UNIVERSITY OF OKLAHOMA
GRADUATE COLLEGE

Development and Testing of a New Transport Protocol Optimized for Multimedia Internet Transactions

A Dissertation

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

By

Seth Bradley Noble

Norman, Oklahoma

1999

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/1329

UMI Number: 3528666

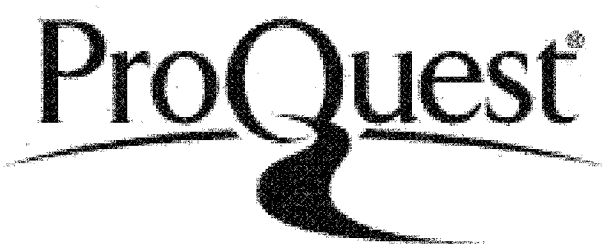
All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



UMI 3528666
Published by ProQuest LLC 2012. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Development and Testing of a New Transport Protocol Optimized for Multimedia Internet Transactions

A Dissertation APPROVED FOR THE
DEPARTMENT OF COMPUTER SCIENCE

BY

K. [illegible]

John M. Canning

[illegible]

[illegible]

Anindya Das

© Copyright by Seth Bradley Noble 1999
All Rights Reserved
Patent Pending

Acknowledgments

To Grandpa, the first Dr. Noble:

Dr. Hyman Shea Noble
July 7, 1914 - November 24, 1999

I would also like to thank my wife for loving me in spite of the many hours of pacing, my family for supporting me through my first two decades in school, my committee and the whole School of Computer Science for giving me the freedom to discover, and finally Dr. Frederick Thompson, Professor Emeritus of the California Institute of Technology, for teaching me that just because something hasn't *been* done doesn't mean it can't *be* done.

Table of Contents

I. Introduction 1

II. Analysis of Traditional IP Transport
2.1 History of TCP 3
2.2 TCP Fundamentals 5
2.3 Violations of TCP Design Assumptions 9
2.4 Summary 16

III. Characteristics of An Alternative Transport 18

IV. Design of the Multimedia Transport Protocol 22

V. Algorithms and Techniques 30

VI. Programming Interface 50

VII. Testing and Simulation
7.1 Methods and Goals 62
7.2 Comparative Throughput 64
7.3 Third Party Effects 68
7.4 Modeling and Simulation 72
7.5 Analysis of Results 77

VIII. Conclusions and Proposals 80

IX. MTP Glossary 82

X. Bibliography 85

Appendix A: mtp.h 88
Appendix B: mtping.c 94
Appendix C: Testing Source Code 98
Appendix D: Empirical Test Results 113
Appendix E: Third Party Test Source Code 127
Appendix F: Simulator 133

Abstract

The TCP/IP protocol, which carries over 95% of data across the Internet, was first published in 1974 at a time when packet-switching was a new technology and computer communications were dominated by the virtual-circuit paradigm. Computer networking has changed dramatically in the past quarter-century, but the underpinnings of TCP have remained virtually unchanged. Many of TCP's most significant design assumptions are no longer valid in the modern Internet. As a result, TCP typically exhibits extremely poor performance including congestion, underutilization of bandwidth, and server overload. Despite these facts, and increasing evidence that TCP/IP is not suited to many of the application protocols it supports, only incremental improvements have been widely researched and no viable alternatives have come to prominence. This dissertation proposes a new transport protocol, the Multimedia Transaction Protocol (MTP), which has been created to meet the needs of modern applications operating in a modern network environment. This new protocol has been designed to handle transaction style client-server interactions across an unreliable, highly congested, packet-switched network. Experimental and simulation results show that MTP provides an order of magnitude improvement in throughput while contributing to network stability and greatly reducing latency. This work characterizes the modern transport environment, describes the design and implementation of MTP, and presents initial test results.

1. Introduction

The TCP/IP protocol, which currently carries over 95% of data across the Internet [21], was first published in 1974 [6] at a time when packet-switching was a new technology and computer communications were dominated by the virtual-circuit paradigm. Several significant design assumptions made in the 1974 protocol and subsequent revisions through the 1981 RFC [23] now in effect, are no longer valid in the modern Internet. Despite this fact, and increasing evidence that TCP/IP is not well suited to many of the application protocols it supports, there has been little research focused on characterizing and improving the Internet transport layer. There are a few alternative transport mechanisms, most of which consist of incremental improvements to TCP and concentrate on the issue of start-up and tear-down costs. None of these alternatives are widely implemented. It therefore necessary to examine the fundamental design of TCP and to create a new transport mechanism optimized for modern applications and networks.

This work begins by examining TCP, the dominant transport protocol of the Internet. First we review the history of TCP in order to characterize its design within the context of its original assumptions about how and where it would be used. We then characterize modern networking applications and the Internet itself and compare this environment to TCP's assumptions. Based on the discrepancies between these two sets of characteristics, we characterize TCP's failure modes and begin to examine alternative transport strategies.

The central part of this work is the introduction of a new mechanism for moving data between processes across packet-switched networks: the Multimedia Transaction Protocol, MTP. It is designed and optimized specifically for modern applications operating across the Internet. Many of the lessons learned from TCP's failure modes are applied to the formation of a narrow set of design goals. From these goals are derived the new transport algorithms which promise to greatly improve the performance and efficiency of data transport for the vast majority of Internet traffic. The implementation of these algorithms as a software library, and the use of this library for application development is discussed in detail.

Finally, the results of direct testing on MTP and TCP under a variety of conditions are examined. Their performance is measured against varying network loss rates and path lengths to show that MTP vastly outperforms TCP for transaction operations while significantly reducing the effects of network congestion. In order to best understand the exact reasons for MTP's performance gains, a simulation using a simplified model of each protocol is presented. This model not only provides correlation between theoretical and actual performance, but allows limited extrapolation into conditions which are beyond the resources of this work.

The Multimedia Transaction Protocol represents a tremendous advance in data transport. In nearly all circumstances it provides an order of magnitude improvement in throughput over TCP while greatly improving the utilization of both network and server resources. The practical effect of using MTP over TCP is to effectively increase, by a large margin, the bandwidth capacity of existing network links and servers without requiring hardware upgrades or significant alteration to existing applications. Moreover, MTP creates opportunities to fully realize cutting edge technologies such as audio/video streaming, real-time data acquisition, and true telephone quality real-time communication. While more work is needed to refine the protocol and develop commercially viable products, the existing implementation discussed here clearly demonstrates the potential of these new techniques to greatly improve the capabilities of existing and future computer networks.

II. Analysis of Traditional IP Transport

2.1 History of TCP

The first wide area computer network (WAN) was created in 1965 by Lawrence Roberts and Thomas Merill connecting computers in Massachusetts and California [18]. The connection was made using the circuit-switched public telephone network which allowed a low quality electrical circuit to be established between the sites. This very first WAN demonstrated both the feasibility of remote communication between computers and the inadequacy of circuit switched technology. This confirmed Leonard Kleinrock's 1961 proposal [17] that packet switching was a more efficient mechanism for digital communication. The next nine years would see the creation of the ARPANET[1] in 1969 and a flurry of research aimed at finding increasingly better ways for computers to exchange data over the emerging network technology. Of particular interest was the development of communication protocols, which proscribe methods for assuring reliable peer to peer communication, and implementations to free applications from having to deal directly with the communication hardware.

Beyond the problems of communicating raw data between two machines, researchers came to recognize that there were numerous issues involved in the transportation of data across networks. Such problems as packet size, data loss, sequencing, buffering, flow control, latency, error checking, and the management of process to process associations were discussed and a wide variety of solutions proposed. Solving these problems required that data transport protocols, sets of standardized conventions for message exchange, be devised. In 1974, Vinton Cerf and Robert Kahn published their landmark Transmission Control Program protocol [6] which not only proposed mechanisms for coping with these problems, but also proscribed a scheme for allowing machines to communicate across different network architectures without requiring their applications to have knowledge of the intervening infrastructure.

Their paper makes several assumptions which reflect the state of computer networking in

[1] The Advanced Research Projects Agency (ARPA) changed its name to Defense Advanced Research Projects Agency (DARPA) in 1971, then back to ARPA in 1993, and back to DARPA in 1996 [19 footnote 4].

the early Seventies. As was noted by the authors, "Much of the thinking about process-to-process communication in the packet switched networks has been influenced by the ubiquitous [circuit-switched] telephone system." [6 page 645]. In particular, the protocol assumed that "processes wish to communicate in full duplex with their correspondents using unbounded but finite length messages" [6 page 640], that message sequences can be viewed "as if they were embedded in an infinitely long stream of bytes" [6 page 641], and that the underlying network will be sufficiently reliable "that the HOST level retransmission mechanism, ..., will not be called upon very often in practice" [6 page 643].

In the following years a specification for the Transmission Control Program was created. In order to promote flexibility and modularity, Internet protocols were divided into four layers, with the original Transmission Control Program split into the transport layer Transmission Control Protocol (TCP) and the network layer Internet Protocol (IP). (See Figure 1.) By 1980 several implementations were available and TCP/IP was adopted as a Defense Department standard. On January 1st 1983, the ARPANET transitioned to TCP/IP from the original ARPANET Network Control Protocol. RFC[2] 793 [23] gives the 1981 specification of TCP/IP which is still in effect. While this specification fills in many of the details left open by the original 1974 publication, it does not modify the underlying assumptions. The specification does leave open some implementation details regarding buffering and timing strategies, however the overall TCP/IP paradigm used today still reflects the assumptions inherent in the 1974 view of computer networking.

While TCP has remained essentially the same, a great deal of research and technology have been devoted to the Application and Link layers. New Application protocols, such has HTTP, have expanded functionality for the end-user. Link level enhancements reflect hardware advances that are ever-increasing the rate of data movement. Even the IP layer has been a focus of attention in recent years due to an exponential growth in the number of Internet nodes and the limited IP address space. The new IPv6 or IP "Next Generation" continues to work its way through development, although the sense of urgency has lessened as the address crisis has proven to be less imminent than originally expected. But while IP has proven adequate even as a new version is developed, TCP has been largely

[2] Request For Comments, a system for creating and publishing standards on the Internet.

left behind. There have been a few research efforts [1, 4, 25, 28] examining incremental mechanisms for reducing connection costs, but there has been no revisitation to the fundamental underpinnings of TCP. In the next section, we will carefully examine the assumptions which TCP makes in its relationship to the surrounding layers.

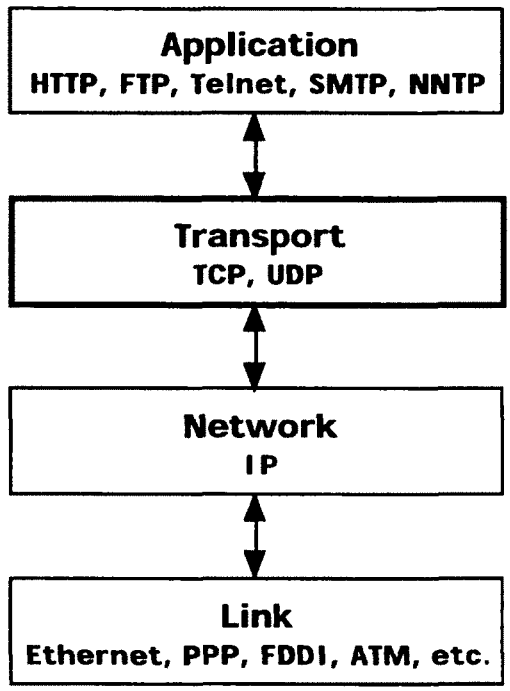


Figure 1: Internet Protocols are classified by four layers. This paper is concerned primarily with the Transport layer which handles the exchange of data between application processes.

2.2 TCP Fundamentals

In order to understand how TCP fits into the modern Internet, we must carefully examine its functional relationship to the other protocols that make up the Internet protocol stack (Figure 1). The top Application layer, which concerns high level data exchange between applications, is the most familiar to typical users. Some of the common application protocols are FTP (file transfer), Telnet (login session), SMTP (electronic mail), HTTP (world wide web), finger (username look up), NNTP (usenet), NFS (distributed file systems), and NTP (clock synchronization). Each of these protocols is built on top of the Transport layer, usually either TCP or UDP (User Datagram Protocol). It is the job of the

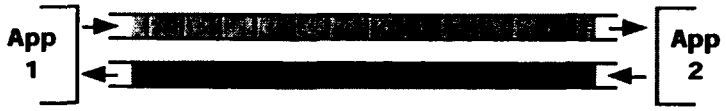


Figure 2: Application view of TCP. Provides a reliable, sequenced, full-duplex data pipe. Assumes that the data is continuous, byte addressable, and unbounded, and that the network is fairly reliable.

Transport layer to provide a mechanism for addressing remote applications and to handle the transport of data on behalf of those applications. Since TCP is the only common transport protocol which provides reliable delivery, it is used for almost all Internet traffic. The transport layer is built on top of the Network layer. The Internet is defined by the use of the Internet Protocol as the mechanism for message routing between nodes. Just as the transport layer provides an abstraction of communication between applications, IP provides an abstraction of communication between network nodes. Below IP is the Physical, or Link layer which provides for the actual transmission of data bits across physical hardware.

TCP was designed to provide applications with a reliable, sequenced, full-duplex delivery of continuous, unbounded data streams over a fairly reliable network (See Figure 2.). To that end, it assumes that two nodes each wish to transmit a continuous, unbounded, byte addressable sequence of data to the other. Thus TCP addresses each stream using a 32-bit Sequence Number which is meant to uniquely identify every byte which may be in transit at any given time. As a TCP connection is established, the two nodes negotiate a starting sequence number for each stream so as to minimize the possibility of old data being mistakenly incorporated into the new stream. Once these sequence numbers have been established the connection is marked as open and the applications can begin sending their data.

At the network level, TCP breaks each stream into packets, each small enough to fit into a single network datagram. Each packet is identified by the sequence number of its first byte. This allows the packets to be correctly identified even if they are subdivided by intervening routers. In the simplest case, a packet of data is transmitted from Node A to Node B. Upon receipt, B returns a datagram containing the sequence number of the last byte it received. When A receives this Acknowledgment (ACK), it knows that all of the

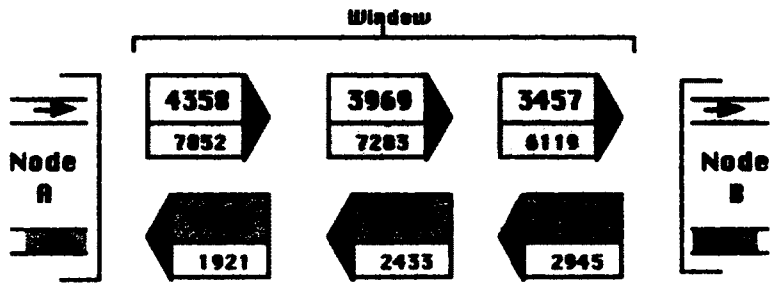


Figure 3: Network View of TCP. The data stream is broken up into packets, each addressed by the index of its first byte of payload (top number in each box). Returning packets carry Acknowledgments: the index numbers of the last byte correctly received (bottom number). Node A has received up through 7852 and is about to receive 8245. Similarly, B has received through 2945 and is about to receive 3457.

data prior to the ACK number was successfully received and that more data can be sent. Since each ACK verifies all prior data, B may choose delay acknowledgement until further data is received and the loss of an ACK is not critical.

In order to fully utilize whatever bandwidth may be available along the path from A to B, it is desirable to continue transmitting data as fast as the network can deliver it, even before the most recent packets have been acknowledged. To accomplish this "pipelining" effect, TCP uses the notion of a "sliding window." A window is the number of bytes which may be "in the pipe" at any one time. If the window size is N, then Node A will send N bytes before stopping and waiting for acknowledgements. As each datagram is received in sequence and without error at Node B, an ACK message is returned. When A receives an ACK number, it knows that all prior data is no longer in the network. It then slides the window forward by transmitting data until there are again N bytes in transit. The value of N starts small, typically around 1,000 bytes, and is increased with each ACK received until the receiver sends a buffer overflow warning or an error occurs. In the ideal case, shown in Figure 3, each node is transmitting data to the other at a steady rate with the ACKs from one stream riding along with the data from the other.

Note that the rate of return of ACKs ultimately regulates the rate of transmission. If a packet is, received out of order, or not received at all, it is detected by the arrival of an unexpected sequence number. Since there is no way to acknowledge out of sequence data, it must be discarded unless the intervening data arrives soon. Given the memory

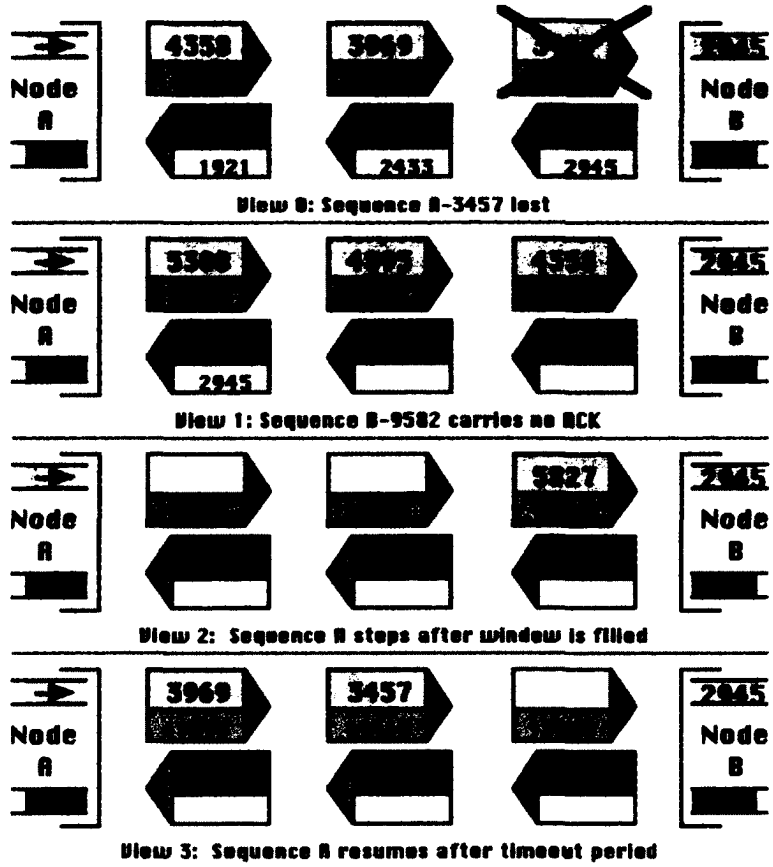


Figure 4: Time sequence of TCP packet loss during steady state exchange.

limitations of twenty years ago, it was assumed that there would not be enough buffer space to reassemble out-of-order data in any case. Typically, implementations buffer out-of-order data in anticipation of the late arrival of the missing piece. But if a packet is lost, then none of the data which follows can be acknowledged. This ACK starvation eventually halts the transmitter and causes it to wait until a timeout period expires. At that point the timer is increased exponentially (called, exponential back-off), the window size is reduced (typically by half or to its minimum value), and a window of data is retransmitted from the point of last acknowledgment. If no ACKs are received for several minutes, then the connection is considered broken and the application notified of the error. This sequence is illustrated in Figure 4.

There are several subtleties worth noting:

- The loss of any packet will halt all data flow in that direction for the duration of the

time out period.

- The loss of a single packet requires that the entire window be retransmitted.
- The available path bandwidth cannot be detected directly. The transmitter must increase its transmission rate until an error occurs, stalling the connection.
- Because the sender can only detect congestion by the failure of an ACK to return, the protocol requires that it exponentially increase successive retransmission intervals to avoid flooding a congested network. As a result, there is no way to quickly detect when the congestion has passed.

Given the original TCP assumptions of a continuous, bidirectional flow of data and a reliable IP network, these issues would not be significant detriments. In the next section, we will discuss how many of these assumptions are no longer accurate in characterizing the modern Internet and the applications which use it. In particular, the assumptions about reliability, data flow, and usage patterns, are inappropriate to the application level HyperText Transmission Protocol (HTTP) which forms the backbone of the World Wide Web and accounts for over 70% of Internet traffic [21].

2.3 Violations of TCP Design Assumptions

The Internet, as we know it today, behaves very differently from the small and well-connected network of 1974. Today's Internet consists of millions of nodes and links with widely varying characteristics. Bandwidth commonly ranges from thousands to billions of bits per second and latencies range from less than a millisecond to several seconds. Packet loss, which is largely a product of router congestion, can be transient or chronic and can vary between one hundred percent and zero in less than a second. (For the purposes of this discussion, 100% packet loss between nodes does not necessarily imply the loss of connectivity. It is possible that a path consisting of working links exists between two nodes but that no packets are currently traversing the entire path due to congestion.) Of equal significance are the patterns of data transfer, which have changed dramatically. The early Internet connected academic and research institutions primarily for the batch transmission of relatively large data files and for real-time remote command execution. Today, the vast majority of traffic is generated by millions of people using the request-response style HyperText Transport Protocol to exchange numerous small requests and

large responses with tens of thousands of servers. Thus the Internet has changed from a sparse, reliable transmission medium to a congested, unreliable retrieval medium. This section will examine how TCP is affected by the characteristics of this modern Internet.

The exponential growth in Internet usage over recent years has consistently out paced efforts at bandwidth expansion. As a result, network congestion at all levels has become a significant problem. This congestion manifests in two areas: servers and network routers. Server overload occurs when client requests exceed the server's capacity, forcing it to delay, drop, or refuse requests. Server overload is relatively rare, occurring for limited periods on popular but underpowered sites. It is easy to detect, however the only common means of coping is the expensive upgrade of the server and its local links. Link level congestion results from more data being sent through a router than it is capable of retransmitting at a given time. When a router's buffer space begins to fill up, the router will start dropping (or "black-holing") an increasing percentage of packets in an effort to force TCP streams to stall and retransmit later. Packet loss or excessive delays in connections across the Internet typically result in TCP retransmission rates of 10 to 20 percent [9]. This clearly violates TCP's assumption of a reliable network in which retransmission would be rarely used.

HyperText Transfer Protocol

This violation is greatly exacerbated by the very different way in which the Internet is now used. The biggest consumer of Internet bandwidth is the HyperText Transport Protocol or (HTTP). One recent observation [21] shows HTTP accounting for approximately 70% of network volume. Since HTTP dominates network usage and in practice is implemented exclusively on top of TCP, it will be used as the primary focus for characterizing how TCP is used by the Application layer. HTTP belongs to a class called Transaction Protocols, meaning that its primary mode of operation is for a client to send a data request to a server and for the server to then send a response. The typical HTTP session consists of the following steps (See Figure 5):

- the client initiates a TCP connection to a server,

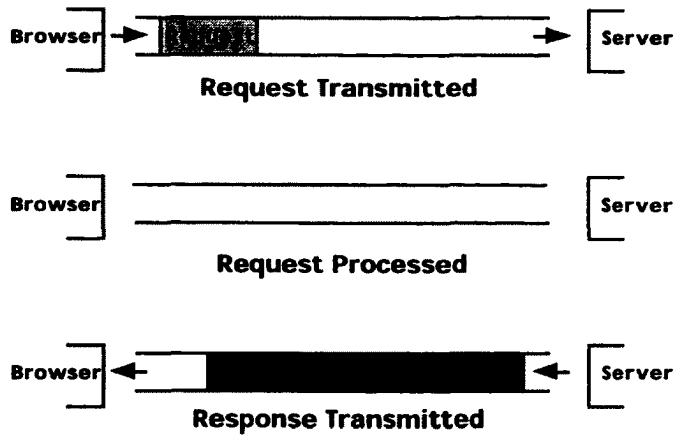


Figure 5: Time sequence of an HTTP transaction. A small request is followed by server processing and the return of a large response.

- the client sends a short request message to the server,
- the server processes the request,
- then the server sends a relatively long response message to the client.

Typically the connection is then closed, although more transactions may take place if both the client and server support "persistent" HTTP and further transactions are pending.

The request-response format classifies transaction protocols such as HTTP as half-duplex, meaning that data is only sent in one direction at a time. For HTTP, the request message is generally on the order of 200 bytes long and contains action and document identification information. The response is typically on the order of 1,000 to 10,000 bytes long and consists of the requested data and usually a small header giving type and processing information [1]. The response may vary from as little as a few dozen bytes for errors or cache checking, to millions of bytes for large multimedia documents. Server response time may very from milliseconds to minutes, depending on the current load. In summary, each HTTP transaction consists of an unbalanced, half-duplex exchange of a finite set of bytes. This is completely contrary to the TCP assumptions of balanced, full-duplex, and unbounded streams

Additional complexities are introduced by both application and user level "optimizations". Seeking to take advantage of IP's best effort bandwidth allocation, all modern browsers

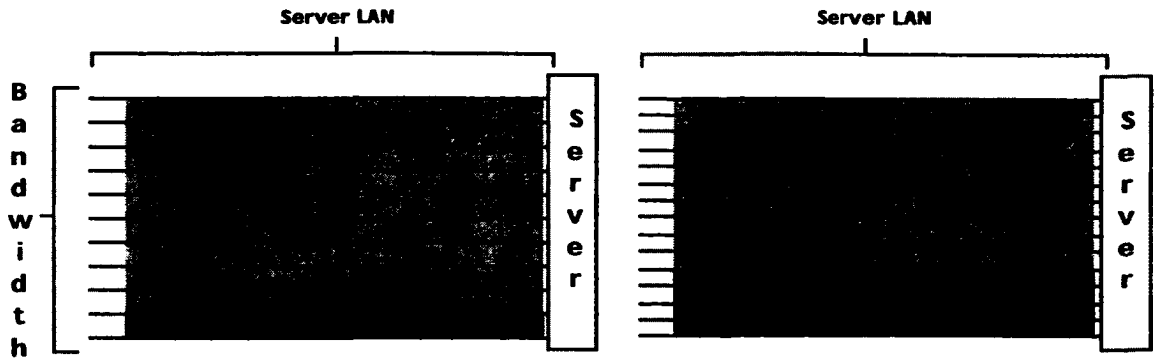


Figure 6: On the left, the allocation of bandwidth with each client utilizing a single TCP connection. On the right, the advantage given to a client which uses four parallel connections. Widespread use of this technique negates the advantage and increases overhead for all connections.

initiate multiple connections (typically four) in parallel rather than conducting transactions in sequence. This practice does produce relative bandwidth gains for an individual user (see Figure 6), but at the cost of greatly reducing overall network efficiency. In addition to significantly increasing both server and network load, parallel connections exacerbate TCP's start-up / tear-down overhead while defeating the efforts of connection caching mechanisms (such as Persistent HTTP and Transaction TCP) to minimize those costs. Parallel TCP connections also cause window synchronization in which many streams make simultaneous, identical flow adjustments. This causes flow oscillations, greatly reducing network efficiency [27 and 31].

At the user level, a frequent solution to a stalled HTTP stream is to abort the current connections and either retry or move on to new requests. Since new connections start in a fresh flow control state, the retry often results in a much faster response than waiting for TCP's retransmission clock to expire. But the server overhead of the old connections will persist until a retransmission discovers the broken connection. Meanwhile, the retransmitted data itself is wasted and contributes to further congestion. Thus an apparent user-level performance gain comes at the cost of increasing server load and wasting bandwidth.

Application / Network Interactions

With both the network below and the applications above TCP behaving very differently

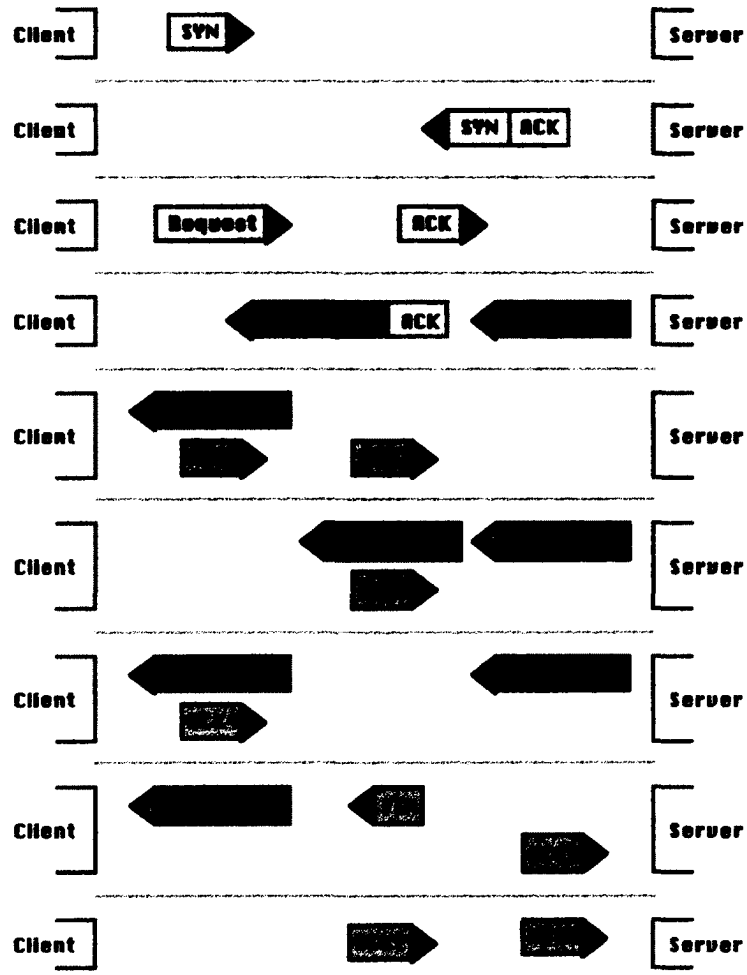


Figure 7: Time sequenced network view of a best case (lossless) HTTP/TCP transaction. TCP's windowing and delayed ACK strategies are not illustrated.

than was originally anticipated, it is not surprising that TCP performs inefficiently even under the best conditions. But when packet loss is introduced, particularly at the high levels often seen during peak Internet usage, TCP's performance becomes outright pathological. The remainder of this section examines in detail how HTTP and TCP interact and how they are affected by packet loss.

Figure 7 illustrates the network level exchange of data in an ideal HTTP/TCP transaction. It begins with the connection set up followed by the transmission of the request, which usually fits into a single datagram. The response then follows as a sequence of datagrams. The client returns acknowledgements to keep the data stream flowing. Usually the client TCP does not acknowledge every datagram and may delay sending some ACKs in the false

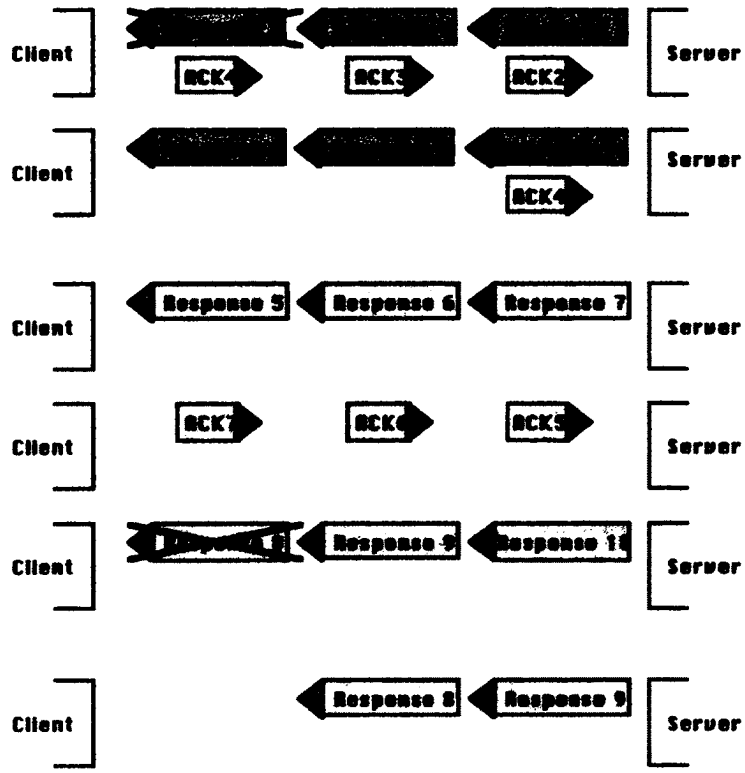


Figure 8: Time sequenced network view of TCP error recovery. The entire data stream stalls after packet 5 is lost. Packets 6 through 10 are retransmitted even though they had been successfully received. Chronic loss of just 10% compounds the problem and force more duplication of data.

expectation that the client application will wish to send more data. These practices significantly reduce throughput. At the end of the response, the server sends a FIN message to close the connection and waits for it to be acknowledged. Note that if this ACKF is lost, the server must wait, usually for at least 30 seconds, before it release the resources associated with the stream. Such resources usually consist of a heavy weight process plus kernel buffers and port space and thus are a significant burden.

While the setup and request segments may consume a significant portion of the transaction time, they represent a very small number of both bytes and packets. Since packet loss is therefore most likely to occur during the response stream, let us consider what happens when a response datagram is lost. Figure 8 begins with the assumption that a transfer of response data is under way with a window size of six. In the first step, one of these datagrams is lost, most likely dropped by an overloaded router. At this point the server is not aware of the problem and continues to transmit datagrams until it has filled the current

window. The client receives these subsequent datagrams but cannot acknowledge them because of the missing packet. After the timeout period expires at the server without any ACKs received, the window is reduced and data retransmitted from the point of loss. Note that in addition to the stall delay caused by the loss of packet 5, packets 6 through 10 are unnecessarily repeated, wasting time, network, and server resources. Some client implementations attempt to shorten the retransmission delay by repeating the last valid ACK as out of sequence data arrives. The server TCP may then interpret duplicate ACKs as a signal to commence early retransmission. But this combined with the duplicate data risks flooding the network and exacerbating congestion. In either case, if we assume a chronic loss rate of just 10% (illustrated here as every tenth packet lost), we see that a loss is repeated at packet 8 before any new data is sent.

Flow Oscillation

After each stall, TCP doubles the timeout period and decreases the window size (usually to half its previous value, sometimes to a preset minimum). When it resumes sending data it increases the window size by one datagram for every ACK it receives, up to a maximum window determined by the receiver. If the window reaches an implementation specific threshold (e.g. 64 kB), the rate of growth is slowed to one datagram per window acknowledged. This approach rapidly increases in the rate of data transmission until congestion forces another loss and stall. The result of this rapid growth and sudden pull-back is a large oscillation in TCP's rate of data flow (see Figure 9).

