator cannot rely on the connections for a valid message order, because between the connections, the messages might get reordered.

The type of ordering constraints that this communication subsystem supports is a set of parallel streams of messages. These messages are multiplexed over the transport layer connections and reordered at the receiver side.

Table 4.1: The percentage distribution of message sizes of distributions 0 to 14.

	Message Size (bytes)								
Distribution	32	128	384	4K	8K	16K	32K	64K	128K
0	8	1	1	5	5	20	40	10	10
1	80	5	5	2	2	2	2	1	1
2	100	0	0	0	0	0	0	0	0
3	0	0	0	100	0	0	0	0	0
4	0	0	0	0	100	0	0	0	0
5	0	0	0	0	0	100	0	0	0
6	0	0	0	0	0	0	100	0	0
7	0	0	0	0	0	0	0	100	0
8	50	0	0	0	0	0	50	0	0
9	0	100	0	0	0	0	0	0	0
10	0	0	100	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	100
12	20	10	10	10	10	10	10	10	10
13	0	0	0	17	17	17	17	16	16
14	40	30	30	0	0	0	0	0	0

4.3.2 Architecture After Integration

Some of the ideas of SMT have been ported to this system. We modified the coordinator and the connection selector. SMT's priority scheduler has been added to the coordinator. SMT's closure creator and connection selector have been integrated into the connection selector. Moreover, UDT has been modified to use the out-of-order mode of communication.

Based on which of the two constraints of the coordinator is the limiting factor, we modify the elements involved in the estimation of the quality of the connections. If the limiting factor is the number of messages, then all of the messages should be sent and delivered as soon as possible. Therefore, the connection with the smallest estimated time-to-delivery is the best connection for any message.

However, if the limiting factor is the size of the messages and not the number of them, then SMT has already exhausted the available bandwidth of its lowest-latency connections. The SMT version in this case falls back to the approaches of the Intentional Network [12] and treats the bulk messages differently. If a message has a low priority and a large size, the SMT version avoids taking the RTT of the connections into account when selecting a connection. Therefore, these messages would be sent over the higher throughput connection and not necessarily the fastest connection.

4.3.3 Benchmarks

We ran some benchmarks to compare the performance of the base version with the version after integrating SMT. Each benchmark consists of sending a set of messages with a distribution of message sizes to the receiver side and measuring the average throughput and the number of IOPS. In this proprietary software system, each input-output operation

corresponds to a message. Therefore, IOPS is the same as messages per second.

The different distributions of message sizes are shown in Table 4.1. Each of these fifteen distributions represents a different type of application and a different use case. There is a huge difference between these different distributions and it is expected to see different results for each distribution.

We use both paths of the test-bed in all of the following benchmarks. Both the base version and the SMT version are capable of using both paths simultaneously. However, the SMT version outperforms the base version when the paths are asymmetric. The error bars in the following diagrams show standard deviation.

Configuration 1: Symmetric Paths

With this configuration, the two paths are exactly the same. The paths are configured as follows:

1. Emulated Network 1: 1 Gbps bandwidth, 50 ms RTT, and 0 packet loss ratio.
2. Emulated Network 2: 1 Gbps bandwidth, 50 ms RTT, and 0 packet loss ratio.

In this configuration, Fig. 4.6 and 4.7 show that the performance of the SMT version is worse than the base version. The error bars in Fig. 4.6 show the standard deviation of the throughput. Throughput decreases in the SMT version for two reasons: First, the SMT version needs more tuning to come up with better estimates of the paths. Second, the underlying transport layer protocol (UDT in this case) has some problems with getting good estimates for the sending rate, bandwidth, and latency of the connections.

Meanwhile, we should note that the difference between the performance of the base version and the SMT version is not huge. For most distributions, the performance is equal. Also, the theoretical upper-bound of the throughput is 256 MBps (1 Gbps + 1 Gbps = 256 MBps). The base version and the SMT version can achieve a throughput close to this theoretical upper-bound in the distributions that have large messages. In other distributions, flow control is the limiting factor and neither of the two mechanisms can get close to the theoretical upper-bound. In summary, the SMT version performs just a little worse than the base version when using symmetric paths.

Configuration 2: Significant Difference in RTTs

The paths are configured as follows:

1. Emulated Network 1: 1 Gbps bandwidth, 50 ms RTT, and 0 packet loss ratio.
2. Emulated Network 2: 1 Gbps bandwidth, 0 ms RTT, and 0 packet loss ratio.

Fig. 4.8 and 4.9 show the throughput and the number of IOPS of the SMT version and the base version. In most cases the SMT version achieves higher throughput and IOPS than the the base version.

