INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Implementing XTP in an IP Environment

Vince Lo Faso

A Thesis

in

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREEE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

APRIL 2001

ABSTRACT

IMPLEMENTING XTP IN AN IP ENVIRONMENT

Vince Lo Faso

XTP is a transport level protocol that can operate over several network and data link layer protocols. This paper examines the issues involved in implementing XTP version 4.0 over the IP network protocol. We investigate transport related issues that are unique to XTP, examine the behavior and performance of XTP/IP in a Linux kernel implementation (LinuxXTP), recommend several enhancements to the XTP Specification revision 4.0, and introduce Transport Traffic Services for XTP for the most common Internet traffic types. Comparisons to TCP are made to highlight the relative strengths and weak points of XTP.

Acknowledgements

I dedicate this thesis in memory of my father, Filippo, and in honor of my mother, Gaetana.

Dedico questa tesi alla memoria di mio padre Filippo, e alla mia cara mamma Gaetana.

I want to acknowledge and sincerely thank Dr. Atwood for introducing me to XTP and for his patience in taking a vague thesis idea and seeing it through to the present work. I want to thank both Dr. Atwood and Halina Monkiewicz (Graduate Secretary) for their assistance in completing the Master program while I lived away from Montreal.

I dearly thank all of my family and close friends for their support and interest, and for all of the times they have asked "So are you done yet?". Special thanks goes to Andrea Nimer for her patience, support, and understanding during the last portion of this work.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

The Xpress Transport Protocol (XTP) is a transport level protocol that can operate over a layer 3 network protocol, such as IP, or directly over a layer 2 delivery mechanism, such as Ethernet, FDDI, and ATM. Initially, XTP was designed as a "transfer" layer protocol, covering both transport and network layer functionality [Ches89, Ches91, Ches92, SDW92]. As of revision 4.0 of the XTP Specification [Stra95], XTP is exclusively a transport protocol [Weav1, Weav2].

XTP has a rich set of protocol mechanisms, enabling it to provide semantics that are usually found in separate protocols such as TCP (connection oriented services) [Post81], UDP (unreliable datagram service) [Post80], VMTP (transaction type service) [Cher88], and NETBLT (bulk data transfer service) [CLZ87]. XTP is an ideal candidate for supporting many of today's Internet traffic types because it is a very flexible and customizable protocol and because it can adapt to the needs of different application layer requirements. Although XTP has existed for several years, there is a dearth of experience and implementation of XTP (more specifically XTP/IP) as a transport protocol for Internet traffic such as HTTP, RPC, SMTP, DNS, SNMP, FTP and TELNET. This is due to XTP's origins as a "transfer" protocol and its focused development in the areas of non traditional Internet traffic such as multicasting and multimedia traffic. This thesis explores the suitability of XTP/IP for the more common Internet traffic.

1.1 Motivation of Thesis

The original impetus of this thesis was whether XTP is a better suited protocol for HTTP than TCP. The need for a better transport protocol for HTTP traffic stemmed from the body of research that highlighted poor interaction between HTTP and TCP. A preliminary review of the XTP Specification indicated that XTP is better suited than TCP for short request/response type traffic, which characterizes HTTP traffic. However, after closer examination and some implementation experience with LinuxXTP, a kernel-based implementation of XTP, it became apparent that XTP requires additional enhancements; namely a congestion control algorithm tailored for IP environments, and clear recommendations on how to utilize existing protocol features. As work progressed, we expanded the scope of XTP's suitability as a transport protocol for other traffic patterns. We selected four types of traffic patterns that are representative of today's Internet traffic: short request/response, short file transfers, multiple request/response, and long file transfers. Hence this thesis now investigates what is needed to safely and optimally implement LinuxXTP in an IP environment for these four traffic types.

1.2 Scope of Thesis

We begin our work by reviewing, in Chapter 2, the XTP Specification from which the LinuxXTP implementation is based. In Chapter 3 we survey recent and previous research activity with XTP. We then present, in Chapter 4, the need for a protocol such as XTP by examining the poor interaction between TCP and today's application requirements. We conclude that chapter with a discussion of XTP's own shortcomings as they pertain to

XTP/IP. Chapter 5 presents the core contribution of this thesis, the enhancements needed for a proper XTP/IP implementation. We introduce a kernel-based XTP implementation, LinuxXTP, present Connection Management and Congestion Management Policies for XTP, and provide a new framework for configuring XTP protocol features called XTP Transport Traffic Services. In Chapter 6 XTP's performance and behaviour is analyzed and compared with TCP. Chapter 7 summarizes our findings and suggests future research areas.

Chapter 2 Overview of Specification 4.0

Revision 4.0 of the XTP Specification [Stra95] changed the XTP protocol exclusively to a transport layer protocol. Several other changes, such as 64 bit fields, new header fields, and elimination of certain packet types were introduced as well. In this chapter we review the XTP Specification to provide a meaningful background to the work presented in this thesis.

2.1 Protocol Walk Through

XTP's fundamental structure is a packet. It consists of a fixed size header of 32 bytes with a variable payload size. There are 7 types of packets, each fulfilling a specific role or function. XTP associations (nomenclature for connections) can be connection oriented as in TCP or datagram based as in UDP. XTP's association setup, data transfer, and teardown phases can be completely separate and distinct, or they can be combined to minimize the number of packet exchanges. For example, when the association selects a combined phase approach, user data is carried in the setup handshake. For user data that is less than maxdata size (maximum packet size) we have the optimum scenario of transmitting one packet with the setup information, user data, and the teardown bit. The receiver responds with one packet to acknowledge the setup, the data received, and the teardown request. With the exchange of two XTP packets, a fully reliable connection is completed. This same connection can also be accomplished with separate phases, which can require up to 9 XTP packets.

An XTP instance is called a context (similar to a TCP process control block). We usually speak of two XTP contexts communicating across the network. XTP contexts have state information and are identified by the XTP Key. The Key is guaranteed to be unique within a given host. Each context manages an inbound queue and an outbound queue.

A major distinguishing feature of XTP is that it is primarily a sender-based controlled protocol. That is, the sender dictates when and how acknowledgements are sent, and when and how data is retransmitted. Unlike TCP, data bearing packet are not automatically protected with a timer. The sender determines which packets will have a timer activated. Also, retransmission of data bearing packets is not automatic when a timeout occurs. Rather than immediately retransmitting timed out packets, the sender "probes" the receiver via the Synchronizing HandShake algorithm.

2.2 XTP Packet Types

XTP packets can be categorized into two broad groups, INFORMATION type packets and CONTROL type packets. INFORMATION packets can carry either user data or protocol diagnostic messages. CONTROL packets carry protocol state information. There are 3 types of INFORMATION packets: FIRST, DATA, and DIAG packets. The FIRST packet is the initial packet sent to initiate an XTP association and is sent only once. DATA packets are sent to transfer user data. DIAG packets are used to transmit protocol error codes and messages, which may be passed on to the end-user application. The CONTROL type packets consist of four packets, CNTL, ECNTL, TCNTL, and JCNTL. The CNTL packets are referred to as Common Control packets. They carry protocol state information about the association. These acknowledgements are usually used to acknowledge received data and

update the flow control or rate control parameters. If there is missing data then an ECNTL packet, Error Control, is sent instead. XTP permits selective acknowledgements (similar to TCP's recent SACK option) or GoBackN acknowledgements (TCP's original acknowledgement mechanism). The TCNTL packets, Traffic Control, carry address information and traffic information, which may optionally carry explicit rate control parameters. TCNTL packets are usually sent in response to a FIRST packet, completing XTP's two way setup handshake. The JCNTL packet is part of XTP's multicast feature and is defined in [Ment98].

2.3 XTP Header

XTP uses a fixed size header 32 bytes long. There are 7 header fields:

KEY Field (64 bits): the KEY field is an unique identifier for a XTP context. Note that this is *not* equivalent to the port number in TCP or UDP. XTP keys are not service port numbers. A key usually serves as a direct index into a host's table of XTP contexts. A key and the sender's IP address uniquely identify a sender. During the setup phase, each context exchanges keys. Therefore, a sender can insert the receiver's key rather than its own key into an XTP packet. Such a key is call a "return key". A return key is identified by setting the most significant bit to 1, called the RETURN bit. When the receiving lookup process parses the key field and finds a return key, it knows that the key identifies a local XTP context and therefore quickly indexes into its table of XTP contexts.

COMMAND Field (32 bits): the COMMAND field has 2 major subfields, the PTYPE subfield (8 bits) and the OPTIONS subfield (24 bits). The PTYPE is further subdivided into PFORMAT field (5 bits) which identifies the packet type, and VER field (3 bits) which contains the protocol version number. For XTP Specification revision 4.0 the VER value must be "001". The OPTIONS subfield contains a reserved 8 bit field and 16 bits of "options". This is similar to TCP's option flags. XTP's OPTIONS control a variety of features and mechanisms. Fifteen of the sixteen OPTIONS bits are defined. They are the following:

NOFLOW: if set indicates that the association does not observe flow control

NOERR: when set indicates that the association performs no error checking, i.e., unreliable service

END: terminates association, similar to TCP's RST bit

EOM: "end of message" indicator

SREQ "status request", triggers a control packet from the receiver

DREQ "data status request", triggers a control packet from the receiver after the data is delivered to the end user

FASTNAK "fast negative acknowledgement", when set a receiver can inform the sender of gaps in the data stream independently of the SREQ bit

WCLOSE: "writer close", the transmitter is requesting to terminate its outbound data stream

RCLOSE: "reader close", acknowledges the transmitter that the receiver has successfully received all data according to the associations error checking policy and it is closing its inbound data stream

EDGE: triggers a control packet from the receiver when its value alternates with its last recorded value

NOCHECK: controls whether the checksum is performed only for the header (if set) or the whole packet (header and payload)

BTAG: if set indicates that the first 8 bytes of data the segment become the btag field

RES: controls whether the ALLOC field refers to actual client buffer space

MULTI: multicast bit to indicate a multicast association

SORT: if set indicates that the SORT field has a valid value

DLEN Field (32 bits): the DLEN field is the payload's length in bytes.

CHECK Field (16 bits): contains an IP checksum of either the XTP header (32 bytes) or of the entire XTP packet (header+payload). This is determined by the setting of the NOCHECK bit field.

SORT Field (16 bits): priority value of a packet, which may be used for delivery priorities.

SYNC Field (32 bits): every time the SREQ bit is set on outgoing packets, the context's SYNC value is incremented by one. The new value is placed into the SYNC field of all outbound packets. When the receiver responds to the SREQ bit, its ECHO should have the same value as the SYNC field. This can be used to match responding control packets to the requesting packet.

SEQ Field (64 bits): for INFORMATION type packets, the sequence number of the first byte in the payload of a FIRST or DATA packet. For CONTROL type packets, the expected sequence number for the reverse data stream.

2.4 XTP Association Types

An XTP association is an active connection between two XTP contexts. Although the XTP Specification does not explicitly discuss association "types", XTP permits 4 types of association. The 3 phases of a connection, Setup Phase (S), Data Transfer Phase (D), and Teardown Phase (T), can all be separate and distinct phases or a mixture of combined phases. The 4 possible association types are:

1) SDT: all three phases are combined

2) S_D_T: all 3 phases are separate and distinct from each other

3) SD_T: the setup and data transfer phases are combined, but the teardown phase is distinct and separate

4) S_DT: the setup phase is separate and the data transfer and teardown phases are combined.

When a connection phase is separate, the context state MUST be synchronized at both ends before the next phase is begun. As we will show, the selection of association types affects the number of XTP packets exchanged and therefore the performance and efficiency of the association. The association type is determined by the sequence of Command Option bit settings.

2.4.1 Association Setup Handshake

During the connection setup phase, several state parameters are exchanged between two contexts. Usually these are their respective keys, traffic parameters, address information, and possibly rate control information.

XTP associations are initiated with a FIRST packet (F pk). When a F pk arrives, the receiving host performs a First Packet Matching (FPM) algorithm which searches for any listening contexts that can accept the F pk. If there is a matching context, then an Address Translation Map is recorded, where the source IP address and the key field in the F pk are mapped to the listening context's key. The listening context responds with a Traffic Control packet (T pk) and places its key value in the XKEY field. In future packet exchanges, the key field in the XTP header will have the destination's local key value with the RETURN bit set. When such a packet is received, the RETURN key will be used to quickly index into the host's table of XTP contexts. If future packets do not utilize the RETURN bit, then a Full Context Lookup takes place. The packet's source IP and KEY field are hashed into the Address Translation Map table to find the appropriate XTP context.