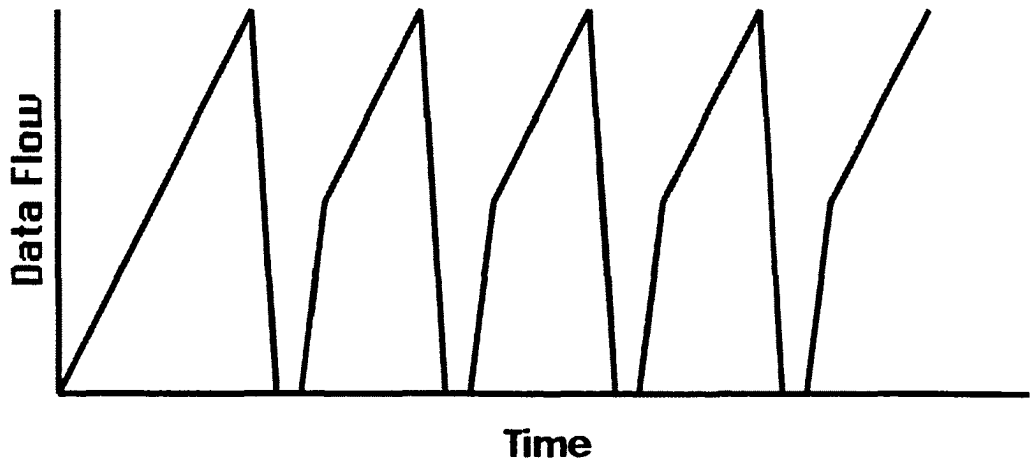


Figure 9: Illustration of TCP flow oscillation

The actual timing of packet losses and recovery will vary depending upon the network path, third party traffic, and the details of the particular TCP implementations involved. But it is clear that this unstable rate of data flow results in TCP averaging a flow rate that is significantly below the available capacity of the network path. Conventional wisdom has held that this apparent excess bandwidth is filled in by other TCP streams and thus helps to assure a fair distribution of bandwidth. But that assumption originated at a time when there were typically only a few third-party streams each with very different network paths. Given the very large number of third-party streams and the practice of opening multiple simultaneous HTTP connections, is highly likely that many of the streams passing through a congested router will share similar attributes. As more recent research [32] has shown, instead of distributing their transmissions evenly over time, parallel TCP streams become synchronized in their flow control actions, stalling and restarting in unison. Even streams with initially different path characteristics may become synchronized as the result of router congestion. Because of this synchronicity, congestion may be forced prematurely as many streams rapidly increase their windows at the same time. This would cause a momentary overflow of router buffers and a burst of packet losses across all the streams. These losses then would force all streams to stall and restart. The net result would be flow oscillation along the entire network path, with routers alternately congested and under-utilized. Figure 9 then becomes an illustration of flow for the entire path, not just a single stream. Thus the entire network would enter the pathological state of being both congested and underutilized.

2.4 Summary

TCP was designed to service full-duplex, continuous, and unbounded data exchanges over a small, well connected network with few routers, little third-party traffic, and almost no packet loss. But HTTP/TCP transactions are brief, bursty, and asynchronous while the modern Internet consists of path lengths spanning dozens of lossy routers servicing thousands of competing data streams. User and application "optimizations" compound congestion, while TCP's own congestion control mechanisms focus the competition for resources at repeating intervals leaving routers under utilized the rest of the time. The client-server model focuses hundreds of transactions per minute on each server, yet TCP places the burden of maintaining most flow-control and all error-recovery efforts at this

congested point. With nearly all HTTP installations using separate heavy-weight processes for each TCP stream, the burden is substantial. Each stall extends the amount of time that these server resources are consumed and each duplicate retransmission further erodes the server's bandwidth capacity. Each duplicate connection or user abort-and-retry adds still more to the server's load. These feedbacks cause the efficiency of both the server and the network to decrease substantially as their loads increase. The only limiting factor in this feedback loop of performance degradation is the patience of the end-user who may ultimately reduce the load by giving up (see Figure 10).

TCP was not designed to service the demands of modern applications nor was it designed to function in the environment presented by the modern Internet. The client-server model, high speed links, long path lengths, significant loss rates, and tremendous quantities of third-party traffic were simply not anticipated twenty-five years ago. Thus the transport protocol which accounts for over 95% of Internet traffic [21] often exhibits the pathological combination of chronic congestion and underutilization. Clearly a new approach is needed to provide reliable network data transport optimized for modern applications and environments.

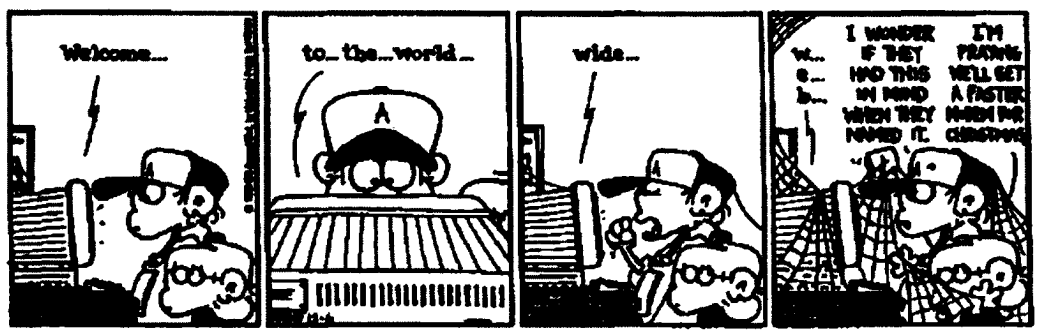


Figure 10: The ultimate limiting factor for Internet congestion
FOXTROT © Bill Amend. Reprinted with permission of UNIVERSAL PRESS SYNDICATE. All rights reserved.

III. Characteristics of An Alternative Transport

Given the properties of modern applications and networks, combined with an understanding of the failure modes of TCP, it is possible to derive a criteria for designing a new transport protocol which would be optimized for today's usage requirements. The first step is to narrow our focus to a specific class of protocols which account for over 80% of Internet traffic [21]. Transaction protocols, such as HTTP, FTP, and NNTP, all share the common property of many clients seeking to retrieve moderate to large amounts of data from relatively few servers. The properties of transaction protocols were discussed previously (recall Figure 5). (Each protocol also allows the uploading, or transmission, of data from the client to the server, but this accounts for a tiny fraction of use and so will be considered separately.) Since HTTP alone accounts for 70% of Internet volume, we will use it as the focal point for characterization and optimization.

Transaction protocols such as HTTP make several requirements of the transport layer, which traditionally only TCP has been able to provide:

- Reliability: Both the request and response must be delivered correctly and completely.
- Sequencing: The transport must assure delivery of data as a contiguous, sequenced object.
- Uniqueness: The client application must experience a one-to-one correspondence between requests and responses.
- Scalability: Response sizes may vary from a few bytes to millions.
- Multiplexing: Data from multiple transactions must be identified and delivered separately.

Other properties are desirable, but these form the critical core functionality that any new protocol must support in order to provide service for HTTP and other common transactions. But just as important as recognizing this lower bound of functionality, is the establishment of an upper bound.

Transaction protocols in general, and HTTP in particular, exhibit behaviors which limit the

the requirements placed on the transport layer. Similarly, the characteristics of the network itself suggest avenues for optimization. The most important observations are as follows:

- Requests are small, typically only about 200 bytes, rarely more than 1000 [14].
- Responses are finite. Requests will result in a fixed sized document. In most cases, the size is known as soon as the response begins. (Multimedia streaming, which almost always uses connectionless protocols, will be discussed separately.)
- Delivered documents will be buffered in their entirety by the client.
- Many clients concentrate on relatively few servers.
- Clients have an abundance of resources, while servers have a scarcity.
- Most requests to a given server concentrate on only a small pool of documents.
- Network packet loss is evenly distributed amongst all datagrams.
- Network latency and data loss vary proportionally to traffic and congestion.

The first few observations can be distilled into the notion of "client-pull". In traditional network models, it is assumed that one node wishes to *send* data to another, and therefore the sender assumes the burden of assuring that the data is delivered correctly. Under the client-pull model this role is reversed: it is assumed that one node wishes to *acquire* data from another. This is a very subtle and yet extremely powerful shift in emphasis. Under this model, it is possible to shift the burden of assuring data delivery from the server to the client. By more closely associating the state management of each transaction with its point of origin, a protocol can obtain much more accurate and timely information about conditions which may affect its performance. It can then use this improved feedback to make informed adjustments to its data flow. This also shifts a significant load away from the over-burdened servers and toward the lightly loaded clients.

Based on this client-pull model and the remaining observations about the nature of transactions and the underlying network, we can define the following terms and characteristics for an alternative transaction transport:

Data Object

A data object is any independently addressable, ordered set of bytes on a server. A data object is assumed to be finite, contiguous, and byte addressable. Because the client intends to use each document in its entirety, it is assumed that sufficient buffer space may be provided by the application such that portions of a document can be assembled by the transport independent of their arrival ordering. Data objects may be dynamically generated, but it is assumed that most exist in their entirety prior to their request.

Request

Since requests are small, we assume that they may fit into a single network datagram. A request message is defined as a single datagram which uniquely identifies a data object, or a portion of a data object, which is to be sent from the server to the client. Provisions can be made for very large requests, but such a case is considered extremely rare.

Response

Upon receipt of a request message, a server is obligated to send to the client exactly one response. The response may consist of a single datagram consisting of an error message or a portion of the requested data object. If the data-object is larger than a single datagram, multiple datagrams constituting the object or a portion of it may be returned. The client is solely responsible for keeping track of which portions of the data-object have been requested and received.

Error Recovery

If a client fails to receive a correct response within a reasonable time frame, it may send a repeat request for the missing data. In the case of a large data object requiring multiple requests, the repeat requests will be made as part of an ongoing flow of data. If no response is received for any part of the data object, then an error is returned to the application.

Flow Control

The client exercises primary control of the data flow by regulating the size and frequency of

its requests. In general, the client will issue a series of requests for portions of the desired data object using a specialized sliding window strategy. The server may enforce limits on the rate at which it is willing to send data. Adjustments due to network congestion are the exclusive responsibility of the client. Variations in network latency and loss rates may be used to assess network congestion.

Proportionality

Adjustments to flow control should be made preemptively whenever possible and in proportion to the observed network conditions. Care should be taken to maintain consistent flow and to avoid oscillations or synchronization. Distinct strategies must exist for coping with transient versus chronic congestion.

These basic characteristics address the needs of modern transaction applications while optimizing to ensure the best possible performance across the modern Internet. These properties form the basis for the design of the Multimedia Transaction Protocol. MTP has been designed from scratch to satisfy the design specification just described. By focusing on the specific needs of the most common applications and by taking into account the behavior and interactions of the network, servers, clients, and the end-user, this new protocol has been optimized to provide full and efficient utilization of existing network infrastructure in support of existing applications. In short, MTP is designed and able to provide a substantial increase in the effective capacity of existing network and server resources through a modular, software solution. The next section builds on the properties discussed to describe the philosophy and design of the Multimedia Transaction Protocol.

IV. Design of the Multimedia Transport Protocol

The characteristics of the modern Internet and the applications which use it suggest many specific requirements which a new transport protocol should meet. In addition to these new requirements, there are a great many basic goals which any acceptable network protocol must encompass. By combining the characteristics of the modern Internet, modern applications, and the successes and failures of TCP's mechanisms, it has been possible to derive a set of design goals for an optimized transaction transport protocol. The most basic goals, such as reliability, a simple yet flexible interface, and code portability, are strengths of TCP. Likewise, the weaknesses of TCP suggest goals such as quickly recovering under a variety of error conditions while assessing and maintaining the highest possible throughput. From the characteristics of modern applications and usage patterns stem the needs for transaction style communication, minimizing server load, stabilizing flow rates, and scalability across a very wide range of network and machine types. These goals present numerous constraints which form a boundary around the space of possible solutions. But some of the goals, particularly those requiring optimization to specific applications and usage patterns, also simplify the design and suggest new solutions. To fully understand the design and functionality of the Multimedia Transport Protocol, it is necessary to understand each of the goals which has shaped the protocol's philosophy and design.

Transactions

Nearly all Internet traffic is generated by data being requested by a client from a server. A transaction generally begins with a client sending a *request* to a server. The request usually specifies data which the client seeks to obtain from the server. Upon receipt of the request, the server processes this specification and returns a *response.* The characteristics of the response may vary, but it typically consists of a sequence of tens of kilobytes. The request is usually no more than a few hundred bytes. From the network perspective, transactions are half-duplex, meaning that data is only sent in one direction at a time, and asymmetric, meaning that the vast majority of data flows in only one direction: from the server to the client. (Recall Figure 5.) In addition, transactions usually (but not always) occur with

many clients seeking data from a few servers.

As the name implies, the Multimedia Transaction Protocol is designed explicitly for handling transactions. Its form and function assume the structure just described. Conversely, MTP is not designed to generally handle full-duplex, symmetric data transport. While it is possible for an application to use MTP efficiently for some tasks in that class (as in the case of IP telephony), such a use requires that the application designer have a much greater awareness of transport level issues [31]. MTP is certainly not intended to create a bidirectional byte pipe, as is the case for TCP.

By optimizing MTP for this specific mode of transport (which accounts for nearly all Internet traffic), it is much easier to achieve the remaining design goals. Throughout the remaining discussion, the philosophy of MTP's functionality rests firmly on the presumption, at least at the transport level, of a small-request for a large-response format.

Reliability

Inherent in the notion of a transaction is the idea that each request will receive some form of response. Assuring that such a response is generated, and taking measures to ensure that the response delivered is the one desired, are core aspects of transaction reliability. Note that this definition of reliability differs significantly from that of byte-pipe transports such as TCP. For those transport protocols, reliability means that data transmitted by one host is received by its peer Since that transport function ends at a different network node than it begins, there exists the added difficulty of conveying its success or failure back to the point of origin. Transaction protocols have no such difficulty because their transport function, the request-response pair, both begins and ends at the same client node. Thus the client is fully capable of evaluating the state of an ongoing transaction.

Since each transaction begins and ends at the client, MTP places the entire burden of assuring reliability on the client side. For each request issued by the client application, the client MTP can guarantee that it will deliver a unique and matching response. The client MTP maintains all state information regarding the transaction and handles timeouts and retransmissions as specified by the application. The MTP server is not required to maintain

any state information about individual transactions. The only responsibility of an MTP server is to generate at most one response for each request message it receives. Once the response has been transmitted, the server need not maintain any further information about the transaction. Not only is the server not responsible for transport reliability, but neither may it make any assumptions about the state of requests or responses. From the perspective of the server application, responses may be lost and requests may be duplicated. Only the client is assured of a one-to-one correlation between requests and responses.

Once again, optimizing for the transaction case greatly simplifies the design of the MTP protocol. Error recovery is handled simply by client retransmission of a request message until a response is received or an application specified timeout is reached. The client always knows the state of a transaction and is able to make accurate assessments of network conditions. The server's transport task is greatly simplified by having no responsibilities and making no assumptions with regard to reliability. By concentrating all of the burden for state maintenance at the client side, the algorithms required are made both simpler and more powerful while moving the overall design closer to its next goal.

Minimal Server Load

The client-server model of networking concentrates the attention of a great number of nodes at a few key points. Busy servers may be required to respond to thousands of clients per minute. As a result, servers are most often stressed to the limits of their resources. Expansion of a server's capacity most often requires costly hardware upgrades or additions. Clients, on the other hand, are often largely idle during network transactions, possessing a relative abundance of processing and memory resources. It is therefore desirable to distribute as much of the transport burden as possible away from the server and toward its clients.

Shifting the burden of reliability and state-maintenance to the client goes a long way toward achieving this goal. The server is not required to maintain any data, buffers, or state with respect to an individual transaction. Contrast this to TCP in which the server must dedicate significant amounts of memory to buffer space and state maintenance, significant CPU

cycles to state processing and data movement, and one of a limited space of network ports to each request. This burden is so significant that a separate heavy-weight process is typically needed to efficiently cope with each TCP transaction. MTP greatly reduces the server burden by allowing each request message to be handled as a distinct and stateless entity. Requests for large blocks of information are broken down into sub-requests by the client, allowing the server to rapidly respond to every request message without the need for extensive algorithms or buffering. Because each request message can be handled asynchronously, it is generally possible for a single process to respond to a very large number of simultaneous requests. MTP does provide transaction and session identification for those applications which require server side processing. However, MTP itself places no state burden on the server.

Removing the requirement for server-side transport state information reduces not only the transport load, but also allows for more efficient use of application level resources. By making it efficient to handle large numbers of transactions within a single heavy-weight process, the server is capable of implementing much more efficient caching and load control mechanisms. MTP itself is able to maintain an awareness of server load and throttle back aggressive clients to ensure steady and fair service. This combination of shifted burden and greater efficiency means that a given server platform is able to handle significantly more traffic with MTP than it could with TCP. When combined with a more efficient use of network resources, MTP allows existing server hardware to enjoy a substantial increase in capacity through a software only solution.

Scalability

The modern Internet exhibits a tremendous amount of variation in its key characteristics. Link speed varies from hundreds to billions of bytes per second, path latency ranges from less than one to thousands of milliseconds, and nodes span a wide spectrum of memory and processing resources. The behavior of the network is also governed by many factors which are not readily observed, such as the path length, router buffer capacities, and the level of third party traffic. Perhaps of greatest importance is the fact that most, if not all, of these characteristics may change many times during the life of a single transaction. For a

protocol to be efficient, it must be capable of quickly adapting to a changing environment.

Traditional TCP is rather inflexible and has difficulty in scaling to today's high-speed, low reliability networks. Without modification, linear increases in link or cpu speed often result in only logarithmic increases in TCP performance. When TCP is modified to perform well under one set of conditions, it may become dramatically less efficient when coping with others. MTP's simplified transport model, with all of the state information maintained at the client side, allows it to garner a much clearer picture of network conditions and thus to adapt quickly to changing environments. For each transaction, the MTP client tracks round-trip times, error occurrences, local buffer levels, and indicators of both client and server load levels. Using this information, it is able to approach and maintain near optimal data flow under a very wide range of network conditions.

Given the tremendous growth and change which is ongoing in Internet usage, it is vital that any transport protocol be capable of adapting to not only transient, but evolutionary changes it its operating environment. MTP is thus designed with as few assumptions as possible about network characteristics. Instead, an emphasis has been placed on discovery and adaptation. This allows the protocol to be utilized not only in a wide variety of current network scenarios, but also promises that it will remain relevant in the face of changing technology.

Efficient Flow Control

Adapting to network conditions in order to achieve efficient and reliable transactions requires a careful regulation of the rate of data flow. The almost exclusive cause of network level packet loss is congestion [32, 27, 12], which is the result of data arriving at a node faster than the node is capable of processing. When a network router's incoming buffer overflows, subsequent datagrams will be lost. Some routers preemptively drop datagrams before their buffers overflow, with the expectation that affected transport protocols will then reduce their flow rates. In any case, overly aggressive flow strategies result in high loss-rates which forces the protocol into excessive and costly error-recover operations. For a reliable protocol, there is no advantage to trying to send data faster than the network is capable of delivering it. It is therefore desirable to approach, but not to

exceed, the maximum utilization of the network path and to adjust as that capacity changes.

TCP uses a very aggressive and unstable control mechanism. It starts slowly and rapidly increases the rate of data flow until a loss is detected, at which point it halts all data flow. After a significant wait, it resumes data flow at a dramatically reduced rate and once again begins increasing it. This oscillation of flow results in high loss rates and chronic under-utilization. (Recall Figure 9.) MTP implements a much more informed and conservative flow control strategy. It begins with an initial flow calculated based upon MTP's previous experience. From the initial response, it estimates current conditions to make sure that there has not been a substantial change in capacity. On subsequent iterations throughout the transaction, MTP usually makes linear adjustments to the flow rate. Increases are allowed only when no errors have been recently detected. If a loss is detected, a linear flow reduction is made and a brief moratorium is placed on most further flow control actions. In addition to observing simple packet loss, increases in the round-trip time are used as an indication of decreasing network capacity. This allows MTP to begin throttling back its flow rates *before* data is lost. Local conditions such as incoming queue lengths and the number of outstanding transactions are also used to regulate data flow. The server may participate in flow control by limiting the rate at which it responds to each request message. These limitations are both enforced at the server and communicated to the client to allow for a coordinated regulation of server load.

The details of MTP's flow control algorithms will be discussed in the next chapter. For now it is sufficient to note that MTP incorporates numerous observations to create a measured response to both transient and chronic conditions which may affect the flow of data. This strategy of informed conservatism allows MTP to discover and maintain near optimal flow rates in the widest possible variety of network conditions. That in turn allows networks which make extensive use of MTP to achieve much higher and much more stable utilization.

Efficient Loss Recovery

Part of maintaining a steady flow of data is coping with the inevitable loss of datagrams. While MTP takes steps to prevent packet loss, it does not assume that such loss will be a

rare event. Losses are detected by the failure of a request to receive a complete response within a calculated time period. The length of the timeout is determined in part by the observed round-trip time and thus is constantly updated to reflect current conditions. It is vital that the timeout be sufficiently long to prevent data duplication, while minimizing the wait before recovery can begin. MTP loss recovery consists simply of sending a request for the missing data. Unlike TCP, only the missing data is resent. Most importantly, while the client is waiting for lost data, other portions of the transaction can continue to flow at a slightly reduced rate. Only under the most severe network conditions is data flow ever completely stalled by the loss of a datagram. Not only does this maintain a higher overall throughput for the individual transaction, but it contributes to stability along the entire network path.

Simple and Flexible Programing Interface

One of the reasons for the tremendous popularity of TCP is its extremely simple interface. A programmer needs only to establish a communications port and connect it to a peer to establish a data link. After that, file-like reads and writes may be used to exchange data. MTP strives for a similar level of simplicity while adding a great deal more flexibility. Once the MTP library has been initialized, a transaction is begun with a single call to the request procedure. Included in the parameters is all of the information needed to form the request as well as information about how the response should be delivered. In the simplest case, MTP handles all of the transport and delivery issues. Upon completion of the transaction, a callback procedure is invoked allowing the application to handle the newly delivered data. Much greater levels of application control and notification are optional. On the server side, the application registers callback procedures to receive requests. In the simplest case, those procedures begin with a description of the incoming request and end by passing response data back to the server side MTP. The server side application is not required to maintain any state information about transactions or to take any action between requests. There are, of course, facilities to allow session identification and flow regulation for those applications which desire it.

A minimal MTP client or server application can function with just six procedure calls.

Most of even the more complex MTP options can be controlled by parameters embedded in these calls. The details of MTP's programing interface will be discussed in the Programing Interface chapter below. The important point is that MTP has been designed to make application development, particularly on the client side, as simple and unobtrusive as possible.

Portability

The final goal of MTP is to be capable of easy adaptation to a wide variety of hardware and operating system architectures. In order to be quickly ported amongst a large number of platforms, MTP has been implemented on top of the UDP/IP protocol. In theory, MTP is not constrained to this protocol and could be implemented on top of any packet switched network, including raw IP datagrams. But since UDP/IP is both ubiquitous and simple, there is no immediate benefit to implementing MTP on top of other networks. The MTP software itself resides in a user level code library. In order to speed development and acceptance, there is no kernel level code. While the implementation might gain some performance benefits from kernel level integration, such marginal gains would be far outweighed by the difficulty of maintaining numerous kernel drivers and interfaces and by the perceived risk that would be posed to operating system stability.

Summary

The complex and changing characteristics of modern computer communications present many challenges to the transport layer of the Internet Protocol stack. MTP has been designed from scratch for the purpose of meeting these challenges. By focusing on the needs of the specific class of transport required by the vast majority of network applications, MTP is able to utilize a simple and robust design philosophy to provide vastly superior transport for transactions. With these ideas in mind, the next chapter, Algorithms and Techniques, describes the MTP implementation in technical detail. The following chapter, Programing Interfaces, documents the software development kit (SDK) for version 2.1 of the MTP library.

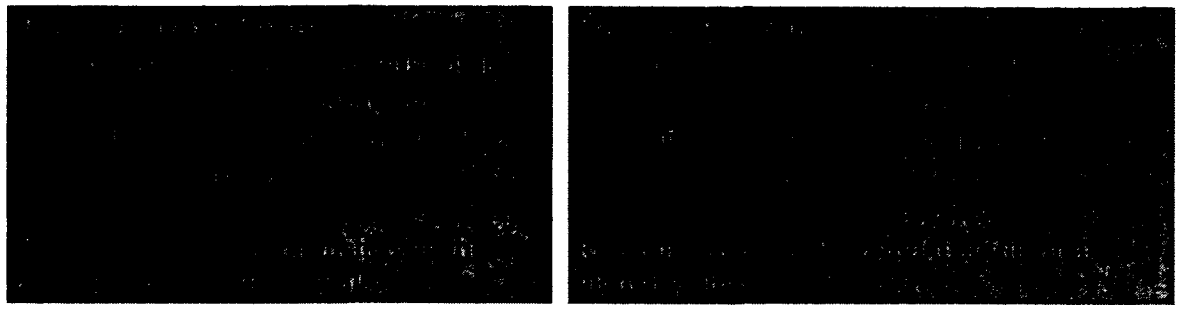
V. Algorithms and Techniques

Many of the goals which MTP has been designed to achieve have only become requirements within the last five years since the advent of the Internet as a public communications medium. Thus, while some of the protocol's approaches to data transport resonate with traditional network models, many of the algorithms have been newly developed based upon more recent constraints. This chapter describes the algorithms and techniques used to achieve the previously discussed design goals. The reader may wish to refer to "Appendix A: mtp.h" for coding references. The following chapter examines the implementation details from an application programing perspective. Except where noted, all of the mechanisms described here have been implemented and tested under version 2.1 of the MTP protocol. The results of these tests, as well as that of a simplified simulation model, are described in the "Testing and Simulation" section.

Transaction Architecture

MTP is designed to facilitate two processes exchanging data over an underlying packet network (currently UDP/IP). Each data transaction is characterized at the application level by the exchange of a *request* message for a *response* message. The node issuing a request for data is the *client* while the node which receives the request and issues a response is the *server.* A given node may act as a client for some transactions while acting as a server for others. However, for any given transaction, there exists exactly one client and exactly one server.

Each request specifies an *action* (such as data retrieval) which the server is requested to take. The request may also describe a *data object*, or potion thereof, to which the action should be applied. The request, describing the action and the data object, must fit into a single network datagram (which current network technologies typically limit to under 8 kilobytes). If the application protocol is able to guarantee that the response will also fit into a *single* network datagram, then MTP places no restrictions upon the data object and will transparently deliver the server's response. *Multi datagram* responses may be up to 2^32 bytes long and must be contiguous and byte addressable. In all cases, data objects must be asynchronously addressable. That is, subsequent identical requests should retrieve exactly



the same data. The client may specify the amount of data it is requesting or it may request all data available. (Facilities also exist for handling semi-infinite and dynamically generated data objects, although they have not yet been extensively tested.) For multi datagram responses, the server may restrict the amount of data that it will transmit at any one time, thus limiting its delivery rate. If the requested data is greater than this limit, or if a portion of the response is lost in transit, then the client MTP will issue subsequent *subrequests* addressing the remaining portions of the data object until the entire request is fulfilled.

The client application specifies a *delivery* method for the response. Data may be delivered to a fixed memory buffer or to an unbounded file (see illustration). (Provisions have been made for dynamic memory buffers and pipes but these have not yet been fully implemented.) The client may elect to receive notification of the progress of a response as it proceeds or to be notified only once the entire response has been delivered. The only action required from the client application after a request has been issued is that the client call the MTP procedure that processes network events with sufficient frequency to service the transaction.

Client & Server Roles

Depending upon the needs of the application, any process may function simultaneously as both a client and a server. But with respect to any single transaction, a given process will either be one or the other. It is expected that most MTP applications will have other functions beyond supporting network communications. Thus both the client and server sides of the MTP code are designed to have a minimal impact on the resources of the parent application. Most importantly, `MTP_Event()`, which performs the bulk of the transport operations, is modeled after the `select()` system call, allowing the application to watch

for file or socket descriptor events without resorting to polling operations. All externally initiated protocol events (such the as receipt of a message) are processed in this procedure via callbacks to application procedures. Programmers familiar with the X-Windows callback and event loop mechanisms will find the MTP mechanisms similar in form and function.

At the application level, the client has the simplest of the two roles. Every transaction is initiated with a request and the client is guaranteed to receive exactly one response (either the requested data or an error). Once the request is issued by calling `MTP_Request()`, the client has only to repeatedly call the `MTP_Event()` procedure to ensure that the transaction continues until the desired response or an error response is received. Responses are delivered to the `MTP_ResponseProc()` specified by the application in its request. The client application may view the request and the response as atomic events with a one-to-one correspondence.

The server application may appear more complex in the sense that it may handle multiple requests and responses per transaction and it has no guarantees that requests will not be repeated or that responses will be delivered. But most of the complexity introduced by asynchronous operation is resolved by good application protocol design, which will be discussed in the next chapter. The actual server code is very simple. It starts by registering `MTP_RequestProc()` handlers for specified action codes via `MTP_Action()`. The server then waits in a loop around `MTP_Event()` as requests are routed to the handlers. Each handler is expected to quickly process the request it is given and then return control to MTP so that other messages may be processed. Generally the last act of a request handler is to call `MTP_Response()`.

At the transport level, the complexity of the roles is reversed. Since the MTP level of the server is stateless, the server side code is extremely simple. It receives requests and routes them to the specified handlers. When the response procedure is called, the message is packaged into one or more datagrams and transmitted toward the client. The client side, on the other hand, is responsible for all of the flow control and error recovery mechanisms of the transaction. The exact status and statistics for each request must be maintained, sub-

requests generated, and error and flow control decisions made. It is thus the client side transport code which makes up the bulk of MTP's implementation.

Messages

At the network level, MTP is characterized by datagram messages exchanged between processes. Both *request* and *response* messages are formatted identically, although some of the fields have different meanings depending on the direction of travel. Each message consists of a 28 byte header followed by zero or more bytes of payload. The header contains information about the protocol versions, message size, flow control options, message identifiers, action specification, response codes, and an address range for the requested data object (see illustration). In general, the description of the data object is contained in the request payload while the data itself is contained in the payloads of one or more response messages.

If there are no network errors and the response fits into a single datagram, then a transaction consists of exactly one request message and one response message. If either the request or response is lost or damaged, then the identical request message will be retransmitted. If the response is too large to fit in a single datagram, then multiple datagrams will be sent, up to a regulated maximum. Each response header carries the starting address corresponding to the payload's position within the requested data object. The header also contains flow control and session identification information specified by



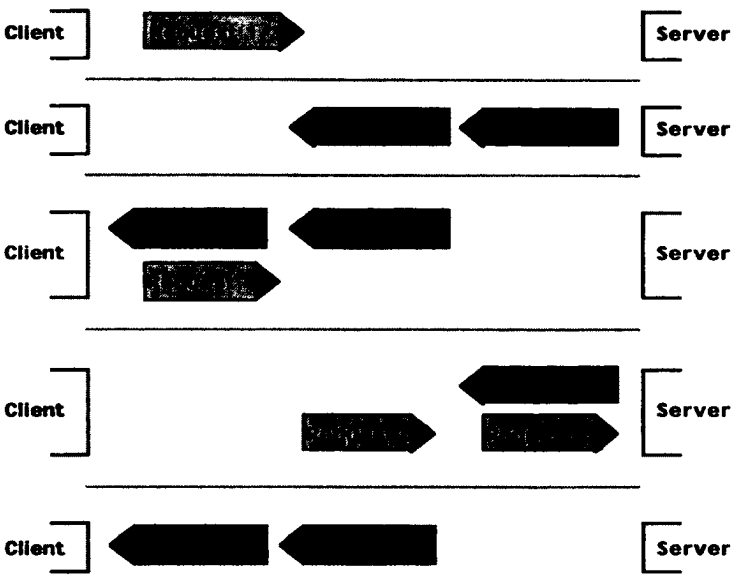
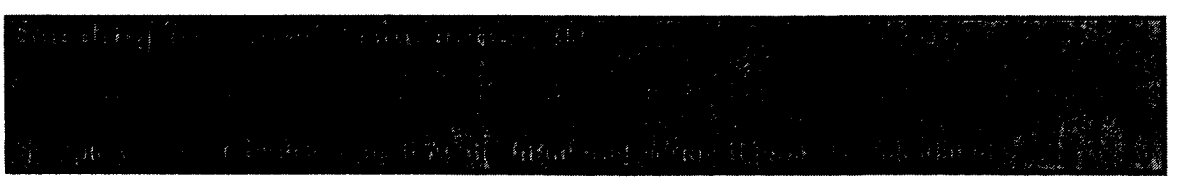


Figure 11: Time sequenced network view of a best case (lossless) HTTP/MTP transaction. For simplicity, it is assumed that the window size starts at 3 and that no window adjustments are made. Compare to HTTP/TCP in Figure 7.

the server. Each response message also specifies the extent of the full response from the given address. If various flow control restrictions prevent the entire response from being transmitted at once, then new request messages are sent specifying the remaining portion. These subsequent request messages are referred to as *subrequests* since they represent only a portion the original request. Figure 11 shows the network view of a lossless MTP transaction with a large response.

Transaction Identifiers

In order to associate messages with their parent transaction, MTP assigns a unique 32-bit identifier, called the transaction ID, when a request is created by the client. It is vital to prevent datagrams from different transactions to simultaneously exist in the network with the same source address, destination address, and transaction ID. Thus the transaction ID must be generated in such a way that it will not roll-over in the expected lifespan of a datagram, especially if the current process fails and a new process begins without



knowledge of the first. The initial value for the identifier is created during the application's call to `MTP_Init()` by using the `lrand48()` function after seeding the random number generator with a 32-bit value consisting of the process id in the upper 16 bits, the lower 8 bits of the system clock in seconds for the middle 8 bits, and the lower 8 bits of the system clock in milliseconds in the final 8-bits of the seed (see illustration). The starting value is then incremented each time a new request is issued. The process must thus issue over four billion requests before reusing an ID, making it unlikely that an ID will be reused within the lifespan of a transaction.

The use of both time and process id elements ensure that a new process is unlikely to choose an initial ID near that of a predecessor Given the disparate elements used in constructing the random number seed and the spectral properties of the of the `lrand48()` random number generator, the odds of any new transaction identifiers crossing the space of datagrams still arriving for the previous process is on the order of one in billions. This already acceptably low probability of conflict is further reduced by the fact that the arrival of a stale response will conflict only if its transaction ID, and server IP and port addresses ALL coincide with those of a newly created request.

The transaction identifier is passed back to the client from MTP_Request() and may be used to identify the response or to make changes to an ongoing transaction (such as aborting it). The server also has access to the transaction identifier of each request message it receives. The combination of the client IP address and port number, server IP address and port number, and the transaction identifier of a message uniquely identifies the transaction to which that message belongs.

Channel Identifiers

Each message also contains a 32-bit channel value which may be assigned by the server application to classify transactions. The initial value in a request message is set by the client. If the server changes the value, then the client MTP will use the new value in all subsequent subrequests for the remainder of the transaction. The new value is also passed back to the client application for possible reuse in future transactions. For example, consider a server which caches frequently requested data objects. It is likely to be

expensive for the server to determine which cache slot contains the object specified by each request payload. By returning an index for the cache slot as the channel id, the client may reuse that ID to reduce the server's lookup overhead. Since the channel ID would be associated with the data object, the same channel ID could be returned to multiple clients across multiple transactions and would not require the server to maintain *transaction* specific state information. The server would, however, need to maintain state information about its *data objects* in order to manage ID and other resource allocation.

Payload Suppression

MTP allows the server to specify that the channel ID it has provided is sufficient to identify the requested data object. Subsequent subrequest messages from the same transaction will then be sent without their payload, relying on the channel ID value to identify the data object being acted upon. This reduces the size of the subrequests to just the 28 byte MTP header plus UDP/IP overhead, comparable to a TCP/IP ACK datagram. Since the channel identifier is a 32-bit value, the server can assure that it will not reuse an identifier in such a way as to conflict with an ongoing transaction. If the server receives a request with a suppressed payload and an invalid channel identifier (as might happen on a very busy server which is thrashing its cache), then it may return a channel error which will cause the client MTP to resend the request with its full payload. Techniques for proper use of the channel identifier with payload suppression are discussed in the "Programing Interface" chapter.