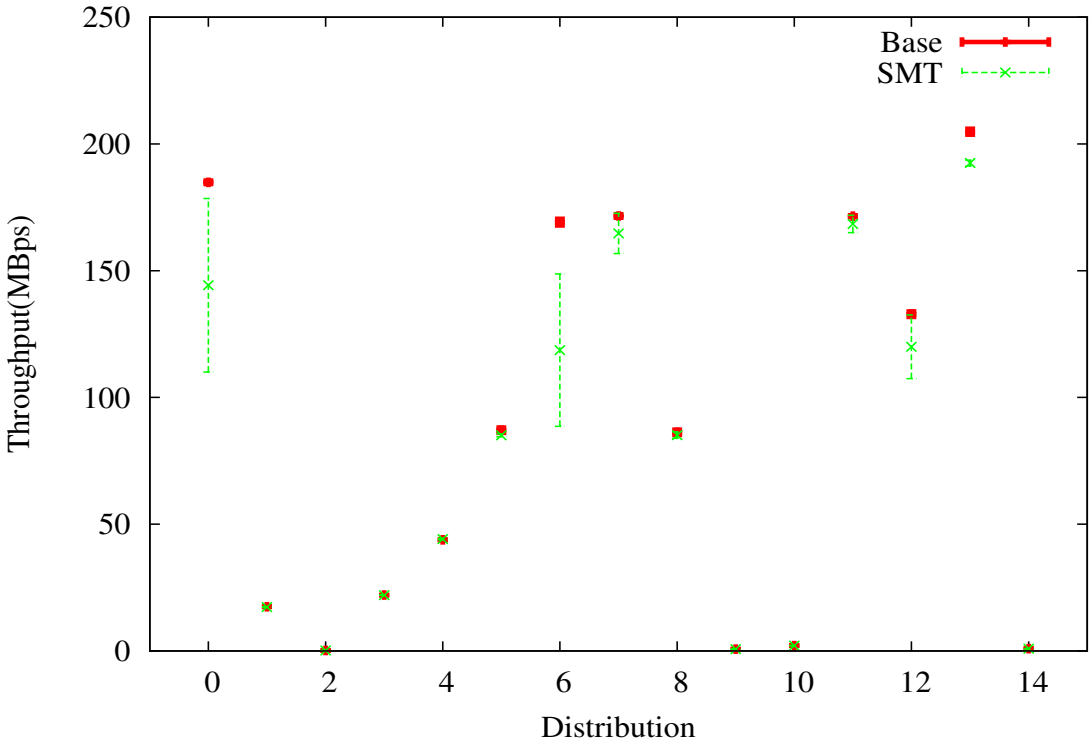


Figure 4.6: Throughput in configuration 1.

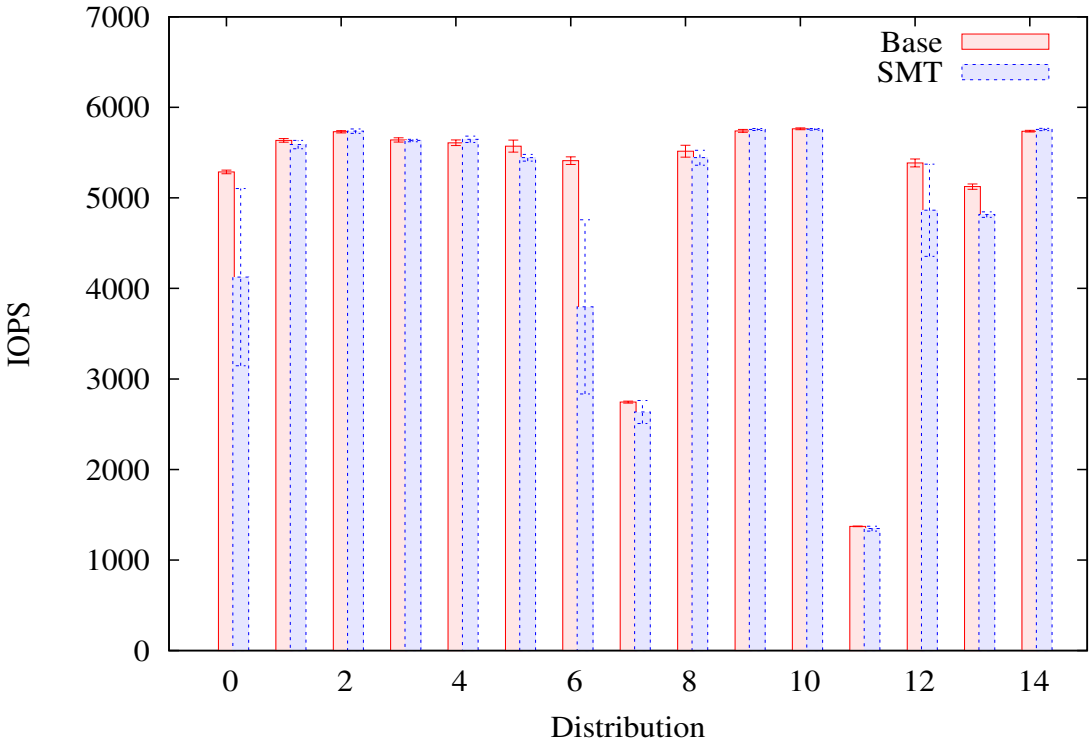


Figure 4.7: IOPS in configuration 1.