2.4.2 Data Transfer

Data is transferred with DATA packets (D pk). To receive acknowledgement of successfully received data, the sender sets the SREQ bit in the outgoing D pk. The SREQ bit starts a "wait timer" WTIMER, and forces the receiver to respond with a CONTROL type packet. If there are no gaps in the inbound sequence, then a Common Control packet (C pk) is sent. This will acknowledge the highest contiguous sequence number (RSEQ) the receiver has seen, and inform the sender of an updated sliding window value via the ALLOC field which

is the highest sequence the receiver is willing to accept. If there are gaps in the inbound queue, then an Error Control packet (E pk) is sent. The E pk packet will indicate which byte sequence numbers the receiver has received. If this is an unreliable connection (NOERR bit is set), then C pks are sent even if gaps exist in the inbound data queue.

2.4.3 Association Teardown Handshake

An XTP association is terminated by closing its outbound and inbound data streams and the transmission of the END bit. When a sender is finished transmitting data, it closes its outbound data stream by setting the WCLOSE bit in the next outbound packet. The receiver will check its error checking policy for this association. If there is no missing data, then the receiver responds with a RCLOSE bit in a C pk, else it sends an E pk. Upon receiving the RCLOSE the sender knows that it can permanently shut down its outbound queue. The sender can then wait for the receiver to initiate teardown of its outbound queue, or it can unilaterally terminate the association by sending a C pk with the END bit set. Any XTP context that receives an END bit must terminate its context and notify the user application. The context sending the END bit goes into a "ZOMBIE" state. [Stra95b] details the expected packet sequence of these 4 teardown methods.

The XTP Specification outlines four possible teardown methods:

1) GRACEFULCLOSE MODE: Each context closes its outbound queue independently and asynchronously from the other. To close the outbound queue, a context sets the WCLOSE bit (and the SREQ bit to force a response) on an outbound packet. If there are no gaps, then the receiving context responds with a C packet and the RCLOSE bit set, else an E packet is sent to indicate missing data. When the RCLOSE is received it confirms the closure of the outbound

queue. The context now waits for its peer context to perform its own outbound queue closure. Once both queues are closed either context can send the END bit to terminate the association.

2) ABRREVCLOSE MODE: This mode is similar to GRACEFULCLOSE in that both perform an independent and asynchronous closure of their outbound queue. The difference lies in the sending of the END bit. Rather than waiting for both RCLOSE bits to be set before sending the END bit in a separate packet, the END bit is piggybacked with the last RCLOSE bearing packet. This abbreviated closure ensures that the context initiating the teardown will be the context that ends up sending the END bit.

3) FORCEDCLOSE MODE: This mode requires only one outbound queue to be gracefully closed. To close an outbound queue, a context sets the WCLOSE bit and waits for a RCLOSE bit in the responding packet. When the RCLOSE packet is received, the context transmits the END bit and therefore terminates the association without waiting for its peer context to gracefully close its outbound queue.

4) ABORTCLOSE MODE: This mode permits a quick and ungraceful termination of an association by the transmission of an END bit without any graceful closure of either context's outbound queues. This occurs when there is a need to abort an association.

2.5 XTP Timers

XTP uses timers to track lost packets and lost associations. XTP has 3 types of timers: WTIMER, CTIMER, and CTIMEOUT. There is a fourth timer, rate timer (RTIMER), which is used for the rate control mechanism. RTIMER does not affect the reliability requirements of the data or association.

2.5.1 WTIMER

The WTIMER (wait timer) is used to track lost packets. Anytime the SREQ bit is set on an outbound packet, the WTIMER is started. If the WTIMER times out before the acknowledge arrives, then the packet is presumed to be lost. XTP does not automatically retransmit a lost packet. Rather it goes into a Synchronizing HandShake algorithm with its peer context to ensure that it is still available and responsive.

2.5.2 CTIMER

The CTIMER (connection timer) is a keep-alive timer for an XTP association. It is started at the establishment of an XTP association. When the CTIMER times out, it checks if the context has received any packets from it peer context. If it has, it resets the packet count and restarts the CTIMER. If it hasn't received any packets then it initiates the Synchronizing HandShake algorithm.

2.5.3 CTIMEOUT

The CTIMEOUT (context timeout) timer puts an upper bound on how long the Synchronizing HandShake can last. The Synchronizing HandShake is executed anytime

there is a WTIMER timeout. Rather than automatically retransmit a timed out packet, XTP probes the peer context. It sends a C pk with the SREQ bit set and starts the CTIMEOUT timer. If the receiver responds successfully, the data transfer resumes and the CTIMEOUT timer is cancelled. If the C pk times out, the context resends the C pk up to K retries. For each retry, the WTIMER is doubled. The context ends its Synchronized HandShake when it reaches the K retry limit or the CTIMEOUT expires.

There is a special case of the CTIMEOUT during the teardown phase. The context that sends the END bit must protect that packet against any loss. XTP reuses the CTIMEOUT by re-arming the timer with a new value and moves the context into a "ZOMBIE" state. Upon its timeout the XTP context releases itself.

2.6 Flow Control

XTP uses a sliding window flow control mechanism similar to TCP's flow control window. The ALLOC field in control type packets inform the sender about the upper bound SEQ number value that the receiver is willing to accept. The sender MUST NOT send this SEQ number or any greater value. The RSEQ field informs the sender about the highest contiguous SEQ number the receiver has seen. The window is calculated as ALLOC – RSEQ.

The SEQ field in the DATA packets contain the SEQ number of the first byte in the payload. Since all XTP associations start with number SEQ 0, the SEQ field in the FIRST packet has a special interpretation. It indicates to the receiver, the sender's initial window size.

2.7 Error Control

XTP can provide reliable and unreliable data delivery semantics. This is determined by the setting of the NOERR option bit. For data retransmission, XTP provides two mechanisms, Selective Retransmission and GoBackN Retransmission.

When a receiver sends an E packet to inform the sender of lost data, it has the choice of using either retransmission policy. When a sender processes the E packet it has the option of retransmitting data selectively or from the first missing sequence byte number. In XTP it is possible for a receiver to use a Selective Retransmission Policy and for the sender to use a GoBackN Retransmission Policy.

Chapter 3 Related XTP Research

Since XTP has been exclusively a transport protocol only since [Stra95], much of the existing research literature is about XTP as a 'transfer' protocol. Nevertheless, the insight and experienced gained during its development as a transfer protocol are relevant for XTP 4.0.

3.1 Implementations

Several researchers have implemented and tested the XTP Specification. [SGC94] introduces SandiaXTP, a publicly available, user-level object-oriented implementation of XTP 3.6. SandiaXTP uses a C++ library called Meta-Transport Library (MTL). Using MTL, the authors implement SandiaXTP, and for testing purposes a raw IP Service. An interesting observation is made from latency performance tests between SandiaXTP and the raw IP Service. The authors report that XTP processing adds about 2 ms for packet sizes less than 1448 bytes, and add about 4 ms for packet sizes greater than 1448 bytes. This is attributed to the segmenting and reassembling of data at the maxdata size boundary. [Mapp95] analyzes the performance of SandiaXTP over ATM. Tests were done using Synchronous Mode with Reliable and Unreliable service semantics and with Blast Mode unreliable service semantics where data is send in short bursts with a wait period between each bursts. The highest throughput was obtained with Blast Mode, second highest with Synchronous Mode with Unreliable semantics, and then Synchronous Mode with Reliable

semantics. Other key observations were: SandiaXTP had good throughput for large data blocks, while low throughout and high latency were experienced for small data blocks. Mapp notes that some of the performance results are due to how the XTP protocol is implemented in SandiaXTP rather than the protocol itself.

A similar observation is made in [SLC94], which describes a commercial implementation of XTP 3.7 by Mentat, MXTP, using the TLI interface. Performance tests showed that XTP and TCP are comparable, and that XTP has an advantage over TCP when connections must be constructed on the fly. Due to the TLI interface, they noted that MXTP uses a 3-way handshake to disconnect, requiring a minimum exchange of 5 packets for a maxdata sized packet transmission. This is in contrast to the 2 packet minimum possible with XTP. It was noted that the TLI interface and implementation design decisions were factors in MXTP not fully exploiting XTP's protocol features. Other XTP implementations include XTP 3.6 over FDDI [DLW94], and a derivative of XTP 3.6 over ATM called XTPX [KB96].

[MN95] implements XTP 4.0 as a multi-threaded user-level daemon. It uses UDP as the data delivery mechanism and has an API geared for multimedia applications. The application layer is expected to control the functionality of XTP and therefore requires the application programmer to be knowledgeable and familiar with the XTP Specification.

3.2 Comparisons to other Protocols

An extensive analytical comparison of XTP 3.6, TP4, and TCP is given in [BD92a, BD92b]. They illustrate the different semantics of connection management and data transfer that each

protocol offers. Throughput performance between XTP and TCP is compared in [SLC94] and [KB96].

[Stra95b] contrasts the four possible Teardown methods permitted by the XTP protocol and made available in SandiaXTP's API.

3.3 Application Environment

While most of XTP's features are exploited in multimedia environments [MN95, Mapp95, DLW94, DSW], XTP can be utilized in a variety of other environments. [SW88, Stra91, SW92] explore XTP's feasibility in a real time distributed system; [SLC94] uses XTP as the transport protocol for distributed parallel processing.

3.4 Protocol Development

As with any dynamic protocol, XTP's development is an ongoing process. Its multicast functionality is continually improved and tested in implementations [ACFS96, DLW94, Ment98]. [CA90, AC93] report on simulation experience of XTP using the Local Area Network Simulation Facility (LANSF). They investigate the effects of XTP's Error Control mechanisms for XTP 3.3 and 3.6.

Chapter 4 Exploring XTP for Internet Traffic

Although XTP has a robust set of transport features, little has been done with XTP as a transport protocol for common Internet traffic. Much of the research and development work has been in support of non traditional Internet traffic. Some have expressed that XTP's potential lies in exploiting its support for next generation traffic patterns such as multimedia and multicasting. While XTP's support for next generation traffic patterns is undoubtedly its greatest advantage, its robust and flexible set of protocol mechanisms also make XTP an ideal candidate for existing Internet traffic types. Even with TCP's continued development and improvements, there are still clear limitations that are inherent in TCP's design and which may not be overcome. Thus investigating XTP as an alternate transport protocol for common Internet traffic is advantageous and beneficial for both the XTP community and the Internet community at large.

An impetus for investigating XTP as an alternative transport protocol is the poor interaction of TCP with some of today's Internet traffic. In this chapter, we examine in detail one such example, the interaction of TCP and HTTP. We also examine some factors that may have hindered XTP from being quickly accepted and implemented as a protocol for traditional Internet traffic. We highlight these factors in preparation for the XTP work presented in chapter 5.

4.1 TCP's Poor Interaction

TCP was originally designed as a general purpose highly reliable protocol. Its primary use was for telnet and ftp traffic patterns. When implemented correctly, TCP is a stable protocol that can satisfactorily respond to varying network conditions [Nagl84, JK88] With the advent of new Internet service protocols such as HTTP, RPC, SMTP, DNS, and SNMP, new demands are placed on TCP. TCP's original strength in providing reliable and stable data transfer is sometimes the source of its poor interaction with these service protocols. In some cases, application programmers utilize UDP instead of TCP to overcome TCP's shortcomings. However, this is done at the expense of duplicating reliable semantics and transmission rate control in the application layer (when called for by the application type). We examine one well known interaction problem between TCP and HTTP.

4.1.1 HTTP Traffic Pattern and TCP

HTTP is the transfer protocol for the WorldWide Web [BFF95]. An HTTP session consists of a request sent from a client to an HTTP server, followed by a response back to the client. An HTTP request can usually be contained within one Ethernet packet. The HTTP reply can be any length but is usually within the range of 1K to 21K [HOT96].

The interaction difficulties between HTTP and TCP have been well documented by several authors. [Sper95] points out several HTTP performance problems. TCP's 3-way handshake for connection setup and its Slow Start mechanism [JK88] interact poorly with HTTP's GET request. These incur multiple round trips before the HTTP server can even transmit its response. Spero also points out that TCP can suffer from a scalability problem in

Web Servers. Each TCP connection remains in a TIME_WAIT state after it is closed by the application. This is to protect itself against delayed arrival of retransmitted packets.