Flow Control

One of the primary goals of MTP is to transfer data as fast as possible. That possibility is limited by the client's ability to accept, the server's ability to send, and the ability of each router in the network path to forward data. If the protocol attempts to retrieve data any faster than the slowest of these entities, then it will cause or contribute to congestion. This will immediately cause delivery delays and eventually lead to packet loss, which will force invocation of time consuming error recovery. Ideally, this "best" value could be calculated by knowing the maximum transfer rates of all of the entities in the network path and the amount of this capacity utilized by third-party traffic. However, there currently does not

exist any practical mechanism by which such information can be directly obtained in a best effort network such as the Internet. The only variable which we can directly observe concerning the network path is the delay between the transmission of a request and the receipt of the corresponding response, also known as the *round trip time* or *rtt*. From this value we may infer information about network queue lengths, congestion, packet loss, throughput, and link speed.

Ultimately, we wish to regulate the rate at which data is transmitted by the server such that it matches the available capacity of the network path. We could attempt this by having the client specify that the server transmit the requested data at a specified rate. However this direct flow control would violate several of our design goals. The most obvious problem is that it would create a server side state burden in maintaining the flow rates for each transaction. But more importantly, maintaining that state at the server would create an excessively long control loop. Given a drop in network capacity, the change must first be detected by the client, then new flow constraints successfully communicated to the server, and finally the results observed and evaluated by the client. While this mechanism might be acceptable for congestion which is slow to evolve, it would not be capable of reacting in a timely manner to sudden or transient congestion. Therefore, instead of directly regulating the transmission rate of data, MTP takes advantage of the relationships between the *round trip time*, congestion, the data arrival rate, and the amount of data which has been requested (as explained in the next section).

Non-contiguous Sliding Window

Each request represents a quantity of data which we expect to be delivered at some point in the future. The total amount of data which we have requested but which has not yet been delivered is the amount in *transit*. For any given *transit* value, the rate at which data will arrive is inversely proportional to the *round trip time*. But the *round trip time* itself is a function of congestion: greater congestion correlates with a larger *round trip time*. Thus if we fix our *transit* value, then increases in congestion will immediately result in a decrease in flow, and vice-versa. This adjustment is intrinsic to the behavior of the network and thus is not dependent upon a lengthy control loop to take effect. The limit on the amount of

data to have in *transit* at any given time is called the *window* size. As each response message arrives, the amount of data in *transit* is momentarily reduced. We then issue a new request for more data, bringing *transit* up to the *window* limit and causing the address range of the data in *transit* to slide forward.

Thus far, this mechanism may sound similar to the TCP "sliding window" strategy. However there are several crucial differences. The TCP window represents a specific RANGE of data addresses. But the MTP *window* applies only to a QUANTITY of data. The MTP data in *transit* is not required to be contiguous and may span any address ranges within the *data object*. As we will see below, this has tremendous advantages when recovering from lost datagrams. Furthermore, the MTP *window* applies to the amount of data requested by the client, NOT the amount transmitted by the server. The amount actually transmitted may be reduced by server load restrictions or by the loss or delay of request datagrams. Such a reduction would be an intrinsic result of network conditions and thus would initiate flow throttling before the client has even detected the problem. Finally, the *window* value is maintained and enforced at the same location where the data is being received: the client.

This *non-contiguous sliding window* algorithm is the key to MTP's efficient flow control and error recovery. The intrinsic feedback created by the relationship between *rtt* and data flow causes the flow to respond immediately to transient or sudden congestion. The concentration of feedback and control at the client allows accurate assessments of network conditions to be made and immediately translated into flow adjustments. As a result MTP is able to minimize network congestion while maintaining a steady rate of data flow.

Figure 12 illustrates the flow control strategies of the MTP algorithm versus the TCP algorithm as they discover the maximum sustainable rate for a given network path at each point in time. Both increase their flows until an error occurs, and then throttle back. In the TCP case, this results in wild swings between congestion and under-utilization. The MTP algorithm just described takes a much more measured approach. The loss of a datagram initially reduces the flow only by the size of that one datagram. This reduction alone may be sufficient to relieve congestion. Once the loss is detected by timeout, the reduced flow

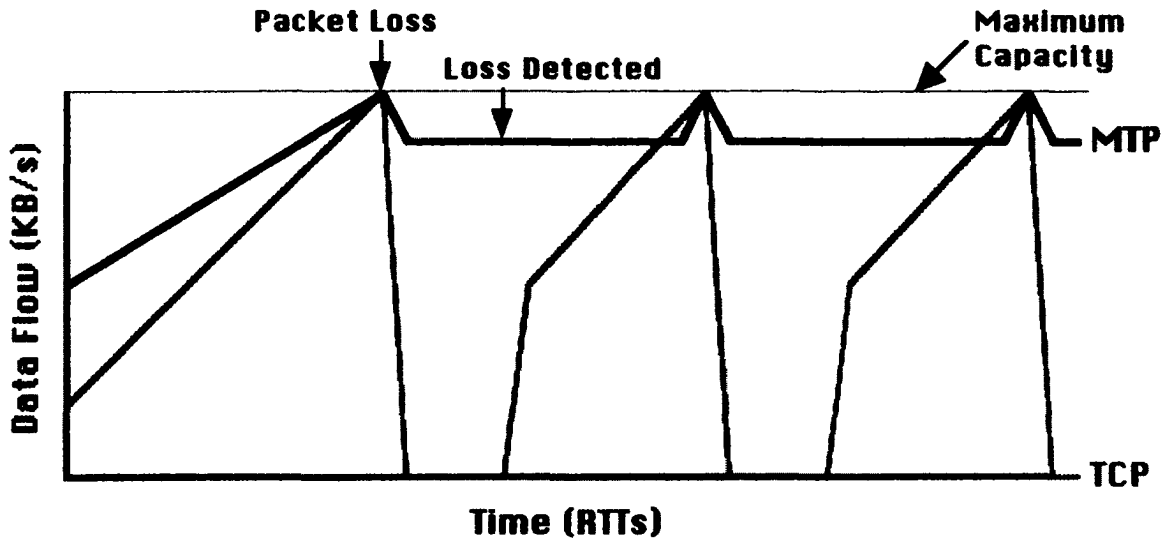


Figure 12: Schematic illustration of MTP and TCP flow oscillation. Both protocols increase their data flow until an error occurs, although TCP increases much more quickly. In the absence of third party traffic, both will encounter loss due to congestion when they reach the maximum capacity of the path. Both protocols reduce their flow after a loss: TCP drops to zero while MTP is reduced only by the amount lost. Upon loss detection (timeout), TCP resumes (depending on implementation) at half the previous rate and continues increasing. MTP ratifies the existing reduction and will not attempt another increase for some time. For the purposes of this schematic, it is assumed that only one datagram is lost per congestion and that MTP and TCP are using identical timeout values.

is confirmed by a reduction of the window size before the missing data is again requested. Thus error recovery is seamlessly integrated into the ongoing flow control. After a sufficient time with no further loss, MTP will again attempt to increase the flow to see if more bandwidth has become available. Chronic congestion and loss would force continued reductions in the MTP window. But recurrent losses due to exceeding the path capacity result in a stable flow of data for MTP at a rate just below that which causes congestion. The overall throughput of each protocol is proportional to the area under the curves in Figure 12. The algorithmic differences between MTP and TCP guarantee that the area under the MTP curve is always greater than that under the TCP curve for the same time period. While this schematic diagram represents a simplified case, the results in Chapter VII of experiments conducted on the Internet confirm that this behavior holds under real world, mixed-traffic conditions.

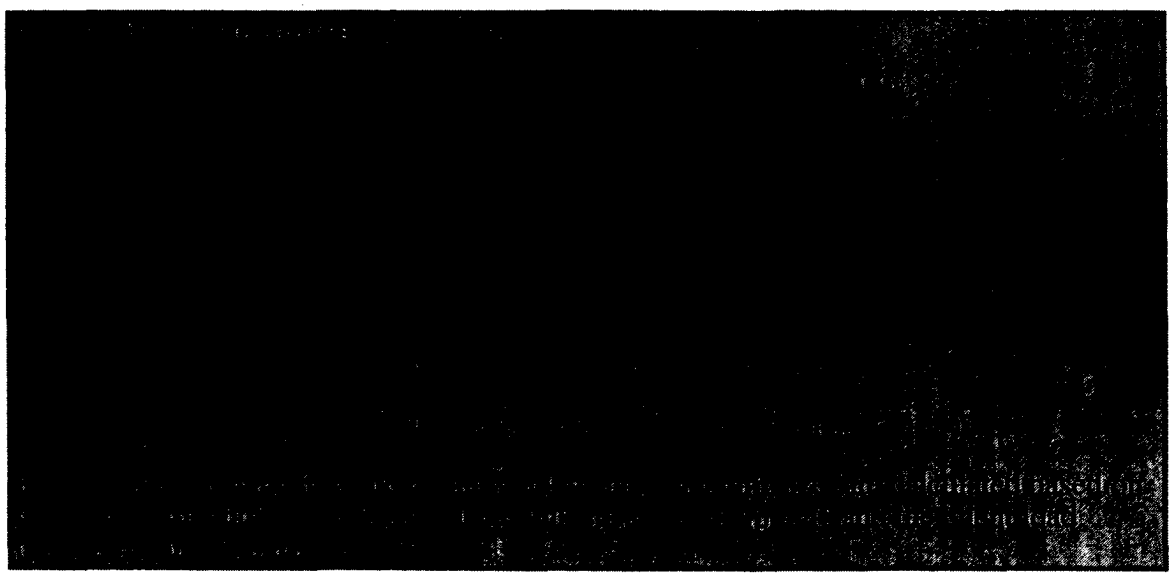
Mechanisms

The initial *window* size for a *multi datagram* transaction is taken from a global value.

Currently this value is fixed as the lesser of half the UDP buffer size and half the per-second throughput of the network link (if provided by the application). Future enhancements of the protocol will change this calculation to incorporate the average window size of recent transactions. If the request specifies a quantity of data less than the *window* size, than this value is presumed to be the amount of data in *transit* once the request is sent. Otherwise the *window* value is considered in *transit*. To make sure that the server does not return more than this amount of a data, the client MTP specifies a *response limit* in the message header. The server MTP will not transmit more than that amount of data in response to this request message.

When the first response datagram is received, its size is compared to the server provided *response size* field to determine if the request has been fulfilled by this one packet. If so, then the data is delivered and the transaction is complete. Otherwise a *subrequest* list is created to keep track of which portions of the requested *data object* have not yet been received. The server also returns its own *response limit* value. The server's *response limit* is the maximum amount of data the server is willing to transmit per request message and will be used as an upper bound on the extent of future subrequests. If it is less than the client's previous *in transit* estimate, then we know that the server did not send as much data as we had requested and we must revise the *transit* number.

As each response datagram arrives, the corresponding address range is removed from the *subrequest list* and the *transit* value is reduced. The data is placed in the delivery area

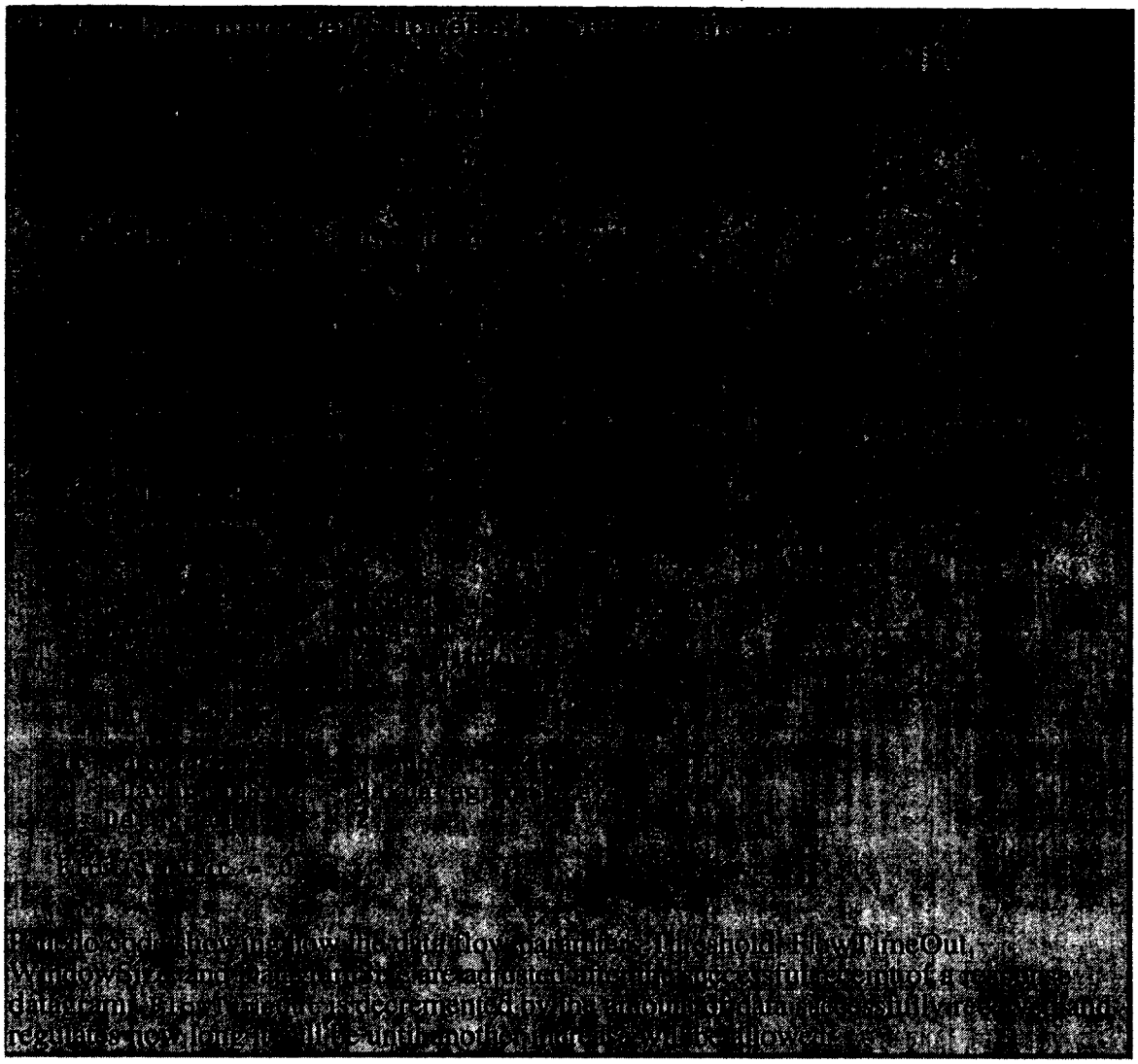


specified by the client. If the amount of data in *transit* is now less than the current *window* size, then a new subrequest is generated for the first address range in the *subrequest list* that is not already in *transit*. The extent of the new subrequest is limited such that the new *transit* value does not exceed the *window*. Ideally, this process will continue until the *subrequest list* is empty or, in the case of a dynamic data object, the server returns an error code to indicate that no more data is available. At that time the client response handler is called to deal with the newly assembled *data object*. The client may also elect to receive notification as to the progress of an ongoing transaction. If the request includes an *update* value, then the response procedure will be called each time the first contiguous portion of arrived data grows by at least *update* bytes.

Window Adjustment

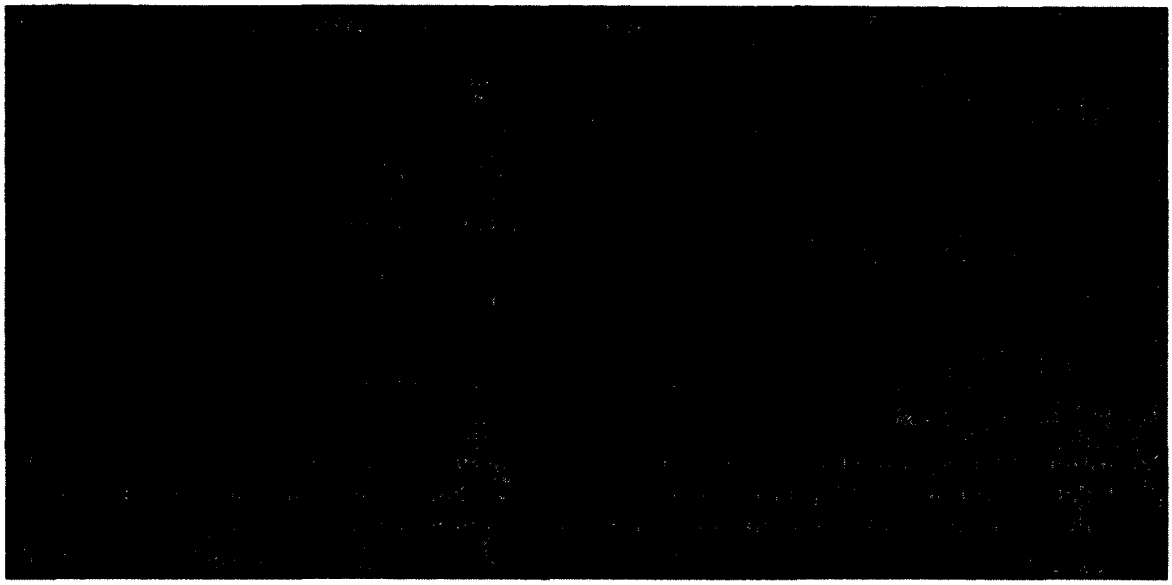
MTP adjusts to changes in the network path capacity by altering the *window size*. In order to create stability, *window size* adjustment is regulated by the *flowtimeout* variable. This value represents how much data must be successfully received before the *window* is eligible for either a decrease or increase. Initially *flowtimeout* is zero and it is decremented by the amount of data received in each response message and by the amount of any data which is declared lost (as described below). The *window* will be increased by the size of one datagram whenever *flowtimeout* reaches a *threshold* equal to an integer value multiplied by *window*. The threshold multiplier used depends upon the condition of the transaction.

If no data loss has been detected during a transaction (no repeats have been required), then a multiplier of one is used. Thus the *window* size (the amount of data allowed in *transit*) is initially incremented for every successful *window* of data received. Note that this fastest rate of MTP window growth is similar to the slowest rate of growth for TCP. If at least one datagram has been lost, then it is presumed that we have come close to the maximum flow of the network path and a multiplier of two is used to cut future growth rates in half. In order to prevent congestion, and thus possibly prevent any packet loss, MTP also monitors the average *round trip time* of the last eight response messages of the transaction. If this value exceeds five times the smallest *rtt* observed during the transaction, then it is presumably because we are filling up data queues along our network path and may be



approaching congestion. In this case, the threshold multiplier is set to three, cutting growth to one third its maximum rate. If the average *round trip time* has grown to over ten times its minimum observed value or if it is beyond a certain maximum (currently four seconds), then no *window* increases are allowed until the delay is reduced. Finally, if the minimum observed *rtt* for the transaction is small (currently less than 40ms), then the *window size* is not allowed to increase above the size of the incoming UDP kernel buffer. This is done to hedge against sudden flooding of the client since low-latency paths such as LANs are likely to be bursty. Otherwise the *window* is limited by the lesser of twice the datagram buffer or twice the per-second speed of the local network link (if provided by the application).

The *flowtimeout* variable is also used to regulate *window* adjustments in response to data loss. When a loss is detected, as is described below, the *window* is reduced by the size of



one datagram and *flowtimeout* is set equal to the positive value of the new *window* size. Except for the case of an incoming buffer overflow, no further *window* reductions are permitted until *flowtimeout* is reduced to zero or less. The positive value also adds to the delay before it is considered safe to increase *window*. If the window has become very small then a minimum *flowtimeout* of three times the current datagram size is used. This brief moratorium on further adjustments prevents the protocol from over reacting to a cluster of lost data which may have been caused by transient congestion. If the congestion is chronic, then continued data loss will drive the *window* lower until it reaches its minimum (one datagram) or conditions allow it to rise. Figure 13 shows a network level view of MTP data loss and recovery.

Datagram size is chosen as a function of the *window* size. Specifically, the amount of payload permitted per response datagram is regulated by *window*. Whenever there are fewer than four datagrams per *window*, the size is cut in half with a minimum payload of 512 bytes. This reduction is done to decrease latency, increase the chances of each datagram finding space in congested queues, and to improve the resolution of our flow control statistics. Whenever there are more than sixteen datagrams per *window*, the size permitted for each payload is doubled. This increase reduces the processing overhead at all points along the path. The maximum payload per datagram is limited to the lesser of one quarter the incoming buffer and the amount of data the local link can deliver in one second. (Currently an 8 kilobyte limit is enforced as well. The next protocol revision will also

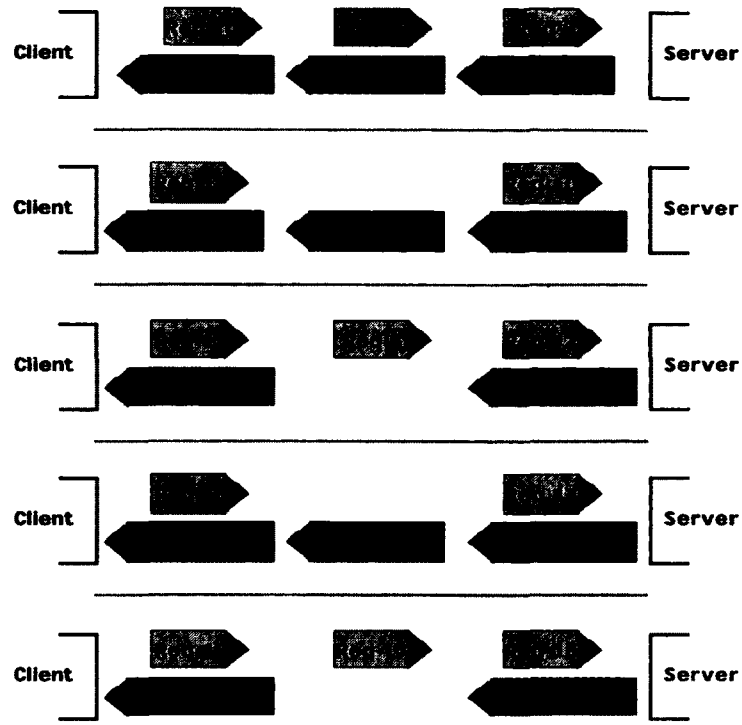


Figure 13: Time sequenced network view of MTP error recovery. Initially the window size is six times the response datagram size (allow six units of data in transit). Thus request 11 is delayed by the failure of response 5 to arrive, but is sent once response 6 arrives. Data flow continues during error recovery and there is no redundant data. Once detected, the missing data is requested as part of the reduced data flow (now with a window size of five). Compare to HTTP/TCP in Figure 8 noting that the time steps are much smaller here. Further errors may further reduce the flow rate. Without errors, it will grow.

regulate datagram size according to the number of outstanding transactions and recent statistics.) Note that not every datagram will contain its maximum permitted payload since the *window* limit on in *transit* data and server limitations may restrict the amounts requested or sent per datagram.

In addition to the parameters of *each* transaction, the client MTP watches the incoming UDP kernel buffer shared by all transactions to ensure that data is not being requested faster than the client is capable of receiving it. Whenever the incoming UDP queue exceeds half the buffer size, the *window* is reduced on the fastest transaction and all active transactions have their *flowtimeout* values reset to zero which delays their normal schedules of flow increase. (Currently, only the *window size* is used to determine the fastest transaction. A future revision will also use the *rtt* to determine the transaction with the

greatest flow.)

The server application may also respond to excessive load by lowering its response *window limit*, which reduces the maximum amount of data sent per request datagram. This forces a slower start up for new transactions and increases the latency (thus reducing the flow) for existing transactions. Overloading of the server will also have the inherent effect of delaying its responses which will also increase latency and reduce flow. In case of extreme overloading, the server may choose (or be forced) to ignore some requests. Note that because MTP handles such loss smoothly, dropping requests at the server will its reduce load without causing extreme delays at the client.

Data Loss and Recovery

The client MTP keeps track of exactly which address ranges of the desired response are in *transit* at any given time. It also maintains a notion of a *repeat interval* by which it judges when a portion of data should be declared lost and its request repeated. The initial *repeat interval* value is currently set by the application. (The next protocol revision will use a calculated initial value unless a *single* response transaction has been specified by the client.) As responses are received, the repeat value is adjusted such that it remains between two to four times the average round trip time. A minimum *repeat* interval (currently 500ms) is enforced due to the tendency for fast links to have have very unstable latencies. (Experiments suggest that this minimum is higher than necessary and so it may be lowered to 400ms in the next protocol revision.)

Experiments have shown that the *repeat interval* must be at least twice the current round trip time and greater than a certain value due to the tendency of the round trip time to increase suddenly when additional third-party traffic is introduced into the data path. Smaller *repeat intervals* result in the duplicate request of data which has not actually been lost. Since such behavior would contribute to congestion at the moment when congestion is already increasing, it is considered vital to avoid duplication.

When datagrams are lost, the *transit* value is not reduced by their arrival and thus the rate at which new subrequests can be generated is diminished. Until the *repeat interval* expires,

the transaction's *window* size is effectively reduced by the amount of missing data. Thus having a *repeat interval* value that is larger than the average round trip time causes the data flow to be temporarily reduced in proportion to the size the burst before the loss has even been detected by the client. Once the loss is detected by the data's failure to arrive within the *repeat interval*, it is no longer considered in *transit*. The window may then be set to one datagram less than it's previous value. In most cases, this means that the effective flow rate will continue at the same slightly reduced level. But after a burst of loss, the new flow rate will actually be an increase compared to what it was during the last *repeat interval*. This prevents the transaction from stalling and promotes continued stability.

But having too large a repeat value delays error recovery and thus reduces the flow below optimum levels. A large timeout may also delay completion of the transaction if data is lost very at the end of the response. Choosing the correct repeat value is most critical during heavy congestion when the window size may be reduced to as little as a single datagram. In such a case, loss of a datagram represents a much larger proportion of the total flow and a timely recovery dramatically affects the overall transaction time. The MTP transaction normally continues until all of the requested data is received If absolutely no responses arrive for a period of time set by the application, then the transaction is considered *expired* and an error response is delivered to the application.

Summary

The various mechanisms described above all work together to provide several levels of data flow regulation and thus maintain a rate which matches the changing capacity of the network path. The actual rate at which data is transferred (in terms of bytes per second) is determined by the combination of the *window* size set by the client and the *round trip time* which is a function of congestion. Thus the first level of flow control is the variation in latency which, for a given *window* size, alters the rate of data transfer in inverse proportion to the level of congestion. When congestion becomes critical, datagrams are lost and the second level of flow control comes into play. By preventing delivery of a response, the network immediately reduces the load on all downstream nodes and reduces the rate of further requests. As a third and final level of flow control, the client responds to chronic

data loss by steadily lowering the *window*, and thus the flow rate, until no more losses occur. As loss subsides, the *window* may be slowly increased. But this increase is regulated directly by changes in the *round trip time*. This may allow congestion to be avoided BEFORE packet loss becomes an issue. Finally, both the client and the server may make adjustments to the flow in response to their own load conditions.

Issues and Considerations

The design goals and techniques thus far discussed have focused on the task of creating a protocol which performs efficient transaction transport across the Internet. But there are other considerations which should be taken into account. Foremost is the effect the protocol may have on network resources and third party traffic. Since MTP has been designed to respond more quickly and proportionately to congestion problems, it is expected that a network using MTP will exhibit much greater stability and throughput than one using primarily TCP. Tests of this assertion are discussed in the "Testing and Simulation" section.

More specifically, there is the concern that the higher network utilization of MTP could result in lower performance for simultaneous TCP streams. However MTP's is also much more stable than TCP. It does not contribute to synchronicity and it's presence along the network path may be able to fill in the valleys while smoothing out the peaks of TCP's flow oscillations. Experiments discussed in later chapters suggest that over WANs, MTP transactions have little impact on simultaneous TCP streams. Over LANs, where it is much easier for a CPU to saturate the link, MTP may interfere with simultaneous TCP streams on the same link. This is discussed more in the chapter "Third Party Effects".

Another concern is CPU overhead. While the server side has been reduced to far less overhead than TCP, the MTP client is relatively heavy. It relies on numerous statistical calculations and the code is far from the instruction optimized leanness of TCP. However this is not considered to be a burden for two very important reasons. First, computers today are many orders of magnitude faster than they were when most TCP implementations were created. It is simply not necessary to worry about integer versus floating point math or how many additions are needed per datagram. While efforts have been made to ensure

that the MTP implementation is as efficient as possible, minimization of CPU use is simply not a design priority. Second, MTP has been implemented as a user level code library with non-blocking system calls. Since it is not part of the kernel, MTP has a greater latitude in its use of resources. The use of non-blocking system calls combined with a design emphasis on short, discrete operations, allows the protocol library to quickly return control to its parent application. In spite of the fact that CPU efficiency has not been a priority of MTP, it never the less exhibits extremely low overhead. On a Sparc 5/70 CPU, a five year old workstation, it uses less than 1% of the CPU even while receiving network data at over one megabyte per second on a 10 megabit ethernet link.

Perhaps the most subtle issue affecting protocol design is that of security. The fact that MTP is meant to operate across the public Internet means that it will inevitably be subject to denial of service, spoofing, packet sniffing, and other attacks. Because the balances between risk, flexibility, and overhead can vary widely, most security issues have been left to the application level. Encryption and authentication schemes, for example, are to be developed as part of the application protocols. MTP provides mechanisms to aid their development, such as providing session and transaction identification, and makes no assumptions about the contents of packet payloads. Future enhancements to the protocol will include an option for creating fully encrypted datagrams, including the MTP headers (which would be decrypted by an application supplied procedure). Again, the details will be left up to the application, but MTP will provide the mechanisms.

A more recent design requirement is for direct regulation of real-time data streams. Although this topic has not been greatly addressed in this dissertation, it is worth noting that MTP began it's life as an IP telephony mechanism and thus contains numerous adaptations and techniques for efficiently supporting real-time multimedia streams. But since such traffic makes up only a tiny fraction of current network volume, description of these techniques has been left for future work.

No doubt there are a great many other considerations which MTP's design and implementation should take into account. Thus the protocol and its implementation have been designed around the core goals described above with an eye toward maximizing

modularity and flexibility. It is expected that while not every problem has been foreseen, every issue can be reasonably accommodated.

VI. Programming Interface

This chapter documents the Software Development Kit for the Multimedia Transaction Protocol library, version 2.1. The Kit consists of a binary code library compiled for specific platforms, a C header file containing all of the necessary structures and function prototypes, and basic documentation. The purpose of this section is to both document how MTP may be used by an application programmer and to provide a greater insight into the implementation of MTP itself. The following chapters examine the use of MTP for web service as well as the performance, testing, and modeling of the protocol. A complete listing of the header file, whose contents will be referred to extensively in this chapter, is provided in Appendix A. This chapter assumes a familiarity with that code as well as the principles discussed in the previous two chapters. Appendix B contains source code for the program "`mtping`" which illustrates the use these interfaces in a simple query-response application. It should be noted that the data structures used in the API are somewhat abstracted from the underlying mechanisms, allowing the format of MTP messages to be slightly altered without disturbing the applications.

Overview

Each MTP application must begin its networking by calling `MTP_Init()`. This establishes MTP networking on a given UDP port and sets the level of debugging output. The procedure also requires a character string which will be used to identify the application to other MTP clients. If it is known, the speed of the local Internet link may also be specified using a `LINK_` descriptor. (More recent revisions of the protocol are able to discover the link speed dynamically.) This information can usually be obtained from the user at the time the application is installed. More enlightened operating systems store the speed globally for each link. When in doubt, a lower link speed should specified. If the application plans to make multiple requests, then this value is not critical as MTP will quickly adapt to whatever conditions it finds.

```
int MTP_Init(uint16     port,            /* Local UDP port number */
             int        link,           /* Link speed description */
             char *     idstr,  /* Description of this application */
             int        debug            /* Debuging detail level */
             );
```

Incoming data is received via callback procedures which must be registered with MTP. Requests are handled on the basis of their *action* code. `mtp.h` lists some standard action codes and describes their functions. By default, `ACT_PING` and `ACT_IDSTR` are handled internally by MTP, although the application may substitute its own procedures.

The application specifies how incoming requests are handled by calling `MTP_Action()`. This procedure takes the action to be registered, the `MTP_RequestProc` that will be called for this action, and a 32-bit field that will be passed unmodified to the action procedure. `MTP_Action()` returns the previous procedure associated with the action (or NULL if there was none). Response handlers are registered at the time a request is made and will be discussed under "Client" below.

```
MTP_RequestProc
   MTP_Action(int       action,          /* Action being registered */
              uint32    user,          /* Data to be passed to rproc */
              MTP_RequestProc rproc     /* Procedure to be called on
                                                          action */
              );
```

The actual processing of MTP messages is handled within the `MTP_Event()` procedure. This procedure must be part of the main application loop and must be frequently called upon in order to maintain MTP networking. This procedure is similar to `select()` in that it accepts file descriptor sets for tracking stream events as well as a timeout value. `MTP_Event()` will return after processing messages, when an event occurs pertaining to one of the file descriptor sets, or after `maxtime` milliseconds have elapsed. If `maxtime` is set to `-1` then `MTP_Event()` may wait indefinitely. If maxtime is set to `0`, then `MTP_Event()` will return immediately after handling pending traffic.

```
int MTP_Event(fd_set *  readfds,           /* Readable descriptors */
              fd_set *  writefds,         /* Writeable descriptors */
              fd_set *  exceptfds,        /* Exception descriptors */
              sint32    maxtime         /* Timeout in milliseconds */
              );
```

It is important to allow `MTP_Event()` to be active as much as possible so that network activity can be handled in a timely manner. An application which is expecting a significant volume of network activity should not go without calling on `MTP_Event()` for longer than a few milliseconds. If MTP networking is expected to be minimal (as for simple control functions), then longer intervals are acceptable, but still should not exceed about 500

milliseconds. In addition to modifying the file descriptor sets, `MTP_Event()` returns the number of messages processed during its call.

Upon completing its network tasks, the application should call `MTP_Close()`. Currently, this function is only needed to release process resources. But future revisions of the library may allocate extra-process resources such as files or shared memory blocks. Therefore `MTP_Close()` should be called prior to the process exiting to ensure that these resources are deallocated.

```
int MTP_Close();
```

Client

The client is defined as the originator of a request and the recipient of a response. All transactions begin with a client application calling `MTP_Request()` and end with the client MTP calling an `MTP_ResponseProc` procedure. In most cases, the client's request identifies a portion of a data object on a server and some action for the server to take. Upon receipt of a request message, the server will return one or more response datagrams. Once the entire response has been received, and optionally at prior intervals, the `MTP_ResponseProc` provided in the `MTP_Request()` call will be invoked.

```
int MTP_Request(mtp_id_t *      id,          /* Request identifier */
                mtp_request_t   request,    /* Request description */
                int             size,            /* Payload size */
                void *          payload,     /* Pointer to payload */
                uint32          repeat,     /* Request repeat rate */
                uint32          expire, /* Request expiration time */
                mtp_delivery_t  delivery,  /* Delivery instructions */
                uint32          user,/* Data to be passed to callback */
                int             update,        /* Update rate, in KB */
                MTP_ResponseProc callback      /* Proc to receive
                                                        response */
                );
```

The `payload` pointer addresses the data which should be transmitted in this request while `size` indicates its extent. If there is to be no payload, then the size should be `0` and the pointer set to `NULL`. The request payload is limited to `MTP_SIZE_REQUEST` bytes (currently `1024`). The `repeat` value provides a timeout, in milliseconds, after which the initial request may be repeated if no response has yet been received. If no responses are received in `expire` milliseconds, then the transaction will terminate with an error response. As a

special case, `expire` may be set to zero. MTP would then send one request message and will not expect or accept any responses. In that case, `callback` would never be invoked and may be `NULL`.