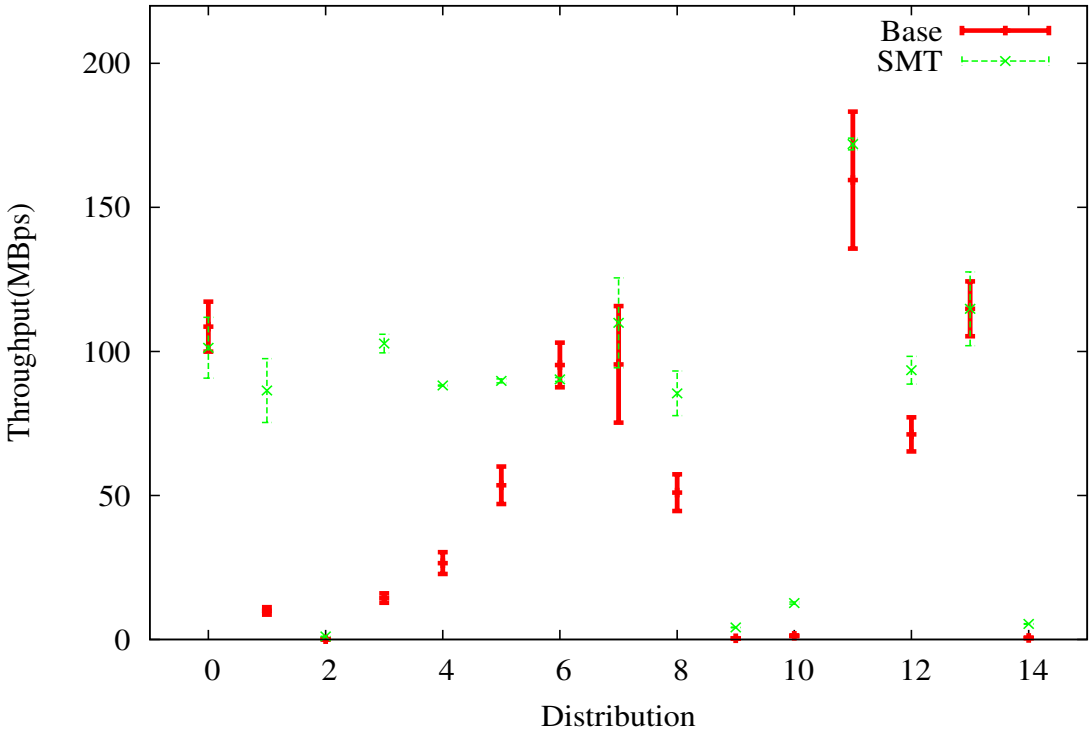


Figure 4.8: Throughput in configuration 2.

In some distributions like distributions 0, 6, 7, 11, and 13 the throughput and IOPS of the base version and the SMT version do not differ by much. In these cases, the distribution contains a set of large messages. The limiting factor for these large messages is the bandwidth of the communication and not the latency of the paths. Therefore, there is not much difference in sending them over either path. This means that the base version's decision approach is good enough for selecting between the two paths. In summary, we see that the SMT version performs better than the base version in most of the distributions when using asymmetric paths with significantly different RTTs when large messages do not dominate the offered traffic.

Configuration 3: Difference in RTTs

The paths are configured as follows:

1. Emulated Network 1: 1 Gbps bandwidth, 50 ms RTT, and 0 packet loss ratio.

2. Emulated Network 2: 1 Gbps bandwidth, 20 ms RTT, and 0 packet loss ratio.

Fig. 4.10 and 4.11 show the results of running the benchmarks in the third configuration. In this configuration, the SMT version sometimes is better than the base version. However, in some other cases, it is worse than the base version. The reason is that these distributions contain large messages and for large messages, bandwidth is more important than the

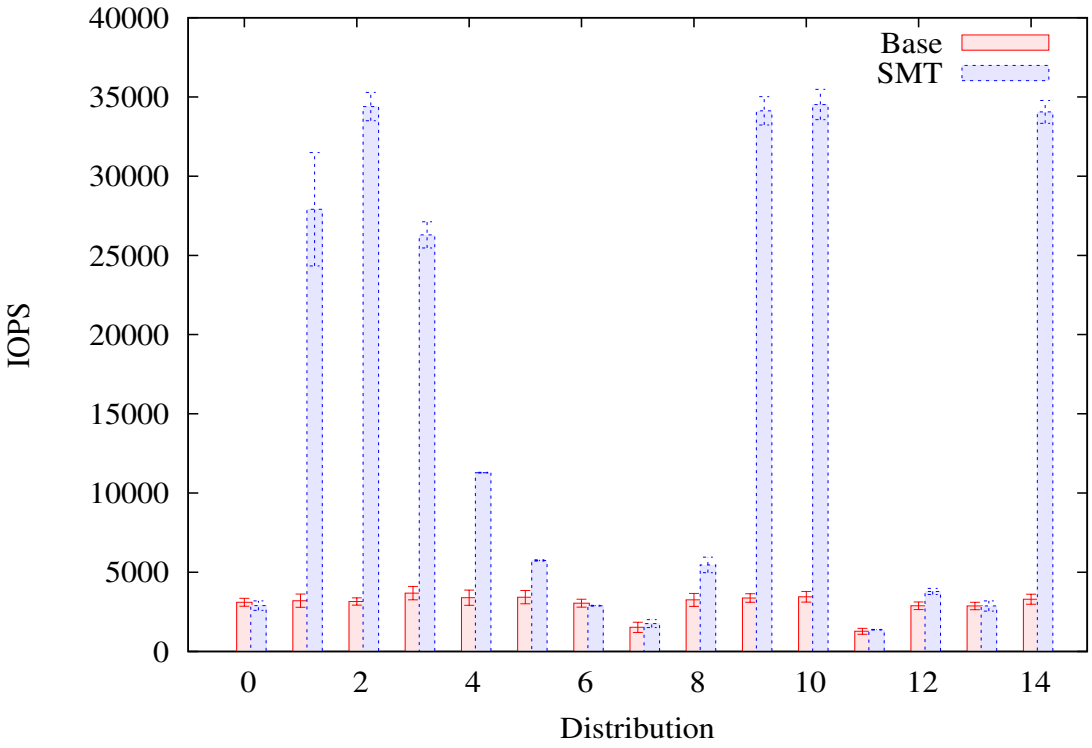


Figure 4.9: IOPS in configuration 2.