[PM94] and [Mogu95] collaborate Spero's findings and state that to avoid network latency, round trips should be minimized for HTTP. This is problematic for TCP since it does not fully utilize the network bandwidth until several round trips have transpired.

[THO96] and [HOT96] provide a detailed analysis of HTTP performance. Both concur the mismatch between the needs of HTTP and the services provided by TCP. [HOT96] state that TCP is optimized for large-scale bulk data transport, while HTTP needs a light weight, request-response protocol. Other request-response style services, such as short SMTP messages and RPC would also benefit from such a protocol.

A consistent recommendation from these authors is the need for using a "persistent" HTTP connection. Since the transport protocol cannot be easily modified to accommodate HTTP's requirements, they propose having a HTTP connection remain open across several GET requests, therefore reducing TCP's overhead for connection setup and Slow Start initiation. This technique became part of the next version of HTTP [FFBGM96]. However, even Persistent HTTP proved to have poor interaction with TCP in certain cases (TCP's Nagle Algorithm and Idle Time Restart) [Heide97, HV97, PK98].

In this particular example of TCP's poor interaction with Internet protocols, HTTP simply highlights TCP's weakness for short request/response traffic and very short file transfers. This is not surprising considering TCP's original design goals.

4.2 Factors Hindering XTP

When one considers XTP's design features in light of TCP's limitations, one would expect that the Internet community would adopt XTP quickly. Unfortunately there are hindering factors to XTP's acceptance. We review three such factors of which the second factor is considered the most significant.

4.2.1 Lack of a Publicly Available Implementation

Currently, most of the XTP implementations are either private or commercial. A clear exception to this is the SandiaXTP implementation. However, SandiaXTP is a user level implementation. While this is definitely beneficial for testing new protocol features, comparisons with existing kernel-based protocol are impractical. Also, its object-oriented nature precludes it from porting it into kernel space.

Nevertheless, we observe that the existence of private and commercial implementations of XTP prove that the protocol is indeed viable. As compared with many other proposed transport protocols VMTP [Cher88], TTCP [Brad92a, Brad92b], and NETBLT [CLZ87], XTP by far has the most documented implementations.

We simply stress that having access to a free source code implementation would encourage further development and enhancements of XTP's features, and encourage greater acceptance within the Internet community.

4.2.2 Lack of Congestion Control Policy

The XTP Specification does not outline a congestion control policy or procedure. This is consistent with the design philosophy of XTP, which is to provide the tools for a policy but

not to dictate the implementation of a specific policy. It is this philosophy that enables XTP to run efficiently over a variety of network layer protocols. Since each underlying network infrastructure has differing characteristics, what may work well in an IP network may not be appropriate or efficient in an ATM or Fiber network.

However, in an Internet-like environment, the lack of a clear and explicit congestion control policy is problematic. Several authors [BCCD98, FF98, MF97] have raised concerns about a trend of new non-TCP based applications that do not implement end-to-end congestion control. These are usually UDP based applications. In [FF98], the authors show that non-congestion-controlled best-effort traffic can have a serious negative impact on the Internet, ranging from extreme unfairness against TCP traffic to potential congestion collapse. The authors categorize 3 types of problematic traffic flows: (1) Unresponsive Traffic Flow – where the flow is not reduced in response to dropped packets, which usually indicates congestion; (2) Non TCP-friendly Traffic Flow – where a flow's arrival rate exceeds that of a conformant TCP in the same circumstances; (3) Disproportionate Bandwidth Flow – where the flow uses more bandwidth than other flows in a time of network congestion. The seriousness of this trend has led researchers to develop techniques to identify such flows and restrict their flow at the router level [BCCD98].

These negative impacts are significant in light of the XTP Specification. By design, XTP has no formal or explicit congestion management algorithms or techniques. While the protocol components for congestion control do exist, it is left up to the application programmer to implement a congestion management scheme. For some network environments, this may not be a concern, however, for best-effort IP environments, misused or abused transmission algorithms can have serious consequences as described above.

Although there is no explicit congestion management outlined for XTP, there is implicit transmission control through the Flow Control mechanism where a sender cannot transmit more than the ALLOC field value. This simplistic transmission control can easily fall prey to a performance-degrading stop-and-go pattern or to a network devastating burst of hundreds of data packets. The primary intent for flow control is to manage the data buffers on the receiving host. This flow control does not handle congestion indicators such as lost packets or increasing round trip times. Another transmission control mechanism is Rate Control. Rate Control is optional and not mandatory for all XTP association. The primary intent for Rate Control is to provide a steady flow of data packets to the receiver. It also does not handle network indicators of congestion.

While for some XTP implementations, like XTP over ATM, this may not be a hinderance, however for XTP/IP the lack of explicit congestion control management is of utmost importance. By designing an explicit congestion management scheme for XTP IP and implementing this into the transport layer, we remove the threat of XTP causing disastrous traffic network conditions. This will ensure that XTP and XTP applications behave as "good Internet citizens" [BCCD98].

4.2.3 Lack of Clear XTP Service Guidelines

A unique and distinguishing aspect of XTP is the way it separates transport protocol paradigms from protocol configuration issues. Usually a transport protocol is tied to a specific set of characteristics and capabilities with a fixed communication paradigm, such as TCP (connection-oriented, reliable service) and UDP (connectionless, unreliable service).

To address new transport paradigms entire new protocols are developed: Multicast [Deer89], T/TCP [Brad92a, Brad92b], VMTP [Cher88]. Even mature, well established

protocols like TCP have required extensions and modification. Since TCP's original specification several protocol extensions have been proposed to accommodate unforeseen network layer characteristics such as high bandwidth*delay networks [VB88, JBB92, Fox89], and high speed networks [VBZ90, JBB92].

XTP's decoupling of transport paradigms from protocol mechanisms overcomes some of these limitations. The XTP Specification provides a rich set of protocol mechanisms and algorithms that can be used as a toolkit from which transport services are "built" to accommodate a variety of transport layer needs and network layer environments. By design, the XTP Specification does not dictate how XTP's mechanisms must be used to implement a specific transport service. It does provide certain guidelines but purposely avoids specific details. It is left up to the programmer to correctly select when and how to use XTP's mechanisms. This lets the programmer determine how to configure XTP to provide a specific and tailored transport service. This design approach brings an incredible amount of adaptability and flexibility to XTP.

However, this "toolkit" design approach can also hinder effective and efficient use of XTP. XTP's effectiveness can be hindered because in order for an XTP application to take full advantage of XTP's features, the application programmer must have a thorough understanding of the XTP Specification, XTP's interface to the operating system, and a sound understanding of network issues such as congestion and connection management. Furthermore, the underlying network layer dramatically impacts the configuration of XTP at the transport level such that an application using XTP over ATM must use and configure XTP differently from an application using XTP over IP. XTP's efficiency can be hindered because incorrectly configured XTP parameters or misused mechanisms can generate XTP

traffic in such a manner that can adversely impact the stability of the Internet or degrade XTP's performance.

Chapter 5 Implementing XTP/IP

In this chapter we examine what is needed to optimally implement XTP in an IP network environment. In chapter 4, we identified several areas where XTP requires enhancements in order to safely and optimally function as a transport protocol for the most common types of Internet traffic.

Our proposed enhancements include connection management policies and congestion management policies tailored for XTP/IP, and a new Transport Traffic Service Framework for interfacing with XTP. We also present an open source kernel-based XTP implementation with XTP-enabled programs as a proof of concept for our proposed enhancements. We begin our presentation by introducing LinuxXTP, and then our enhancements to XTP.

5.1 XTP Implementation

To investigate and enhance XTP's behaviour in an IP environment we needed access to a kernel implementation of the XTP Specification. Previous to this thesis work, no publicly available kernel implementation of XTP existed. This provided the impetus for the LinuxXTP development and public release. This work is a clean and fresh implementation of the XTP Specification 4.0, and is presented as an original contribution for this thesis work. We feel that a native kernel level implementation of XTP is the only way to provide a fair basis for performance comparisons with existing kernel level protocols such as TCP and

UDP. Various network tools and programs were developed to initially aid in the development of LinuxXTP and then to optimize and refine XTP/IP.

In this section we present and describe the LinuxXTP implementation, a network utility for capturing XTP packets, and XTP aware applications that use the XTP protocol in LinuxXTP. We feel that the availability of an open source kernel XTP implementation and XTP-enabled user programs will aid the further development of XTP and deepen the understanding of XTP's capabilities and transport services. The software can be accessed freely from Concordia University's HSPL website:

http://www.cs.concordia.ca/~faculty/bill/hspl/xtplinux

5.1.1 LinuxXTP

LinuxXTP is a kernel based implementation of the XTP Specification revision 4.0, for the Linux Operating System. It implements all XTP features except for the multicast functionality. One design goal was to fully integrate the XTP code into the Linux kernel. Therefore XTP exists alongside other Internet layer 4 protocols such as TCP and UDP. We also strove to maintain a modular structure to the XTP code. This facilitates the implementation of future XTP features and mechanisms.

There are 6 modules of the XTP code in LinuxXTP. The `xtp` module contains the Operating System API which implements the BSD socket interface. The `xtp_inb` and `xtp_outb` modules contain functions for managing the inbound packets and outbound packets respectively. Module `xtp_ip` implements the interface to the IP layer, and `xtp_linux` contains functions for manipulating the XTP data structures and integrating

them with the Linux networking data structures. The final module, `xtp_func`, contains a variety of functions required for executing the XTP protocol. There is an additional module, `xtp_debug` which is not necessary or required, but contains all debugging functions essential for developing and testing the XTP kernel code.

For this thesis, LinuxXTP is implemented for the 2.4.0pre-test10 Linux kernel release. After its initial release we intend to keep LinuxXTP current with the kernel versions.

5.1.2 BSD Application Programming Interface

We had several choices for implementing an API for applications to interface with LinuxXTP. We could have used a library implementation or introduced new XTP-centric system calls. This would have given us extreme flexibility in configuring XTP's parameters and activating XTP's mechanisms. The SandiaXTP [SGC94] and "Real-Time Threads" implementation of XTP [MN95] are two examples that have used the library approach.

In our implementation, we decided to utilize the well known BSD socket interface. The primary motive for this is to leverage the existing code base of Internet applications and utilities. This will make the porting of existing TCP/IP programs easier. If a new API were introduced it would require a learning curve and significant program recoding. Another reason for selecting the BSD interface is that it easily supports our proposed "packaged" approach to utilizing XTP's mechanisms and features. Yet, BSD also provides a way to have direct access to XTP's mechanisms—which is essential for the development, testing, and optimization of the xtp code and the Transport Traffic Service configurations.

5.1.2.1 BSD System Calls

There are 8 primary system calls in the BSD socket API. The `socket()` system call creates a BSD socket. To create an XTP socket we specify the XTP protocol, IPPROTO_XTP, and the XTP service, SOCK_XTP as arguments to socket():

```
socket(AF_INET, SOCK_XTP, IPPROTO_XTP)
```

This creates and initializes a BSD socket with a link to the XTP Context. The XTP Context is initialized with default parameters. To modify specific XTP parameters the setsockopt() call is used. To create an XTP Context tailored for a specific traffic pattern, we use the setsockopt() call and specify a Transport Traffic Service, for example:

```
/* for File Transfer Service */
service=xtp_FILETRANSFER_SERVICE;
setsockopt(s, IPPROTO_XTP, XTPSOCK_TTS, &service, sizeof(int))
/* for Request/Respond Service */
service=xtp_REQRESP_SERVICE;
setsockopt(s, IPPROTO_XTP, XTPSOCK_TTS, &service, sizeof(int))
```

The remaining system calls (connect(), bind(), accept(), listen(), read() recv(), write()/send(), shutdown()) are called exactly as in TCP or UDP applications.

5.1.2.2 Get/Setsockopt System Calls

The setsockopt() and getsockopt() system calls allow direct access to XTP parameters. This is necessary for testing the behaviour and performance of XTP, and is meant for R&D purposes. Appendix A lists all possible XTP parameters that can be modified in LinuxXTP by an application. In our proposed XTP Service Framework, the application layer should not manipulate XTP's features directly via setsockopt() except for selecting a Transport Traffic Service.

The getsockopt() call is used to retrieve the current values of a given XTP context. We use this capability to retrieve performance metrics, configuration detail, and protocol statistics about a given XTP Context. We instrumented the xtp code in LinuxXTP to track a variety of counters and metrics that can be used to evaluate XTP performance, and the effectiveness of different optimization techniques. We obtain this information by passing an empty XTP data structure which is returned with the metric values. Appendix B lists all variables returned in the data structure.