The destination for the request is given in the `mtp_id_t` structure:

```
typedef struct {                /* Uniquely identify a transaction */
   struct in_addr  peerip;     /* IP address of peer - Network Order!*/
   uint16          peerport;    /* UDP port number -  Network Order! */
   uint8           mtpversion;  /* MTP protocol version */
   uint8           appversion;  /* APP protocol version */
   uint32          reqid;       /* Request id number */
} mtp_id_t;
```

The `peerip` and `peerport` identify the IP address and UDP port number of the server. The `appversion` specifies the revision level of the application protocol. This value is passed to the server where it may be ignored or used to determine the context in which the request should be interpreted. The `mtpversion` and `reqid` fields will be set by `MTP_Request()`.

The nature of the request itself is described in the `mtp_request_t` structure:

```
typedef struct {                /* Describe a request */
   uint32          channel;     /* Session or object identifier */
   uint32          address;     /* Address within data object */
   uint32          rsize;       /* Size of requested data object */
                                /* (usually EOF - address) */
   uint8           action;      /* Action being requested */
   uint8           flag;        /* Action modifiers */
   uint16          reserved;
   int             single;      /* Flow control disabled flag */
   int             stream;      /* Dynamic object flag */
   int             suppress;    /* Payload suppression flag */
} mtp_request_t;
```

The `action` field, optionally supplemented by the `flag` field, defines the purpose of the request. Several standard actions are defined in `mtp.h` and these values should be used whenever they apply. Application specific actions may begin at `ACT_APP` (currently 64). The meaning of the rest of the fields, as well as MTP's behavior, depend upon the value of the `single` flag. When set true, the application promises that the response payload will fit into a single network datagram. This causes MTP to disable most flow control for this transaction and to accept the first response message it receives as the entire response. IP permits a payload size of up to approximately 63 kB, but the network path may have tighter

restrictions. Either the client or the server may have kernel level restrictions on the maximum datagram size they can handle. Generally, up to 8 kB of payload is safe, but `single` should really only be used for minimal (at most 1 kB) responses.

For a `single` request, MTP ignores the values of `channel`, `address`, `rsize`, and `flag`, passing them without modification between the client and the server. For non-`single`, also known as *multi*, requests, MTP requires that `address` be a byte offset within the requested data object and that `rsize` be the number of bytes requested. Alternatively, `rsize` may be set to zero to indicate that all data available from `address` to the end of the data object is requested. The initial `channel` value may be specified by the client application, but if the server returns a different value during its response, then this new number will be used for the remainder of the transaction. The `suppress`, `stream`, and `reserved` fields are ignored in the request call.

As response data arrives, it will be delivered to the application as specified in the `mtp_delivery_t` structure. Currently, two types of delivery are supported: fixed memory buffer, and file. The method of delivery is given by the type, while the remaining fields describe the destination and boundaries of the destination.

```
typedef struct {
   uint32       type;
   uint32       handle;          /* File descriptor or buffer address */
   uint32       offset;          /* Location for msg.address = 0 */
   uint32       bound;           /* Maximum allowed.  0 for unlimited */
                                 /* on return, max sequenced size */
} mtp_delivery_t;
```

Type `MTP_DEL_BUFFER` is used to deliver data to a fixed size memory buffer. The data will be delivered starting at the address `handle + offset` and may extend no further than `bound` bytes. If more than `bound` bytes are explicitly requested, then an error response will be returned. If the request is for all available data (`rsize` set to zero) then the response may be truncated. The other currently supported delivery type is `MTP_DEL_FILE`. In this case `handle` contains the file descriptor of a file opened for writing. As before, `offset` and `bound` further specify the location and limitation of the response destination. In future protocol revisions, support is planned for dynamically allocated buffers using a mechanism similar to `writev()` `iovec` structures. Support is also planned for pipelines which will

allow responses to be delivered to specified sockets descriptors.

The final portion of the request call specifies how the application shall be notified of response delivery. `callback` is an `MTP_ResponseProc` procedure which will be invoked once the transaction has completed. If `update` is non-zero, then the `callback` will also be invoked each time the initial contiguous range of received data grows by at least `update` bytes. This allows the application to keep track of the progress of a transaction and to provide user feedback. The `user` field is an arbitrary 32-bit value that will be passed unmodified to the `callback`.

```
typedef void (* MTP_ResponseProc)                    /* Receive response */
       (mtp_id_t msg_id,          /* Uniquely identify the transaction */
        mtp_request_t request,      /* Info from the response header */
        int error,                          /* Error classification */
        int response,                             /* Response code */
        uint32 size,                     /* Total size of response */
        int final,                        /* True if this is final */
        mtp_delivery_t delivery,        /* Deliver info from request */
        uint32 user,                     /* Application defined data */
        mtp_stats_t stats);      /* Information about the transaction */
```

The response `callback` procedure has a similar structure to the request call. The `msg_id` contains the same identification as before, but with the `appversion` set to that of the server. The `request` structure is more extensively modified. The `channel`, `rsize`, `action`, and `flag` fields may all have been changed by the server application. If the request was successful and not designated as `single`, then `rsize` will have been set to the amount of data available from `address` to the end of the data object. The server may set the `stream` flag to indicate that this was a dynamic data object. The `suppress` flag indicates that the returned `channel` identified the data object and allowed payload suppression, at least during this transaction. Both flags have meaning to MTP during the transaction. Whether or not they are significant to the client application depends upon the design of the application protocol.

The `error` and `response` fields characterize the nature of the response. If `error` is zero then the requested action was successful. `response` will always be non-zero and may be set by the application server to further comment on the nature of the response. Application specific values begin at `RES_APP` (currently 64). If `error` is non-zero, then it indicates which portion of the network path generated an error. In that case, the value of `response`

would give the specific type of error as defined in `mtp.h`. Note that the error `response` codes vary with the `error` class. Class `ERR_APP` refers to errors generated by the server application. Several standard error `response` codes are defined in `mtp.h` and should be used whenever they apply. Application specific error responses begin at `ERR_APP_APP` (currently 64).

The `final` flag is set when the transaction has been completed. Depending on the size of the response and the value of the request `update` field, the response procedure may be called many times with `final` set to zero, but it will only be called once with `final` set to one. On the final invocation, if there have been no errors, `size` will be set to the amount of data which has been delivered. For update or error calls, `size` will only give the number of contiguously delivered bytes starting from `address`. Additional data may have been delivered, but may have "holes". Only the first `size` bytes are guaranteed to be contiguous.

For the current delivery types, the `delivery` structure will be identical to the request except that `bound` will be set equal to `size`. For an `MTP_DEL_FILE` delivery type, it is very important to note that during an update or after an error the size of the file may be larger than `size`. The application should not attempt to read or map any data beyond `size` as portions of the file may not have been allocated. The dynamically allocated delivery type, when implemented, may change other fields as well.

The `stats` structure provides statistical information about the transaction, including error recovery and flow control information. Most applications will be primarily interested in the `received` and `etime` fields as these provide up-to-date information on the total number of bytes delivered during the life of the transaction as given in milliseconds. Most of the `stats` fields have no meaning for `single` responses and their values are undefined if `single` is set.

Once the response procedure has been called with `final` set, the transaction is considered complete and MTP will remove all state information regarding it. It is then up to the application to do whatever it wants with the data and to deallocate any resources associated with its delivery.

While a transaction is ongoing, the client may make limited changes to its request. Currently, the only supported change is to abort the request in one of several ways.

```
int MTP_Abort(uint32 reqid, int action);
```

The `reqid` must be the transaction identifier of an ongoing request as returned in the `id` field of an `MTP_Request()` call. The `action` specifies one of four mechanisms by which the transaction may be terminated. `ABORT_FINISH` is the least drastic, allowing the transaction to complete up to the furthest address which has already been delivered. Depending on network conditions, the transaction may continue for several seconds. The response procedure will then be invoked without error, assuming no further errors occur. `ABORT_STOP` terminates more quickly, usually within one second, by both truncating the request and by forcing the request to expire very soon. Data which is already in transit will likely be accepted (so that its bandwidth is not wasted), but no further subrequests will be sent. The response procedure will then be called as though the request had timed out. `ABORT_EXPIRE` expires the transaction immediately. During the next `MTP_Event()` call, the response procedure will be invoked with a timeout error. Some data which is already in transit may be discarded. `ABORT_DESTROY` completely wipes out the transaction. The response callback will not be invoked and no data will be delivered. Delivery buffers or files provided by the application will be close to the state of their last update. Dynamically allocated buffers will be deallocated and must not be accessed.

Server

The server is defined as the recipient of requests and the originator of responses. The server registers `MTP_RequestProc` request handlers for each `action` code it supports, as was discussed under "Overview". Each handler is expected to process a single request message per invocation and then make one call to `MTP_Respond()` to send its response. The server may perform additional tasks, but it must call `MTP_Event()` with sufficient frequency to service its transaction load.

```
typedef void (* MTP_RequestProc)              /* Handle action request */
     (mtp_id_t msg_id, mtp_request_t request,
      int size, void *payload, uint32 user);
```

The `msg_id` field contains the IP and UDP port numbers of the client which issued this

request along with its MTP version, application version, and the id of the transaction to which this request message belongs. The combination of the `peerip`, `peerport`, and `reqid` fields unique identifies the transaction. The request field describes the nature of the request itself. The meaning of the fields is as was described in under "Client". If the server is making use of the `channel` field for some form of session or object identification, it should anticipate that the information may be stale and be prepared to recover gracefully. If the `suppress` flag is set, then there is no payload and the server must rely on the channel to identify the data object or session. If this can't be done, the server should respond with `ERR_APP_CHANNEL`, which will cause the client MTP to resend the request with the full payload. Any request payload is pointed to by `payload` and is of length `size`. The `user` field contains whatever data was declared in the `MTP_Action()` call.

Once the server has interpreted the request and prepared a response, it invokes `MTP_Respond()`.

```
int MTP_Respond(mtp_id_t        id,          /* Request identifier */
                mtp_request_t   request,    /* Request description */
                int             error,          /* True for error */
                int             response,        /* Response code */
                int             size,             /* Payload size */
                void *          payload     /* Pointer to payload */
                );
```

The `id` field should be identical to that passed into the response procedure except that the server may set its own `appversion`. As noted in the Client section, the meaning of the `request` fields depends upon whether or not this was a `single` request. If it was, then the server can set any values for the `channel`, `address`, `rsize`, `action`, and `flag` fields. If this is to be an error response, then the server also has a broad discretion in the values it returns for these fields. But for normal *multi* responses, `address` must be the byte address within the data object of the beginning of the response payload while `rsize` is the extent of the *entire* data object starting at `address`. Note that `rsize` is NOT the amount of data being returned by this response, but rather the amount that COULD have been returned. MTP also requires that the `reserved` field passed in the `request` structure be returned without modification in the response. In general, it is easiest to call `MTP_Respond()` using the same `id` and `request` data structures as were passed to the request handler, making whatever modifications are necessary to individual fields.

For dynamically generated data objects, the server must notify the client of the amount of data which the server is willing to buffer. It does so by setting the `stream` flag to one and `rsize` to the maximum amount of data which the server will buffer. Note that unlike a static response, the `rsize` value is fixed and does not depend on `address`. The client MTP will assure that the range of requests outstanding during the transaction does not exceed `rsize`. Thus if the server receives a request for dynamic data which is some amount beyond its declared buffer, it is assured that an equal amount at the beginning of the buffer has been received by the client and may be discarded. In practice, the server may wish to buffer at least 512 bytes more data than it declares in `rsize` so that it will immediately have room to slide the window forward as new requests arrive.

If there is no further data available at the time a request is received, but more is expected to be generated in the future, then a response should be sent with no payload and `rsize` set to zero. The client will then repeat its request without expiring. This state can be maintained indefinitely so long as the server continues to send back these zero payload messages. If a request arrives for an `address` beyond the end of the dynamic object, the server must return `ERR_APP_EOD`. In this special case, `rsize` should be set to the TOTAL size of the object. This lets the client know where to terminate its requests. Once the client has received all of the responses up to EOD, it will send a final request with both `address` and `rsize` set to `~0` (all bits set to one). The server must respond with a zero length reply and may then regard the transaction as complete.

Application Protocol Design

When creating an application which will use MTP, the designer must begin with the design of their application level protocol. While byte-pipe protocols such as TCP allow for the free-form exchange of messages between peers, MTP is much more structured and requires the designer to be aware of some issues hidden by TCP. Most obvious is the fact that MTP is a transaction protocol and thus is best suited to request-response exchanges. More specifically, MTP is structured around the notion of one process seeking to perform an action upon a data object located at another process. The action and the data object are described in the request and the results are returned in the response. This is a somewhat

broader description than the client-server model discussed previously and reflects the fact that MTP can be used for a wide range of applications beyond simple data retrieval.

The first step to designing an MTP application is thus to decide what data objects are going to be operated upon and how they will be identified. In the case of HTTP, the data objects are files or CGI executable programs and they are identified by text strings. For network telephony, the objects may be call sessions identified by the user. A networked file system might operate directly on files, directories, and links by name or node number. A database system might use complex queries or direct indices into data files. The designer should also be aware of the likely distribution of their data objects. In the classic client-server model, there will be many clients accessing a few central servers. But transactions may also occur between peers or even in few-to-many layouts.

The next step is to decide what actions may be taken upon the data objects. The most obvious is the retrieval of data, but there will likely be a need for many control functions as well. Data locking and exclusion, session creation and tear down, differing query types and formats, and other data management functions must be considered. For example, it may be desirable to create an MTP server which manges the use of a hardware data source (such as a video feed). Remote clients would need actions to perform various shared and exclusionary operations on the data feed, while a local client might be used as a user interface to send control and configuration commands. (This is referred to as a "proxy interface" or "proxy control" in other MTP documentation.) When designing actions, it must be decided whether the actions are intended to retrieve arbitrarily large responses, or single datagram responses. In general, it is best to use `single` type transactions for control functions since they require less overhead and allow many of the header to fields to be used for application data. Requests for potentially large responses must be more structured. Whenever possible, the standard action and response codes listed in `mtp.h` should be used. This simply makes the code more portable and can be extremely useful in debugging.

The final design step is to determine how the server will actually process the action requests it receives. This may be an iterative process involving revision to the nature of objects and actions as the designers familiarize themselves with the particular issues important to an

MTP server. The most pressing issue is the fact that the server cannot assume that responses will be delivered and must cope with repeated requests. Thus an action must never perform an incremental change on a data object without providing a synchronization mechanism. For example, if a client wishes to increment the value of a register on the server, the server must NOT simply increment the value for each request it receives. Repeated requests due to network errors would then cause unintended state changes. Instead, the client could request the current value and then submit a new updated value. If the value was session or client specific, an exclusionary lock function could be included in the value retrieval and change. Alternatively, the server could record the transaction id of the increment command and take no action upon repeats but still send confirmation that the job was done. In practice, a small amount of planning can make dealing with the asynchronous nature of an MTP server very simple.

Another consideration unique to MTP is the limitation on the request size. If there is a need for one process to initiate the transfer of a large quantity of data to another, it can do so simply by implementing a reversed request. The client would set up its own request handler and assign a channel id to the data it wishes to send. It would then issue a request to the server instructing the server to send its own request for the data to the client. This simple mechanism is used by the MTP implementation of HTTP to handle `POST` methods. While it may sound convoluted, it only adds one round trip time to the transaction and gains all the speed benefits of MTP's transaction optimizations.

Different applications may use MTP in very different ways, but the basic principles of actions on objects should remain constant. An HTTP server, a TCP/MTP protocol gateway, a file transfer client, and a suite of telephony applications have already been created and demonstrate the tremendous power and flexibility conveyed by MTP. As more applications are developed, it is expected that the interfaces above will change to accommodate the needs of developers. Thus it should be carefully noted that this description applies to version 2.1 of MTP and is subject to change.

VII. Testing and Simulation

7.1 Methods and Goals

MTP's design process has been closely tied to empirical testing and the iterative refinement of its operational parameters. Many of the design choices discussed in the previous section were made as a result of experiments comparing various other techniques and options. With the completion of revision 2.1, it has been possible to begin systematic testing for the purposes of validating the design choices and quantifying the performance gains over TCP. The primary goal of this testing has been to evaluate the throughput of MTP versus TCP under varying conditions of network congestion as measured by packet loss rates. A simplified computational model of both MTP and TCP has been created and used to simulate conditions that cannot be adequately tested *in situ* at this time. The modeling and simulation has also been useful in the further refinement of the protocol itself and in validating key performance assumptions. This chapter discusses the resources and methods used to perform the comparative testing and modeling of TCP and MTP. The following chapters discuss the results of the throughput test, observations of the impact of MTP on third-party traffic, and the results of the modeling and simulation. Finally, the results of the tests are analyzed and conclusions about the performance of MTP are made.

Validation of MTP as a high performance transaction protocol requires the examination of three key performance characteristics: throughput, network overhead, and its affect on third party traffic. Of these, throughput provides the most insight into the performance of flow control mechanisms and is the simplest to observe. Therefore the principle goal of the empirical testing has been to measure the rate at which large data quantities may be transferred using both MTP and TCP. Network overhead is difficult to observe directly, particularly over long path lengths. Therefore simulation testing has been used to obtain estimates of how much bandwidth is wasted by protocol overhead. Any protocol which claims high throughput rates must also be evaluated for any detrimental effects it may have on third-party traffic. For MTP, third party effects have been measured by observing transfer rates during multiple, mixed transactions.

A related performance characteristic which is important in many multimedia applications is latency, both during initial setup and error recovery. The setup latency problems with TCP are well known and various mechanisms (discussed in Chapter II) have been employed to limit their effects. By it's design, MTP's setup overhead is non-existant and thus it's setup latency is an optimal one round trip. The effects of error recovery latency for TCP are dramatic and have a severe impact on it's throughput. As discussed in previously, MTP makes only incremental adjustments to data flow, and thus does not have a concept of latency in error recovery. For these reasons, latency testing is not examined here except to note that MTP exhibts far better performance in latency critical applications than TCP. Instead, we concentrate on throughput as the primary measure of performance.

Empirical testing has been done using the network path between the University of Oklahoma and the California Institute of Technology. Workstations at both ends are connected to 10baseT subnets. The path length between nodes is approximately 18 hops with latencies varying between 80ms and 210ms. Datagram loss rates between nodes vary between 0 and 60% but are typically around 2 to 10%. This path is taken to be representative of the general case of Internet HTTP traffic.

Since the principal design goal of MTP is to provide optimal data transfer in spite of network congestion, both the empirical and simulation testing have been evaluated relative to ambient network path congestion. In the empirical case, congestion is measured by the rate of packet loss as observed by a continuous ICMP (Internet Control Message Protocol) fast ping performed throughout the course of the data transfer. For the simulation, a packet loss rate is set as a parameter of the simulation run. Observations of loss rate versus throughput have been correlated for both MTP and TCP and their comparison provides the principal means of quantifying the relative performance of the two protocols.

The primary development platform for MTP has been a Sparc Station 5/70 running SunOS 4.1.3_U1 which was used as the client during the testing described here. For the server side, the code was ported to IRIX 6. Other testing has been performed under SunOS 5, HP-UX, Linux, and OSF. In all cases, the platforms were connected via 10baseT ethernet to an IP subnet and the code was compiled using `gcc` version 2.7.

7.2 Comparative Throughput

For the purposes of testing, each HTTP transaction is modeled as an independent data transfer from the server node to the client node. For the MTP case, this simply consists of a request for a fixed sized data object. For the TCP case, each transaction consists of the establishment of a connection, the transfer of the data file, and the close of the connection. The time required for each data transfer is measured along with the ambient loss rate. This data is recorded in a log file and later correlated to obtain ambient loss versus throughput statistics.

Both protocols are serviced by the same client and server applications. Appendix C provides the source code for the server, `tserve`. The application asynchronously accepts both MTP and TCP transaction requests and returns a fixed quantity of data (1 megabyte by default). The TCP handling is non-blocking and allows up to eight simultaneous TCP streams. MTP handling is naturally asynchronous and no limit is placed on the number of transactions. Appendix C also contains the source code for the throughput testing client, `tclient`. This client performes a continuous series of transactions, alternating between MTP and TCP protocols. Before each transfer begins, a fast ICMP[1] ping is begun to measure ambient packet loss. (A fast ping generates a new ping message as soon as the previous one has been returned or one second has passed.) The ping is terminated at the end of the transaction and the results recorded along with the amount of data received, the time elapsed, and the calculated throughput. It is assumed that ICMP packet loss rates are at least proportional to those experienced by TCP since routers are indiscriminate when choosing datagrams to drop. Since MTP also provides statistics for its own observed round-trip-time and loss rates, these are also recorded. Transactions which last longer than sixty seconds are terminated and results recorded for the data received up to that point. In order to reduce the impact of testing on third-party users (particularly on the local networks at either end of the path), a ten second pause occurs between MTP and TCP tests while a sixty second pause occurs before returning to MTP.

[1] Internet Control Message Protocol is a specialized IP transport protocol used for this type of diagnostic function. Ping responses are usually generated at the kernel level without application intervention.

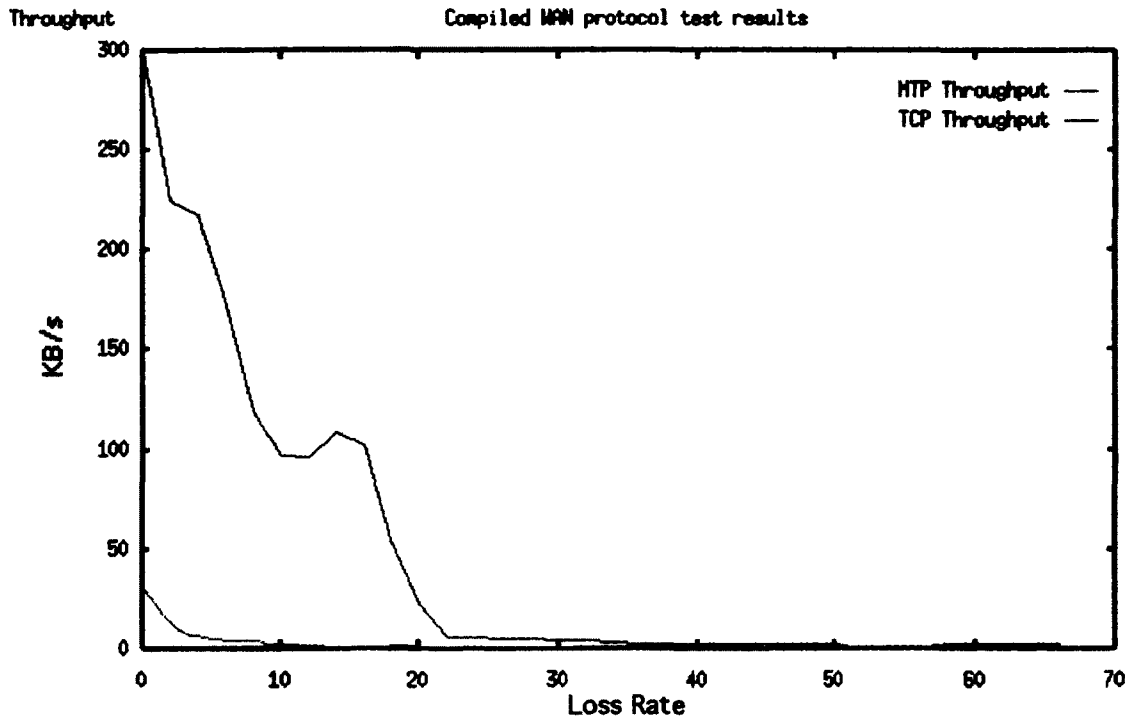


Figure 14a: Comparison of MTP and TCP throughput versus loss rate. Data has been sampled for every 2 percentage points of loss and a blur applied to fill in missing samples.

Since Internet network conditions are beyond this researcher's control, tests were run at various times of day for up to several hours in the hopes of recording a wide variety of congestion conditions. The observations were collated using the `tplot.c` program shown in Appendix C. This application collects the output of `tclient` and averages the throughput for each observed loss rate at 0.5% sample increments. Since only one long network path was tested and there was no ability to control the level of congestion, the data set lacks measurements for all possible loss rates. Therefore `tplot` performs a smoothing operation (roughly equivalent to a five point Gaussian blur) by averaging each point with its nearby neighbors using the weights 1, 2, 4, 2, and 1. Figure 14a shows the aggregate results of test trials using one megabyte data transactions between `cs.ou.edu` and `ugcs.caltech.edu`. Figure 14b shows a scatter plot of the actual data points including an exponential fit generated by Microsoft Excel version 5. Figure 14c shows the same scatter plot using a logrithmic scale on the throughput axis. The raw results are given in Appendix D.

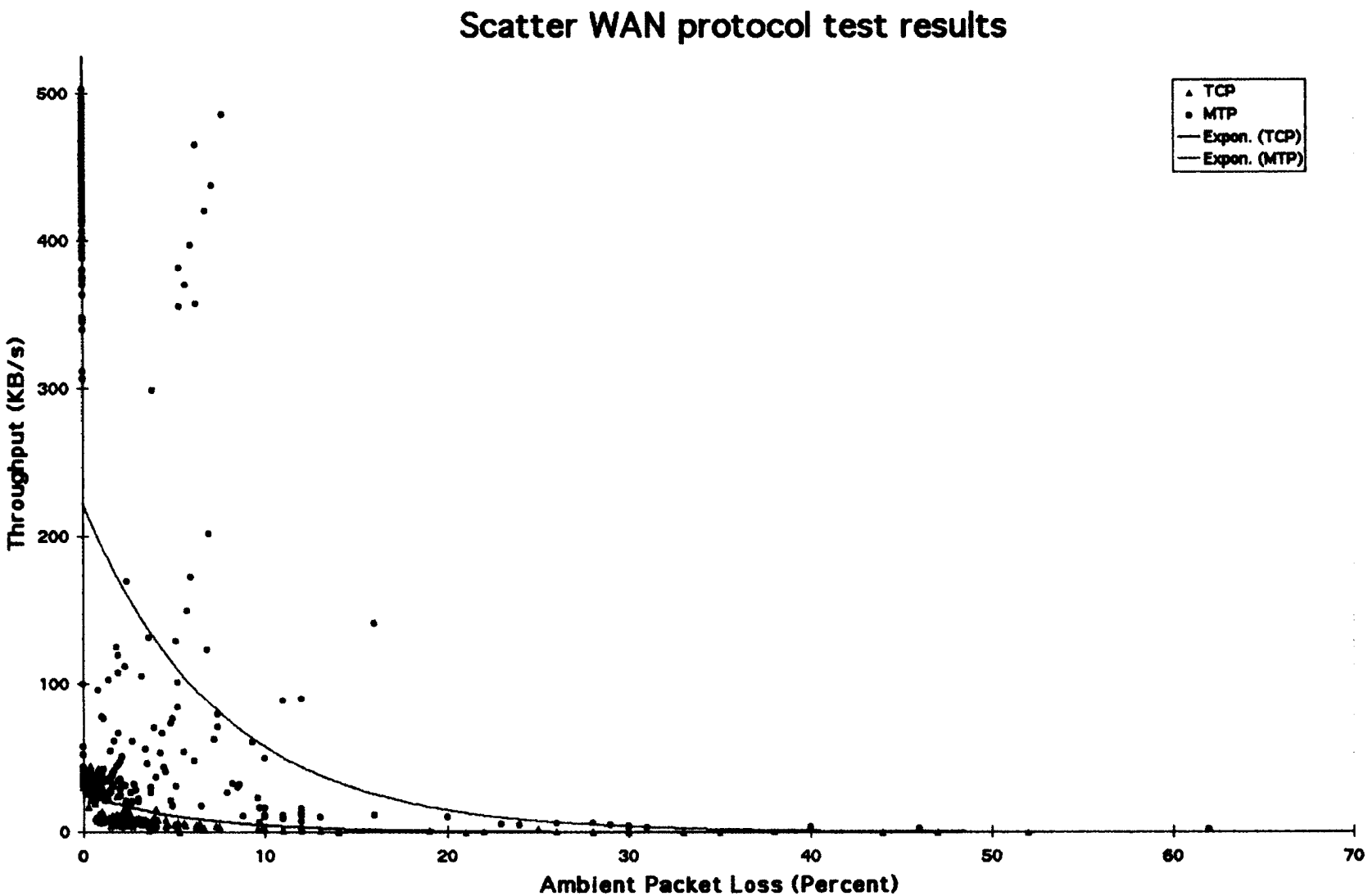


Figure 14b: Comparison of MTP and TCP throughput versus loss rate. Scatter plot with exponential fit by Microsoft Excel 5.0.

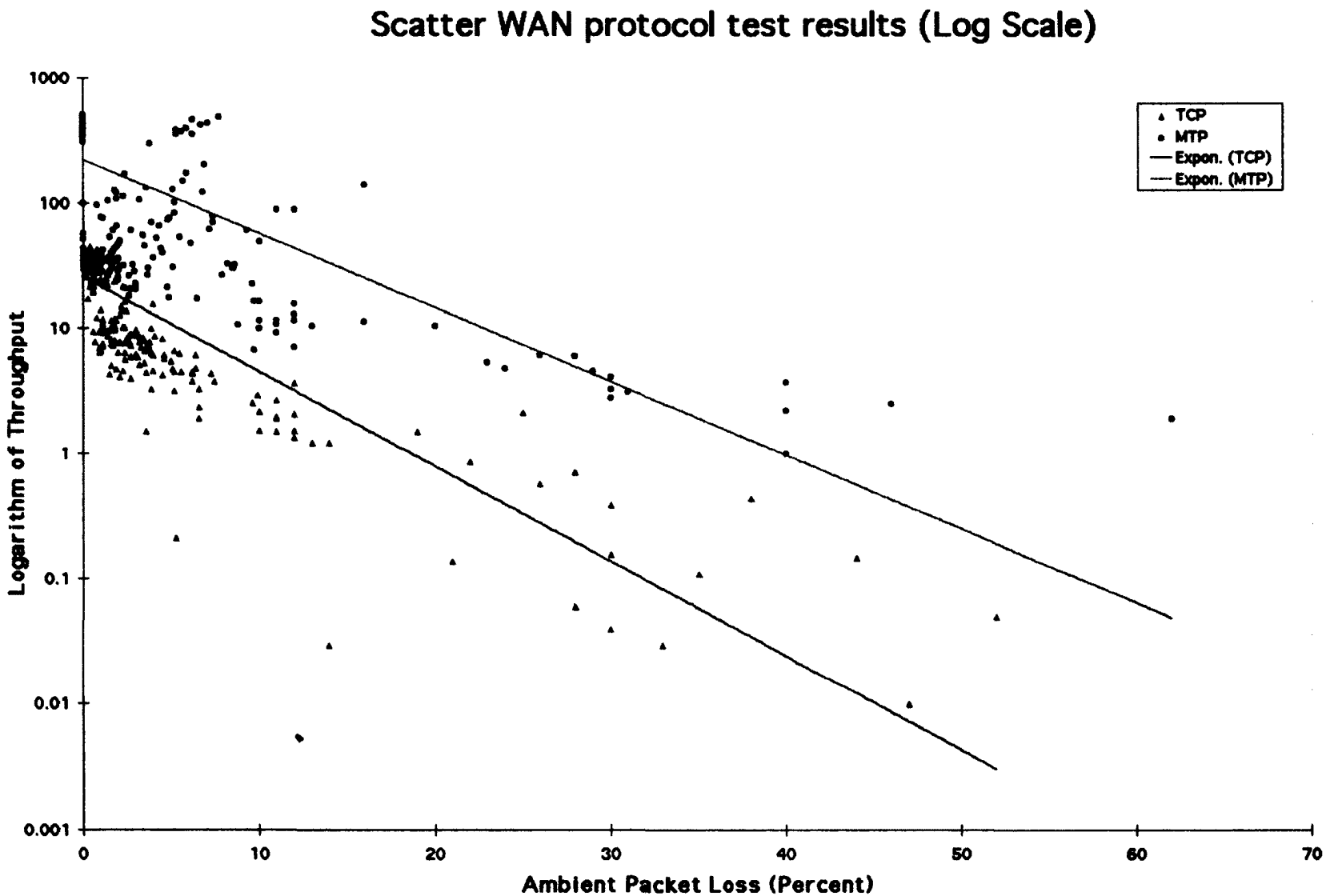


Figure 14c: Comparison of MTP and TCP throughput versus loss rate. Same as 14b but with logrithmic throughput scale.

The most obvious characteristic of these results is that MTP average throughput is greater than that of TCP at all observed loss rates. Under ideal conditions of zero ambient packet loss, MTP manages maximum throughputs as high 500 kilobytes per second while TCP is consistently below 45 kilobytes per second. As ambient loss increases, both protocols experience a degradation of throughput. In the case of TCP, throughput is reduced to less than 3 kilobytes per second (the speed of a 28.8 kbaud modem) for loss rates above 8%. MTP is able to consistently maintain at least this level of performance for losses up to 30%. MTP's observed throughput never drops below 1.9 kilobytes per second even up to 60% packet loss. TCP begins to fall below this level of performance around 10% packet loss. For all loss levels, MTP provides at least 10 times better throughput than TCP.

As can been seen from a careful examination of the data in Appendix D, there is a shortage of data points for loss rates above 10%. Both ends of the test path are located at well connected academic institutions with little local third-party traffic. Moreover, much of the network path between the two sites appears to be non-commercial. Ideally, we would like to test data transfer between numerous distant network nodes across purely commercial paths with the servers located topologically close to high traffic networks. Because of these limitations, it is difficult to exactly quantify MTP's performance gains. However it is clear that MTP does perform at least an order of magnitude better than TCP and provides consistently better throughput during times of elevated congestion and loss.

7.3 Third Party Effects

With such tremendous performance gains, the question of third-party performance degradation must be explored. As discussed in previous sections, an overly aggressive protocol will flood routers and will itself experience substantial data loss in addition to forcing losses in third-party traffic. Since MTP makes its own loss statistics available to its parent application, we can see from the data in Appendix D (under Raw Data, column `Oloss`) that MTP's observed loss rates are comparable to the ambient rates simultaneously observed by the `ping` program (column `Ploss`). If MTP were achieving its increased throughput by flooding the network, it's own loss rates would be dramatically higher than the ambient level. However, even without flooding, it is reasonable to expect that MTP's

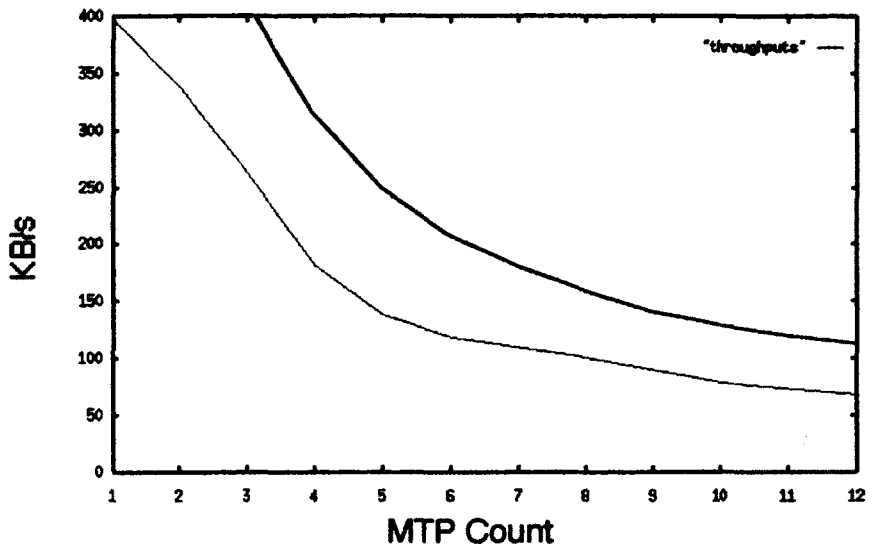


Figure 15: MTP throughput per transaction as a function of increasing self competition. The level of ambient traffic along the path could not be measured, but ICMP loss was observed to be zero indicating little or no congestion. The grey line shows the theoretical maximum throughput per MTP transaction assuming a no third-party traffic and a 1250 KB/s maximum path capacity.

gains might come at the expense of third-party TCP traffic by forcing the TCP traffic to throttle back prematurely.