latency. The SMT version gives precedence to delivering the messages faster. Therefore, it uses the lower latency path more often than it should. This results in lower throughput for some distributions. The SMT version needs further tuning in order to be able to achieve high throughput and IOPS in all cases. In summary, the SMT version performs almost the same as the base version when using asymmetric paths with different RTTs.

Configuration 4: Different Bandwidths

The paths are configured as follows:

1. Emulated Network 1: 1 Gbps bandwidth, 50 ms RTT, and 0 packet loss ratio.

2. Emulated Network 2: 100 Mbps bandwidth, 50 ms RTT, and 0 packet loss ratio.

Fig. 4.12 and 4.13 show the results of running the benchmarks in the fourth configuration. Here, in most cases the SMT version achieves better throughput and IOPS than the base version. In distributions 2, 9, 10, and 14, however, the SMT version achieves lower throughput and a lower number of IOPS. The reason is that in these cases, the traffic does not include any large messages to increase the throughput. The limiting factor is only the latency of the communication and not the bandwidth of the path. Therefore, the SMT version gets lower throughput because it takes the bandwidth of the paths into account. Again, the SMT version needs further tuning to be able to distinguish when it should use

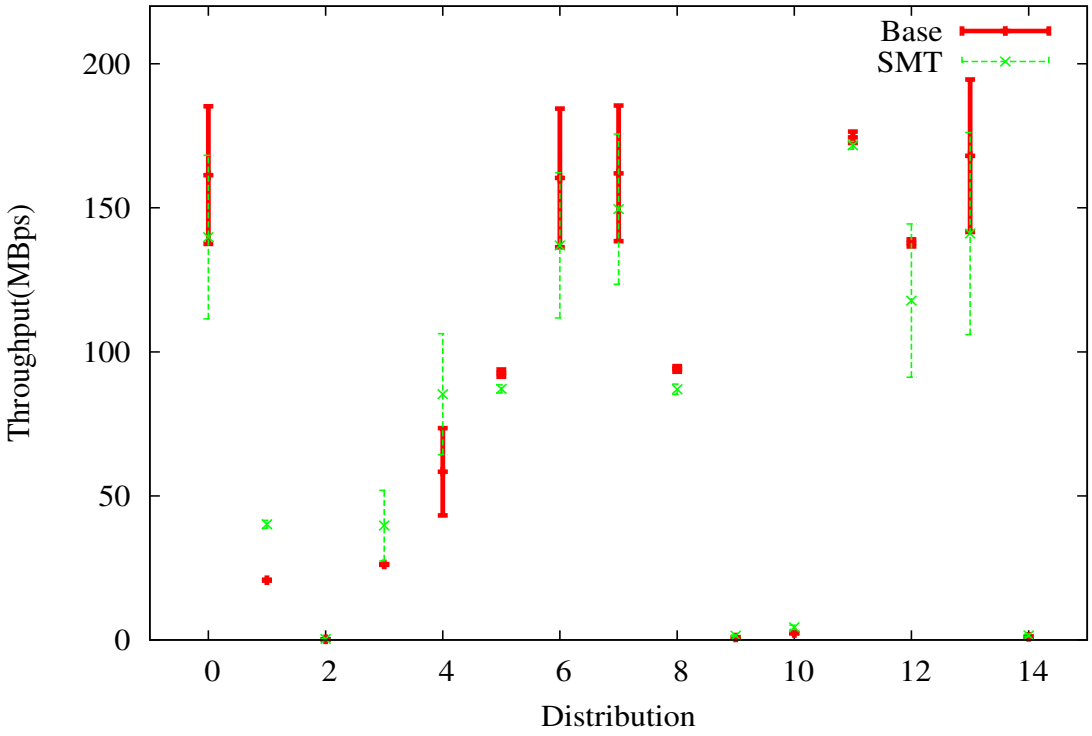


Figure 4.10: Throughput in configuration 3.