5.1.3 TCPDUMP for LinuxXTP

The tcpdump utility captures network packets and interprets all packet information. To monitor and capture XTP traffic we modified tcpdump to understand XTP revision 4.0 packets. We display XTP packets in a similar fashion to tcpdump's output for TCP. Each output line for XTP contains the date/timestamp, sending and receiving hosts, XTP packet type, XTP header fields with option bits, and XTP payload fields. User data is not displayed. Appendix C explains the tcpdump output in detail for XTP packets.

5.1.4 XTP Enabled Utilities and Applications

To evaluate our implementation of XTP/IP, we wrote new test programs and ported some of the common Internet applications to utilize XTP as the transport protocol.

We ported the following Internet applications: ftp, finger, and inetd. We selected these applications to generate a representative set of Internet traffic. With these programs, and the four newly created XTP programs (xft, xftd, xtpsend, xtprecv), we can generate file transfer traffic and request/response traffic. We describe these programs and their port in Appendix D.

Another reason for porting the TCP/IP application is to demonstrate the flexibility of XTP and of a 'packaged' Transport Traffic Service Framework. The modifications to the source code were minimal. The Transport Traffic Service Framework allows us to implement the BSD semantic within the XTP implementation. This is only possible because of the XTP Specification design. If the XTP Specification were more rigid, as compared to other transport protocol specifications, the porting of code would have required much greater effort. We feel that the minimal amount of code modification is an objective measurement of the flexibility of a new protocol.

There is one temporary deviation to this. Due to some interaction dependencies between the initialization of network devices, IP addresses configuration, and the spawning of the Inetd process during the Linux boot sequence, we had to manually configure the server's IP address in the Inetd and ftpdx code. We implement this workaround as a temporary solution and plan to eliminate this upon a further detailed investigation of these dependencies. Even so, this workaround requires only two extra lines of code.

5.2 Connection Management

Connection management is the management of packet exchanges between two communicating hosts. It governs how a transport session is established, maintained, and terminated.

In section 5.2.1 we examine combinations of XTP's configuration that provide the best performance and efficiency for XTP/IP. We examine 3 areas: Teardown Method, Association Type, and Packet Loss Detection. We also cover other parameter settings that

would normally be considered "fixed" features in other transport protocols but are flexible and configurable in XTP. We treat these XTP settings as "fixed" for XTP/IP.

To test the advantages/disadvantages of XTP's parameters we perform several latency tests to measure, if any, performance improvements. We use an Ethernet network (LAN), a simulated congestion delayed network (CD), and a simulated fat pipe network (FP). This test environment is described in section 6.1

We end this section by identifying a First Packet Replay Hazard condition.

5.2.1 Background

It is important to understand well the "mechanics" of a transport session since the number of packets exchanged significantly impact the performance of short data transfers. It some cases the type of connection management can also impact the performance and resources of the host system and application layer.

The components of connection management follow the three phases or stages of a transport session:

1) A setup phase to initiate the connection between two hosts
2) A transfer phase to transfer user data from one host to another
3) A teardown phase to terminate the connection between two hosts

Each of these phases have distinct characteristics that impact a protocol's performance and behavior.

The XTP Specification inherently provides a large set of connection management options. This provides XTP the flexibility to provide connection semantics that are traditionally provided by a single protocol specification. In traditional Internet protocols, such as TCP, many of the connection management options are fixed. For example, TCP uses

a fixed three-way handshake to initiate a connection with no data transfer to the application layer. In contrast, XTP has several ways to manage its connection setup.

5.2.2 Teardown Methods

The XTP Specification allows for 4 different connection teardown semantics. [Stra95b] illustrates the packet exchanges sequence for the 4 closure semantics. For XTP/IP, we examine 3 teardown methods: 1) Fully Graceful Close (GracefulClose), 2) Abbreviated Graceful Close (AbbrevClose), and 3) Forced Close (ForcedClose). The fourth possible teardown method, Abortive Close, is not considered a configurable option since it is primarily reserved for error conditions and is not utilized for reliable data transfers.

We run several tests to observe XTP's teardown behavior. We keep all other parameter settings constant to prevent any unwanted influence. In our tests, we use a reliable S_D_T association type and three network environments, LAN, FP1, and CD1 (see section 6.1 for test environment). We first analyze the packet traces of the teardown methods to understand the exact impact on the network and then review the performance analysis. We present a packet sequence diagram and an actual packet trace from tcpdump for teardown method. Appendix C describes the legend for the XTP packet traces. Our criteria for evaluating the teardown method policies are the number of packets exchanged, performance, and overall efficiency.

5.2.2.1 Packet Trace Analysis of Teardown Methods

Figures 1 to 6 illustrate the packet exchange sequence for the 3 Teardown Polices for XTP/IP. Figure 1 shows the exchange of XTP's Control packets during the Teardown Phase for the GraceFulClose Method and figure 2 is a trace of a complete XTP connection using the

GraceFulClose Method. Figure 3 shows the exchange of XTP's Control packets the AbbrevClose Method and figure 4 is a trace of a complete XTP connection using the AbbrevClose Method. Figure 5 shows the exchange of XTP's Control packets for the ForcedClose Method and figure 6 is a trace of a complete XTP connection using the ForcedClose Method.

We first make a few observations common to all closure methods. All are functionally valid for reliable connections. Each ensures that all user data is successfully transferred and that each XTP context is fully synchronized with respect to sequence numbers. For protocol correctness we protect the Control packets carrying the closure option bits with the WTIMER, therefore we use the SREQ bit. The XTP Specification does not require the setting of the SREQ bits with the closure option bits, however for reliable connections this is necessary.

The GracefulClose teardown method has the lengthiest packet exchange. The teardown method begins in line #5 by the client side (Figure 2). The teardown is completed with the END bit carrying packet in line #9. Due to a harmless race condition, there is one extra packet that is transmitted (line #10). The race condition occurs because either context can transmit the final END bit once the inbound and the outbound data streams are successfully terminated. By examining the timestamps we can pinpoint the time of the race condition. From the FIRST packet and the TRAFFIC packet we determine that there is a 1.093 ms RTT (Round Trip Time). The first END bit is transmitted by the client at 90.880890 seconds. The server transmitted its END bit at about 90.881150 seconds (90.881697 sec (arrival time) – 0.546 ms (1/2 of RTT)). The race condition occurs within 0.3 ms.

The AbbrevClose teardown method has the smallest number of packet exchanges (see figure 3 and figure 4). The END bit is sent by the client as soon as both data streams are terminated successfully.

The ForcedClose teardown method has two extra packets transmitted during its teardown sequence (see figure 5 and figure 6). We observe from the actual packet trace in figure 6, that the first extra packet (line #8) is the expected response by the server after it has determined that both data streams are terminated (compare line #8 with line #7 from GracefulClose packet trace, figure 2). The second extra packet is from the client while it is in the Zombie state. Since the server's packet (line #8) arrives at the client after the client has transmitted its END bit, it is processed according to the ZOMBIE state procedure. The server upon receiving the second extra packet discards it quietly.

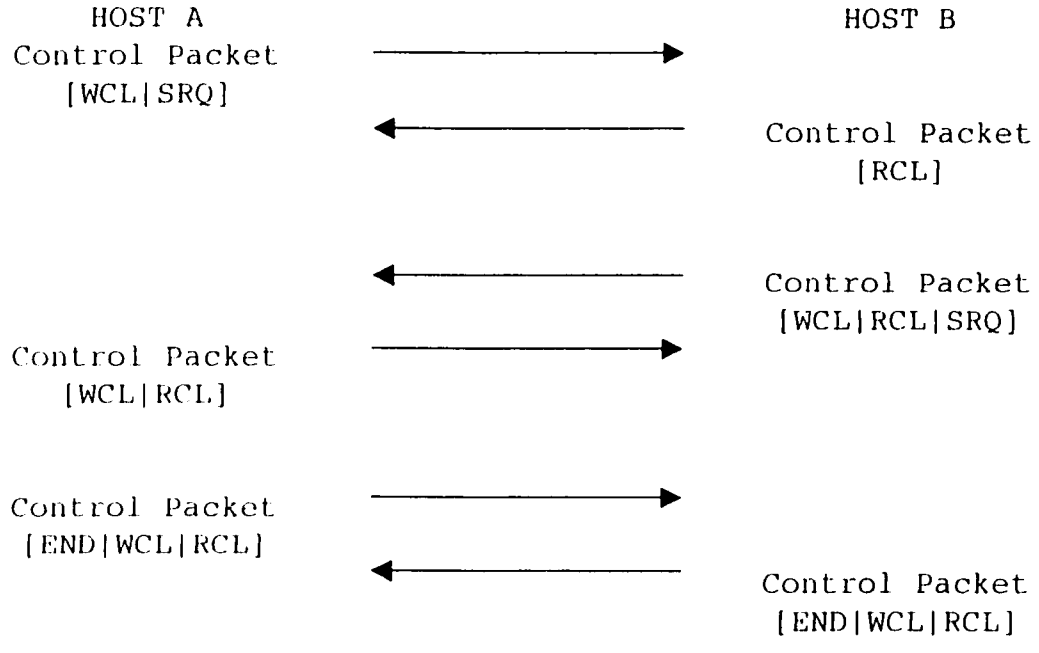

Figure 1 Packet Sequence for GracefulClose Teardown Method

1	959656590.873738 client > server: F ky1257 in0 d124 sy1 sq90000 SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp64520 sh:192.100.100.100 t18 sv4 t10
2	959656590.874831 server > client: T ky1257 in0+RTN d140 sy0 sq0 | rq24 a190024 ec1 xk1326 t18 sv4 t10
3	959656590.875363 client > server: D ky1326 in0+RTN d11000 sy2 sq24 EOM|SRQ|
4	959656590.879343 server > client: C ky1257 in0+RTN d120 sy0 sq0 | rq1024 a191024 ec2
5	959656590.879493 client > server: C ky1326 in0+RTN d120 sy3 sq1024 WCL|SRQ| rq0 a190000 ec0
6	959656590.880423 server > client: C ky1257 in0+RTN d120 sy0 sq0 RCL| rq1024 a191024 ec3
7	959656590.880669 server > client: C ky1257 in0+RTN d120 sy1 sq0 WCL|RCL|SRQ| rq1024 a191024 ec3
8	959656590.880701 client > server: C ky1326 in0+RTN d120 sy3 sq1024 WCL|RCL| rq0 a190000 ec1
9	959656590.880890 client > server: C ky1326 in0+RTN d120 sy3 sq1024 END|WCL|RCL| rq0 a190000 ec1
10	959656590.881697 server > client: C ky1257 in0+RTN d120 sy1 sq0 END|WCL|RCL| rq1024 a191024 ec3

Figure 2 Trace of Teardown Method: GracefulClose [Network Type: LAN]

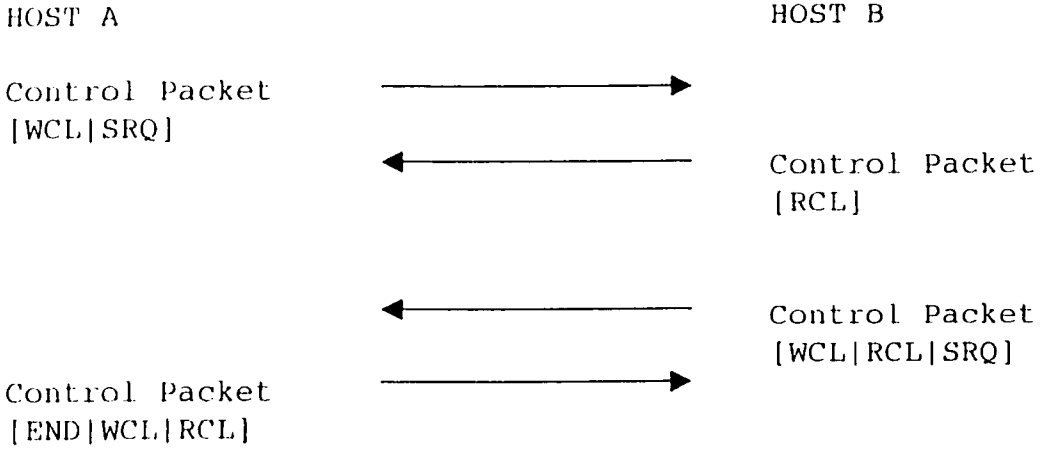

Figure 3 Packet Sequence for AbbrevClose Teardown Method

1	959656592.043403 client > server: F ky1258 in0 d124 sy1 sq90000 SRQ| a116 ad9216 af256 dp9999 dh:192.200.200.200 sp64776 sh:192.100.100.100 t18 sv4 t10
2	959656592.044507 server > client: T ky1258 in0+RTN d140 sy0 sq0 | rq24 a190024 ec1 xk1327 t18 sv4 t10
3	959656592.045080 client > server: D ky1327 in0+RTN d11000 sy2 sq24 EOM|SRQ|
4	959656592.049051 server > client: C ky1258 in0+RTN d120 sy0 sq0 | rq1024 a191024 ec2
5	959656592.049202 client > server: C ky1327 in0+RTN d120 sy3 sq1024 WCL|SRQ| rq0 a190000 ec0
6	959656592.050133 server > client: C ky1258 in0+RTN d120 sy0 sq0 RCL| rq1024 a191024 ec3
7	959656592.050362 server > client: C ky1258 in0+RTN d120 sy1 sq0 WCL|RCL|SRQ| rq1024 a191024 ec3
8	959656592.050395 client > server: C ky1327 in0+RTN d120 sy3 sq1024 END|WCL|RCL| rq0 a190000 ec1