In order to test MTP's effect on third party traffic, a client application capable of performing multiple simultaneous transactions was created. `mclient` is shown in Appendix E. This application closely resembles `tclient` except that it creates and maintains a user specified number of parallel MTP and TCP transactions. The program outputs statistics for each transaction as it completes, plus the results of a single fast ping test. Because MTP transactions always complete before their TCP counterparts, each MTP transaction is restarted upon completion until all TCP transactions have completed. Tests were run using multiple MTP and TCP transactions during periods of no detectable ambient packet loss over the network path previously described. (The tests were conducted several months after the throughput tests. During that time, the network path changed slightly resulting in a slightly higher capacity.)

Figure 15 examines MTP's throughput behavior across the same network path when it is in competition with itself and whatever ambient traffic was present on the WAN. This graph

TCP Count \ MTP Count	0	1	2	3	4	8
0	-	*397.1*	*338.0*	*262.1*	*182.1*	*101.3*
1	45.5	41.7	37.9	32.5	32.2	33.9
2	44.3	42.0	35.7	30.9	30.2	33.8
3	44.9	41.7	35.2	30.7	33.5	31.0
4	45.1	40.4	34.8	30.2	30.9	33.0

Figure 16: TCP throughput in KB/s during MTP traffic across a lightly utilized WAN. The first row (in italics) shows the MTP throughput when competing with itself.

shows the throughput per transaction as a function of the number of simultaneous transactions. As expected, the throughput of MTP transactions diminish as the number of transactions increases, essentially dividing the available bandwidth amongst themselves. Note that this reduction in per-transaction throughput holds for the entire range of tested multiplicity.

Figure 16 shows how multiple TCP transactions are affected by multiple simultaneous MTP transactions. It should first be noted that TCP's throughput without MTP (column 0) does not change significantly as the number of TCP test transactions is increased, at least up to the small number tested here. This is not entirely surprising given that even a lightly utilized WAN path likely contains dozens if not hundreds of simultaneous TCP streams and thus the addition of a few more is significant only in that these are crossing exactly the same path. However it does confirm that TCP and MTP have very different notions as to what constitutes available bandwidth. This raises the important question as to where MTP is finding it's extra capacity.

The remaining columns of Figure 16 show how TCP reacts to the presence of MTP. (The base throughputs of MTP are shown in the first row in italics.) It is clear that the addition of MTP transactions does have some detrimental effect on the throughput of TCP. However, the effects are relatively minor at 8 simultaneous MTP connections and appear to level off around a 30% reduction from the peak rate of 45 KB/s for a single TCP connection. Not shown here, though implicit in the testing conditions, is that these tests did *not* result in any measurable ambient packet loss. That is, while individual MTP

transactions were observed to experience loss as a normal part of their flow control (and it can be inferred that TCP did as well), a simultaneous flood ping (one packet per rtt) measured no loss.

Several conclusions can be drawn from these results. MTP's ability to compete with itself and TCP without causing measureable ambient loss shows that it's flow control mechanisms are able to prevent network flooding and thus efficiently utilize the available link level capacity. The reduction in TCP throughput shows that MTP's gains come at some expense to the bandwidth available to TCP. However the fact that the impact on TCP is only a 30% reduction at 8 competing MTP streams shows that MTP is exhibiting good-neighbor behavior by limiting it's consumption of link bandwidth. Thus MTP has achieved it's design goal of efficiently utilizing available bandwidth and has at least partially satisfied the goal of being a good transport neighbor.

What cannot be determined from these tests is exactly how much raw bandwidth is available along the network path and what proportion of it is being utilized by true third-party traffic. The zero ambient packet loss suggests that there is very little third party traffic, but the negligible reduction in TCP throughput as new TCP streams are introduced suggests that there may be many TCP streams. This apparent contradiction may be explained by the inability of any individual TCP to fully utilize unused bandwidth. This supports other observations, discussed earlier, that TCP tends to under-utilize links. It also supports the notion that some portion of MTP's throughput gains are coming not from squelching TCP but from utilization of unused bandwidth. To definitevly answer these questions requires that testing be conducted under much more controlled circumstances. It is hoped that as interest in MTP grows, resources for such testing will become available. If such testing were to show that MTP were having too large an impact on existing TCP traffic, MTP could be adjusted such that it leaves a larger window of opportunity for other transport traffic. This could be easily accomplished by having MTP reduce it's window size by more than one datagram when encountering packet loss.

7.4 Modeling and Simulation

While the results above offer solid evidence of MTP's performance gains, simulation of a simplified model allows us to test the core principals of its design in a completely controlled environment. By validating the model under empirically verifiable conditions, we can extend the analysis to explore performance characteristics under conditions which have not yet been tested. This chapter describes the simplified computational model used to simulate MTP and TCP as well as the results of simulation trials under various circumstances. The simulator source code is provided in Appendix F. The simulation model is still being tuned and the results presented hear should be considered preliminary.

Model

In order to simplify the feedback effects between the network and protocol algorithms, the network is modeled as a black box with a single interface. A single node may insert a message and at some future time the network will either return another message or timeout. In the case of MTP, the client is modeled as sending requests and receiving responses. For TCP, the server is the modeled node and it exchanges data for acknowledgements. The network has five characteristics:

- loss rate - The fraction of messages which will be arbitrarily lost.
- bandwidth - The maximum rate in KB/sec at which messages may be delivered. Messages transmitted in excess of the network capacity will also be lost.
- round trip time - The minimum time between message insertion and response receipt.
- congestion factor - The fraction of total bandwidth which is not being consumed by third-party traffic.
- buffer padding - A measure of the effect of node buffer sizes on the data capacity of the network.

Messages are the interaction points between the network and the modeled node. The fate of each message is determined at the time it is created by the node. Depending on the loss

rate, a random number generator, and the protocol's timeouts, the message will either arrive back as a response/acknowledgement, timeout due to a drop, arrive after a timeout, or it will be ignored by the protocol. The time at which this event will occur is calculated based upon the state of the network, including the states of other messages.

The behavior of the modeled node depends on which protocol is being simulated. Both protocols are characterized by a retransmission interval, a transaction expiration/abort timer, an initial window size, and a fixed buffer padding, although these parameters are used differently for each. The algorithms for the protocols have been simplified to interact with the limited information available from the black box network. For example, kernel buffer levels, flow control messages, routing messages, connection times, datagram sizes, and other factors are not considered. Both protocols begin in a steady state with a set number of messages already in transit and presumed to arrive correctly. As each preset message completes its round trip, the protocol reacts by generating new messages and changing its state. The state engine for the MTP simulation is derived directly from the MTP 2 implementation. The TCP code is derived from a combination of RFC 793, the BSD 4.4 source code, and the Linux 2 source code. In addition to the usual protocol behaviors, the network applies the congestion factor a second time to TCP to simulate the effects of synchronicity.

Execution

During the course of the simulation, events are logged in variable levels of detail and statistics are calculated for the end report. The simulation continues for either a given period of time or until a given number of messages have successfully transited. Ultimately, the goal of the simulation is to produce an estimate of the relationship between the loss rate and the throughput of the protocol. The simulation was repeated for a range of loss rates to produce a table of values suitable for plotting loss rate versus throughput.

The simulation is executed by specifying a protocol (`-m` for MTP or `-t` for TCP), a loss rate (`-l` with 1.0 equal to 100%), a bandwidth (`-b` in KB/sec), a round trip time(`-r` in ms), and a congestion factor (`-c` described below). If set to graphing mode, the loss rate

parameter sets the high end of the range to be simulated. By default, a bandwidth of 1250 kilobytes per second, equivalent to a 10baseT network is used. The minimum round trip time, congestion factor, and the buffer padding then determine characteristics of the network path. The following examples show execution for a transferring a 1 megabyte file across a LAN with no ambient loss:

```
# sim -t -l 0.0 -c 1 -r 1 -p 5120
Elapsed Time: 4.1s
Requested 5184 messages:        Received 5120   Lost 0  Late 0
Efficiency: 100%        Throughput: 1249.7      Utilization: 100.0%
Delay: 10/441/512       Max Window: 64

# sim -m -l 0.0 -c 1 -r 1 -p 5120
Elapsed Time: 4.1s
Requested 5133 messages:        Received 5120   Lost 0  Late 0
Efficiency: 100%        Throughput: 1249.7      Utilization: 100.0%
Delay: 10/102/104       Max Window: 13
```

The results give the time of the transaction, the number of messages sent, the number received, the number dropped due to ambient loss, and the number arriving after a timeout. The `efficiency` value expresses the percentage of messages generated which successfully transited the network. `Throughput` is the number of 1 kilobyte messages received per second. `Utilization` divides observed throughput by the theoretical maximum for the given ambient loss rate (which in this case is zero). The `Delay` values show the minimum, average, and maximum round trip times in tenths of milliseconds. `Max window` is the maximum number of 1 kilobyte messages which were in transit at any one time. In order to isolate throughput performance, these numbers do not reflect startup or tear down costs and the simulation is terminated with up to one full window still in transit. This eliminates the effects of connection time and initial ramp up and is equivalent to performing empirical measurements over long (greater than ten second) periods.

In this case, with a quiet network containing no routers, both protocols reach their maximum window sizes before saturating the network. MTP halts its window expansion early when the round trip time exceeds 10 times the minimum observed, which could indicate that buffers are filling up. This significantly reduces the message latency without reducing throughput. Given more capacity, MTP's maximum window size is dependent on the UDP kernel buffer sizes which may range from 48 to 248 kilobytes depending on

the platform. These results can be compared to empirical tests performed using the `tserve` and `tclient` programs discussed previously:

```
# MTP vs. TCP network test
# ritchie  Thu May 27 12:32:01 1999
#
#MTP                                                    TCP
#Recv    Time     Thput    RTT      Oloss    Ploss     Recv     Time     Thput
Ploss
5242880 4.8      1069.8   43       0.0      0.0       5242880  5        1024.00
0.0
```

A more interesting case involves a congested, two router path with 2ms latency, and approximately 3% ambient loss. The simulated and actual results are shown below:

```
# sim -t -l 0.03 -c 5 -r 2 -p 5120
Elapsed Time: 186.1s
Requested 6460 messages:        Received 5120   Lost 165        Late 0
Efficiency:  96%        Throughput: 27.5        Utilization: 11.3%
Delay: 20/252/480       Max Window: 13

# sim -m -l 0.03 -c 5 -r 2 -p 5120
Elapsed Time: 21.1s
Requested 5126 messages:        Received 5120   Lost 148        Late 0
Efficiency:  97%        Throughput: 243.0       Utilization: 100.2%
Delay: 20/225/1240      Max Window: 6

# MTP vs. TCP network test
# michelle  Thu May 27 12:53:02 1999
#
#MTP                                                    TCP
#Recv    Time     Thput    RTT      Oloss    Ploss     Recv     Time     Thput
Ploss
1048576 4.2      242.6    7        12.8     2.8       1048576  57       17.96
2.7
```

Of course, the case we are most interested in is the wide area network path. Here we have congested routers, a highly variable loss rate, and a much larger round trip time. Because this path is much more volatile than the local ones, we must refer to our aggregate test results in Appendix D, rather than relying on any one test. Note that we choose a round trip time of 80ms because it was the minimum observed for the route.

```
Loss     MTP              TCP
Perc.    Rate     Count   Rate     Count
-------------------------------------
8        119      11      2.973    2
10       96.96    10      2.132    7
12       96.24    10      1.585    4
```

```
# sim -t -l 0.10 -c 20 -r 80 -p 1024
Elapsed Time: 576.5s
Requested 2179 messages:        Received 1024   Lost 432        Late 0
Efficiency:  70%        Throughput: 1.8 Utilization: 3.2%
Delay: 800/1006/1120    Max Window: 5

# sim -m -l 0.10 -c 20 -r 80 -p 1024
Elapsed Time: 31.3s
Requested 1030 messages:        Received 1024   Lost 114        Late 0
Efficiency:  89%        Throughput: 32.7        Utilization: 58.2%
Delay: 800/1407/8800    Max Window: 7
```

In this case, the simulation underestimates MTP's performance by approximately a factor of three while coming very close to TCP's performance. Figure 17 shows the full results of the simulation and may be qualitatively compared to the empirical results in Figure 14. Both the simulation and empirical results show an exponential decay in performance as loss increases with MTP out performing TCP by an order of magnitude. Note that the simulation shows a more rapid early decay for MTP when compared with the empirical curve. This is likely due to the fact that the simulation assumes a fixed amount of available bandwidth and does not attempt to model MTP's effect on that bandwidth. The TCP

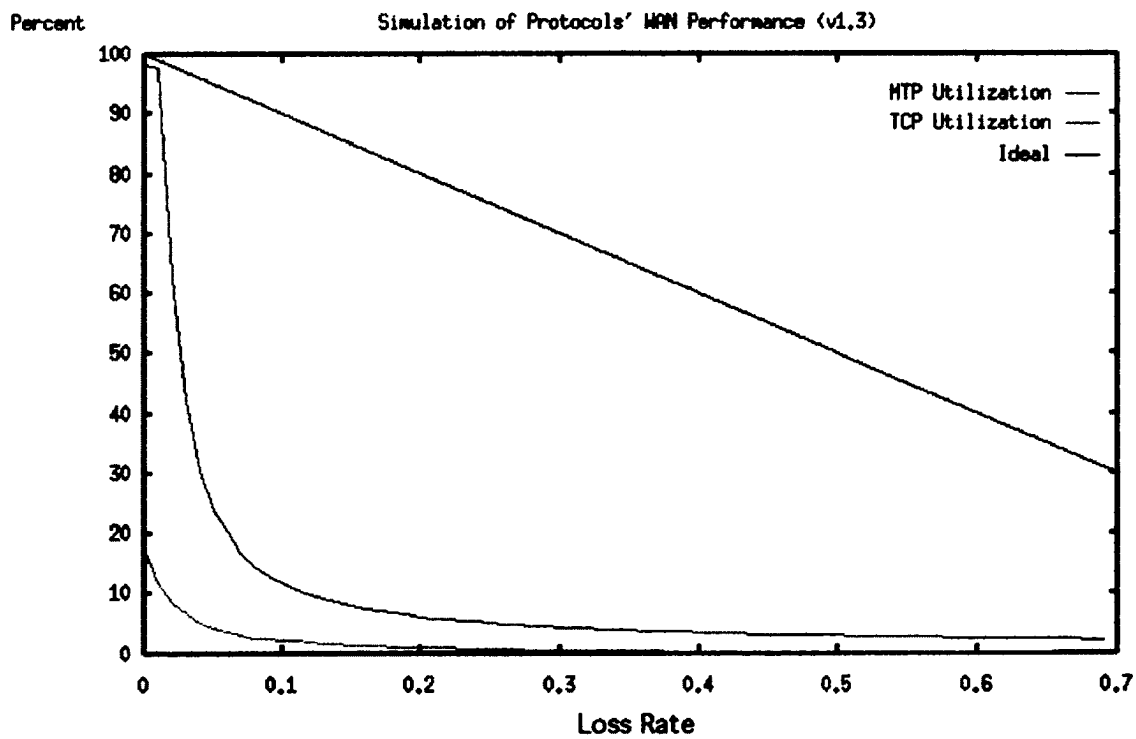


Figure 17: Simulation of MTP and TCP bandwidth utilization versus loss rate.

simulation curve more closely correlates to the TCP empirical curve, most likely because the model does account for TCP's synchronicity effect.

7.5 Analysis of Results

The empirical testing backed by the simulation models clearly demonstrates that MTP provides superior throughput over TCP, especially during periods of high congestion and packet loss. However quantifying the results is difficult because of the limited resources available for testing. The empirical tests themselves have been limited to a single long network path. While this path is believed to be representative of most such paths, the test results show a tremendous amount of variability. This suggests several flaws in the quantitative data. First, it is clear that packet loss alone is not an adequate measurement of path congestion. A careful examination of the results reveals that there is a significant amount of variation in the round trip time which does not correlate with loss rates. More careful study is needed to understand how loss rates and round trip times relate to third-party traffic and available bandwidth.

The most exhaustive approach would be to test MTP in a completely controlled environment in which the third-party traffic is known and the effects upon the observable parameters can be assessed. However, such testing would not only be extremely resource intensive, but it would require a great deal of study to ensure that the test conditions accurately reflect the behaviors experienced on the Internet at large. Thus the best solution might be to deploy MTP at numerous topologically dispersed locations across the Internet. Ideally, this would be combined with measurements of traffic flow and congestion levels at key routers along the paths to gain an accurate understanding of exactly how MTP and TCP respond to a wide variety of real world circumstances.

While the empirical results are insufficient to provide a precise quantitative comparison, the fact that they can be qualitatively verified via simulation using a relatively simple network model is very reassuring. The model was deliberately designed not to attempt simulation of router level decision making precisely because there was no way to empirically verify the correctness of its behavior. Instead, the model uses as its inputs only those variables which are known or observable. The one exception is the "congestion factor," which may

be inferred but cannot be observed given current resources. This measure of third-party traffic has a significant effect on the throughput of both protocols, primarily by adding to the round trip time which they observe (as opposed to the minimum rtt which is given as a parameter). Variation of this parameter has a second order effect on the performance of both protocols, which further verifies that the performance differences are intrinsic to the algorithms and not artifacts. For example, below are otherwise identical simulation with the congestion factor set at 10 and 1. As might be expect, it has a larger influence on TCP, but the performance difference between the two protocols remains an order of magnitude.

```
# sim -m -l 0.03 -c 10 -r 80 -p 1024
Elapsed Time: 6.0s
Requested 1082 messages:        Received 1024   Lost 26 Late 0
Efficiency:  97%        Throughput: 171.1       Utilization: 141.2%
Delay: 800/881/896      Max Window: 16

# sim -t -l 0.03 -c 10 -r 80 -p 1024
Elapsed Time: 101.9s
Requested 1461 messages:        Received 1024   Lost 75 Late 0
Efficiency:  93%        Throughput: 10.0        Utilization: 8.3%
Delay: 800/811/824      Max Window: 7

# sim -m -l 0.03 -c 1 -r 80 -p 1024
Elapsed Time: 5.4s
Requested 1078 messages:        Received 1024   Lost 23 Late 0
Efficiency:  97%        Throughput: 188.8       Utilization: 15.6%
Delay: 800/809/824      Max Window: 16

# sim -t -l 0.03 -c 1 -r 80 -p 1024
Elapsed Time: 52.7s
Requested 1305 messages:        Received 1024   Lost 35 Late 0
Efficiency:  96%        Throughput: 19.4        Utilization: 1.6%
Delay: 800/811/840      Max Window: 16
```

In spite of the less than ideal test conditions, the empirical and simulation results leave no question that MTP has achieved its design goals with respect to network performance. Additional testing is necessary to provide a more precise quantization of this performance gain, but it is clearly on the order of a ten times improvement for all but the most ideal network conditions. One important question which remains unanswered is what performance gains MTP will offer at the application level. For those deployments where network bandwidth is the limiting factor, MTP will certainly offer a dramatic improvement. But to assess its effects on CPU limited applications, further work must be done to develop an HTTP/MTP server with equivalent functionality to existing HTTP/TCP servers.

Preliminary work in this area suggests that there are significant performance advantages, but is too early to provide a firm estimate. Likewise, MTP's success in terms of ease-of-use of its interface remains to be tested. It is hoped that with the distribution of an MTP Software Development Kit, extensive feedback can be obtained regarding the interface as well as other aspects of the protocol's performance.

VIII. Conclusions and Proposals

There is no doubt that the nature of data networks and their use has changed dramatically in the time since TCP was envisioned. But in spite of these tremendous changes, the Internet transport layer in general, and TCP in particular, have been almost completely overlooked in the continuing efforts for improved Internet performance. It is evident from an examination of both TCP's design and its behavior that a new network paradigm is needed to fulfill today's networking needs. It is this author's belief that the client-pull model, along with the other innovations represented by the Multimedia Transport Protocol, provides this new paradigm and that its further exploration and development are imperative to the expansion of the Internet.

The initial testing and simulation of MTP support this belief and suggest that the gains to be had are very substantial. Even with no ambient packet loss, TCP's performance drops to a small fraction of what should be the available bandwidth. MTP demonstrates the vast underutilized potential of current network infrastructure by providing throughput an order of magnitude grater than TCP. Most importantly, MTP has been optimized for modern applications and usage patterns. Not only does this allow MTP to outperform TCP with customized testing applications, but it allows these gains to be quickly realized in current application environments.

Much more work needs to be done to develop and deploy the Multimedia Transaction Protocol. The research done to date demonstrates its advantages and the ease with which it may be integrated into the existing network infrastructure. But the implementation still requires fine tuning and testing under more rigorous circumstances. Additional applications, beyond simple file transfer, must be explored to determine how MTP may interact with and benefit a broader range of Internet functions. Some of this work has been begun as part of a the development of MTP. A document server capable of handling both FTP and HTTP/1.0 style transactions has been developed and tested. Given MTP's tremendous performance advantages over traditional applications, it has been a valuable tool in transferring source code across the network for porting and testing. To demonstrate

the ease with which MTP can be incorporated underneath existing applications, a protocol gateway has also been developed. This gateway accepts HTTP/TCP proxy connections and translates the transactions it receives into HTTP/MTP requests which are forwarded to any remote HTTP/MTP server. This allows a standard TCP browser to be used to view web pages via MTP, with all of the attendant performance gains, without any modification to the browser beyond setting its proxy preferences. MTP's development heritage carries with it other high performance applications. The protocol began its development life as an IP telephone protocol. The fully functional telephoning application provides real-time, full-duplex, telephone quality audio even under conditions of up to 50% packet loss.

In addition to further developing these applications, efforts must be made to integrate MTP into existing programs. This would allow MTP to be tested in widely used applications under much more rigorous and varied circumstances. Wide spread development and use of MTP amongst researchers must be encouraged in order to generate a high level of interest and feedback. Given strong, real-world evidence of its performance advantages and the development of an easy to adopt implementation, commercial software developers will have strong incentives utilize MTP and bring its advantages to the consumer level. It is the hope of this researcher that such testing and development will allow MTP become a new standard for Internet transactions and greatly improve the way the Internet works.

IX. MTP Glossary

application A program, or specification for a program, which uses network communications to perform some task.

buffer Memory in a network node used to store data while it is awaiting transmission. See "congestion".

client A network node or application which generates a request for data that is stored on a server.

congestion A condition where buffers along the network path are increasing in size due to inflow exceeding outflow.

data object A collection of data maintained in some form by a server which may be requested by a client.

datagram limit The largest permissible datagram size. The actual limit is the minimum of the limits set by the client, the server, the network architecture, and the inherent limitations of the implementation.

dynamic A data object which is generated in real-time. For example: the output of a program. Handling of dynamic data objects was not yet fully implemented for MTP version 2.1.

full-duplex Data travels along a path in both directions simultaneously.

flow The rate of data traveling past a given point as measured over a time span significantly greater than one round trip time. Units are kilobytes per second.

flowtimeout Value which controls when it is permissible for MTP to adjust a transaction's window size. A positive value prevents any adjustment. The value is reduced by the number of bytes received or declared lost. If a loss is detected while this value is zero or less, the window is reduced and flowtimeout reset to a postivie multiple of the window value. If it reaches a negative Threshold value (indicating no loss in some time), the window size may be raised and the value reset to zero. See Chapter V, "Window Adjustment".

half-duplex Data travels along a path in only one direction at a time.

loss The failure of data which is intended for transmission by an application to reach its destination. Typically measured as a percent of total data transmitted.

message At the transport protocol level, a message is a single network datagram which contains an MTP header and possibly a data payload. At the application level, a message is may be any unit of communication

between applications or between the application and the MTP transport layer.

multidatagram response transaction
A type of transaction in which the response may consist of more data than can be transmitted in a single datagram. (See also "single datagram response transaction".) The client MTP will regulate the flow of data to achieve maximum throughput with minimum loss. As of MTP version 2.1, the response is limited to 2^32 bytes.

network path The sequence of nodes and links over which data travels between two communicating nodes.

node Any uniquely identified device capable of communicating on a network.

protocol The specification and/or implementation of a mechanism for communicating.

request An action taken by an application (the client) seeking to obtain data from another application (the server).

request message
A datagram generated by the client MTP which specifies a portion of a request transaction.

response The data returned by a server in response to a client's request.

response message
A datagram generated by the server MTP which carries a portion of a server's response to a client's request.

round trip time The amount of time required for a datagram to travel from one network node to another and back again. For MTP it is the time between transmission of a request and receipt of a response. If a single request datagram results in multiple response datagrams, then the RTT is measured for each response against the time of the original request datagram.

router A network node which forwards data from one link to another.

semi-infinite A data object which is unbounded. See also "stream" and "dynamic".

server A network node or application which receives requests for data and which sends back responses.

single datagram response transaction
A type of transaction in which the application protocol guarantees that the response will fit into a single network datagram. When this option is specified in the request, the client MTP will ignore flow control and accept the first valid response datagram that it receives as the entire

response.

static A data object whose size and composition is known and will not change during the life of a transaction. For example: a fixed data file.

stream An MTP transaction which permits the transfer of a semi-infinite dynamic data object or an object which exceeds 2^32 bytes. This mode was not yet fully implemented for MTP version 2.1.

subrequest An MTP request datagram sent to retrieve those portions of a requested data object which are not yet in transit or have been declared lost. At the server side, a subrequest is indistinguishable from an initial request message.

telephony The real-time, full-duplex exchange of telephone quality audio over a network path.

threshold The amount of data which must be succesfully received in a transaction before the window size is permitted to increase. See Chapter V, "Window Adjustment".

throughput The rate at which data is successfully received, typically measured over the life the transaction in kilobytes per second.

transaction An exchange of network data in which a client sends a request to a server and the server returns a response.

transit The amount of data "in transit" is the number of bytes which have been transmitted by the server but not yet received or declared lost by the client.

window size The maximum amount of data which the client MTP desires to be in transit for the purposes of flow control.

window limit The maximum amount of data that the server will transmit in response to an individual request datagram. The actual limit is the minimum of the values requested by the client and the server.

X. Bibliography

[1] Martin F. Arlitt, and Carey L. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications," IEEE Transactions on Networking, Vol. 5, No. 5, October 1997.

[2] Dimitri Bertsekas, Robert Gallager, "Data Networks: Second Edition", Prentice-Hall, 1992.

[3] J-C. Bolot, "End-to-end Packet Delay and Loss Behavior in the Internet", In Proc. ACM SIGCOMM93, pp.289--298, Sept.1993.

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

[5] E. G. Britton, J. Tavs, and R. Bournas, "TCP/IP: The next generation," IBM Systems Journal, Vol. 34, No. 3, pp 452-471, 1995.

[6] V. Cerf, and R. Kahn, "A Protocol for Packet Network Intercommunication", IEEE Transactions on Communications, Vol. COM-22, No. 5, pp 637-648, May 1974.

[7] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead", IEEE Communications, Vol. 27, No. 6, pp 23-29, June 1989.

[8] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in SIGCOMM Symposium on Communications Architectures and Protocols, (Philadelphia, Pennsylvania), pp. 200-208, IEEE Computer Communications Review, Vol. 20 (4), Sept. 1990.

[9] Adrian Cockcroft, "Performance Q&A" SunWorld Online, December 1996, <http://www.sun.com/sunworldonline/swol-12-1996/swol-12-perf.html>, Viewed November 1997.

[10] S. Crocker, "Host Software", RFC 001, April 7, 1969.

[11] S.Floyd and V.Jacobson, "On traffic phase effects in packet-switched gateways", Internetworking:Research and Experience, vol.3, pp.115--156, Sept.1992.

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

[13] George Gopen and JudithSwan, "The Science of Scientific Writing", American Scientist, Volume 78, pp 550-558.

[14] John Heidemann, Katia Obraczka, and Joe Touch, "Modeling the Performance of HTTP Over Several Transport Protocols," IEEE/ACM Transactions on Networking, Vol. 5, No. 5, October 1997.

[15] V. Jacobson, "Congestion avoidance and control", *Proceedings of Symposium on Communication Architectures and protocols*, ACM SIGCOMM, August 1988.

[16] Brian Kernighan, Dennis Ritchie, "The C Programming Language: Second Edition", Prentice-Hall, 1988.

[17] L. Kleinrock, "Information Flow in Large Communication Nets", RLE Quarterly Progress Report, July 1961.

[18] B. Leiner, V. Cerf, D. Clark, R. Kahn, L. Kleinrock, D. Lynch, J. Postel, L. Roberts, S. Wolff, "The Past and Future History of the INTERNET", Communications of the ACM, Vol. 40, No. 2, pp 102-108, February 1997.

[19] B. M.Leiner, .V. G. Cerf, D. D. Clark, R. E. Kahn, L Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, S. Wolff , "A Brief History of the Internet, version 3.1" <http://info.isoc.org/internet-history/>, Internet Society, Viewed November 1997.

[20] Mamoru Maekawa, Arthur Oldehoeft, Rodney Oldehoeft, "Operating Systems: Advanced Concepts", Benjamin/Cummings, 1987.

[21] Greg Miller and Kevin Thompson, "The nature of the beast: recent traffic measurements from an Internet backbone" Cooperative Association for Internet Data Analysis (CAIDA), April 1998, <http://www.caida.org/Papers/Inet98/index.html>, Viewed April 1999.

[22] J. Postel (ed.), "User Datagram Protocol" RFC 768, August 1980.

[23] J. Postel (ed.), "Transmission Control Protocol" RFC 793. September 1981.

[24] J. Postel, and J. Reynolds, "Telnet Protocol Specification", RFC 854. May 1983.

[25] H. Nielsen, "MUX Overview" The World Wide Web Consortium, January 1999, <http://www.w3.org/Protocols/MUX/>, Viewed April 1999.

[26] H. Neilsen and J. Gettys, "HTTP- Hypertext Transport Protocol" The World Wide Web Consortium, March 19 1999, <http://www.w3.org/Protocols/>, Viewed April 8 1999.

[27] H. Sawashima, et al. "Characteristics of UDP Packet Loss: Effect of TCP Traffic" 1997, <http://www.niit.pref.nara.jp/~hidena-s/inet.html>, Viewed April 1999.

[28] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 1889, Network Working Group , January 1996.

[29] Mischa Schwartz, "Telecommunication Networks: Protocols, Modeling and Analysis", Addison-Wesley, 1987.

[30] Abraham Silberschatz and Peter Galvin, "Operating System Concepts: Fourth Edition", Addison-Wesley, 1994.

[31] Jean Walrand and Pravin Varaiya, "High-Performance Communication Networks, Second Edition", Morgan Kaufmann Publishers, 1999.

[32] L.Zhang and D.Clark, "Oscillating Behavior of Network Traffic: A Case Study Simulation", Internetworking: Research and Experience, Vol.1,pp.101-112,1990.