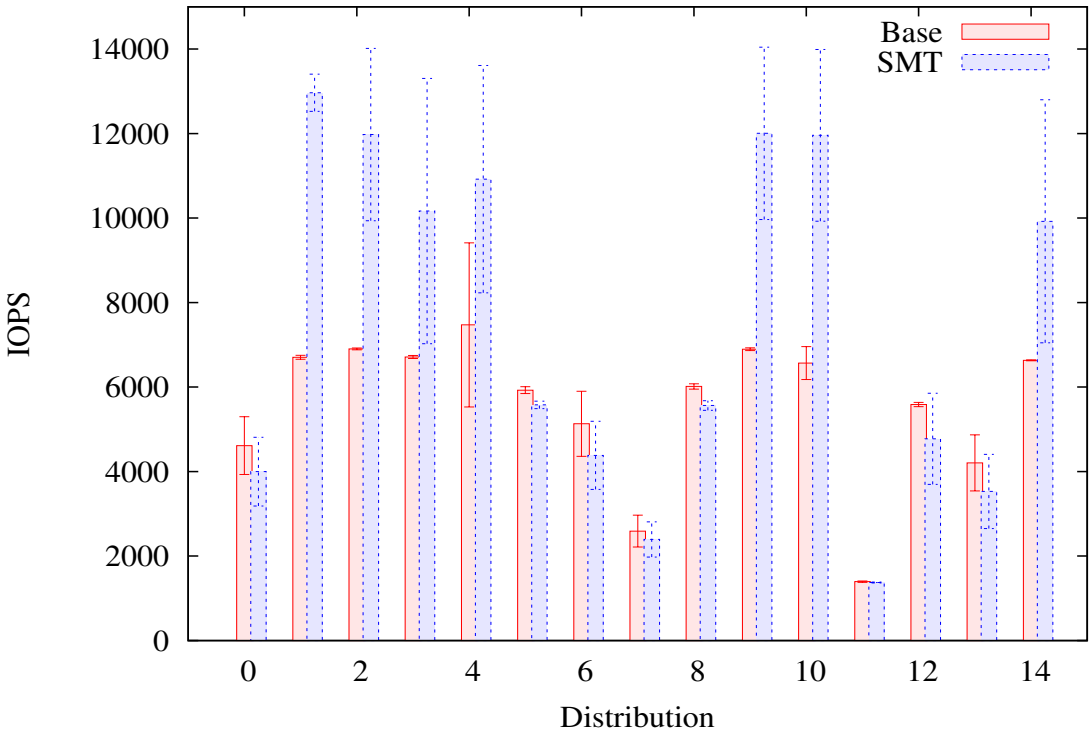


Figure 4.11: IOPS in configuration 3.

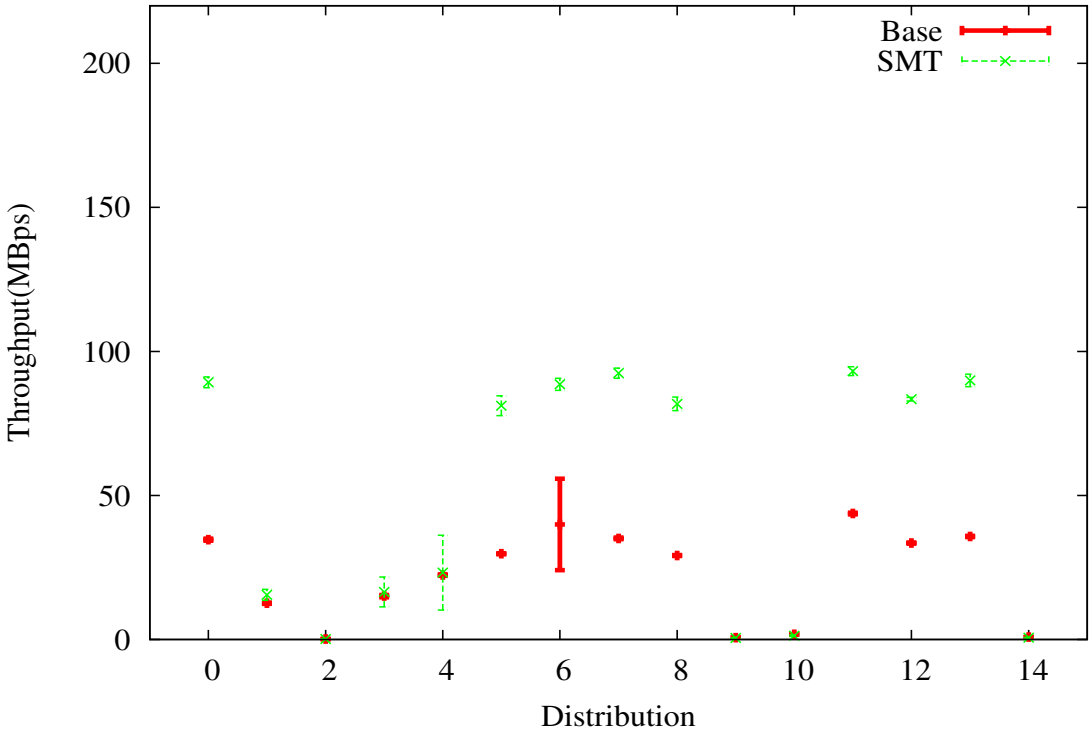


Figure 4.12: Throughput in configuration 4.

both paths instead of only one of them. In summary, the SMT version performs better than the base version in most of the distributions when using asymmetric paths with different bandwidths.

Configuration 5: Different Packet Loss Ratios

The paths are configured as follows:

1. Emulated Network 1: 1 Gbps bandwidth, 50 ms RTT, and 0 packet loss ratio.
2. Emulated Network 2: 1 Gbps bandwidth, 50 ms RTT, and 0.01 packet loss ratio.