Figure 4 Trace of Teardown Method: AbbrevClose [Network Type: LAN]

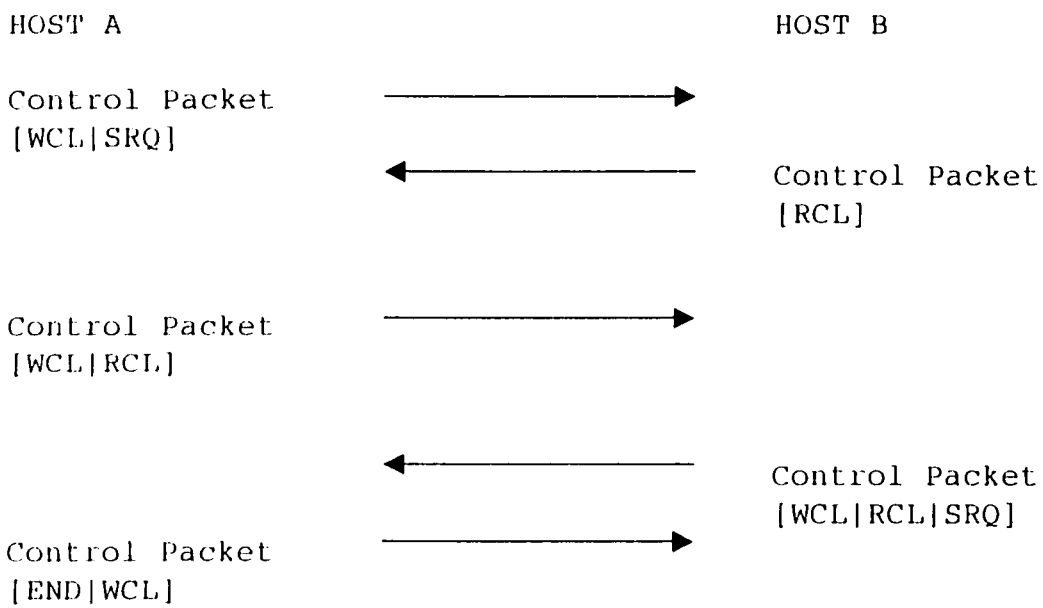

Figure 5 Packet Sequence for ForcedClose Teardown Method

1	959656593.213339 client > server: F ky1259 in0 dl24 sy1 sq90000 SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp65032 sh:192.100.100.100 tl8 sv4 tf0
2	959656593.214450 server > client: T ky1259 in0+RTN dl40 sy0 sq0 | rq24 al90024 ec1 xk1328 tl8 sv4 tf0
3	959656593.214975 client > server: D ky1328 in0+RTN dl1000 sy2 sq24 EOM|SRQ|
4	959656593.218943 server > client: C ky1259 in0+RTN dl20 sy0 sq0 | rq1024 al91024 ec2
5	959656593.219097 client > server: C ky1328 in0+RTN dl20 sy3 sq1024 WCL|SRQ| rq0 al90000 ec0
6	959656593.220055 server > client: C ky1259 in0+RTN dl20 sy0 sq0 RCL| rq1024 al91024 ec3
7	959656593.220093 client > server: C ky1328 in0+RTN dl20 sy3 sq1024 END|WCL| rq0 al90000 ec0
8	959656593.220346 server > client: C ky1259 in0+RTN dl20 sy1 sq0 WCL|RCL|SRQ| rq1024 al91024 ec3
9	959656593.220375 client > server: C ky1328 in0+RTN dl20 sy3 sq1024 END|WCL| rq0 al90000 ec1

Figure 6 Trace of Teardown Method: ForcefulClose [Network Type: LAN]

5.2.2.2 Performance Analysis of Teardown Methods

We test the 3 teardown methods across three different network environments and keep all other XTP settings constant. In Table 1, the performance results show that there is a very minor variance between the three teardown methods. The transfer time results are within 10% of one another. The only significant distinction between the teardown methods is the number of unnecessary transmitted packets for GracefulClose and ForcedClose. In the LAN environment, due to the low latency (and therefore low round trip time) the race condition for the extra CONTROL packet does not always occur. Although the transfer time results are non conclusive, we select the AbbrevClose method as a preferred teardown method since there is no extra packet involved and it represents a "clean" packet exchange for XTP.

Table 1 Teardown Method Performance

Teardown Method	Data Bytes Transferred	Number of Packets Exchanged	Number of Extra Packets Transmitted	Transfer Time (s)	Percentage Difference
LAN Network					
ForcedClose	2000	8.0	1.9	0.0088	0%
AbbrevClose	2000	9.0	0.0	0.0092	4%
GracefulClose	2000	10.0	0.1	0.0095	7%
FP1 Network					
ForcedClose	2000	8.0	2.0	0.0180	0%
AbbrevClose	2000	9.0	0.0	0.0196	8%
GracefulClose	2000	10.0	1.0	0.0198	10%
CD1 Network					
ForcedClose	2000	8.0	2.0	0.1728	0%
AbbrevClose	2000	9.0	0.0	0.1772	2%
GracefulClose	2000	10.0	1.0	0.1773	2%

5.2.3 Association Types

Unlike traditional Internet protocols (UDP, TCP), XTP can be quite flexible in how an association between two hosts is initiated, maintained, and terminated. The XTP Specification permits four different association types. The term "association type" is not part of the XTP Specification, so we define it in this paper as a configuration type of the connection phases. In XTP, it is possible to have the connection phases combined or separate. We examine each of these association types to determine the advantages and disadvantages of each one. The four association types are:

1) SDT: combined setup, data transfer, and teardown phases

2) S_D_T: separate setup, data transfer, and teardown phases

3) S_DT: separate setup with combined data transfer and teardown phases

4) SD_T: combined setup and data transfer phases with a separate teardown phase.

To analyze the packet exchange sequence of these four association types we present a packet sequence diagram and an actual packet trace from tcpdump for each association type. Appendix C describes the legend for the XTP packet traces.

For our packet traces and performance tests, we use the AbbrevClose teardown method and run the tests across three network environments (LAN, CD1, FP1).

5.2.3.1 Packet Trace Analysis of Association Types

Figures 7 to 14 show the packet traces for the 4 association types. In these trace examples 1000 bytes of user data is transferred.

The SDT Association Type has the shortest packet exchange sequence (figure 7 and 8). It has a combined setup, data transfer, and teardown phase. The FIRST packet is packed with information. It contains the user data, the EOM bit to push the data to the application

layer, the WCLOSE to indicate that it has no more data to transmit, and the SREQ to request a return CONTROL packet and to protect the FIRST packet with a WTIMER. This FIRST packet establishes the connection, transfers the user data and initiates the teardown procedure. The TRAFFIC packet acknowledges the connection request and the user data, and responds to the teardown request with the RCLOSE bit. The remaining CONTROL packets complete the abbreviated graceful teardown. This is the most efficient association type (for a generic reliable connection). XTP's capability to carry and deliver user data in the initiating packet is in sharp contrast to TCP's setup packet exchange. A strict implementation of the TCP protocol [RFC793], permits user data to be carried in the initiating TCP packet, however, no data is allowed to be passed on to the application layer until the three-way handshake is completed. To date, no TCP implementation carries user data in the initial SYN segment.

The S_D_T Association Type has the longest packet exchange sequence for XTP (figure 9 and 10). It has separate setup, data transfer, and teardown phases. Each phase is fully completed before initiating the next phase. In figure 10, lines #1 and #2 constitute the setup phase. Lines #3 and #4 comprise the data transfer phase, and Lines #5 to #8 contain the teardown sequence.

The S_DT Association Type has a separate setup phase and a combined data and teardown phase (figure 11 and 12). In figure 12, lines #3 to #6 contain the combined data transfer and teardown phase. This association type is the most similar to TCP in terms of packet sequence.

The SD_T Association Type has a combined setup and data transfer phase and a separate teardown phase (Figure 13 and 14). In figure 14, lines #1 and #2 comprise the setup and data transfer phase.

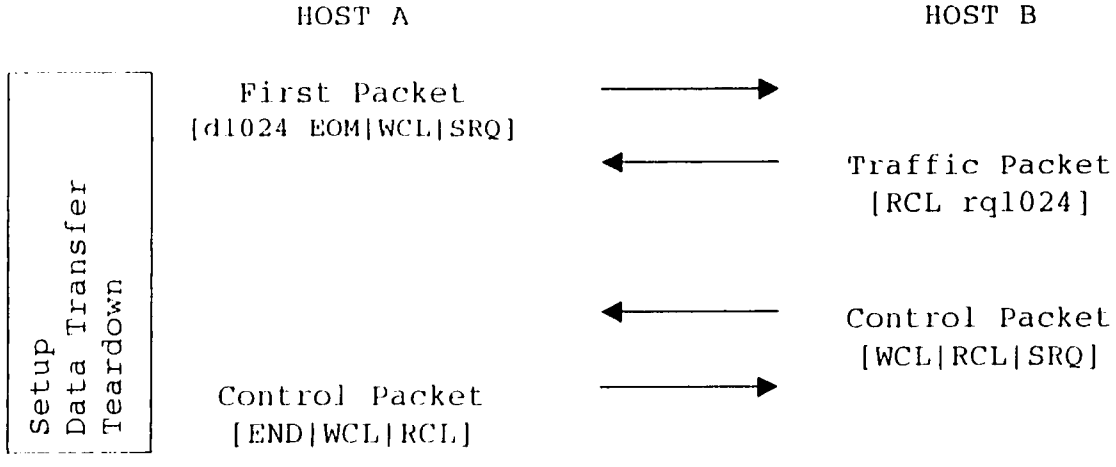

Figure 7 Packet Sequence for Association Type SDT

Setup Data Transfer Teardown	1	959656588.553034 client > server: F ky1255 in0 dl1024 sy1 sq90000 EOM|WCL|SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp64008 sh:192.100.100.100 t18 sv4 t10
	2	959656588.557254 server > client: T ky1255 in0>RTN dl40 sy0 sq0 RCL| rq1024 al91024 ecl xk1324 t18 sv4 t10
	3	959656588.557709 server > client: C ky1255 in0>RTN dl20 sy1 sq0 WCL|RCL|SRQ| rq1024 al91024 ecl
	4	959656588.557749 client > server: C ky1324 in0>RTN dl20 sy1 sq1024 END|WCL|RCL| rq0 al90000 ecl

Figure 8 Packet Trace of Association Type SDT [Network Type: LAN]