Appendix A: mtp.h

```
/*
**  mtp.h - Public headers for libmtp
**
**  Version 2.1  -  Multimedia Transport Protocol  -  Seth Noble
**
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#if defined(irix)
#include <strings.h>
#endif

/*  Storage types
**
**  Make sure we know exactly how big everything is.
*/
typedef signed char        sint8;
typedef signed short       sint16;
typedef signed long        sint32;
typedef signed long long   sint64;
typedef unsigned char      uint8;
typedef unsigned short     uint16;
typedef unsigned long      uint32;
typedef unsigned long long uint64;

extern const uint8 MTP_VERSION;

#define MTP_SIZE_REQUEST (1024)     /* Largest allowable request payload
*/
#define MTP_SIZE_RESPONSE (16*1024) /* Largest response payload per
datagram */
#define MTP_SIZE_DEFAULT (1024)     /* Default response payload per
datagram */
#define MTP_SIZE_MINIMUM (512)      /* Minimum response payload per
datagram */

/*
**  Protocol header values
**
**  These codes define the nature and action of all nwtp messages.
*/

/* Error Codes - Denotes an error and where it originated */
#define ERR_NONE        0       /* No error */
#define ERR_CLIENT      1       /* From Client MTP */
#define ERR_NET         2       /* From Network */
#define ERR_SERVER      3       /* From Server MTP */
#define ERR_APP         4       /* From Server Application */
#define ERR_ROUTE       5       /* Message delivered to the wrong system
*/
#define ERR_count       6
/* Note that LOCAL, NET, and SERVER _only_ denote delivery problems.
   Errors in fulfilling the request itself should always be DEVICE. */
```

```
/* Response Codes -  Comment on the nature of a response. */
#define RES_REQUEST     0       /* This is a request */
#define RES_REPLY       1       /* Reply, no comment */
#define RES_APP         64

/*  There are two classes of other responses:
**    Error -  Specify the nature of an Error.
**    Reply -  Comment on a successful response based on Type of server.
*/

/* Error Daemon (ERR_CLIENT and ERR_SERVER) - Message could not be
delivered */
#define ERR_DMN_SYS     2       /* Error in system call */
#define ERR_DMN_ACTION  3       /* Invalid Action */
#define ERR_DMN_DEL     4       /* Unknown delivery */
#define ERR_DMN_RSPSZ   5       /* Unable to deliver due to size */
#define ERR_DMN_VERSION 6       /* Unsupported version or feature */
#define ERR_DMN_NORSRC  7       /* Insufficient resources for request */
#define ERR_DMN_REQSZ   8       /* Request exceeds maximum size */
#define ERR_DMN_PROTO   9       /* Protocol violation */
#define ERR_DMN_CHECK   10      /* Checksum failed */
#define ERR_DMN_count   11

/* Error Net */
#define ERR_NET_CLIENT  2       /* Client address is not valid */
#define ERR_NET_SERVER  3       /* Server address is not valid */
#define ERR_NET_UNAVAIL 4       /* Network is unavailable */
#define ERR_NET_DOWN    5       /* Network is down */
#define ERR_NET_TRANS   6       /* Transient network error */
#define ERR_NET_EXPIRE  7       /* Request expired without valid
response */
#define ERR_NET_count   8

/* Error APP - 2 through 63 are generic, others depend on type */
/*                 Generic codes should be used whenever applicable */
#define ERR_APP_ACTION  2       /* Unknown action code */
#define ERR_APP_CHANNEL 3       /* Invalid channel code */
#define ERR_APP_FLAG    4       /* Invalid flag settings */
#define ERR_APP_ADDRESS 5       /* Invalid address */
#define ERR_APP_DATA    6       /* Invalid or corrupt payload */
#define ERR_APP_EOD     7       /* Address + Size larger than object */
#define ERR_APP_SIZE    8       /* Object too large to be delivered */
#define ERR_APP_UNAVAIL 9       /* Device is unavailable - hard error */
#define ERR_APP_AUTH    10      /* Client lacks authorization */
#define ERR_APP_BUSY    11      /* Device is busy - user try again later
*/
#define ERR_APP_HOLD    12      /* Transient unavail (hold) - keep
trying */
#define ERR_APP_TIMEOUT 13      /* Authorization timed out */
#define ERR_APP_PROXY   14      /* Required proxy action failed */
#define ERR_APP_VERSION 15      /* Unspported application version or
feature */
#define ERR_APP_count   16
#define ERR_APP_APP     64      /* Application dependent errors */

/* Error Route - Give Type of entity that received the request */


/*  Response delivery types  */
```

```
#define MTP_DEL_BUFFER  0       /* Fixed size memory buffer */
#define MTP_DEL_ALLOC   1       /* Allocate a buffer */
#define MTP_DEL_FILE    2       /* Place in a file */
#define MTP_DEL_SINGLE  3       /* Buffer a single datagram */
                                /* address and rsize need not be bytes
*/
                                /* delivery is optimized for speed */

/* Actions - Standard action requests for all devices and applications
*/
#define ACT_PING        0       /* Reply if up */
#define ACT_IDSTR       1       /* Reply with description string */
#define ACT_REGISTER    2       /* Register as a client- data has id
string */
#define ACT_TERMINATE   3       /* Unregister as a client */
#define ACT_DATA        4       /* Generic request code for addressed
data */
#define ACT_REQAUTH     5       /* Device proxying usage authorization
*/
#define ACT_AUTHORIZE   6       /* Allow proxy use of device */
#define ACT_REVOKE      7       /* Remove proxy authorization */
#define ACT_HOLD        8       /* Suspend proxy authorization */
#define ACT_RESUME      9       /* Resume authorization */
#define ACT_NOTIFY      10      /* Notify of change in proxy state */
#define ACT_REQKEY      11      /* Request public encryption key */
#define ACT_STAT        12      /* Get information about data */
#define ACT_count       13
#define ACT_APP         64      /* Application dependent actions */

/* Abort actions - Different levels of aborting a request */
#define ABORT_FINISH    0       /* Don't extend end of object, fill in
holes */
#define ABORT_STOP      1       /* Send no more subrequests, wait for
extant */
#define ABORT_EXPIRE    2       /* Immediately callback */
#define ABORT_DESTROY   3       /* Abandon the request, don't callback
*/

/* Link Speed descriptors */
#define LINK_UNKNOWN    0       /* Unknown, will use default */
#define LINK_MODEM      1       /* 14.4 to 56 Kb/s modem */
#define LINK_ISDN       2       /* Single ISDN channel - 64 Kb/s  */
#define LINK_ISDN2      3       /* Dual channel ISDN -  128 Kb/s  */
#define LINK_DSL        4       /* DSL - 256 to 1024 Kb/s  */
#define LINK_CABLE      5       /* Cable - 1 to 8 Mb/s */
#define LINK_ETHERNET   6       /* 10baseT ethernet  10 Mb/s  */
#define LINK_FAST       7       /* 100baseT, fiber, or other fast link
*/

/*
**  Message data structures
*/

typedef struct {                        /* Uniquely identify a transaction
*/
   struct in_addr       peerip;         /* IP address of peer - Network
Order!*/
   uint16               peerport;       /* UDP port number -  Network
Order! */
   uint8                mtpversion;     /* MTP protocol version */
```

```
   uint8                appversion;  /* APP protocol version */
   uint32               reqid;       /* Request id number */
} mtp_id_t;

typedef struct {                     /* Describe a request */
   uint32       channel;             /* Session or object identifier */
   uint32       address;             /* Address within data object */
   uint32       rsize;               /* Size of requested data object */
                                     /* (usually EOF - address) */
   uint8        action;              /* Action being requested */
   uint8        flag;                /* Action modifiers */
   uint16       reserved;
   int          single;              /* Flow control disabled flag */
   int          stream;              /* Dynamic object flag */
   int          suppress;            /* Payload suppression flag */
} mtp_request_t;

typedef struct {
   uint32       type;
   uint32       handle;              /* File descriptor or buffer address */
   uint32       offset;              /* Location for msg.address = 0 */
   uint32       bound;               /* Maximum allowed.  0 for unlimited */
                                     /* on return, max sequenced size */
} mtp_delivery_t;

typedef struct {                     /* Transaction statistics */
   uint32       received;            /* Bytes received */
   uint32       span;                /* Address range of responses */
   uint32       size;                /* Full size of data object (if known)
*/
   uint32       requests;            /* Number of request messages sent */
   uint32       repeats;             /* Number of requests repeated */
   uint32       duplicates;          /* Number of duplicates received */
   uint32       avgdelay;            /* Average RTT */
   uint32       minwindow;           /* Smallest window size */
   uint32       avgwindow;           /* Average window size */
   uint32       maxwindow;           /* Largest window size */
   uint32       etime;               /* Elapsed Time */
} mtp_stats_t;

/* Template for viewing payload by different data types */
typedef union data_s {
   char         str[0];
   uint8        byte[0];
   uint16       sword[0];
   uint32       lword[0];
} data_t;

/*
** Callback Functions
*/

typedef void (* MTP_RequestProc)                /* Handle action request
*/
     (mtp_id_t msg_id, mtp_request_t request,
      int size, void *payload, uint32 user);
typedef void (* MTP_ResponseProc)               /* Receive response */
     (mtp_id_t msg_id, mtp_request_t request, int error, int response,
      uint32 size, int final, mtp_delivery_t delivery, uint32 user,
      mtp_stats_t stats);
```

```
/*
** Public Interfaces
*/

extern struct sockaddr_in MTPname;
extern char * MTPidstr;
extern int MTPidlen;

/* MTP Initialization */
int MTP_Init(uint16     port,                    /* Local UDP port
number */
             int        link,                   /* Link speed
description */
             char *     idstr,          /* Description of this
application */
             int        debug                    /* Debuging detail
level */
             );

/* Generate a request */
int MTP_Request(mtp_id_t *      id,                /* Request
identifier */
                mtp_request_t   request,          /* Request
description */
                int             size,                   /* Payload
size */
                void *          payload,           /* Pointer to
payload */
                uint32          repeat,           /* Request repeat
rate */
                uint32          expire,         /* Request expiration
time */
                mtp_delivery_t  delivery,         /* Delivery
instructions */
                uint32          user,     /* Data to be passed to
callback */
                int             update,           /* Update rate, in
KB */
                MTP_ResponseProc callback      /* Proc to receive
response */
                );

/* Generate a response to a request */
int MTP_Respond(mtp_id_t        id,                /* Request
identifier */
                mtp_request_t   request,          /* Request
description */
                int             error,                /* True for
error */
                int             response,              /* Response
code */
                int             size,                  /* Payload
size */
                void *          payload           /* Pointer to
payload */
                );

/* Main processing procedure */
int MTP_Event(fd_set *  readfds,                  /* Readable
```

```
descriptors */
              fd_set *  writefds,                  /* Writeable
descriptors */
              fd_set *  exceptfds,                 /* Exception
descriptors */
              sint32    maxtime                  /* Timeout in
milliseconds */
              );

/* Abort an ongoing transaction */
int MTP_Abort(uint32 reqid, int action);

/* Terminate network services */
int MTP_Close();

/* Register a request action procedure */
MTP_RequestProc
   MTP_Action(int       action,                 /* Action being
registered */
              uint32    user,                /* Data to be passed to
rproc */
              MTP_RequestProc rproc    /* Procedure to be called on
action */
              );

/* Translate an error into a text string */
char * MTP_Error(int    error,                                  /* Error
code */
                 int    response                             /* Response
code */
                 );

/* Get local host id info */
mtp_id_t MTP_Version();


/* Default procedure for ACT_PING and ACT_IDSTR */
void MTP_Ping(mtp_id_t msg_id, mtp_request_t request,
         int size, void *ptr, uint32 user);

/* Simple payload checksum procedure */
uint32 MTP_Sum(char *payload, int length);

/* Procedure to output statistics */
extern void MTP_Stats(FILE *fd, mtp_stats_t stats);
```

Appendix B: mtping.c

```
/*
**  mtping  -  MTP sample application
**
**  gcc mtping.c -o mtping libmtp.a
**
**  Demonstration of basic MTP interfaces.  Also useful for debugging an
**  MTP application.
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include "mtp.h"

int err,Debug,done=0;
char buffer[MTP_SIZE_RESPONSE];

void
Response(mtp_id_t msg_id, mtp_request_t request, int error, int
response,
         uint32 size, int final, mtp_delivery_t delivery, uint32 user,
         mtp_stats_t stats)
/*
**  Procedure to handle incoming responses
*/
{
   if (error) {
      printf("Ping failed: %s\n",MTP_Error(error,response));
      printf("\tafter %lu repeats\n",stats.repeats);
   } else {
      printf("Response after %lu repeats (Version %u/%u)",
             stats.repeats,msg_id.mtpversion,msg_id.appversion);
   }
   if (size > 0)
     printf(": %s\n",(char *)delivery.handle);
   else
     printf("\n");

   done = 1;
}
```

```
void
Server(mtp_id_t msg_id, mtp_request_t request,
      int size, void *ptr, uint32 user)
/*
**  Procedure to handle incoming requests
**
**  Note that we index our response data directly from its source after
**  making sure we have a valid address and reseting rsize.
**  There is no need to make a copy of the response data.
*/
{
   printf("Received Message %.8lx from %s.%u\n",msg_id.reqid,
          inet_ntoa(msg_id.peerip),ntohs(msg_id.peerport));
   printf("\tAction: %u\tFlag: %u \tChannel: %.8lx\n",
          request.action,request.flag,request.channel);
   printf("\tAddress: %lu\tRsize: %lu\n",request.address,request.rsize);
   printf("\tPayload %d bytes\n",size);

   /* The code below is identical to MTP_Ping */
   switch (request.action) {
    case ACT_PING:
      MTP_Respond(msg_id,request,0,RES_REPLY,0,NULL);
      break;
    case ACT_IDSTR:
      if (request.address > MTPidlen) {
         MTP_Respond(msg_id,request,1,ERR_APP_ADDRESS,0,NULL);
         return;
      }
      if (request.rsize == 0 || MTPidlen - request.address <
request.rsize)
        request.rsize = MTPidlen - request.address;
      MTP_Respond(msg_id, request, 0, RES_REPLY,
                  request.rsize,MTPidstr+request.address);
      break;
    default:
      MTP_Respond(msg_id,request,1,ERR_APP_ACTION,0,NULL);
   }
}
```

```
int
Get_Options(int argc, char **argv, char **host, int *port)
/*
**  Parse the command line
**
**  Return true if server mode
*/
{
   int c,server=0;
   extern char *optarg;
   extern int optind/*, opterr*/;
   static const char * usage =
     "usage: %s [-s][-d debug_level] host port\n";

   Debug = 0;

   while ((c = getopt(argc,argv,"sd:")) != -1)
     switch (c) {
      case 's':                  /* server mode */
        server = 1;
        break;
      case 'd':                  /* Set debug output level */
        Debug = atoi(optarg);
        break;
      default:
        fprintf(stderr,usage,argv[0]);
        exit(-1);
     }

   if (server) {
      *host = NULL;
      if (optind >= argc) { fprintf(stderr,usage,argv[0]); exit(-1); }
      *port = atoi(argv[optind]);
   } else {
      *host = argv[optind];
      if (optind+1 >= argc) { fprintf(stderr,usage,argv[0]); exit(-1); }
      *port = atoi(argv[optind+1]);
   }

   return server;
}  /* Get_Options */
```

```
int
main(int argc, char **argv, char **envp)
{
   int Port,server;
   struct hostent *     serverhost;
   mtp_id_t id;
   mtp_request_t request;
   mtp_delivery_t delivery;
   char *host;

   server = Get_Options(argc,argv,&host,&Port);

   if (server) {
      if (MTP_Init(Port,LINK_UNKNOWN,
                  "mtping - 2.0 August 1998 - Seth Noble",Debug) < 0) {
         fprintf(stderr,"MTP_Init Failed\n");
         exit(-1);
      }
      /* Comment thse out to use the default, MTP_Ping */
      MTP_Action(ACT_PING, 0, Server);
      MTP_Action(ACT_IDSTR, 0, Server);

   } else {

      serverhost = gethostbyname(host);
      if (serverhost == NULL) {
         fprintf(stderr,"Invalid server: %s\n",argv[1]);
         exit(1);
      }
      if (MTP_Init(0,LINK_UNKNOWN,
                  "mtping - 2.0 August 1998 - Seth Noble",Debug) < 0) {
         fprintf(stderr,"MTP_Init Failed\n");
         exit(-1);
      }
      memcpy(&id.peerip,serverhost->h_addr_list[0],4);
      id.peerport = htons(Port);
      id.appversion = 0;
      request.action = ACT_IDSTR;
      request.flag = 0;
      request.channel = 0;

      /* Change these to index a portion of the string */
      request.address = 0;
      request.rsize = 0;

      delivery.type = MTP_DEL_SINGLE;
      delivery.handle = (uint32)buffer;
      delivery.offset = 0;
      delivery.bound = MTP_SIZE_RESPONSE;

      printf("Sending ACT_IDSTR to %s\n",serverhost->h_name);
      MTP_Request(&id, request, 0, NULL, 500, 2100, delivery, 0, 0,
Response);
   }

   while (!done) MTP_Event(NULL,NULL,NULL,-1);

   return 0;

}
```

Appendix C: Testing Source Code

Transaction Test Server

```
/*
**  tserve.h  -  Headers for test server
*/

#include <limits.h>
#include <errno.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/file.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/mman.h>
#include <search.h>
#include <signal.h>
#include <string.h>
#ifdef sunos4
#include "sunos4.h"
#endif

#define MAX_CLIENTS     (8)
#define BLOB            (32*1024)

#define TCP_NONE        (0)     /* Unused */
#define TCP_READ        (1)     /* Waiting for "request" */
#define TCP_WRITE       (2)     /* Writing "response" */

typedef struct {
   int  state;
   int  socket;
   int  sent;
} tcp_t;

/* Display Message */
#define CHECK(ret, code, error) \
        { if (((ret) = (code)) < 0) { \
          fprintf(stderr,"%s: %s\n",error,strerror(errno)); } }

/* Display message and exit */
#define GUARD(ret, code, error) \
        { if (((ret) = (code)) < 0) { \
          fprintf(stderr,"%s: %s\n",error,strerror(errno)); exit(1);} }

#if defined(solaris) || defined(linux)
#define bzero(ptr,len) memset((char *)(ptr),'\0',(len))
#endif
```

```
/*
**  tserve.c  - Test server for MTP and TCP trials
**
**  Version 1.0  -  February 1999  --  Seth Noble
**
*/

static char *id = "tserve - 1.0 February 1999 - Seth Noble";

#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include "mtp.h"
#include "tserve.h"

int err,Debug,done=0,Port,Link,Size;

char Data[BLOB];        /* Arbitrary data blob */
tcp_t Main;
tcp_t Client[MAX_CLIENTS];
fd_set Readfds[2],Writefds[2],Exceptfds[2];

void
MTP_Data(mtp_id_t msg_id, mtp_request_t request,
         int size, void *ptr, uint32 user)
/*
**  Send data blob.
*/
{
   uint32 address,rsize,run;

   address = request.address;
   rsize = request.rsize;

   request.rsize = Size - address;
   if (rsize != 0 && rsize < request.rsize)
     run = rsize;
   else
     run = request.rsize;

   MTP_Respond(msg_id, request, 0, RES_REPLY, run, Data);
}  /* MTP_Data */

void
TCP_Accept()
/*
**  Accept an incoming TCP connection.
*/
{
   struct sockaddr addr;
   int i,size=sizeof(addr);

   for (i=0; i<MAX_CLIENTS; i++)
     if (Client[i].state == TCP_NONE) break;

   if (i == MAX_CLIENTS) {
      fprintf(stderr,"Cannot support another TCP client\n");
      return;
   }
```

```
   CHECK(Client[i].socket,accept(Main.socket,&addr,&size),"accept");
   CHECK(err,fcntl(Client[i].socket,F_SETFL,O_NDELAY),"fcntl");
   Client[i].state = TCP_WRITE;
   Client[i].sent = 0;
   FD_SET(Client[i].socket,&Writefds[1]);
   FD_SET(Client[i].socket,&Exceptfds[1]);
}  /* TCP_Accept */

void
TCP_Clean(int i)
{
   CHECK(err,close(Client[i].socket),"close");
   Client[i].state = TCP_NONE;
   FD_CLR(Client[i].socket,&Writefds[1]);
   FD_CLR(Client[i].socket,&Exceptfds[1]);
}

void
TCP_Write(int i)
/*
**  Send data blob.
*/
{
   int run,sent;

   if (i < 0 || Client[i].sent >= Size) {
      if (i<0) i = -i;
      TCP_Clean(i);
      return;
   }

   run = Size - Client[i].sent;
   if (run > BLOB) run = BLOB;

   sent = write(Client[i].socket,Data,run);

   if (sent < 0)
     if (errno == EPIPE)
       TCP_Clean(i);
     else
       perror("write");
   else
     Client[i].sent += sent;

}  /* TCP_Write */

void
Get_Options(int argc, char **argv)
/*  INTERNAL
**  Parse the command line
*/
{
   int c;
   extern char *optarg;
   extern int optind/*, opterr*/;

   Debug = 0; Port = 8080; Link = LINK_ETHERNET;

   while ((c = getopt(argc,argv,"vd:p:l:")) != -1)
     switch (c) {
```

```
      case 'd':                  /* Set debug output level */
        Debug = atoi(optarg);
        break;
      case 'p':                  /* Set port number */
        Port = atoi(optarg);
        break;
      case 'v':                  /* Display version number */
        fprintf(stderr,"%s: %s\n",argv[0],id);
        break;
      case 'l':
        Link = atoi(optarg);
        break;
      default:
        fprintf(stderr,
        "usage: %s [-v] [-d debug_level] [-p port] [-l link] [size]\n",
                argv[0]);
    }

  if (optind < argc)
    Size = atoi(argv[optind]);
  else
    Size = 1024*1024;

}  /* Get_Options */

int
main(int argc, char **argv, char **envp)
{
  int i;
  struct sockaddr_in name;
  struct sigaction oldaction, newaction;

  Get_Options(argc,argv);

  if (MTP_Init(Port,Link,id,Debug) < 0) {
    fprintf(stderr,"Can't init MTP\n");
    exit(-1);
  }
  MTP_Action(ACT_DATA,0,MTP_Data);

  bzero(&Main,sizeof(tcp_t));
  bzero(Client,sizeof(tcp_t)*MAX_CLIENTS);
  GUARD(Main.socket,socket(PF_INET,SOCK_STREAM,0),"TCP socket");
  name.sin_family = PF_INET;
  name.sin_port = Port;
  name.sin_addr.s_addr = INADDR_ANY;
  GUARD(err,bind(Main.socket,(struct sockaddr *)&name,sizeof(name)),
        "TCP bind");
  GUARD(err,listen(Main.socket,5),"listen");
  FD_ZERO(&Exceptfds[1]);
  FD_ZERO(&Writefds[1]);
  FD_ZERO(&Readfds[1]);
  FD_SET(Main.socket,&Readfds[1]);
#if defined(sunos4) || defined(linux)
  newaction.sa_mask = 0;
#else
  {  /* Bloody stupid solaris kludge */
    sigset_t nullsmask={{0,0,0,0}};
    newaction.sa_mask = nullsmask;
  }
```

```
#endif
   newaction.sa_handler = SIG_IGN;
   newaction.sa_flags = 0;
   sigaction(SIGPIPE,&newaction,&oldaction);


   do {
      Exceptfds[0] = Exceptfds[1];
      Writefds[0] = Writefds[1];
      Readfds[0] = Readfds[1];
      MTP_Event(&Readfds[0],&Writefds[0],&Exceptfds[0],-1);
      if (FD_ISSET(Main.socket,&Readfds[0]))
        TCP_Accept();
      for (i=0; i<MAX_CLIENTS; i++)
        if (Client[i].state == TCP_WRITE &&
            FD_ISSET(Client[i].socket,&Exceptfds[0]))
          TCP_Write(-i);
      for (i=0; i<MAX_CLIENTS; i++)
        if (Client[i].state == TCP_WRITE &&
            FD_ISSET(Client[i].socket,&Writefds[0]))
          TCP_Write(i);
   } while (!done);

   return 0;
}
```

Transaction Test Client

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <time.h>
#include <sys/time.h>
#include <netdb.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>
#include <signal.h>
#include "mtp.h"
#if defined(sunos4)
#include <sys/stdtypes.h>
#include "sunos4.h"
#endif


#define PING_SLOW       0       /* Ping at 1 sec intervals */
#define PING_MEDIUM     1       /* Fast ping for MTP */
#define PING_FAST       2       /* Fast ping for both */

typedef struct {
   int          mode;
   int          output;
   int          pid;
   double       result;
} ping_t;

typedef struct {
   int          size;   /* Size of transfer */
   int          time;   /* Milliseconds */
   int          rtt;    /* Observed RTT (mtp only) */
   float        loss;   /* Observed loss (mtp only) */
   float        aloss;  /* Ambient loss rate (ping) */
   int          fast;   /* True if fast ping was used */
} stat_t;

typedef struct {
   stat_t       mtp;
   stat_t       tcp;
} datum_t;


/* Display Message */
#define CHECK(ret, code, error) \
        { if (((ret) = (code)) < 0) { \
          fprintf(stderr,"%s: %s\n",error,strerror(errno)); } }

/* Display message and exit */
#define GUARD(ret, code, error) \
        { if (((ret) = (code)) < 0) { \
          fprintf(stderr,"%s: %s\n",error,strerror(errno)); exit(1);} }
```

```
/*
**  tclient  -  Run trials of MTP vs TCP transfer rates
**
**  Alternately issue MTP and TCP data requests to a given server.  Run
a
**  simultaneous ping to measure ambient packet loss during each.
**
*/

#include "tclient.h"

int err,Debug,Port,Verbose,Link,Null,Going,Timeout,Count;
char *Server;
struct hostent * Serverhost;
mtp_id_t Peer;
char buffer[MTP_SIZE_RESPONSE];
ping_t Ping;
static char *Id="mtptest - 1.0 February 1999 - Seth Noble";

void
Start_Ping()
/*
**  Spawn a ping process to keep track of ambient packet loss.
**
**  Use pipe(2) to create a socket pair, then fork(2) and execl(3) to
pass
**  the pipe to sh for execution of ping.
*/
{
   static char *shell="/bin/sh";
/*   static char *sping="ping -q -t 1 %s >&%d";*/
   static char *fping="ping -qF %s >&%d";
   char pingstr[80];
   int fd[2];

   GUARD(err,pipe(fd),"pipe");;

   Ping.output = fd[0];

   if ((Ping.pid = fork())) {
      /* This is the original process */
      close(fd[1]);
      return;
   } else {
      /* This is the new child process */
      sprintf(pingstr,fping,Server,fd[1]);

#if defined(irix)
      BSDsetpgrp(0,getpid());
#elif defined(solaris)
      setpgrp();
#else
      setpgrp(0,getpid());
#endif
      execl(shell,"sh","-c",pingstr,NULL);
   }
}  /* Start_Ping */
```

```
void
Stop_Ping()
/*
**  Halt the ping and collect its output
**
*/
{
   int size;
   char pout[512], *ptr;
#if defined(solaris)
   extern int killpg(pid_t pgrp, int sig);
#endif

   CHECK(err,killpg(Ping.pid,SIGINT),"killpg");
   waitpid(-1,NULL,0);
   size = read(Ping.output,pout,511);
   pout[511] = '\0';

   ptr = strchr(pout,'%');
   if (ptr == NULL) {
      fprintf(stderr,"Unexpected ping output %s\n",pout);
      exit(-1);
   }

   while (*ptr != ' ') ptr--;
   ptr++;

   Ping.result = atof(ptr);
   close(Ping.output);

}  /* Stop Ping */

void
mtpdone(mtp_id_t msg_id, mtp_request_t request, int error, int response,
        uint32 size, int final, mtp_delivery_t delivery, uint32 user,
        mtp_stats_t stats)
{
   if (error) {
      printf("\nGET failed: %s\n",MTP_Error(error,response));
      exit(-1);
   }

   if (final)  {
      Going--;
      printf("%lu\t%.1f\t%.1f\t%lu\t%.1f\t",
             size,stats.etime/1000.0,(size/1024.0)/(stats.etime/1000.0),
             stats.avgdelay,
             stats.repeats*100.0/(stats.requests+stats.repeats));
/*  printf("Received %lu of %lu in %.1f seconds\tAvg RTT %lu\tLoss
%.1f%%\n",
             stats.received, stats.size, stats.etime/1000.0,
stats.avgdelay,
             stats.repeats*100.0/(stats.requests+stats.repeats));*/
   }
}
```

```
void
Run_MTP()
{
   mtp_request_t request;
   mtp_delivery_t delivery;
   int start, abort;

   request.action = ACT_DATA;
   request.flag = 0;
   request.channel = 0;
   request.address = 0;
   request.rsize = 0;
   delivery.type = MTP_DEL_FILE;
   delivery.handle = Null;
   delivery.offset = 0;
   delivery.bound = 0;
   for (Going = 0; Going < Count; Going++)
     MTP_Request(&Peer,request,0,NULL,2000,30000,delivery,0,0,mtpdone);

   start = time(NULL); abort = 0;
   while (Going) {
      MTP_Event(NULL,NULL,NULL,-1);
      if (time(NULL) > start + Timeout && !abort) {
         MTP_Abort(Peer.reqid,ABORT_FINISH);
         abort = 1;
      }
   }
}  /* Run_MTP */

void
Run_TCP()
{
   int s,start,end,size;
   char buffer[8192];
   struct sockaddr_in name = { PF_INET, Port, Peer.peerip };

   GUARD(s,socket(PF_INET,SOCK_STREAM,0),"TCP socket");
   CHECK(err,fcntl(s,F_SETFL,0),"fcntl");
   GUARD(err,connect(s,&name,sizeof(name)),"connect");

   start = time(NULL);
   size = 0;
   for (err = 1; err >0; ) {
      GUARD(err,read(s,buffer,8192),"read");
      size += err;
      end = time(NULL);
      if (end - start > Timeout) break;
   }

   printf("%d\t%d\t%.2f\t",
         size, end-start, (double)size/(1024.0*(end-start)));
/*   printf("TCP read %d bytes in %d seconds\n",size,end - start);*/

   close (s);

}  /* Run_TCP */
```

```
void
Get_Options(int argc, char **argv)
/*  INTERNAL
**  Parse the command line
*/
{
   int c;
   extern char *optarg;
   extern int optind;/*, opterr;*/
   Debug = 0; Port = 8080; Verbose = 0; Link = LINK_ETHERNET;
   Ping.mode = PING_FAST; Timeout = 60; Count = 1;

   while ((c = getopt(argc,argv,"vd:p:st:l:m:t:c:")) != -1)
     switch (c) {
      case 'd':                  /* Set debug output level */
        Debug = atoi(optarg);
        break;
      case 'p':                  /* Set peer udp/tcp port number */
        Port = atoi(optarg);
        break;
      case 'v':                  /* Verbose mode */
        Verbose = 1;
        break;
      case 'l':
        Link = atoi(optarg);
        break;
      case 'm':
        Ping.mode = atoi(optarg);
        break;
      case 't':
        Timeout = atoi(optarg);
        break;
      case 'c':
        Count = atoi(optarg);
        break;
      default:
        fprintf(stderr,"%s: %s\n",argv[0],Id);
        fprintf(stderr,
    "usage: %s [-v] [-d debug_level] [-p port] [-l link]
testhost\n",argv[0]);
        exit(1);
     }

   if (argc - optind != 1) {
      fprintf(stderr,
    "usage: %s [-v] [-d debug_level] [-p port] [-l link]
testhost\n",argv[0]);
      exit(-1);
   }

   Server = argv[optind];
   Serverhost = gethostbyname(Server);
   if (Serverhost == NULL) {
      fprintf(stderr,"Invalid server: %s\n",Server);
      exit(1);
   }

}  /* Get_Options */
```

```
int
main(int argc, char **argv, char **envp)
{
   time_t now;

   Get_Options(argc,argv);

   if (MTP_Init(0,Link,Id,Debug) < 0) {
      fprintf(stderr,"MTP_Init Failed\n");
      exit(-1);
   }
   CHECK(Null,open("/dev/null",O_WRONLY),"/dev/null");
   memcpy((char *)&Peer.peerip,Serverhost->h_addr_list[0],4);
   Peer.peerport = Port;
   Peer.appversion = 0; /*FILE_VERSION;*/

   now = time(NULL);
   printf("# MTP vs. TCP network test\n# %s  %s#\n",
          Server,ctime(&now));
   printf("#MTP\t\t\t\t\t\tTCP\n");

printf("#Recv\tTime\tThput\tRTT\tOloss\tPloss\tRecv\tTime\tThput\tPloss\
n");
   fflush(stdout);

   while (1) {
      Start_Ping();
      Run_MTP();
      Stop_Ping();
      printf("%.1f\t",Ping.result);
      fflush(stdout);
      sleep(10);
      Start_Ping();
      Run_TCP();
      Stop_Ping();
      printf("%.1f\n",Ping.result);
      fflush(stdout);
      sleep(60);
   }

   return 0;
}
```

Data Analyzer for Test Client

```
/*
**  tplot.c  -  Format the output from tclient into a plottable form
**
**  We will want to plot ambient loss versus throughput for both MTP and
TCP.
**
**  First we coallate the data points by 0.5% increments.  Optionally,
**  we then apply a gaussian blur to fill to smooth the graph and fill
in
**  gaps.
*/

#include <stdio.h>
#include <string.h>

int err;
#define CHECK(ret, code, error) \
        if (((ret) = (code)) < 0) {perror(error);}
#define GUARD(ret, code, error) \
        if (((ret) = (code)) < 0) {perror(error);exit(1);}
#define SENTRY(ret, code, error, tag) \
        if (((ret) = (code)) < 0) {perror(error);return(-tag);}

typedef struct {
   int          count;
   double       sample;
} perc_t;

extern double atof(char *str);

perc_t mtp[201], tcp[201];

void
get_data(FILE *fd)
/*
**  Each line contains a data point for one MTP and one TCP run
*/
{
   char line[80], *thputstr, *lossstr;
   int index, field;
   double percent,tmp,thput;

   while (fgets(line, 80, fd)) {
      if (line[0] == '#' || line[0] == '=' || line[0] == '\n') continue;
      strtok(line," \t");
      strtok(NULL," \t");
      thputstr = strtok(NULL," \t");
      strtok(NULL," \t");
      strtok(NULL," \t");
      lossstr = strtok(NULL," \t");
      if (!thputstr || !lossstr) continue;

      percent = atof(lossstr);
      index = percent * 2;
      thput = atof(thputstr);
      mtp[index].sample += thput;
      mtp[index].count++;
```

```
        /*printf("%.1f\t%.1f\t",percent,thput);*/

        strtok(NULL," \t");
        strtok(NULL," \t");
        thputstr = strtok(NULL," \t");
        lossstr = strtok(NULL," \t\n");
        if (!thputstr || !lossstr) continue;

        percent = atof(lossstr);
        index = percent * 2;
        thput = atof(thputstr);
        tcp[index].sample += thput;
        tcp[index].count++;
        /*printf("%.1f\t%.1f\n",percent,thput);*/
    }
}

void
Smooth(perc_t data[])
{
    int index,count;
    double sum;
    perc_t smoothed[201];

    bzero(smoothed,sizeof(perc_t)*201);

    for (index=0; index < 201; index++ ) {
        count=0;
        sum=0.0;

        if (index > 1 && data[index-2].count > 0) {
            sum += data[index-2].sample / data[index-2].count;
            count++;
        }
        if (index > 0 && data[index-1].count > 0) {
            sum += 2.0*data[index-1].sample / data[index-1].count;
            count+=2;
        }
        if (data[index].count > 0) {
            sum += 4.0*data[index].sample / data[index].count;
            count+=4;
        }
        if (index < 200 && data[index+1].count > 0) {
            sum += 2.0*data[index+1].sample / data[index+1].count;
            count+=2;
        }
        if (index < 199 && data[index+2].count > 0) {
            sum += data[index+2].sample / data[index+2].count;
            count++;
        }
        if (count) {
            smoothed[index].count = data[index].count;
            smoothed[index].sample = sum/count;
        }
    }
    memcpy(data,smoothed,201*sizeof(perc_t));
}
```

```
void
Output1(FILE *fd)
{
   int index;

   for (index=0; index < 201; index++) {
      if (!mtp[index].count && !tcp[index].count) continue;
      fprintf(fd,"%3.1f\t",(double)index/2.0);
      if (mtp[index].count)
        fprintf(fd,"%.1f\t%d\t",
                mtp[index].sample/mtp[index].count,mtp[index].count);
      else
        fprintf(fd,"-\t-\t");
      if (tcp[index].count)
        fprintf(fd,"%.2f\t%d\n",
                tcp[index].sample/tcp[index].count,
                tcp[index].count);
      else
        fprintf(fd,"-\t-\n");
   }
}

void
Output2(FILE *fd)
{
   int index;

   for (index=0; index < 201; index++) {
      if (mtp[index].sample == 0.0 && tcp[index].sample == 0.0)
continue;
      fprintf(fd,"%.3g\t",(double)index/2.0);
      if (mtp[index].sample > 0.0)
        fprintf(fd,"%.4g\t%d\t",mtp[index].sample,mtp[index].count);
      else
        fprintf(fd,"0\t0\t");
      if (tcp[index].sample > 0.0)
        fprintf(fd,"%.4g\t%d\n",tcp[index].sample,tcp[index].count);
      else
        fprintf(fd,"0\t0\n");
   }
}
```

```
int
main(int argc, char **argv, char **envp)
{
   int arg;
   FILE *fd;

   bzero(mtp,201*sizeof(perc_t));
   bzero(tcp,201*sizeof(perc_t));

   for (arg=1; arg<argc; arg++) {
      if ((fd = fopen(argv[arg],"r"))) {
         get_data(fd);
         close(fd);
      } else
        fprintf(stderr,"%s: Could not open %s, skipping.\n",
                argv[0],argv[arg]);
   }


   Output1(stderr);

   Smooth(mtp);
   Smooth(tcp);