Fig. 4.14 and 4.15 show the results of running the benchmarks in the fifth configuration. The SMT version is better than the base version for all distributions of message sizes. The reason is that the base version tries to distribute messages evenly over the two connections. However, the throughput of the two connections are different and the aggregate throughput of the base version is less than the SMT version. Moreover, the lost packets result in head-of-line blocking in the base version. However, if a message is lost in the SMT version and the future messages are received completely, they can be delivered to the application layer if they do not belong to the same stream. In summary, we see that the SMT version performs better than the base version in all of the distributions when using asymmetric paths with different packet loss ratios.

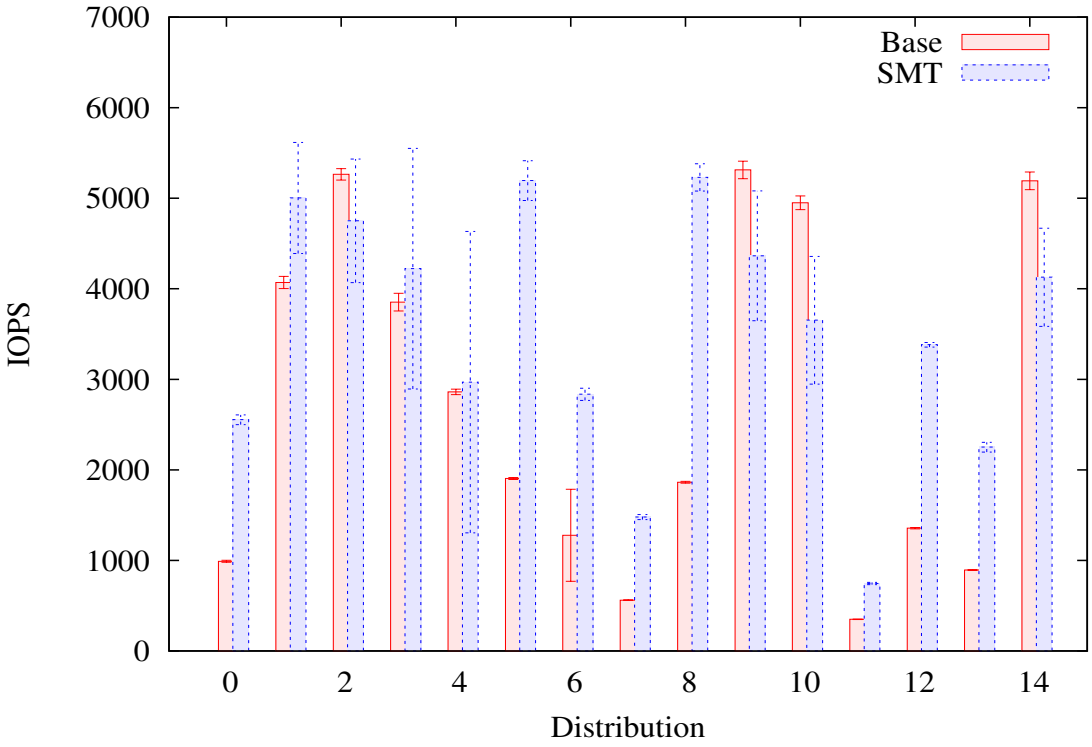


Figure 4.13: IOPS in configuration 4.

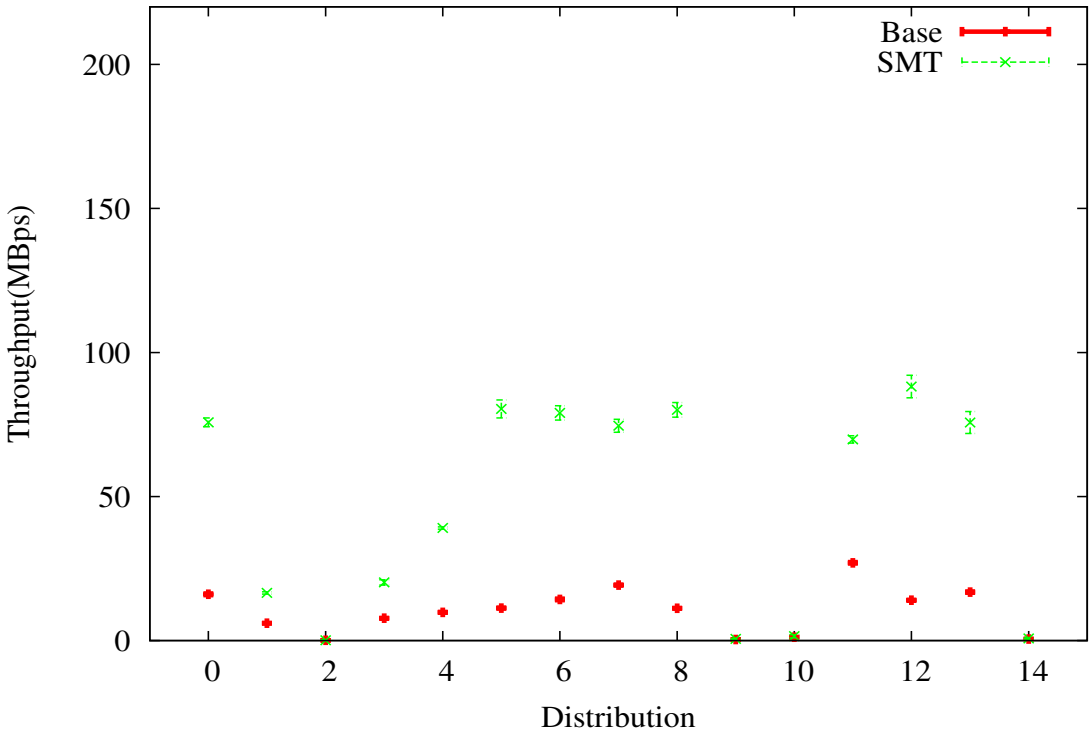


Figure 4.14: Throughput in configuration 5.