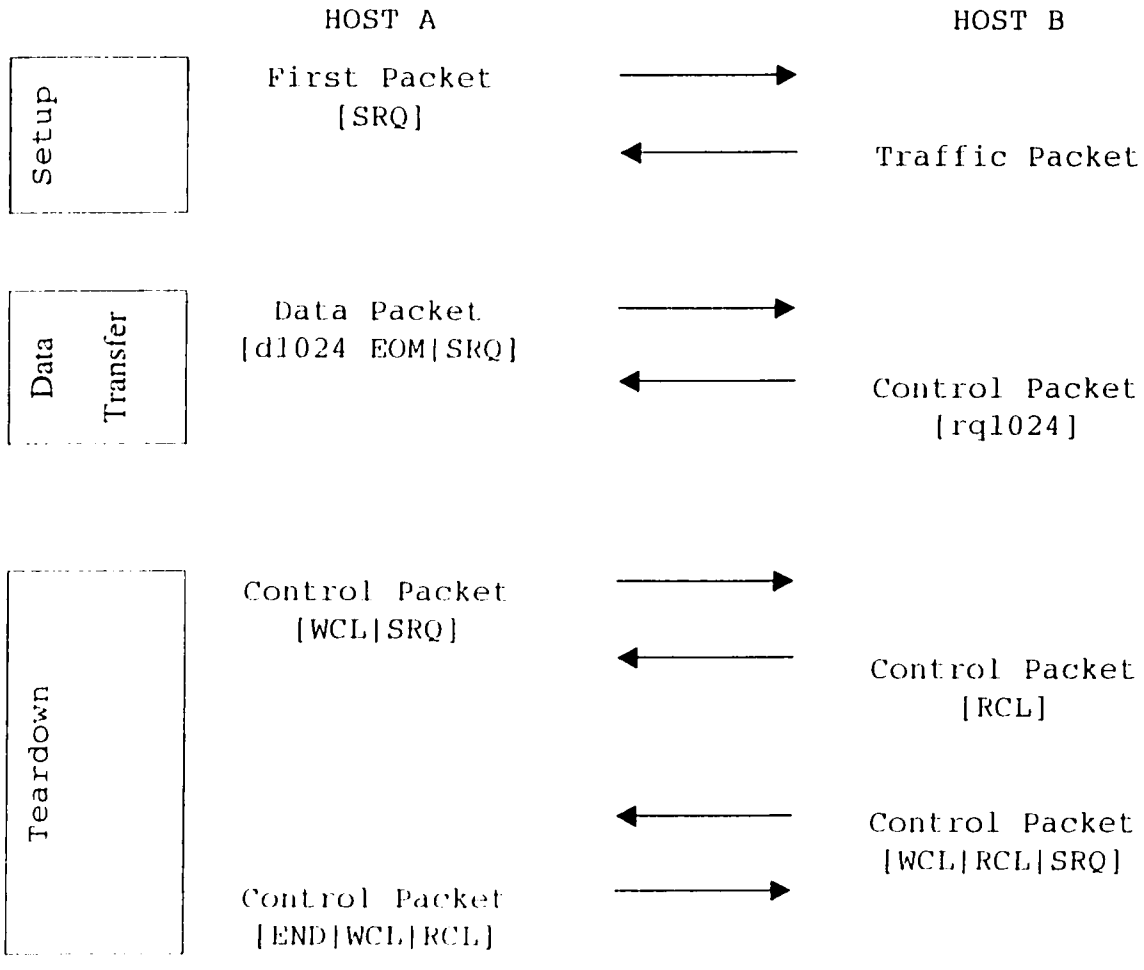

Figure 9 Packet Sequence for Association Type S_D_T

Setup	1	959656592.043403 client > server: F ky1258 in0 dl24 sy1 sq90000 SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp64776 sh:192.100.100.100 t18 sv4 tf0
	2	959656592.044507 server > client: T ky1258 in0+RTN dl40 sy0 sq0 | rq24 al90024 ec1 xk1327 t18 sv4 tf0
Data Transfer	3	959656592.045080 client > server: D ky1327 in0+RTN dl1000 sy2 sq24 EOM|SRQ|
	4	959656592.049051 server > client: C ky1258 in0+RTN dl20 sy0 sq0 | rq1024 al91024 ec2
Teard own	5	959656592.049202 client > server: C ky1327 in0+RTN dl20 sy3 sq1024 WCL|SRQ| rq0 al90000 ec0
	6	959656592.050133 server > client: C ky1258 in0+RTN dl20 sy0 sq0 RCL| rq1024 al91024 ec3
	7	959656592.050362 server > client: C ky1258 in0+RTN dl20 sy1 sq0 WCL|RCL|SRQ| rq1024 al91024 ec3
	8	959656592.050395 client > server: C ky1327 in0+RTN dl20 sy3 sq1024 END|WCL|RCL| rq0 al90000 ec1

Figure 10 Trace of Association Type: S_D_T [Network Type: LAN]

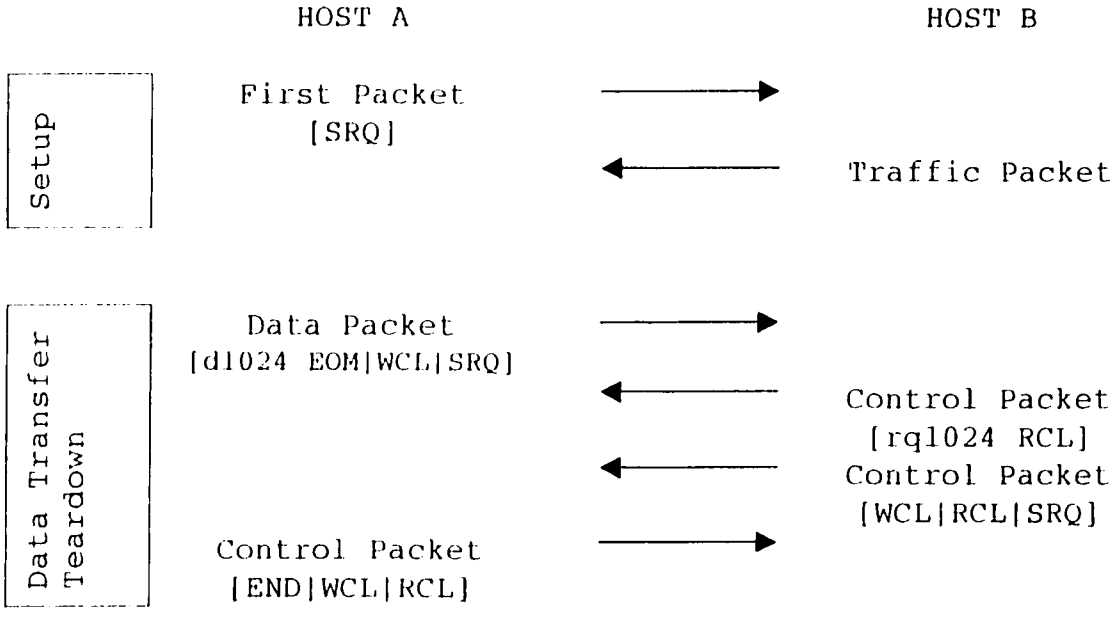

Figure 11 Packet Sequence for Association Type S_DT

Setup	1	959656595.543647 client > server: F ky1261 in0 d124 sy1 sq90000 SRQ| a116 ad9216 af256 dp9999 dh:192.200.200.200 sp777 sh:192.100.100.100 t18 sv4 t10
	2	959656595.544744 server > client: T ky1261 in0+RTN d140 sy0 sq0 | rq24 a190024 ec1 xk1330 t18 sv4 t10
Data Transfer Teardown	3	959656595.545314 client > server: D ky1330 in0+RTN d11000 sy2 sq24 EOM|WCL|SRQ|
	4	959656595.549308 server > client: C ky1261 in0+RTN d120 sy0 sq0 RCL| rq1024 a191024 ec2
	5	959656595.549597 server > client: C ky1261 in0+RTN d120 sy1 sq0 WCL|RCL|SRQ| rq1024 a191024 ec2
	6	959656595.549633 client > server: C ky1330 in0+RTN d120 sy2 sq1024 END|WCL|RCL| rq0 a190000 ec1

Figure 12 Trace of Association Type: S_DT [Network Type: LAN]

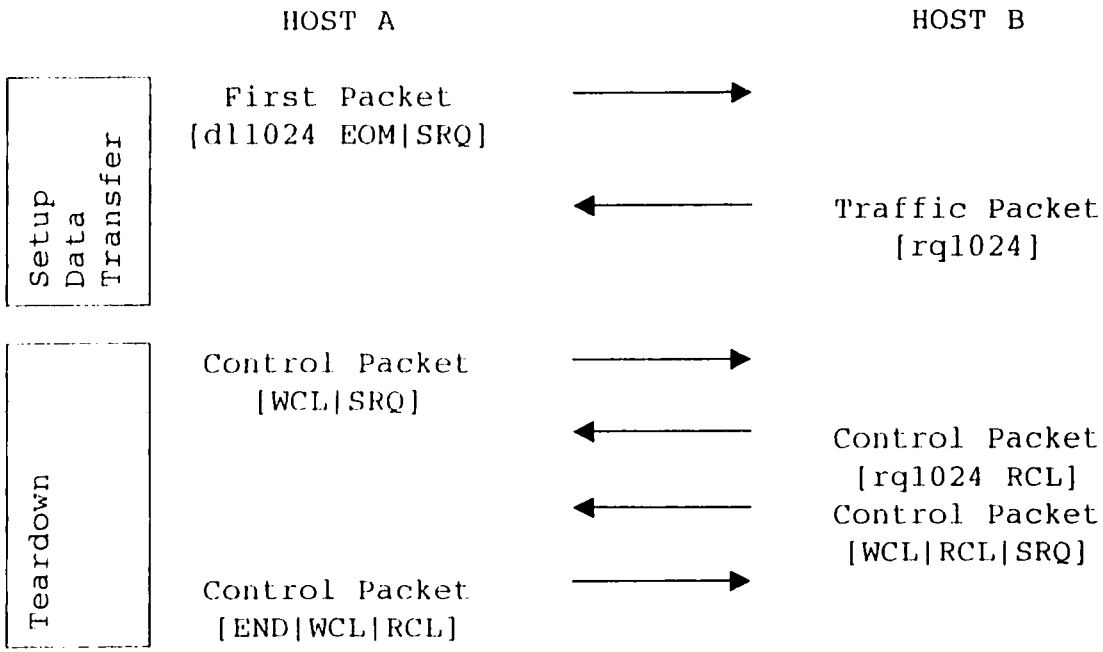

Figure 13 Packet Sequence for Association Type SD_T

Setup Data Transfer	1	959656599.023736 client > server: F ky1264 in0 dl1024 sy1 sq90000 EOM|SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp1545 sh:192.100.100.100 t18 sv4 tf0
	2	959656599.027941 server > client: T ky1264 in0+RTN dl40 sy0 sq0 | rq1024 al91024 ec1 xk1333 t18 sv4 tf0
Teardown	3	959656599.028170 client > server: C ky1333 in0+RTN dl20 sy2 sq1024 WCL|SRQ| rq0 al90000 ec0
	4	959656599.029132 server > client: C ky1264 in0+RTN dl20 sy0 sq0 RCL| rq1024 al91024 ec2
	5	959656599.029360 server > client: C ky1264 in0+RTN dl20 sy1 sq0 WCL|RCL|SRQ| rq1024 al91024 ec2
	6	959656599.029394 client > server: C ky1333 in0+RTN dl20 sy2 sq1024 END|WCL|RCL| rq0 al90000 ec1

Figure 14 Trace of Association Type: SD_T [Network Type: LAN]

5.2.3.2 Performance Analysis of Association Types

Figure 15 and tables 2 to 6 show the performance results for the four association types. In the 5 performance tables we shows that results of performance tests for data transfers of 1000 to 10 000 bytes in increments of 1000 bytes. In figure 15, we graph the performance result of data transfers between 1000 to 30 000 bytes. Overall, we observe that the type of association has a significant performance impact for short data transfers. As the data transfer portion of the association increases and becomes a larger percentage of the overall association, the association type is less significant. As expected, the SDT association type produces the best performance and the S_D_T is the slowest.

In a LAN environment, the Association Type has a high impact on performance for data transfer of 3000 bytes or less (see table 2). By the 10 000 byte data transfer the transfer times are within 15% of the best/worst case.

In a fat-pipe network environment, the influence of the association type is stronger (see table 3 and 4). The higher the bandwidth*delay product is, the stronger the impact on transfer times. However both environments show a tapering off as the data byte transfer size increases.

For congestion delayed networks, the impact of the association type on performance quickly dissipates (see table 5 and 6). In this environment, the bottleneck is truly the network and little can be done to improve transfer times. Data transfers involving more than 1 maxdata size packet are unaffected by the association type selected.

From these performance results, we can make general observations about XTP's association types. For short data transfer, the SDT association type performs best in all

network environments. As the data transfer size increases, other factors, such as congestion management, become the dominant variable.