   Output2(stdout);
}
```

Appendix D: Empirical Test Results

Smoothed Averages

Loss Perc.	MTP Rate	Count	TCP Rate	Count
0	299.3	201	24.07	233
2	225	28	13.73	43
4	217.6	36	8.637	15
6	172	17	4.299	10
8	119	11	2.973	2
10	96.96	10	2.132	7
12	96.24	10	1.585	4
14	108.7	0	1.145	2
16	102	3	0.99	0
18	54.6	0	0.9713	1
20	22.54	1	0.66	1
22	6.3	2	0.725	1
24	5.72	1	0.5675	0
26	5.293	1	0.5183	1
28	4.983	2	0.347	2
30	4.286	4	0.2307	3
32	3.983	0	0.1243	1
34	3.325	0	0.1421	1
36	2.3	0	0.226	0
38	2.3	0	0.374	1
40	2.3	3	0.3433	0
42	2.367	0	0.1875	0
44	2.433	0	0.1033	1
46	2.5	1	0.05667	1
48	2.5	0	0.055	0
50	2.5	0	0.03667	0
52	0	0	0.05	1
54	0	0	0.05	0
56	0	0	0.05	0
58	1.9	0	0	0
60	1.9	0	0	0
62	1.9	1	0	0
64	1.9	0	0	0
66	1.9	0	0	0

Raw Averages

Loss Perc.	MTP Rate	Count	TCP Rate	Count
0.0	398.0	201	36.22	233
2.0	122.3	28	9.15	43
4.0	258.5	36	5.32	15
6.0	223.0	17	3.92	10
8.0	53.4	11	2.73	2
10.0	82.1	10	2.06	7
12.0	107.2	10	1.54	4
14.0	-	-	0.63	2
16.0	123.5	3	-	-
18.0	-	-	1.50	1
20.0	10.5	1	0.14	1
22.0	5.0	2	0.86	1
24.0	4.8	1	-	-
26.0	6.1	1	0.58	1
28.0	5.3	2	0.39	2
30.0	3.3	4	0.20	3
32.0	-	-	0.03	1
34.0	-	-	0.11	1
38.0	-	-	0.44	1
40.0	2.3	3	-	-
44.0	-	-	0.15	1
46.0	2.5	1	0.01	1
52.0	-	-	0.05	1
62.0	1.9	1	-	-

Raw Data

#19990219

#MTP #Recv	Time	Thput	RTT	Oloss	Ploss	TCP Recv	Time	Thput	Ploss
1048576	17.0	60.3	112	10.6	9.3	169472	62	2.67	11.0
1048576	13.0	78.6	120	9.7	9.0	412160	61	6.60	5.2

#MTP #Recv	Time	Thput	RTT	Oloss	Ploss	TCP Recv	Time	Thput	Ploss
1048576	13.0	78.8	111	6.7	5.9	198144	61	3.17	5.2
1048576	17.5	58.5	106	8.1	8.4	270848	61	4.34	7.3
1048576	8.5	119.9	112	8.6	12.0	273408	61	4.38	6.2
1048576	12.1	84.4	110	6.4	3.3	399360	62	6.29	5.5
1048576	13.7	74.7	113	5.7	7.6	15360	72	0.21	5.3
1048576	10.7	95.5	116	10.9	6.1	345088	62	5.44	5.0
1048576	4.1	250.6	121	5.1	3.3	463360	62	7.30	3.7
1048576	5.6	182.9	116	8.0	5.9	451584	61	7.23	2.7
1048576	8.6	119.7	116	7.9	5.5	411648	61	6.59	3.5
1048576	5.1	202.4	116	5.9	2.5	322048	61	5.16	3.2
1048576	5.1	201.4	125	5.1	6.2	543232	63	8.42	3.4
1048576	8.1	126.0	123	9.3	6.0	551936	61	8.84	2.6
1048576	7.8	130.9	129	6.0	3.4	1048576	48	21.33	0.6
1048576	4.5	228.0	124	4.4	2.9	799232	61	12.80	2.1
1048576	4.1	249.1	117	4.0	0.0	723456	61	11.58	1.9
1048576	4.9	210.0	114	6.9	2.6	883712	62	13.92	1.0
1048576	2.9	348.5	113	3.0	0.0	1048576	51	20.08	0.6
1048576	2.5	407.0	109	2.2	0.0	1048576	31	33.03	0.0
1048576	3.4	302.9	111	2.8	0.0	1048576	35	29.26	0.3
1048576	3.0	339.6	112	3.6	4.8	1048576	27	37.93	0.7
1048576	2.4	426.3	111	1.3	0.0	1048576	26	39.38	0.4
1048576	2.6	394.8	111	1.3	5.9	1048576	43	23.81	1.8
1048576	2.2	463.8	135	0.0	0.0	1048576	38	26.95	1.4

1048576	3.0	346.3	138	1.2	5.3	1048576 33	31.03
0.6							
1048576	2.9	356.3	112	0.4	0.0	1048576 28	36.57
1.1							
1048576	2.2	460.6	147	0.0	0.0	1048576 28	36.57
0.3							
1048576	3.0	340.9	110	3.2	0.0	1048576 29	35.31
0.3							
1048576	2.9	354.1	116	1.6	5.0	1048576 34	30.12
0.6							
1048576	8.1	126.4	102	8.5	0.0	298496 62	4.70
1.9							
1048576	13.7	74.6	97	7.6	1.5	420352 62	6.62
1.0							
1048576	8.6	119.5	95	6.4	0.0	1048576 60	17.07
0.3							
1048576	2.6	395.8	130	0.9	5.3	314368 61	5.03
1.6							
1048576	4.1	252.2	119	4.0	6.9	1048576 42	24.38
1.0							
1048576	3.4	302.8	153	1.6	0.0	1048576 27	37.93
0.3							
1048576	2.9	350.4	118	2.2	0.0	1048576 46	22.26
1.4							
1048576	3.0	337.8	116	1.6	0.0	1048576 37	27.68
0.8							
1048576	2.9	352.1	111	2.8	0.0	1048576 28	36.57
0.0							
1048576	2.6	390.7	118	0.8	6.2	1048576 31	33.03
0.3							
1048576	2.3	437.0	128	0.4	0.0	1048576 30	34.13
0.0							
1048576	2.4	421.9	131	0.9	0.0	1048576 37	27.68
0.9							
1048576	5.7	178.8	120	9.4	10.0	1048576 45	22.76
1.2							
1048576	2.8	364.4	136	5.9	5.6	852992 61	13.66
2.5							
1048576	5.3	194.4	127	7.6	12.0	514560 61	8.24
2.5							
1048576	4.4	234.6	128	7.7	3.4	1048576 29	35.31
0.3							
1048576	3.2	323.3	163	4.0	0.0	1048576 27	37.93
0.3							
1048576	2.6	398.1	117	1.6	0.0	1048576 29	35.31
0.3							
1048576	2.3	454.1	124	0.0	0.0	1048576 43	23.81
0.9							
1048576	2.4	432.8	116	0.9	0.0	1048576 46	22.26
0.6							
1048576	3.0	337.7	114	3.2	4.3	1048576 36	28.44
1.1							
1048576	2.5	409.1	144	0.0	0.0	1048576 31	33.03
0.9							
1048576	2.3	436.9	147	0.9	0.0	1048576 26	39.38
0.0							
1048576	2.5	406.0	131	1.3	0.0	1048576 25	40.96
0.4							
1048576	2.3	452.7	139	0.5	0.0	1048576 31	33.03
0.3							

1048576	2.2	459.6	129	0.0	0.0	1048576 27 37.93 0.3
1048576	5.5	184.9	118	8.0	11.0	1048576 47 21.79 1.2
1048576	2.9	348.1	120	2.4	0.0	1048576 36 28.44 0.3
1048576	3.0	342.8	131	2.6	0.0	1048576 33 31.03 0.3
1048576	3.9	261.8	115	7.0	10.0	1048576 37 27.68 0.6
1048576	3.8	271.0	131	5.5	3.7	1026048 61 16.43 2.4
1048576	3.5	292.9	117	4.7	0.0	1048576 43 23.81 0.5
1048576	4.7	218.2	130	6.6	17.0	1048576 48 21.33 2.3
1048576	6.2	163.9	113	9.8	2.0	859136 61 13.75 2.2
1048576	3.1	327.6	126	5.3	5.0	1048576 42 24.38 0.5
1048576	2.6	391.6	128	2.6	5.6	1048576 31 33.03 0.3
1048576	4.3	239.5	121	10.4	8.7	1048576 34 30.12 0.3
1048576	3.0	344.0	128	1.8	0.0	1048576 29 35.31 0.0
1048576	3.2	317.1	125	6.6	12.0	925696 61 14.82 2.2
1048576	3.3	308.1	129	1.2	0.0	1048576 35 29.26 0.3
1048576	3.0	344.1	126	3.8	0.0	1048576 34 30.12 0.3
1048576	3.2	321.1	123	6.2	0.0	1048576 26 39.38 0.0
1048576	3.5	292.2	122	4.4	4.2	1048576 27 37.93 0.0
1048576	3.4	297.7	124	6.2	4.2	1048576 32 32.00 0.0
1048576	3.6	284.3	120	3.2	3.6	1048576 28 36.57 0.0
1048576	2.8	371.7	146	2.1	0.0	1048576 29 35.31 0.0
1048576	2.3	451.1	147	0.0	0.0	1048576 29 35.31 0.0
1048576	2.8	364.4	121	0.8	0.0	1048576 34 30.12 0.0
1048576	3.0	342.4	119	1.6	5.0	1048576 31 33.03 0.6
1048576	2.2	473.6	137	0.0	0.0	1048576 36 28.44 1.5
1048576	2.9	350.9	114	4.0	0.0	1048576 34 30.12 0.9
1048576	2.2	469.1	125	0.0	0.0	1048576 40 25.60 0.2
1048576	2.1	478.3	121	0.0	0.0	1048576 28 36.57 0.3
1048576	2.3	453.7	128	0.0	0.0	1048576 31 33.03 0.3
1048576	3.2	320.4	121	2.9	4.5	1048576 28 36.57 0.0

```
1048576 2.7     381.1   127     0.8     0.0     1048576 29      35.31
0.3
1048576 2.3     445.4   117     0.4     0.0     1048576 28      36.57
0.0
1048576 2.9     353.8   127     3.1     0.0     1048576 28      36.57
0.0
1048576 3.2     319.8   122     2.4     0.0     1048576 33      31.03
0.6
1048576 2.9     357.0   130     1.8     0.0     1048576 26      39.38
0.0
1048576 2.5     405.2   115     4.3     0.0     1048576 28      36.57
0.7
1048576 2.8     370.7   117     3.7     0.0     1048576 33      31.03
0.3
1048576 2.2     472.3   127     0.0     0.0     1048576 32      32.00
0.0
1048576 3.0     345.5   109     7.8     4.5     1048576 24      42.67
0.0
1048576 2.2     461.9   125     0.0     0.0     1048576 25      40.96
0.0
1048576 2.8     371.8   112     3.6     5.6     1048576 27      37.93
0.0
1048576 2.6     393.4   129     1.3     0.0     1048576 29      35.31
0.3
1048576 2.9     355.4   118     4.2     0.0     1048576 29      35.31
0.3
1048576 3.3     307.0   120     4.4     4.5     1048576 40      25.60
0.2
1048576 3.0     336.3   112     5.2     4.8     1048576 39      26.26
0.2
1048576 3.0     340.8   109     3.2     0.0     1048576 53      19.32
0.6
1048576 2.9     356.7   109     3.6     4.8     1048576 48      21.33
0.4
1048576 2.3     448.1   119     0.0     0.0     1048576 41      24.98
0.7
1048576 2.6     386.4   146     4.2     5.9     1048576 33      31.03
0.9
1048576 2.8     367.4   115     2.0     0.0     1048576 31      33.03
0.7
1048576 3.8     268.1   110     6.6     3.7     1048576 41      24.98
1.3
1048576 3.5     293.5   109     6.6     12.0    1048576 32      32.00
0.6
1048576 2.8     361.2   124     3.0     0.0     1048576 25      40.96
0.0


# MTP vs. TCP network test
# claustro.ugcs.caltech.edu  Mon Feb 22 10:13:15 1999
#
#MTP                                                     TCP
#Recv   Time    Thput   RTT     Oloss   Ploss   Recv    Time    Thput
Ploss
136704  61.2    2.2     111     40.7    40.0    17408   107     0.16
30.0
# MTP vs. TCP network test
# claustro.ugcs.caltech.edu  Mon Feb 22 11:14:58 1999
#
#MTP                                                     TCP
```

#Recv	Time	Thput	RTT	Oloss	Ploss	Recv	Time	Thput	Ploss
714240	61.1	11.4	183	9.2	16.0	93184	61	1.49	3.6
1048576	39.0	26.3	138	7.5	7.9	343552	61	5.50	5.0
1048576	34.3	29.9	192	9.8	8.5	356352	61	5.70	4.6
1048576	9.1	112.5	183	10.4	2.3	248832	61	3.98	2.7
1048576	15.5	65.9	209	7.5	4.3	205312	61	3.29	6.6
1048576	8.6	119.3	128	10.4	1.9	384512	62	6.06	4.0
1048576	9.5	108.2	189	8.4	1.9	558592	61	8.94	2.7
1048576	7.3	140.9	148	10.0	16.0	514048	61	8.23	4.5
1048576	6.0	169.9	128	9.0	2.4	460800	61	7.38	3.7
1048576	7.8	131.5	145	8.7	3.6	798720	61	12.79	2.1
1048576	5.1	201.7	126	4.7	6.9	566784	64	8.65	4.1
1048576	6.9	149.2	172	10.1	5.7	632832	62	9.97	3.9
1048576	8.2	125.0	119	8.3	1.8	550400	61	8.81	3.1
1048576	5.9	172.9	143	10.1	5.9	532480	61	8.52	3.2
1048576	14.8	69.4	123	5.5	3.9	481792	61	7.71	3.8
1048576	13.5	75.6	148	7.3	1.1	505344	62	7.96	3.5
1048576	7.9	128.9	154	7.7	5.1	473600	61	7.58	2.4
1048576	13.5	75.6	176	9.5	4.9	331264	61	5.30	3.5
1048576	39.2	26.1	200	5.9	2.6	205824	61	3.30	3.9
1048576	14.6	70.3	184	9.4	7.4	299520	61	4.80	6.4
1048576	23.7	43.2	170	6.4	4.4	291328	63	4.52	6.2
1048576	20.9	49.1	181	8.1	10.0	281600	61	4.51	5.6
733696	61.5	11.6	192	12.8	10.0	214016	109	1.92	6.6
1048576	32.1	31.9	180	6.9	8.6	378368	61	6.06	6.4
1048576	31.4	32.6	189	8.9	8.2	170496	71	2.35	6.6
1048576	11.6	88.5	141	9.3	11.0	236032	61	3.78	7.5
1048576	11.4	89.4	191	10.7	12.0	296960	63	4.60	4.0
1048576	13.0	78.6	209	10.1	7.4	425472	61	6.81	3.8
1048576	12.3	83.2	156	7.9	5.2	280576	61	4.49	5.2

```
1048576 16.6    61.6    176     9.6     7.2     383488  61      6.14
4.6
1048576 48.8    21.0    203     6.4     4.8     236544  61      3.79
6.2
# MTP vs. TCP network test
# claustro.ugcs.caltech.edu  Mon Feb 22 14:13:09 1999
#
#MTP                                                     TCP
#Recv   Time    Thput   RTT     Oloss   Ploss   Recv    Time    Thput
Ploss
118784  60.3    1.9     123     44.6    62.0    3072    66      0.05
52.0
417792  60.6    6.7     209     14.4    9.7     84480   61      1.35
12.0
434688  60.9    7.0     203     14.8    12.0    94720   61      1.52
12.0
573440  61.0    9.2     202     13.2    11.0    136192  61      2.18
10.0
231936  61.4    3.7     139     29.2    40.0    512     97      0.01
47.0
# MTP vs. TCP network test
# claustro.ugcs.caltech.edu  Mon Feb 22 15:21:15 1999
#
#MTP                                                     TCP
#Recv   Time    Thput   RTT     Oloss   Ploss   Recv    Time    Thput
Ploss
62464   62.3    1.0     127     40.6    40.0    27648   61      0.44
38.0
# MTP vs. TCP network test
# claustro.ugcs.caltech.edu  Mon Feb 22 15:24:30 1999
#
#MTP                                                     TCP
#Recv   Time    Thput   RTT     Oloss   Ploss   Recv    Time    Thput
Ploss
204288  61.4    3.3     127     29.9    30.0    10752   96      0.11
35.0
194560  60.9    3.1     126     37.8    31.0    5632    93      0.06
28.0
300544  60.8    4.8     128     23.7    24.0    26624   66      0.39
30.0
287232  60.5    4.6     139     24.1    29.0    46592   64      0.71
28.0
172032  60.8    2.8     128     35.2    30.0    2560    96      0.03
33.0
153600  60.9    2.5     132     36.1    46.0    93184   61      1.49
11.0
810496  60.7    13.0    203     11.2    12.0    114688  91      1.23
14.0
709632  60.2    11.5    145     11.8    12.0    36352   61      0.58
26.0
378368  60.7    6.1     128     23.3    26.0    23552   164     0.14
21.0
340992  61.6    5.4     119     24.1    23.0    93696   61      1.50
19.0
374272  61.0    6.0     133     22.6    28.0    14848   95      0.15
44.0
256000  60.4    4.1     141     29.1    30.0    3072    95      0.03
14.0
650240  60.7    10.5    126     14.0    20.0    57856   66      0.86
22.0
```

```
665600  60.6    10.7    164     11.8    8.8     78336   62      1.23
13.0
1016832 60.8    16.3    113     11.6    10.0    119808  61      1.92
11.0
1048576 45.2    22.7    123     8.1     9.6     130560  62      2.06
12.0
666624  60.5    10.8    113     12.3    11.0    168448  62      2.65
11.0
1026560 60.8    16.5    118     10.4    9.7     157696  61      2.52
9.6
1048576 21.6    47.3    115     11.7    6.1     128000  63      1.98
11.0
973312  60.5    15.7    110     11.0    12.0    183296  61      2.93
9.9
657408  60.9    10.5    120     10.9    13.0    96768   62      1.52
10.0
630272  61.9    10.0    137     12.1    10.0    4096    95      0.04
30.0


# MTP vs. TCP network test
# mono.ugcs.caltech.edu  Thu Mar  4 10:53:53 1999
#
#MTP                                                     TCP
#Recv   Time    Thput   RTT     Oloss   Ploss   Recv    Time    Thput
Ploss
1048576 2.1     479.4   128     0.0     0.0     1048576 26      39.38
0.0
1048576 2.2     461.5   137     0.0     0.0     1048576 24      42.67
0.0
1048576 2.3     449.3   112     0.9     0.0     1048576 26      39.38
0.0
1048576 2.2     455.7   101     0.0     0.0     1048576 23      44.52
0.0
1048576 2.4     432.4   142     0.0     0.0     1048576 24      42.67
0.0
1048576 2.2     458.6   141     0.0     0.0     1048576 36      28.44
0.8
1048576 2.4     435.4   155     0.0     0.0     1048576 23      44.52
0.0
1048576 2.2     464.2   129     0.0     0.0     1048576 24      42.67
0.0
1048576 2.1     481.0   118     0.0     0.0     1048576 24      42.67
0.0
1048576 2.2     462.5   124     0.9     0.0     1048576 23      44.52
0.4
1048576 2.4     427.0   150     0.0     0.0     1048576 25      40.96
0.0
1048576 2.6     392.8   111     0.4     0.0     1048576 23      44.52
0.0
1048576 2.7     380.1   118     0.4     0.0     1048576 24      42.67
0.0
1048576 2.4     425.4   210     0.0     0.0     1048576 24      42.67
0.4
1048576 2.1     486.7   119     0.0     0.0     1048576 25      40.96
0.0
1048576 2.3     449.1   170     0.0     0.0     1048576 24      42.67
0.0
1048576 2.1     490.2   139     0.0     0.0     1048576 23      44.52
0.0
```

```
1048576 2.2     471.5   122     0.0     0.0     1048576 23      44.52
0.0
1048576 2.2     457.6   189     0.0     0.0     1048576 23      44.52
0.0
1048576 2.1     485.5   120     0.0     0.0     1048576 24      42.67
0.0
1048576 2.1     496.6   115     0.0     0.0     1048576 23      44.52
0.0
1048576 2.1     483.0   146     0.0     0.0     1048576 23      44.52
0.0
1048576 2.2     455.9   148     0.0     0.0     1048576 23      44.52
0.0
1048576 2.1     486.7   127     0.0     0.0     1048576 24      42.67
0.0
1048576 2.8     370.1   135     0.4     0.0     1048576 23      44.52
0.0
1048576 2.3     446.8   111     0.0     0.0     1048576 23      44.52
0.0
# MTP vs. TCP network test
# mono.ugcs.caltech.edu  Thu Mar  4 12:57:50 1999
#
#MTP                                                     TCP
#Recv   Time    Thput   RTT     Oloss   Ploss   Recv    Time    Thput
Ploss
1048576 2.3     447.4   146     0.0     0.0     1048576 29      35.31
0.3
1048576 2.4     430.8   172     0.0     0.0     1048576 27      37.93
0.3
1048576 2.1     482.1   121     0.0     0.0     1048576 24      42.67
0.0
1048576 2.1     476.3   119     0.0     0.0     1048576 24      42.67
0.0
1048576 2.3     438.2   166     0.0     0.0     1048576 23      44.52
0.4
1048576 2.2     459.0   155     0.0     0.0     1048576 23      44.52
0.0
1048576 2.2     471.2   138     0.0     0.0     1048576 24      42.67
0.0
1048576 2.9     355.8   118     0.9     5.3     1048576 30      34.13
0.3
1048576 2.3     441.4   126     0.0     0.0     1048576 31      33.03
1.6
1048576 3.4     298.6   171     3.5     3.8     1048576 34      30.12
1.5
1048576 2.2     462.1   165     0.0     0.0     1048576 33      31.03
2.0
1048576 2.9     348.1   188     0.0     0.0     1048576 36      28.44
0.9
1048576 2.4     428.1   134     0.0     0.0     1048576 24      42.67
0.0
1048576 2.6     396.4   147     0.0     0.0     1048576 26      39.38
0.4
1048576 2.5     417.8   180     0.0     0.0     1048576 24      42.67
0.0
1048576 2.1     483.2   130     0.0     0.0     1048576 24      42.67
0.0
1048576 2.1     484.2   126     0.0     0.0     1048576 24      42.67
0.0
1048576 2.3     446.2   126     0.0     0.0     1048576 24      42.67
0.0
```

1048576	2.4	427.2	170	0.0	0.0	1048576 24	42.67
0.0							
1048576	2.2	459.2	133	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.6	388.2	121	0.4	0.0	1048576 26	39.38
0.7							
1048576	2.0	502.7	121	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.2	469.5	201	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.2	457.1	109	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.5	401.9	126	0.9	0.0	1048576 25	40.96
0.0							
1048576	2.7	380.0	152	0.4	0.0	1048576 23	44.52
0.0							
1048576	2.5	412.9	132	0.8	0.0	1048576 25	40.96
0.0							
1048576	2.3	448.3	123	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.3	445.6	198	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.4	425.4	116	0.4	0.0	1048576 24	42.67
0.0							
1048576	2.2	460.2	128	0.0	0.0	1048576 24	42.67
0.0							
1048576	2.2	464.0	120	0.0	0.0	1048576 31	33.03
0.6							
1048576	2.4	424.2	154	0.0	0.0	1048576 33	31.03
0.3							
1048576	2.9	357.4	164	0.0	6.2	1048576 28	36.57
1.1							
1048576	2.9	347.5	174	1.2	0.0	1048576 26	39.38
0.4							
1048576	2.5	413.7	132	0.4	0.0	1048576 23	44.52
0.4							
1048576	2.1	481.4	159	0.0	0.0	1048576 28	36.57
0.3							
1048576	2.1	493.3	110	0.0	0.0	1048576 29	35.31
0.3							
1048576	2.3	443.9	140	0.0	0.0	1048576 36	28.44
0.5							
1048576	2.2	468.2	166	0.0	0.0	1048576 24	42.67
0.0							
1048576	2.8	363.8	96	1.2	0.0	1048576 24	42.67
0.0							
1048576	2.1	484.8	113	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.2	470.6	120	0.0	0.0	1048576 24	42.67
0.0							
1048576	2.3	438.5	179	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.2	472.3	122	0.0	0.0	1048576 23	44.52
0.0							
1048576	2.1	486.9	119	0.0	0.0	1048576 24	42.67
0.0							
1048576	2.7	381.8	114	0.8	5.3	1048576 23	44.52
0.0							
1048576	2.7	379.0	254	0.0	0.0	1048576 25	40.96
0.0							

1048576	2.1	476.7	139	0.0	0.0	1048576	24	42.67	0.0
1048576	2.5	415.8	168	0.4	0.0	1048576	25	40.96	0.0
1048576	2.3	438.2	124	0.0	0.0	1048576	26	39.38	0.4
1048576	2.3	442.1	120	0.9	0.0	1048576	25	40.96	0.0
1048576	2.2	470.8	125	0.5	0.0	1048576	24	42.67	0.4
1048576	3.3	311.4	164	1.2	0.0	1048576	25	40.96	0.8
1048576	2.4	429.5	133	0.0	0.0	1048576	25	40.96	0.0
1048576	2.7	373.9	115	0.4	0.0	1048576	26	39.38	0.0
1048576	2.2	455.7	134	0.0	0.0	1048576	24	42.67	0.0
1048576	2.2	461.7	147	0.0	0.0	1048576	25	40.96	0.4
1048576	2.5	410.9	113	0.8	0.0	1048576	26	39.38	0.0
1048576	2.3	452.7	163	0.0	0.0	1048576	24	42.67	0.0
1048576	2.4	423.1	188	0.0	0.0	1048576	25	40.96	0.8
1048576	2.3	454.3	111	0.0	0.0	1048576	25	40.96	0.0
1048576	2.4	426.7	164	0.0	0.0	1048576	25	40.96	0.0
1048576	2.2	465.7	135	0.0	0.0	1048576	25	40.96	0.0
1048576	2.1	485.1	123	0.0	7.7	1048576	23	44.52	0.0
1048576	2.2	471.2	134	0.0	0.0	1048576	26	39.38	0.0
1048576	2.3	445.6	149	0.0	0.0	1048576	24	42.67	0.0
1048576	2.2	469.7	132	0.0	0.0	1048576	25	40.96	0.4
1048576	2.3	439.1	184	0.0	0.0	1048576	23	44.52	0.0
1048576	2.6	391.7	111	0.8	0.0	1048576	25	40.96	0.0
1048576	2.4	419.2	132	0.0	0.0	1048576	25	40.96	0.0
1048576	2.3	446.8	142	0.0	0.0	1048576	23	44.52	0.0
1048576	2.2	459.6	128	0.0	0.0	1048576	25	40.96	0.0
1048576	2.3	437.2	135	0.4	7.1	1048576	23	44.52	0.0
1048576	2.2	458.2	125	0.9	0.0	1048576	23	44.52	0.0
1048576	2.3	438.2	133	0.0	0.0	1048576	24	42.67	0.4
1048576	2.2	473.9	114	0.0	0.0	1048576	24	42.67	0.4
1048576	2.1	484.2	120	0.0	0.0	1048576	23	44.52	0.0

1048576	3.0	345.1	116	3.6	0.0	1048576	25	40.96	0.0
1048576	2.5	416.4	127	0.4	0.0	1048576	25	40.96	0.4
1048576	2.7	375.8	110	0.8	0.0	1048576	24	42.67	0.4
1048576	2.2	472.1	128	0.0	0.0	1048576	24	42.67	0.4
1048576	2.6	397.5	106	0.4	5.9	1048576	25	40.96	0.0
1048576	2.2	464.8	120	0.4	6.2	1048576	26	39.38	0.0
1048576	2.2	467.4	120	0.0	0.0	1048576	24	42.67	0.0
1048576	2.4	429.7	120	0.4	0.0	1048576	25	40.96	0.4
1048576	2.3	449.9	123	0.0	0.0	1048576	24	42.67	0.0
1048576	2.2	473.0	115	0.4	0.0	1048576	23	44.52	0.0
1048576	3.3	306.3	128	0.9	0.0	1048576	23	44.52	0.0
1048576	2.3	437.4	146	0.0	0.0	1048576	23	44.52	0.0
1048576	2.2	465.5	127	0.0	0.0	1048576	23	44.52	0.4
1048576	2.6	393.4	130	0.4	0.0	1048576	24	42.67	0.4
1048576	2.3	439.5	113	0.0	0.0	1048576	24	42.67	0.0
1048576	2.7	374.8	121	0.9	0.0	1048576	25	40.96	0.8
1048576	2.3	452.9	157	0.0	0.0	1048576	27	37.93	0.4
1048576	2.5	411.1	138	0.0	0.0	1048576	24	42.67	0.0
1048576	2.4	423.5	146	1.3	0.0	1048576	26	39.38	0.4
1048576	2.4	425.2	141	0.0	0.0	1048576	25	40.96	0.0
1048576	10.7	95.9	114	1.1	0.8	1048576	33	31.03	1.7
1048576	2.5	402.2	116	0.4	0.0	1048576	38	26.95	2.0
1048576	2.5	415.8	143	0.0	0.0	1048576	36	28.44	1.4
1048576	2.4	429.2	115	0.0	0.0	1048576	30	34.13	1.0
1048576	2.4	427.7	194	0.0	0.0	1048576	32	32.00	0.3
1048576	2.4	427.7	120	1.3	0.0	1048576	27	37.93	0.3
1048576	2.5	406.3	106	0.4	0.0	1048576	28	36.57	0.0
1048576	2.1	486.2	119	0.4	0.0	1048576	27	37.93	0.3
1048576	2.3	454.3	173	0.0	0.0	1048576	35	29.26	0.8
1048576	2.4	431.0	126	0.0	0.0	1048576	24	42.67	0.8

1048576	2.8	370.2	114	2.6	5.6	1048576	31	33.03	0.3
1048576	2.4	433.5	127	0.0	0.0	1048576	23	44.52	0.0
1048576	2.3	448.7	158	0.0	0.0	1048576	23	44.52	0.0
1048576	2.3	453.3	106	0.4	0.0	1048576	25	40.96	0.8
1048576	2.4	420.0	121	0.8	6.7	1048576	23	44.52	0.4
1048576	2.1	477.8	157	0.0	0.0	1048576	29	35.31	0.0
1048576	2.2	466.5	147	0.0	0.0	1048576	27	37.93	0.7
1048576	3.0	339.7	120	0.8	0.0	1048576	27	37.93	0.7
1048576	2.3	436.1	124	0.0	0.0	1048576	31	33.03	0.3

Appendix E: mclient

mclient.h

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <time.h>
#include <sys/time.h>
#include <netdb.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>
#include <signal.h>
#include "mtp.h"
#if defined(solaris)
#include <sys/file.h>
#endif
#if defined(sunos4)
#include <sys/stdtypes.h>
#include "sunos4.h"
#endif

typedef struct {
   int          output;
   int          pid;
   double       result;
} ping_t;

typedef struct {
   int          size;    /* Size of transfer */
   int          time;    /* Milliseconds */
   int          rtt;     /* Observed RTT (mtp only) */
   float        loss;    /* Observed loss (mtp only) */
   float        aloss;   /* Ambient loss rate (ping) */
   int          fast;    /* True if fast ping was used */
} stat_t;

typedef struct {
   stat_t       mtp;
   stat_t       tcp;
} datum_t;


/* Display Message */
#define CHECK(ret, code, error) \
        { if (((ret) = (code)) < 0) { \
          fprintf(stderr,"%s: %s\n",error,strerror(errno)); } }

/* Display message and exit */
#define GUARD(ret, code, error) \
        { if (((ret) = (code)) < 0) { \
          fprintf(stderr,"%s: %s\n",error,strerror(errno)); exit(1);} }
```

mclient.c

```
/*
**  mclient  -  Test MTP and TCP performance under multiple loads.
**
**  User specifies number of MTP and TCP transactions to initiate.
**  Statistics are gathered for each.
**
*/

#include "mclient.h"

int err,Debug,Port,Verbose,Link,Null,Going,Timeout,Mcount,Tcount,Noping;
int Start;
int TCP[16],TCPcount=0;
int Size[16];
char *Server;
struct hostent * Serverhost;
mtp_id_t Peer;
char buffer[MTP_SIZE_RESPONSE];
ping_t Ping;
fd_set Readfds[2],Writefds[2],Exceptfds[2];
static char *Id="mclient - 1.0 March 1999 - Seth Noble";

void
Start_Ping()
/*
**  Spawn a ping process to keep track of ambient packet loss.
**
**  Use pipe(2) to create a socket pair, then fork(2) and execl(3) to
pass
**  the pipe to sh for execution of ping.
*/
{
   static char *shell="/bin/sh";
/*   static char *sping="ping -q -t 1 %s >&%d";*/
   static char *fping="ping -qF %s >&%d";
   char pingstr[80];
   int fd[2];

   GUARD(err,pipe(fd),"pipe");;

   Ping.output = fd[0];

   if ((Ping.pid = fork())) {
      /* This is the original process */
      close(fd[1]);
      return;
   } else {
      /* This is the new child process */
      sprintf(pingstr,fping,Server,fd[1]);

#ifdef irix
      BSDsetpgrp(0,getpid());
#elif defined(solaris)
      setpgrp();
#else
      setpgrp(0,getpid());
#endif
```

```
      execl(shell,"sh","-c",pingstr,NULL);
   }
}  /* Start_Ping */

void
Stop_Ping()
/*
**  Halt the ping and collect its output
**
*/
{
   int size;
   char pout[512], *ptr;
#if defined(solaris)
   extern int killpg(pid_t pgrp, int sig);
#endif

   CHECK(err,killpg(Ping.pid,SIGINT),"killpg");
   waitpid(-1,NULL,0);
   size = read(Ping.output,pout,511);
   pout[511] = '\0';

   ptr = strchr(pout,'%');
   if (ptr == NULL) {
      fprintf(stderr,"Unexpected ping output %s\n",pout);
      exit(-1);
   }

   while (*ptr != ' ') ptr--;
   ptr++;

   Ping.result = atof(ptr);
   close(Ping.output);

}  /* Stop Ping */

void
mtpdone(mtp_id_t msg_id, mtp_request_t request, int error, int response,
        uint32 size, int final, mtp_delivery_t delivery, uint32 user,
        mtp_stats_t stats)
{
   if (error) {
      printf("\nGET failed: %s\n",MTP_Error(error,response));
      exit(-1);
   }

   if (final)  {
      printf("MTP: %lu\t%.1f\t%.1f\t%lu\t%.1f\t%lu/%lu/%lu\n",
             size,stats.etime/1000.0,(size/1024.0)/(stats.etime/1000.0),
             stats.avgdelay,
             stats.repeats*100.0/(stats.requests+stats.repeats),
             stats.minwindow,stats.avgwindow,stats.maxwindow);
      Mcount--;
   }
}

void
Start_MTP()
{
   mtp_request_t request;
```

```
    mtp_delivery_t delivery;

    request.action = ACT_DATA;
    request.flag = 0;
    request.channel = 0;
    request.address = 0;
    request.rsize = 0;
    delivery.type = MTP_DEL_FILE;
    delivery.handle = Null;
    delivery.offset = 0;
    delivery.bound = 0;

    MTP_Request(&Peer,request,0,NULL,2000,30000,delivery,0,0,mtpdone);

}   /* Start_MTP */

void
Start_TCP()
{
    struct sockaddr_in name = { PF_INET, Port, Peer.peerip };

    GUARD(TCP[TCPcount],socket(PF_INET,SOCK_STREAM,0),"TCP socket");
    CHECK(err,fcntl(TCP[TCPcount],F_SETFL,FNDELAY),"fcntl");
    err = connect(TCP[TCPcount],&name,sizeof(name));
    if (err && errno != EINPROGRESS) {
        perror("connect");
        abort();
    }

    FD_SET(TCP[TCPcount],&Readfds[1]);
    Size[TCPcount] = 0;
    TCPcount++;
}

void
Stop_TCP(int t)
{
    int end;

    end = time(NULL);

    printf("TCP: %d\t%d\t%.2f\n",
           Size[t], end-Start, (double)Size[t]/(1024.0*(end-Start)));
/*    printf("TCP read %d bytes in %d seconds\n",size,end - start);*/

    FD_CLR(TCP[t],&Readfds[1]);
    Tcount--;

    close (TCP[t]);

}   /* Run_TCP */

void
Get_Options(int argc, char **argv)
/*  INTERNAL
**  Parse the command line
*/
{
    int c;
    extern char *optarg;
```

```
   extern int optind;/*, opterr;*/
   Debug = 0; Port = 8080; Verbose = 0; Link = LINK_ETHERNET;
   Timeout = 60; Mcount = 1; Tcount = 1; Noping = 0;

   while ((c = getopt(argc,argv,"vd:p:t:l:m:q")) != -1)
     switch (c) {
      case 'd':                    /* Set debug output level */
        Debug = atoi(optarg);
        break;
      case 'p':                    /* Set peer udp/tcp port number */
        Port = atoi(optarg);
        break;
      case 'v':                    /* Verbose mode */
        Verbose = 1;
        break;
      case 'l':
        Link = atoi(optarg);
        break;
      case 'm':
        Mcount = atoi(optarg);
        break;
      case 't':
        Tcount = atoi(optarg);
        break;
      case 'q':
        Noping = 1;
        break;
      default:
        fprintf(stderr,"%s: %s\n",argv[0],Id);
        fprintf(stderr,
    "usage: %s [-v] [-d debug_level] [-p port] [-l link]
testhost\n",argv[0]);
        exit(1);
     }

   if (argc - optind != 1) {
      fprintf(stderr,
    "usage: %s [-v] [-d debug_level] [-p port] [-l link]
testhost\n",argv[0]);
      exit(-1);
   }

   Server = argv[optind];
   Serverhost = gethostbyname(Server);
   if (Serverhost == NULL) {
      fprintf(stderr,"Invalid server: %s\n",Server);
      exit(1);
   }

}  /* Get_Options */

int
main(int argc, char **argv, char **envp)
{
   int m,t;
   char buffer[8192];
   time_t now;

   Get_Options(argc,argv);
```

```
    FD_ZERO(&Exceptfds[1]);
    FD_ZERO(&Writefds[1]);
    FD_ZERO(&Readfds[1]);
    if (MTP_Init(0,Link,Id,Debug) < 0) {
       fprintf(stderr,"MTP_Init Failed\n");
       exit(-1);
    }
    CHECK(Null,open("/dev/null",O_WRONLY),"/dev/null");
    memcpy((char *)&Peer.peerip,Serverhost->h_addr_list[0],4);
    Peer.peerport = Port;
    Peer.appversion = 0; /*FILE_VERSION;*/

    now = time(NULL);
    printf("# MTP vs. TCP multiplicity test\n# %s  %s#\n",
           Server,ctime(&now));
    fflush(stdout);

    m = Mcount; t = Tcount;
    Start = time(NULL);
    if (!Noping)
      Start_Ping();
    while (m || t) {
       if (m) {
          Start_MTP();
          m--;
       }
       if (t) {
          Start_TCP();
          t--;
       }
    }

    while (Mcount || Tcount) {
       Exceptfds[0] = Exceptfds[1];
       Writefds[0] = Writefds[1];
       Readfds[0] = Readfds[1];
       MTP_Event(&Readfds[0],&Writefds[0],&Exceptfds[0],-1);

       for (t = 0; t<TCPcount; t++)
         if (FD_ISSET(TCP[t],&Readfds[0])) {
            GUARD(err,read(TCP[t],buffer,8192),"read");
            if (err == 0) Stop_TCP(t);
            else Size[t] += err;
         }
    }

    if (!Noping) {
       Stop_Ping();
       printf("PING: %.1f\n",Ping.result);
    }

    return 0;
}
```