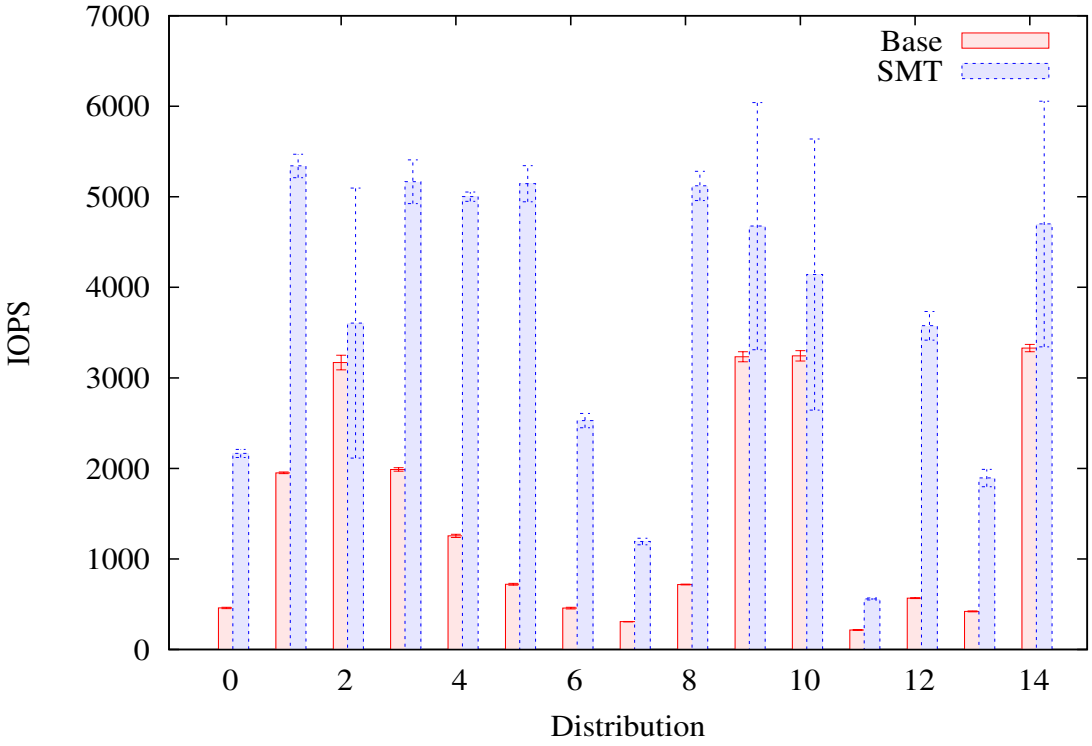


Figure 4.15: IOPS in configuration 5.

4.4 Summary

In this chapter we described the results of the comparison of SMT with some other transport mechanisms with our prototype and in the proprietary software system. In summary, we have shown that in the application with synthetic ordering constraints, when the latency of the path and its packet loss ratio increase, SMT maintains smaller average RTT for the messages. Moreover, in the proprietary software system:

1. The SMT version performs just a little worse than the base version when using symmetric paths.

2. The SMT version performs better than the base version in most of the distributions when using asymmetric paths with significantly different in RTTs.

3. The SMT version performs almost the same as the base version when using asymmetric paths with different RTTs.

4. The SMT version performs better than the base version in most of the distributions when using asymmetric paths with different bandwidths.

5. The SMT version performs better than the base version in all of the distributions when using asymmetric paths with different packet loss ratios.

As discussed in Section 4.3.1, the traffic that the coordinator sends and receives consists of a set of parallel streams of messages. However, even in this traffic many of the ordering constraints are not inherent. SMT supports this special type of ordering graph as well as any other ordering graph. Removing the artificial ordering constraints of the traffic remains as future work.

Chapter 5

Concluding Remarks

In this thesis, we present Structured Message Transport (SMT), a transport layer protocol coordinator that supports partial ordering between messages. SMT provides the applications with an API to specify which messages are the dependencies of a message. SMT does not introduce any other ordering constraints between messages. SMT's explicit dependency tracking avoids the head-of-line blocking and can potentially reduce the average latency of the messages in the totally ordered protocols like TCP.

Moreover, tracking the dependencies between messages has more benefits if more than one path is available. With multiple paths, they might have different bandwidth, latencies, and packet loss ratios. Mechanisms like Multi-Path-TCP try to multiplex one stream of messages over all of the paths and ensure the valid order of the segments sent over different paths at the receiver side. However, as described in the previous chapters, with asymmetric paths, the chances of head-of-line blocking increases. If there is a significant difference between the delivery speed of the paths, even the aggregate throughput of all of the paths might be less than the throughput of the best path alone.