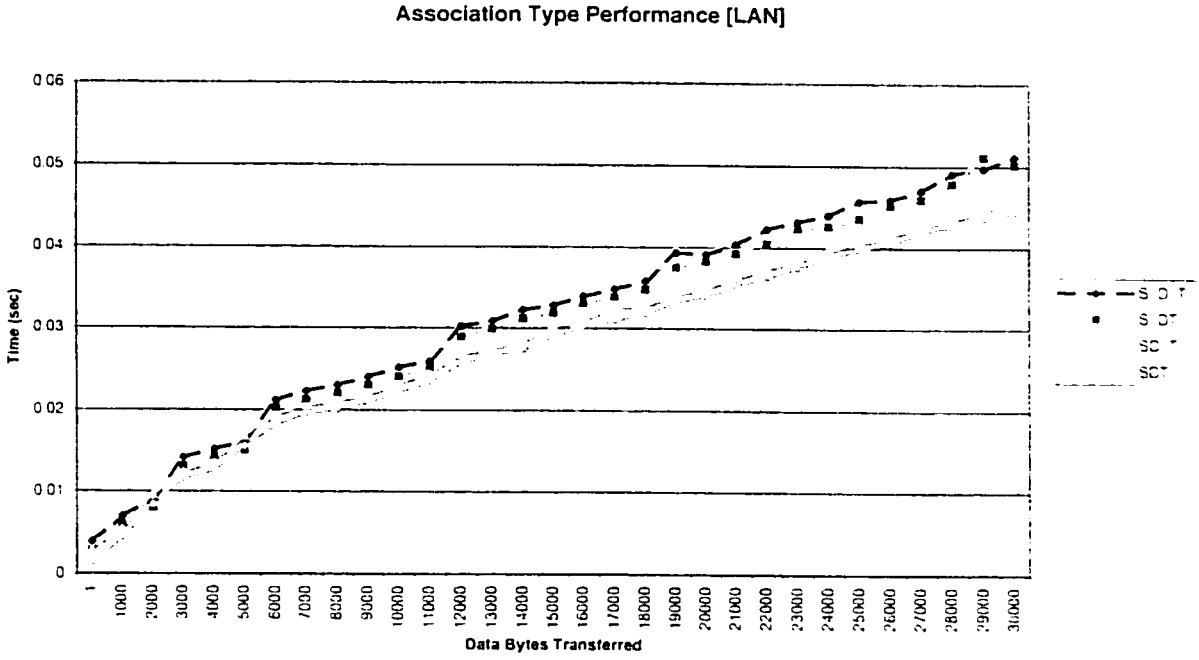

Figure 15 Performance Graph for Association Types [LAN Network]

Table 2 Association Type Performance [Network Type: LAN]

Association Type	Data Bytes Transferred	Number of Packets Exchanged	Transfer Time (s)	Percentage Difference
SDT	1	4.0	0.0017	0%
SD_T	1	6.0	0.0029	70%
S_DT	1	6.0	0.0029	70%
S_D_T	1	8.0	0.0039	129%
SDT	1000	4.0	0.0048	0%
S_DT	1000	6.0	0.0060	25%
SD_T	1000	6.0	0.0061	27%
S_D_T	1000	8.0	0.0072	50%
S_DT	2000	7.0	0.0081	0%
SDT	2000	6.0	0.0091	12%
S_D_T	2000	9.0	0.0091	12%
SD_T	2000	8.0	0.0098	20%
SDT	3000	7.0	0.0118	0%
SD_T	3000	9.0	0.0128	8%
S_DT	3000	10.0	0.0132	11%
S_D_T	3000	12.0	0.0143	21%
SDT	4000	7.0	0.0131	0%
SD_T	4000	9.0	0.0141	7%
S_DT	4000	10.0	0.0144	9%
S_D_T	4000	12.0	0.0153	16%
S_DT	5000	12.0	0.0151	0%
S_D_T	5000	14.0	0.0161	6%
SDT	5000	9.0	0.0162	7%
SD_T	5000	11.0	0.0169	11%
SDT	6000	10.0	0.0186	0%
SD_T	6000	12.0	0.0197	5%
S_DT	6000	16.0	0.0205	10%
S_D_T	6000	18.0	0.0216	16%
SDT	7000	10.0	0.0200	0%
SD_T	7000	12.0	0.0211	5%
S_DT	7000	16.0	0.0214	6%
S_D_T	7000	18.0	0.0226	12%
SDT	8000	12.0	0.0205	0%
SD_T	8000	14.0	0.0213	3%
S_DT	8000	17.0	0.0222	8%
S_D_T	8000	19.0	0.0234	14%
SDT	9000	13.0	0.0214	0%
SD_T	9000	15.0	0.0224	4%
S_DT	9000	19.0	0.0232	8%
S_D_T	9000	21.0	0.0243	13%
SDT	10000	13.0	0.0225	0%
SD_T	10000	15.0	0.0235	4%
S_DT	10000	19.0	0.0245	8%
S_D_T	10000	21.0	0.0254	12%

Table 3 Association Type Performance [Network Type: FP1]

Association Type	Data Bytes Transferred	Number of Packets Exchanged	Transfer Time (s)	Percentage Difference
SDT	1	4.0	0.0057	0%
SD_T	1	6.0	0.0100	75%
S_DT	1	6.0	0.0100	75%
S_D_T	1	8.0	0.0141	147%
SDT	1000	4.0	0.0089	0%
S_DT	1000	6.0	0.0131	47%
SD_T	1000	6.0	0.0134	50%
S_D_T	1000	8.0	0.0172	93%
S_DT	2000	7.0	0.0154	0%
SDT	2000	6.0	0.0161	4%
SD_T	2000	8.0	0.0203	31%
S_D_T	2000	9.0	0.0204	32%
SDT	3000	7.0	0.0190	0%
SD_T	3000	9.0	0.0234	23%
S_DT	3000	10.0	0.0235	23%
S_D_T	3000	12.0	0.0273	43%
SDT	4000	7.0	0.0209	0%
SD_T	4000	9.0	0.0256	22%
S_DT	4000	10.0	0.0258	23%
S_D_T	4000	12.0	0.0292	39%
SDT	5000	9.0	0.0282	0%
S_DT	5000	12.0	0.0284	0%
SD_T	5000	11.0	0.0317	12%
S_D_T	5000	14.0	0.0330	17%
SDT	6000	10.0	0.0303	0%
SD_T	6000	12.0	0.0344	13%
S_DT	6000	16.0	0.0364	20%
S_D_T	6000	18.0	0.0418	37%
SDT	7000	10.0	0.0322	0%
SD_T	7000	12.0	0.0371	15%
S_DT	7000	16.0	0.0374	16%
S_D_T	7000	18.0	0.0417	29%
SDT	8000	12.0	0.0354	0%
S_DT	8000	17.0	0.0394	11%
SD_T	8000	14.0	0.0395	11%
S_D_T	8000	19.0	0.0435	22%
SDT	9000	13.0	0.0367	0%
SD_T	9000	15.0	0.0417	13%
S_DT	9000	19.0	0.0430	17%
S_D_T	9000	21.0	0.0477	29%
SDT	10000	13.0	0.0379	0%
SD_T	10000	15.0	0.0426	12%
S_DT	10000	19.0	0.0436	15%
S_D_T	10000	21.0	0.0480	26%

Table 4 Association Type Performance [Network Type: FP2]

Association Type	Data Bytes Transferred	Number of Packets Exchanged	Transfer Time (s)	Percentage Difference
SDT	1	4.0	0.0465	0%
S_DT	1	6.0	0.0778	67%
SD_T	1	6.0	0.0780	67%
S_D_T	1	8.0	0.1091	134%
SDT	1000	4.0	0.0496	0%
SD_T	1000	6.0	0.0810	63%
S_DT	1000	6.0	0.0811	63%
S_D_T	1000	8.0	0.1124	126%
SDT	2000	6.0	0.0840	0%
S_DT	2000	7.0	0.0968	15%
SD_T	2000	8.0	0.1155	37%
S_D_T	2000	9.0	0.1281	52%
SDT	3000	7.0	0.1003	0%
SD_T	3000	9.0	0.1316	31%
S_DT	3000	10.0	0.1451	44%
S_D_T	3000	12.0	0.1766	76%
SDT	4000	7.0	0.1028	0%
SD_T	4000	9.0	0.1336	29%
S_DT	4000	10.0	0.1471	43%
S_D_T	4000	12.0	0.1788	73%
SDT	5000	9.0	0.1362	0%
SD_T	5000	11.0	0.1675	22%
S_DT	5000	12.0	0.1789	31%
S_D_T	5000	14.0	0.2097	53%
SDT	6000	10.0	0.1530	0%
SD_T	6000	12.0	0.1837	20%
S_DT	6000	16.0	0.2416	57%
S_D_T	6000	18.0	0.2713	77%
SDT	7000	10.0	0.1544	0%
SD_T	7000	12.0	0.1855	20%
S_DT	7000	16.0	0.2421	56%
S_D_T	7000	18.0	0.2727	76%
SDT	8000	12.0	0.1854	0%
SD_T	8000	14.0	0.2165	16%
S_DT	8000	17.0	0.2619	41%
S_D_T	8000	19.0	0.2890	55%
SDT	9000	13.0	0.2037	0%
SD_T	9000	15.0	0.2321	13%
S_DT	9000	19.0	0.2929	43%
S_D_T	9000	21.0	0.3227	58%
SDT	10000	13.0	0.2052	0%
SD_T	10000	15.0	0.2332	13%
S_DT	10000	19.0	0.2949	43%
S_D_T	10000	21.0	0.3247	58%

Table 5 Association Type Performance [Network Type: CD1]

Association Type	Data Bytes Transferred	Number of Packets Exchanged	Transfer Time (s)	Percentage Difference
SDT	1	4.0	0.0172	0%
S_DT	1	6.0	0.0267	55%
SD_T	1	6.0	0.0283	64%
S_D_T	1	8.0	0.0376	118%
SDT	1000	4.0	0.0865	0%
S_DT	1000	6.0	0.0960	10%
SD_T	1000	6.0	0.0973	12%
S_D_T	1000	8.0	0.1073	24%
SDT	2000	6.0	0.1654	0%
S_DT	2000	7.0	0.1670	0%
SD_T	2000	8.0	0.1763	6%
S_D_T	2000	9.0	0.1777	7%
SDT	3000	7.0	0.2362	0%
SD_T	3000	9.0	0.2475	4%
S_DT	3000	10.0	0.2477	4%
S_D_T	3000	12.0	0.2583	9%
SDT	4000	7.0	0.3048	0%
SD_T	4000	9.0	0.3154	3%
S_DT	4000	10.0	0.3157	3%
S_D_T	4000	12.0	0.3270	7%
SDT	5000	9.0	0.3828	0%
S_DT	5000	12.0	0.3903	1%
SD_T	5000	11.0	0.3936	2%
S_D_T	5000	14.0	0.4009	4%
SDT	6000	10.0	0.4538	0%
SD_T	6000	12.0	0.4642	2%
S_DT	6000	16.0	0.4753	4%
S_D_T	6000	18.0	0.4864	7%
SDT	7000	10.0	0.5218	0%
SD_T	7000	12.0	0.5323	2%
S_DT	7000	16.0	0.5429	4%
S_D_T	7000	18.0	0.5534	6%
SDT	8000	12.0	0.5965	0%
SD_T	8000	14.0	0.6069	1%
S_DT	8000	17.0	0.6128	2%
S_D_T	8000	19.0	0.6241	4%
SDT	9000	13.0	0.6670	0%
SD_T	9000	15.0	0.6777	1%
S_DT	9000	21.1	0.7012	5%
S_D_T	9000	22.3	0.7067	5%
SDT	10000	13.0	0.7337	0%
SD_T	10000	15.0	0.7449	1%
S_D_T	10000	23.0	0.7775	5%
S_DT	10000	23.0	0.7792	6%

Table 6 Association Type Performance [Network Type: CD2]

Association Type	Data Bytes Transferred	Number of Packets Exchanged	Transfer Time (s)	Percentage Difference
SDT	1	4.0	0.0537	0%
S_DT	1	6.0	0.0816	51%
SD_T	1	6.0	0.0860	60%
S_D_T	1	8.0	0.1142	112%
SDT	1000	4.0	0.2732	0%
SD_T	1000	6.0	0.3059	11%
S_DT	1000	9.4	0.3577	30%
S_D_T	1000	9.6	0.3599	31%
SDT	2000	6.0	0.5212	0%
SD_T	2000	8.0	0.5535	6%
S_D_T	2000	13.0	0.6265	20%
S_DT	2000	13.0	0.6266	20%
SDT	3000	7.0	0.7506	0%
SD_T	3000	9.0	0.7829	4%
S_DT	3000	13.2	0.8414	12%
S_D_T	3000	15.0	0.8704	15%
SDT	4000	7.0	0.9693	0%
SD_T	4000	9.0	1.0017	3%
S_DT	4000	16.2	1.1093	14%
S_D_T	4000	16.4	1.1104	14%
SDT	5000	9.0	1.2168	0%
SD_T	5000	11.0	1.2491	2%
S_DT	5000	18.3	1.3590	11%
S_D_T	5000	18.6	1.3624	11%
SDT	6000	10.0	1.4461	0%
SD_T	6000	12.0	1.4784	2%
S_DT	6000	20.2	1.5972	10%
S_D_T	6000	22.4	1.6336	12%
SDT	7000	10.0	1.6648	0%
SD_T	7000	12.0	1.6973	1%
S_DT	7000	24.2	1.8761	12%
S_D_T	7000	24.3	1.8832	13%
SDT	8000	12.0	1.9088	0%
SD_T	8000	14.0	1.9413	1%
S_D_T	8000	25.8	2.1173	10%
S_DT	8000	25.6	2.1215	11%
SDT	9000	13.0	2.1367	0%
SD_T	9000	15.0	2.1691	1%
S_D_T	9000	27.4	2.3541	10%
S_DT	9000	28.0	2.3644	10%
SDT	10000	13.0	2.3553	0%
SD_T	10000	15.0	2.3877	1%
S_D_T	10000	29.6	2.6088	10%
S_DT	10000	29.8	2.6121	10%

5.2.4 Packet Loss Detection

XTP has two methods for detecting packet loss. The first is through the use of a packet timer. The second is through explicit notification via the Error Control packet and the FASTNAK bit option.