Appendix F: Simulator

sim.h

```
/*
**  sim.h  -  Datastructures and headers for a simple TCP vs. MTP
simulation
**
*/

#include <stdio.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include <limits.h>
#include "sunos4.h"

#define MODE_NONE       0
#define MODE_MTP        1
#define MODE_TCP        2

#define STATE_NONE      0       /* Unused slot */
#define STATE_SENT      1       /* Request or datagram has been sent */

#define EVENT_NONE      0       /* Non-event */
#define EVENT_ARRIVE    1       /* Message arrives correctly */
#define EVENT_TIMEOUT   2       /* Message times-out after being dropped
*/
#define EVENT_LATE      3       /* Message times-out before arriving */
#define EVENT_IGNORE    4       /* Message arrives but is ignored */

#define TICKS           10000   /* Number of clock ticks per second */
#define KBYTES          1       /* Number of kilobytes per message */
#define DEFAULT_BW      1000    /* Default bandwidth in KB/s */
#define DEFAULT_CONG    1       /* Default to no third-party traffic */
#define DEFAULT_RTT     0.01    /* Default round trip time in seconds */
#define INIT_WINDOW     4       /* Initial window size */
#define BUFFER          8       /* Network buffer adjustment */
#define MAX_LOSS        0.7     /* Max loss in graph mode */

#define MAXMSG          128
#define MAXWINDOW       96

typedef struct {
   int          state;          /* State of this subrequest */
   int          id;             /* id number */
   int          ctime;          /* Time this was created */
   int          sendtime;       /* Time this was last sent */
   int          expect;         /* Time this message should arrive */
   int          event;          /* Next expected event */
   int          eventtime;      /* Time event will occur */
} slot_t;

typedef struct {
   double       lossrate;       /* Packet loss factor */
   int          bw;             /* Message delivery rate */
   int          rtt;            /* Round trip time */
   int          cong;           /* Congestion factor */
```

```
   int          period;         /* Minimum intermessage arrival time */
   int          maxload;        /* Maximum permissible load value */

   int          transit;        /* Number of pending arrivals */
   int          load;           /* Number of messages sent in past rtt/2
*/
   int          drop;           /* Number of pending timeouts */
   int          lastarrive;     /* Expected time of most future message
*/
   int          firstsend;      /* Earliest transiting send time */
   int          window;         /* Maximum messages in transit */
   int          total;          /* Total number of messages sent */
   int          received;       /* Total received */
   int          lost;           /* Total lost */
   int          maxwindow;
   int          late;           /* Messages arriving late */
   int          mindelay;       /* Shortest receive time */
   int          maxdelay;       /* Longest receive time */
   int          totdelay;       /* Total of receive times */
} stat_t;

#define dprintf(level,format,args...) \
  if (Debug >= (level)) printf(format , ## args);

extern slot_t   Msg[MAXMSG];
extern stat_t   Stat;
extern int      Debug, CurrT, Mode, Graph, Flow;

extern int Find_Blank();
```

sim.c

```
/*
**  sim.c  -  Simple simulation of TCP vs. MTP responses to data loss
**
*/

#include "sim.h"
#include "tcp.h"
#include "mtp.h"

static char *id = "sim - 1.3 February 1998 - Seth Noble";

/*
**  1.3  -  Add congestion parameter.
**          Assume RTT parameter is minimum, not observed, RTT.
**          Support TCP synchronicity.
**
**  1.2  -  Calculate load based on transit.
**          TCP employs exponential backoff.
**          Increase time resolution to 1/10,000 of a second.
**          Employ network buffer.
**          TCP uses fast/slow start algorithm.
**
**  1.1  -  Loss rate increases with transit.
**          TCP adjust window on every other ACK
**
**  1.0  -  Core error correction and flow control for TCP & MTP
**          Fixed transmission rate and loss rates for network.
*/

slot_t  Msg[MAXMSG];
stat_t  Stat;
int     Debug, CurrT, Mode, Graph, Flow, Packet;


int err;
#define CHECK(ret, code, error) \
        if (((ret) = (code)) < 0) {perror(error);}
#define GUARD(ret, code, error) \
        if (((ret) = (code)) < 0) {perror(error);exit(1);}
#define SENTRY(ret, code, error, tag) \
        if (((ret) = (code)) < 0) {perror(error);return(-tag);}

int
Find_Blank()
{
   int i;
   for (i= MAXMSG-1; i>=0; i--)
     if (Msg[i].state == STATE_NONE) return i;
   return -1;
}
```

```
void
Init_Stat1()
{
   bzero(&Stat,sizeof(stat_t));
   Stat.bw = DEFAULT_BW / KBYTES;
   Stat.cong = DEFAULT_CONG;
   Stat.rtt = DEFAULT_RTT * TICKS;
   Stat.lossrate = 0.0;
   Stat.mindelay = INT_MAX;
   Stat.window = INIT_WINDOW;
}

void
Init_Stat2()
{
   /* First we have the standard bandwidth product */
   Stat.maxload = Stat.bw * Stat.rtt / (Stat.cong * TICKS);

   /* Now we add a buffering factor */
/*   Stat.maxload += BUFFER;

   if (!Stat.maxload) Stat.maxload = 1;*/
   if (Stat.maxload < BUFFER) Stat.maxload = BUFFER;

   Stat.period = (Stat.cong*TICKS)/Stat.bw;
}

void
Report()
{
   int transmitted = Stat.received+Stat.lost;
/* double max = ((1.0 - Stat.lossrate) * CurrT *
Stat.window)/Stat.rtt;*/
/*   double max = (1.0 - Stat.lossrate) * transmitted;*/
   double max = (((double)Stat.bw * CurrT)/(Stat.cong*TICKS)) * (1.0-
Stat.lossra
te);

   printf("Elapsed Time: %0.1fs\n",(double)CurrT/TICKS);
   printf("Requested %d messages:\tReceived %d\tLost %d\tLate %d\n",
         Stat.total,Stat.received,Stat.lost,Stat.late);
   if (transmitted > 0 && max > 0)
     printf("Efficiency: %3d%%\tThroughput: %.1f\tUtilization:
%.1f%%\n",
            100*Stat.received/(transmitted),
            (100.0 * TICKS *Stat.received)/((double)CurrT*Stat.bw),
            (100.0*Stat.received)/max);
   if (Stat.received > 0)
     printf("Delay: %d/%d/%d\tMax Window: %d\n",
            Stat.mindelay,Stat.totdelay/Stat.received,Stat.maxdelay,
            Stat.maxwindow);
}
```

```
void
Plot()
{
   int transmitted = Stat.received+Stat.lost;
   double max = (((double)Stat.bw * CurrT)/(Stat.cong*TICKS)) * (1.0-
Stat.lossra
te);

   if (transmitted == 0 || max == 0) {
      fprintf(stderr,"Cannot compute statistics:  Transmitted %d, Max
%.0f\n",
              transmitted, max);
      return;
   }

   printf("%.2f\t%.2f\t%.2f\t%.2f\n",
          Stat.lossrate,(100.0*Stat.received)/transmitted,
          (TICKS * 100.0*Stat.received)/((double)CurrT*Stat.bw),
          (100.0*Stat.received)/max);
}

int
Get_Options(int argc, char **argv)
/*
**  Parse the command line
*/
{
   int c,EndT;
   char *usage = "usage:\n  %s [-vgp] [-m | -t] [-d debug] [-l loss
rate] [-b ba
ndwidth] runlength\n";
   extern char *optarg;
   extern int optind/*, opterr*/;

   Debug = 0; Mode = MODE_MTP; Graph = 0; Flow = 1; Packet = 0;

   while ((c = getopt(argc,argv,"hd:vmtl:w:b:r:c:gp")) != -1)
     switch (c) {
      case 'm':
        Mode = MODE_MTP;
        break;
      case 't':
        Mode = MODE_TCP;
        break;
      case 'd':                  /* Set debug output level */
        Debug = atoi(optarg);
        break;
      case 'l':
        Stat.lossrate = atof(optarg);
        break;
      case 'b':                  /* Maximum Bandwidth */
        Stat.bw = atoi(optarg) / KBYTES;
        break;
      case 'r':                  /* Minimum Round Trip Time in ms */
        Stat.rtt = atoi(optarg) * TICKS/1000;
        break;
      case 'c':                  /* Congestion Factor */
        Stat.cong = atoi(optarg);
        break;
      case 'w':                  /* Disable flow control */
```

```
        Stat.window = atoi(optarg);
        Flow = 0;
        break;
      case 'g':
        Graph = 1;
        break;
      case 'p':
        Packet = 1;
        break;
      case 'v':                    /* Display version number */
        fprintf(stderr,"%s: %s\n",argv[0],id);
        break;
      case 'h':
        fprintf(stderr,"%s: %s\n",argv[0],id);
        fprintf(stderr,usage,argv[0]);
        exit(0);
      default:
        fprintf(stderr,usage,argv[0]);
        exit(-1);
    }

  if (optind < argc)
    EndT = atoi(argv[optind]);
  else {
    fprintf(stderr,
            "usage: %s [-v] [-d debug_level] runtime\n",argv[0]);
    exit(-1);
  }

  return EndT;
}  /* Get_Options */

int
main(int argc, char **argv, char **envp)
{
  int EndT, NextT;
  double elow, ehi, einc, rate;
  stat_t Template;

  srand48(time(NULL));

  Init_Stat1();
  EndT = Get_Options(argc,argv);
  Init_Stat2();
  Template = Stat;

  if (Graph) {
    elow = 0.0; ehi = MAX_LOSS; einc = Stat.lossrate;
    setlinebuf(stdout);
  }
  else { elow = Stat.lossrate; ehi = Stat.lossrate+0.5; einc = 1.0; }

  if (Packet) {
    dprintf(2,"Simulation %d messages\n",EndT);
  } else {
    dprintf(2,"Simulation %ds\n",EndT);
    EndT *= TICKS;
  }

  for (rate = elow; rate < ehi; rate += einc) {
```

```
        Stat.lossrate = rate;
        bzero(Msg,sizeof(slot_t)*MAXMSG);
        switch (Mode) {
         case MODE_MTP:
           NextT = Init_MTP();
           break;
         case MODE_TCP:
           NextT = Init_TCP();
           break;
         default:
           abort();
        }
        for (CurrT = NextT;
             Packet?(Stat.received<EndT):(CurrT < EndT);
             CurrT = NextT) {
           dprintf(4,"==== Time: %d ===\n", CurrT);
           switch (Mode) {
            case MODE_MTP:
              NextT = Process_MTP();
              break;
            case MODE_TCP:
              NextT = Process_TCP();
              break;
           }
           if (Stat.window > Stat.maxwindow) Stat.maxwindow = Stat.window;
        }

        if (CurrT == INT_MAX) abort();
        if (Graph) {
           Plot();
           Stat = Template;
        } else Report();
     }

     return 0;
}
```

mtp.h

```
typedef struct {
   int          repeat;          /* Repeat interval */
   int          expire;          /* Transaction expiration interval */
   int          flowtimeout;     /* MTP flowtimeout variable */
} mtp_t;
```

mtp.c

```
extern int Init_MTP();
extern int Process_MTP();

#include "sim.h"
#include "mtp.h"

static mtp_t    Mtp;
static int lr/*,rbw*/;

int
Init_MTP()
/*
**  Initialize the MTP simulator to a steady state.
**
**  Space out arrival rate in proportion to bw: Send/arrive every 1/bw
seconds
**
**  NOTE:  MTP's actual repeat timer is based upon observed RTT
*/
{
   int i;

   Stat.transit = Stat.window;
   Mtp.repeat = 4*(Stat.rtt+Stat.period);
   if (Mtp.repeat < 500) Mtp.repeat = 500;
   Mtp.expire = 30*TICKS;
   Mtp.flowtimeout = 0;
   Stat.total = Stat.transit;

   for (i=0; i < Stat.transit; i++) {
      Msg[i].state = STATE_SENT;
      Msg[i].id = i;
      Msg[i].ctime = i*Stat.period;
      Msg[i].sendtime = Msg[i].ctime;
      Msg[i].event = EVENT_ARRIVE;
      Msg[i].eventtime = Msg[i].ctime + Stat.rtt;
      dprintf(3,"Init %d:  Sent %d, Arrives %d\n",
              i,Msg[i].sendtime, Msg[i].eventtime);
   }
   Stat.lastarrive = Msg[i-1].eventtime;
   lr = Stat.lossrate*INT_MAX;
/*   rbw = TICKS/Stat.bw;*/
/*printf("BW: %d\trbw: %d\n",Stat.bw,rbw);*/
   return Msg[0].eventtime;
}  /* Init_MTP */
```

```
static inline void
Calc_Load()
/*
**  Calculate the immediate network load by counting the number of
messages
**  which were sent more than rtt/2 ago but less than rtt.
**
**  Note that MTP requests are sent at their sendtime while the message
**  itself is sent rtt/2 later.  The message is received after rtt.
*/
{
/*
   int i, cutoff1=CurrT-Stat.rtt/2,cutoff2=CurrT-Stat.rtt;

   Stat.load = 0;

   for (i=0; i< MAXMSG; i++)
     if (Msg[i].state > STATE_NONE &&
         Msg[i].sendtime >= cutoff2 &&
         Msg[i].sendtime < cutoff1 &&
         Msg[i].event != EVENT_TIMEOUT)
       Stat.load++;
*/
   Stat.load = Stat.transit/2;
   if (Stat.load >= Stat.maxload && Debug >= 2)
     printf("MAXLOAD\n");
} /* Calc_Load */

static inline void
Decide_Message(int i)
/*
**  Decide whether or not a message will be delivered
*/
{
   int currbw;
   int dropit,earlyslot;

   if (Debug >= 3) {
      if (Stat.lastarrive > CurrT)
        currbw = TICKS * Stat.transit / (Stat.lastarrive - CurrT);
      else {
         /* This can happen even if transit is 1, provided this message
            has expired at the same the one in transit will arrive */
         if (Stat.transit > 1 ||
             (Stat.transit == 1 && Msg[i].event != EVENT_TIMEOUT)) {
            fprintf(stderr,"Bad arrival times\n");
            abort();
         }
         currbw = 0;
      }
      printf("\tTransit %d / (Last %d - %d) = Current BW %d\n",
              Stat.transit,Stat.lastarrive,CurrT,currbw);
   }

   Msg[i].expect = Msg[i].sendtime + Stat.rtt;
   earlyslot = Stat.lastarrive + Stat.period;
   if (Msg[i].expect < earlyslot) Msg[i].expect = earlyslot;

   dropit = (Stat.load >= Stat.maxload || lrand48() < lr);
```

```
/*
printf ("Load %d\tMaxload %d\tDrop?
%d\n",Stat.load,Stat.maxload,dropit);
printf ("Send %d\tRTT %d\tRepeat
%d\n",Msg[i].sendtime,Stat.rtt,Mtp.repeat);
*/
   if (dropit || Msg[i].expect > Msg[i].sendtime + Mtp.repeat) {
      Msg[i].event = EVENT_TIMEOUT;
      Msg[i].eventtime = Msg[i].sendtime + Mtp.repeat;
   } else {
      Msg[i].event = EVENT_ARRIVE;
      Msg[i].eventtime = Msg[i].expect;
   }

   if (dropit) Stat.drop++;
   else {
      /* We only count this message against available bandwidth if
         it is going to arrive. */
      Stat.transit++;
   }
   /* But we leave a hole even if it is dropped */
   if (Msg[i].expect > Stat.lastarrive) Stat.lastarrive = Msg[i].expect;

}  /* Decide_Message */

static void
New_Message(int i)
/*
**  Generate a new message and determine its fate
*/
{
   Msg[i].state = STATE_SENT;
   Msg[i].id = Stat.total;
   Msg[i].ctime = CurrT;
   Msg[i].sendtime = Msg[i].ctime;
   Decide_Message(i);
   Stat.total++;
   dprintf(2,"%.4d: Requested %.3d (will %s at %d)\n",

CurrT,Msg[i].id,(Msg[i].event==EVENT_TIMEOUT)?"timeout":"arrive",
           Msg[i].eventtime);
}  /* New_Message */

int
Process_MTP()
/*
**  Update the state of MTP messages
*/
{
   int i, next = INT_MAX, delay, new;

   /* Calc_Load();*/
   Stat.load = Stat.transit/2;
   if (Stat.load >= Stat.maxload && Debug >= 2)
     printf("MAXLOAD %d %d\n",Stat.load,Stat.maxload);

   for (i=0; i < MAXMSG; i++) {
      if (Msg[i].state > STATE_NONE) {
         if (Msg[i].eventtime < Msg[i].sendtime) {
            fprintf(stderr,"Bad Event/Send times\t%d %d\n",
```

```
                Msg[i].eventtime, Msg[i].sendtime);
            abort();
         }
         if (Msg[i].eventtime <= CurrT) {
            switch (Msg[i].event) {
             case EVENT_ARRIVE:
               Stat.received++;
               Stat.transit--;
               delay = CurrT - Msg[i].ctime;
               if (delay < Stat.mindelay) Stat.mindelay = delay;
               if (delay > Stat.maxdelay) Stat.maxdelay = delay;
               Stat.totdelay += delay;
               dprintf(2,"%.4d: Received %.3d after %d\n",
                     CurrT,Msg[i].id,delay);
               if (Flow) {
                  Mtp.flowtimeout--;
                  if (Mtp.flowtimeout < -2*Stat.window &&
                      Stat.window < MAXWINDOW) {
                     Stat.window++;
                     dprintf(1,"Raise_Flow %d\n",Stat.window);
                     if (Stat.window <= Stat.transit+Stat.drop) {
                        fprintf(stderr,"Still over window after
raise\n");
                        abort();
                     }
                     new = Find_Blank();
                     if (new < 0) {
                        fprintf(stderr,"Message Table Full!\n");
                        abort();
                     }
                     New_Message(new);
                     if (Msg[new].eventtime < next) next =
Msg[new].eventtime;
                     Mtp.flowtimeout = 0;
                  }
               }

               if (Stat.transit+Stat.drop <= Stat.window) {
                  New_Message(i);
               } else {
                  Msg[i].state = STATE_NONE;
                  Msg[i].eventtime = INT_MAX;
               }
               break;
             case EVENT_TIMEOUT:
               Stat.lost++;
               Stat.drop--;
               dprintf(1,"%.4d: Lost %.3d\n",CurrT,Msg[i].id);

               if (Msg[i].expect > CurrT) {
                  /* This message is still in transit */
                  new = Find_Blank();
                  if (new < 0) {
                     fprintf(stderr,"Message Table Full!\n");
                     abort();
                  }
                  Msg[new] = Msg[i];
                  Msg[new].event = EVENT_LATE;
                  Msg[new].eventtime = Msg[new].expect;
               }
```

```
            if (Flow) {
               Mtp.flowtimeout--;
               if (Mtp.flowtimeout < 0 && Stat.window > 1) {
                  Stat.window--;
                  dprintf(1,"Lower_Flow %d (%d)\n",
                          Stat.window,Stat.transit);
               }
               if (Stat.window > 3)
                 Mtp.flowtimeout = Stat.window;
               else
                 Mtp.flowtimeout = 3;
            }

            Msg[i].sendtime = CurrT;
            Decide_Message(i);
            break;
          case EVENT_LATE:
            Stat.transit--;
            Stat.late++;
            dprintf(1,"%.4d: Late %.3d\n",CurrT,Msg[i].id);
            Msg[i].state = STATE_NONE;
            break;
          default:
            fprintf(stderr,"Double Crap!\n");
            abort();
         } /* switch */
       } /* if */
       if (Msg[i].state > STATE_NONE &&
           Msg[i].eventtime < next) next = Msg[i].eventtime;
    } /* if */
  } /* for */
  return next;
}  /* Process_MTP */
```

tcp.h

```
typedef struct {

   int          repeat;         /* Repeat interval */
   int          expire;         /* Transaction expiration interval */
   int          dropcount;      /* Count of datagrams that will be lost
*/
   int          flowtimeout;
   int          backoff;        /* Seconds to wait for retransmission */
   int          ssthresh;       /* Slow-start threshold */
} tcp_t;

extern int Init_TCP();
extern int Process_TCP();
```

tcp.c

```
#include "sim.h"
#include "tcp.h"

static tcp_t    Tcp;
static int lr/*,rbw*/;

#define TCPTV_MIN       (1)     /* Minimum repeat interval */
#define TCPTV_REXMTMAX  (64)    /* Maximum repeat interval */
#define WINDOW_MIN      (1)     /* Minimum window size */

int
Init_TCP()
/*
**  Initialize the MTP simulator to a steady state.
*/
{
   int i;

   Stat.transit = Stat.window;
   Tcp.repeat = TCPTV_MIN*TICKS;
   Tcp.expire = 3*60*TICKS;
   Tcp.dropcount = 0;
   Tcp.backoff = 1;
   Tcp.ssthresh = 2;
   Stat.total = Stat.transit;

   /* SYNCRHONICITY EFFECT */
   Stat.maxload /= Stat.cong;
   if (Stat.maxload < 1) Stat.maxload = 1;

   for (i=0; i < Stat.transit; i++) {
      Msg[i].state = STATE_SENT;
      Msg[i].id = i;
      Msg[i].ctime = i*Stat.period;
      Msg[i].sendtime = Msg[i].ctime;
      Msg[i].event = EVENT_ARRIVE;
      Msg[i].eventtime = Msg[i].ctime + Stat.rtt;
      dprintf(3,"Init %d:  Sent %d, Arrives %d\n",
```

```
                i,Msg[i].sendtime, Msg[i].eventtime);
    }
    Stat.lastarrive = Msg[i-1].eventtime;
    lr = Stat.lossrate*INT_MAX;
/*    rbw = TICKS/Stat.bw;*/
    return Msg[0].eventtime;
}  /* Init_TCP */

static inline void
Calc_Load()
/*
**  Calculate the immediate network load by counting the number of
messages
**  which were sent in the past rtt/2.
**
**  Note that TCP messages are sent at their sendtime arrive rtt/2
later.
**  The ACK is received after rtt.
**
*/
{
/*
    int i, avgrtt, cutoff;

    if (Stat.lastarrive > CurrT) {
       avgrtt = (Stat.lastarrive - CurrT)/2;
       cutoff=CurrT-avgrtt/2;
    } else
      cutoff = CurrT -Stat.rtt/2;

    Stat.load = 0;

    for (i=0; i< MAXMSG; i++)
      if (Msg[i].state > STATE_NONE &&
          Msg[i].sendtime >= cutoff &&
          Msg[i].event != EVENT_TIMEOUT)
        Stat.load++;
*/
    Stat.load = Stat.transit/2;
/*printf("LOAD %d\n",Stat.load);*/
    if (Stat.load >= Stat.maxload && Debug >= 2)
      printf("MAXLOAD\n");
} /* Calc_Load */

static inline void
Decide_Message(int i)
/*
**  Decide whether or not a message will be delivered
*/
{
    int currbw;
    int dropit,earlyslot;

    if (Debug >= 3) {
       if (Stat.lastarrive > CurrT)
         currbw = TICKS * Stat.transit / (Stat.lastarrive - CurrT);
       else {
          /* This can happen even if transit is 1, provided this message
             has expired at the same the one in transit will arrive */
          if (Stat.transit > 1 ||
```

```
            (Stat.transit == 1 && Msg[i].event != EVENT_TIMEOUT)) {
            fprintf(stderr,"Bad arrival times\n");
            abort();
         }
         currbw = 0;
      }
      printf("\tTransit %d / (Last %d - %d) = Current BW %d\n",
             Stat.transit,Stat.lastarrive,CurrT,currbw);
   }

   Msg[i].expect = Msg[i].sendtime + Stat.rtt;
   earlyslot = Stat.lastarrive + Stat.period;
   if (Msg[i].expect < earlyslot) Msg[i].expect = earlyslot;
   dropit = (Stat.load >= Stat.maxload || lrand48() < lr);

   if (dropit || Msg[i].expect > Msg[i].sendtime + Tcp.repeat) {
      /* Message cannot be delivered in time */
      Msg[i].event = EVENT_TIMEOUT;
      Msg[i].eventtime = Msg[i].sendtime + Tcp.repeat;

   } else {
      /* Message may be delivered on time */
      Msg[i].event = EVENT_ARRIVE;
      Msg[i].eventtime = Msg[i].expect;
   }

   if (Tcp.dropcount) {
      /* This message is part of a window that will not be acknowledged
*/
      if (dropit) {
         /* This is a non-event.  The message will never arrive and the
            receiver will never notice. */
         Msg[i].state = STATE_NONE;
         Msg[i].event = EVENT_NONE;
         Msg[i].eventtime = INT_MAX;
      } else {
         /* The message will arrive (consuming bandwidth) but be ignored
*/
         Msg[i].event = EVENT_IGNORE;
         Msg[i].eventtime = Msg[i].expect;
      }
/*      Tcp.dropcount--;*/
   } else if (dropit || Msg[i].expect > Msg[i].sendtime + Tcp.repeat) {
      /* Timeout will cause loss of the rest of the window */
      Tcp.dropcount = 1; /*Stat.window-1;*/
      /* Note that it is possible for a late ACK to arrive in the
         middle of transmitting a redundant window.  It's not clear
         what the typical TCP behavior would be, but it is very
         unlikely to happen, so we ignore it. */
   }

   if (Msg[i].state == STATE_NONE) return;

   if (dropit) Stat.drop++;
   else {
      /* We only count this message against available bandwidth if
         it is going to arrive. */
      Stat.transit++;
   }
   /* But we leave a hole even if it is dropped */
```

```
      if (Msg[i].expect > Stat.lastarrive) Stat.lastarrive =
Msg[i].expect;

}  /* Decide_Message */
```

```
static void
New_Message(int i)
/*
**  Generate a new message and determine its fate
*/
{
   char *event;

   Msg[i].state = STATE_SENT;
   Msg[i].id = Stat.total;
   Msg[i].ctime = CurrT;
   Msg[i].sendtime = Msg[i].ctime;
   Decide_Message(i);
   Stat.total++;
   if (Debug >= 2) {
      switch (Msg[i].event) {
       case EVENT_NONE:
         event = "never arrive";
         break;
       case EVENT_ARRIVE:
         event = "arrive";
         break;
       case EVENT_TIMEOUT:
         event = "timeout";
         break;
       case EVENT_LATE:
         event = "arrive late";
         break;
       case EVENT_IGNORE:
         event = "be ignored";
         break;
       default:
         event = "UNKNOWN EVENT!!!!!";
      }
      printf("%.4d: Sending %.3d (will %s at %d)\t[dc %d]\n",
            CurrT,Msg[i].id,event,Msg[i].eventtime,Tcp.dropcount);
   }
}  /* New_Message */

int
Process_TCP()
{
   int i, j, next = INT_MAX, delay, new, growthresh;

   /* Calc_Load();*/
   Stat.load = Stat.transit/2;
   if (Stat.load >= Stat.maxload && Debug >= 2)
     printf("MAXLOAD %d %d\n",Stat.load,Stat.maxload);

   for (i=0; i < MAXMSG; i++) {
      if (Msg[i].state > STATE_NONE) {
         if (Msg[i].eventtime < Msg[i].sendtime) {
            fprintf(stderr,"Bad Event/Send times\t%d %d\n",
                    Msg[i].eventtime, Msg[i].sendtime);
            abort();
         }
         if (Msg[i].eventtime == INT_MAX) {
            fprintf(stderr,"Bad send time\n");
            abort();
         }
```

```
            if (Msg[i].eventtime <= CurrT) {
               switch (Msg[i].event) {
                case EVENT_ARRIVE:
                  /* This data was accepted and acknowledged */
                  Stat.received++;
                  Stat.transit--;
                  delay = CurrT - Msg[i].ctime;
                  if (delay < Stat.mindelay) Stat.mindelay = delay;
                  if (delay > Stat.maxdelay) Stat.maxdelay = delay;
                  Stat.totdelay += delay;
                  dprintf(2,"%.4d: Received ACK %.3d after %d\n",
                          CurrT,Msg[i].id,delay);
                  if (Flow) {
                     Tcp.flowtimeout--;
                     if (Tcp.backoff > 1) {
                        Tcp.backoff = 1;
                        Tcp.repeat = TCPTV_MIN*TICKS;
                     }

                     /* It appears that the typical slow start threshold is
                        set equal to the minimum window size, meaning that
                        the exponential growth will never happen. */
                     if (Stat.window > Tcp.ssthresh)
                       growthresh = -Stat.window;
                     else
                       growthresh = -1;
                     if (Tcp.flowtimeout <= growthresh &&
                         Stat.window < MAXWINDOW) {
                        Stat.window++;
                        dprintf(1,"Raise_Flow %d\n",Stat.window);
                        new = Find_Blank();
                        if (new < 0) {
                           fprintf(stderr,"Message Table Full!\n");
                           abort();
                        }
                        New_Message(new);
                        if (Msg[new].eventtime < next) next =
Msg[new].eventtime;
                        Tcp.flowtimeout = 0;
                     }
                  }

                  /* Send the next datagram */
                  if (Stat.transit+Stat.drop <= Stat.window) {
                     New_Message(i);
                  } else {
                     Msg[i].state = STATE_NONE;
                     Msg[i].eventtime = INT_MAX;
                  }
                  break;
                case EVENT_TIMEOUT:
                  Stat.lost++;
                  dprintf(1,"%.4d: Lost %.3d\n",CurrT,Msg[i].id);

                  if (Msg[i].expect > CurrT) {
                     /* This message is still in transit */
                     Msg[i].event = EVENT_LATE;
                     Msg[i].eventtime = Msg[i].expect;
                  } else {
                     /* The message is simply gone */
```

```
                    Msg[i].state = STATE_NONE;
                    Msg[i].eventtime = INT_MAX;
                    Stat.drop--;
                 }

                 if (Flow /*&& !Tcp.dropcount*/) {
                    Tcp.ssthresh = Stat.window/2;
                    if (Tcp.ssthresh < 2) Tcp.ssthresh = 2;
                    Stat.window = WINDOW_MIN;

                    if (Tcp.repeat < TCPTV_REXMTMAX * TICKS) {
                       Tcp.backoff = Tcp.backoff<<1;
                       Tcp.repeat = Tcp.backoff * TICKS;
                    }

                    dprintf(1,"Lower_Flow %d (%d+%d) < %d\n",
Stat.window,Stat.transit,Stat.drop,Tcp.ssthresh);

                    Tcp.flowtimeout = 0;
                 }
                 Tcp.dropcount = 0;

                 /* Now we have to resend the entire window */
                 /* The repeated messages should have been marked for
IGNORE */
                 for (j=0; j < Stat.window; j++) {
                    new = Find_Blank();
                    if (new < 0) {
                       fprintf(stderr,"Message Table Full!\n");
                       abort();
                    }
                    New_Message(new);
                    if (Msg[new].eventtime < next) next =
Msg[new].eventtime;
                 }
                 break;
              case EVENT_LATE:
                 Stat.transit--;
                 Stat.late++;
                 dprintf(1,"%.4d: Late %.3d\n",CurrT,Msg[i].id);
                 Msg[i].state = STATE_NONE;
                 break;
              case EVENT_IGNORE:
                 Stat.transit--;
                 dprintf(1,"%.4d: Ignored %.3d\n",CurrT,Msg[i].id);
                 Msg[i].state = STATE_NONE;
                 break;
              default:
                 fprintf(stderr,"Double Crap!\n");
                 abort();
              } /* switch */
           } /* if */
           if (Msg[i].state > STATE_NONE &&
               Msg[i].eventtime < next) next = Msg[i].eventtime;
        } /* if */
     } /* for */
     return next;
}  /* Process_TCP */
```