We created an application with some synthetic ordering constraints between its messages. Based on our evaluations, in some configurations SMT can achieve lower RTTs than the underlying transport layer protocol. This experiment shows that SMT results in lower RTTs even in case of a single path.

We also integrated some of the ideas from SMT into the communication subsystem of a proprietary software system. We showed that in some configurations, SMT is much better than the previous communication subsystem.

5.1 Future Work

SMT is not a mature protocol yet. In order to be relevant in today's Internet, SMT should have an integrated congestion manager that makes sure a single SMT association does not take more bandwidth than the fair share of a single TCP connection. UDT has a mechanism to override the congestion controller behavior and we plan to implement MP-TCP's *coupled congestion control* for SMT over UDT.

Another future work is making the case of more than three dependencies efficient. A

directed acyclic graph (DAG) ID can be used to isolate independent messages. With a DAG-ID, SMT can prevent the messages with a large in-degree in the ordering graph from creating head-of-line blocking.

The evaluation of the remaining parts of SMT remains a future work of this thesis. SMT supports multiple preemptive and WDRR priorities and has a traffic redundancy elimination module. Comparing the effectiveness of these mechanisms with the related work remains a future work.

Tuning SMT to get better results in the proprietary software system also remains a future work. While SMT can achieve comparable throughput and IOPS in all cases, with SMT there are lots of opportunities for relaxing the ordering constrains of the messages and taking advantage of SMT's explicit dependency tracking.

Bibliography

[1] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 419–432, 2010.

[2] P. D. Amer and R. Stewart. Why is SCTP needed given TCP and UDP are widely available? ISOC briefing, apr 2004.

[3] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '99, pages 175–187, New York, NY, USA, 1999. ACM.

[4] T. Connolly, P. Amer, and P. Conrad. An Extension to TCP : Partial Order Service. RFC 1693 (Historic), Nov. 1994. Obsoleted by RFC 6247.

[5] L. Eggert. Moving the Undeployed TCP Extensions RFC 1072, RFC 1106, RFC 1110, RFC 1145, RFC 1146, RFC 1379, RFC 1644, and RFC 1693 to Historic Status. RFC 6247 (Informational), May 2011.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[7] B. Ford. Structured Streams: a New Transport Abstraction. *SIGCOMM Comput. Commun. Rev.*, 37:361–372, August 2007.

[8] F. Gont. On the Implementation of TCP Urgent Data. www.gont.com.ar/talks/IETF73/ietf73-tcpm-urgent-data.ppt. [Online; accessed Jun.-2012].

[9] Y. Gu and R. L. Grossman. UDT: UDP-Based Data Transfer for High-Speed Wide Area Networks. *Computer Networks*, 51(7):1777 – 1799, 2007. Protocols for Fast, Long-Distance Networks.

[10] Y. Gu, X. Hong, and R. L. Grossman. Experiences in Design and Implementation of a High Performance Transport Protocol. *SC Conference*, 0:22, 2004.

[11] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, and D. Towsley. Multi-path TCP: A Joint Congestion Control and Routing Scheme to Exploit Path Diversity in the Internet. *IEEE/ACM Trans. Netw.*, 14:1260–1271, December 2006.

[12] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson. Intentional Networking: Opportunistic Exploitation of Mobile Network Diversity. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, MobiCom '10, pages 73–84, New York, NY, USA, 2010. ACM.

[13] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[14] A. Kleen and N. Singhvi. TCP Protocol, Linux Manual Page. `http://linux.die.net/man/7/tcp`, Sept. 2011. [Online; accessed Sep.-2011].

[15] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. *Computer Networks*, 31(11-16):1737–1751, 1999.

[16] Single Exponential Smoothing. http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc431.htm. [Online; accessed Jul.-2012].

[17] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Amin, and B. Ford. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS. In *Proceedings of the 9th USENIX conference on Networked systems design and implementation*, NSDI'12, pages 383–398, Berkeley, CA, USA, 2012. USENIX Association.

[18] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proceedings of the 9th USENIX conference on Networked systems design and implementation*, NSDI'12, pages 399–412, Berkeley, CA, USA, 2012. USENIX Association.

[19] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[20] L. Rizzo. Dummynet. http://info.iet.unipi.it/~luigi/dummynet/, Sept. 2011. [Online; accessed Sep.-2011].

[21] S. Samudrala. SCTP Protocol, Linux Manual Page. http://linux.die.net/man/7/sctp, Sept. 2011. [Online; accessed Sep.-2011].

[22] M. Shreedhar and G. Varghese. Efficient Fair Queuing using Deficit Round-Robin. *Networking, IEEE/ACM Transactions on*, 4(3):375–385, jun 1996.

[23] J. Sommers, H. Kim, and P. Barford. Harpoon: A Flow-Level Traffic Generator for Router and Network Tests. *SIGMETRICS Perform. Eval. Rev.*, 32:392–392, June 2004.

[24] SPDY. http://www.chromium.org/spdy/spdy-whitepaper. [Online; accessed Jun.-2012].

[25] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007. Updated by RFC 6096.

[26] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 99–112, Berkeley, CA, USA, 2011. USENIX Association.