Packet timers are based on the measured round trip time (RTT). The subject of packet loss detections via RTT measurements and packet timers has been well documented by several researchers [Zha86, KP87, FB90]. [Zha86] shows that while timers are mandatory for any reliable protocol, they are a poor mechanism for detecting packet loss and should only be used as a last resort. The reason for this is that a timeout does not provide clear insight into the source and cause of a particular packet timeout. In addition to packet loss, a timeout can be attributed to a sudden increase in network congestion or a delay in the receiving host's processing. Both conditions produce needless packet retransmissions to an already congested network or overloaded host. While maintaining a larger timeout value would alleviate false packet loss detection, it slows down the detection of a true packet loss when it is the only mechanism for packet loss detection.

For this reason, we configure XTP/IP to rely more heavily upon its Error Control Packet mechanism to detect packet loss. XTP can also use its Fast Negative Acknowledgement mechanism. However, since switched IP networks can re-order packets, we decided not to utilize this mechanism.

Even though XTP/IP uses the Error Control packet for detecting packet loss, it is still not "free" from using a packet timer. Error Control packets are effective for notifying the sender of lost data packet, but the sender still requires a packet timeout for detecting Control

type packet loss. XTP uses the WTIMER and the SREQ bit to protect against Control type packet loss. The SREQ bit can be set on any type of packet, data-bearing or Control type. Although there are implicit indications in the XTP Specification for setting the SREQ bit in order to maintain protocol correctness, the Specification does not dictate or state explicitly when to set the SREQ bit. In this section we explicitly identify when a XTP Context MUST set the SREQ bit. We defer the discussion of the effects and implications of setting the SREQ bit on data bearing packets to section 5.3. There are specific scenarios where XTP is only able to detect packet loss with a timeout. For these scenarios XTP must use the SREQ and WTIMER to protect against loss. We identify three scenarios where XTP can only detect a packet loss via the WTIMER:

1 – Loss of a FIRST Packet: since the F packet is the initial packet transmitted its only detection is through a timer.

2 – Loss of the last Data packet in a packet "burst": if a D packet containing the last few bytes in the Allocation window is lost, then a timer is the only method for detection.

3 – Loss of a Control packet: since CONTROL Type packets do not consume sequence numbers, the only recourse for detection is through a packet timer.

For these scenarios, we explicitly require XTP/IP to set the SREQ bit on such outbound packets to ensure packet loss detection.

5.2.5 Fixed Policies for XTP/IP

The XTP Specification provides much flexibility in selecting XTP features in order to optimize XTP's adaptability for various network layer environments. For XTP/IP we fix some of these features across all Transport Traffic Service types.

5.2.5.1 Data Retransmission

XTP permits the receiver and the sender to independently use either Selective Retransmission or Go-Back-N Retransmission. There is a large body of research that clearly show the benefits of using Selective Retransmission for the Internet [FF96, MMFR96]. We explicitly use Selective Retransmission at both the receiver and the sender.

5.2.5.2 Packet Checksumming

For XTP/IP we perform a checksum over the entire packet. This is consistent with existing Internet protocols such as UDP and TCP. We acknowledge however that checksumming is an expensive operation and that for some application layer services, such as multimedia or video streaming, it may be advantageous to disable the full packet check summing. However, for the purpose of this thesis and for the application layer requirements under consideration we explicitly use the full packet checksumming technique.

5.2.5.3 First Packet Traffic Format

Since we do not explore XTP's Rate Control mechanisms in this work, we utilize the NULL Traffic format for XTP/IP. This reduces the packet byte overhead for the FIRST and Traffic Control packets.

5.2.6 XTP's First Packet Hazard Condition

We conclude the discussion of Connection Management for XTP/IP with a hazard condition that occurs with a replay of XTP's FIRST packet. This hazard condition requires that the XTP First Packet Matching Algorithm be enhanced when utilized in an IP network. The source of the problem is that IP networks can replay a duplicate FIRST packet, either accidentally or by malicious intent. If a duplicate FIRST packet arrives at a host shortly after

that association has terminated and the host is not in a Zombie State (because it did not send the END bit), the duplicate FIRST packet will be processed as a valid packet and will set up a new XTP association.

This condition is a well known problem for transport protocols over IP networks. The TCP specification specifically deals with this condition through the use of Initial Sequence Numbers and a 3-way handshake:

> The protocol places no restriction on a particular connection being used over and over again. A connection is defined by a pair of sockets. New instances of a connection will be referred to as incarnations of the connection. The problem that arises from this is -- "how does the TCP identify duplicate segments from previous incarnations of the connection?" This problem becomes apparent if the connection is being opened and closed in quick succession, or if the connection breaks with loss of memory and is then reestablished.
>
> To avoid confusion we must prevent segments from one incarnation of a connection from being used while the same sequence numbers may still be present in the network from an earlier incarnation. We want to assure this, even if a TCP crashes and loses all knowledge of the sequence numbers it has been using. When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32 bit ISN. The generator is bound to a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds. Thus, the ISN cycles approximately every 4.55 hours. Since we assume that segments will stay in the network no more than the Maximum Segment Lifetime (MSL) and that the MSL is less than 4.55 hours we can reasonably assume that ISN's will be unique. . .A three way handshake is necessary because sequence numbers are not tied to a global clock in the network, and TCPs may have different mechanisms for picking the ISN's. The receiver of the first SYN has no way of knowing whether the segment was an old delayed one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN. [Post81]

Figure 16 shows the normal TCP packet exchange for a typical data transfer. Figure 17 shows how TCP reacts to a duplicate SYN packet.

Server		Client
	< - - - -	SYN1
SYN2+ack	- - - - >	
	< - - - -	ack
	... data transferred ...	
	... connection terminated ...	

Figure 16 Typical TCP Data Transfer

Server		Client
	< - - - -	SYN1 (duplicate packet from connection in figure 16)
SYN2+ack	- - - - >	
	< - - - -	RST (the client sends a Reset packet since it did not initiate a connection)

Figure 17 TCP Handling of Duplicate SYN packet

The non-data bearing 3-way handshake protects TCP from passing "old" user data to the application layer in case of duplicate SYN packets.

Since XTP permits user data in its FIRST packet and also permits a 2-way handshake we examine now how XTP would currently handle a duplicate FIRST packet. We first note that XTP does not have a unique ISN. This is due to the fact that the role of the unique ISN in TCP is performed by the KEY field in XTP. The XTP Specification states that the KEY value monotonically increases for each new Context instance. Since the KEY field is 64-bit, the wrap around time about 1.3 days on a Pentium II computer (observed from empirical testing), well beyond the 2 minute Maximum Segment Life (MSL) recommended. However, since XTP permits a 2-way handshake, there is a potential hazard for duplicate FIRST packets in certain environments. Figure 18 illustrates the typical packet exchange for a data bearing FIRST packet transfer using a 2 packet connection setup.

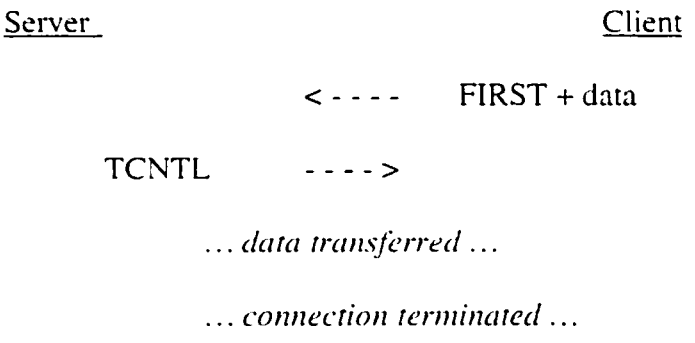

Figure 18 Typical XTP Data Transfer

The FIRST packet hazard occurs when the FIRST packet in Figure 18 is duplicated and delivered to the host Server after the association ends. Figure 19 shows what would occur if such a duplicate packet is received by the host Server.

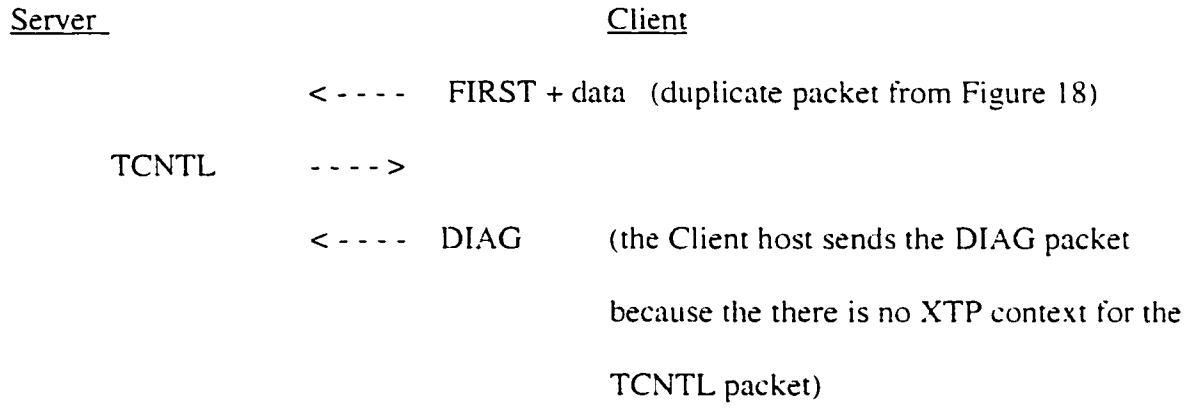

Figure 19 XTP Handling of Duplicate FIRST packet

The danger in Figure 19 is that the XTP Context in host Server has already processed the FIRST packet and delivered the user data to the application layer. This can create potentially devastating resource consumption problems on the host. One way to prevent this is to introduce TCP like policies of requiring a 3-way handshake for the setup phase and preventing user data in the FIRST packet from being passed on to the application layer until the completion of the handshake phase. Although such policies are permitted and legal according to the XTP Specification, it reduces XTP to the TCP protocol and negates any possible performance improvements.

We suggest a different approach that keeps the 2-way handshake for the setup phase, delivers user data in the FIRST packet, and eliminates the FIRST Packet Hazard condition.

We accomplish this by introducing a new timer for the Address Translation Mapping table. This timer delays the deletion of an entry in the Address Translation Mapping table at the end of an XTP connection. All incoming FIRST packets are processed according to the First Packet Matching (FPM) algorithm. FIRST packets go through a Full Context Lookup

to determine if it belongs to an active association (i.e., if it is a duplicate while the connection is current). We introduce a new condition check to identify a duplicate FIRST packet from previous connections.

All FIRST packets are looked up on two criteria, the IP address (IP1) and the KEY field (KEY1). The modified FPM algorithm is as follows:

For an incoming FIRST packet with a given IP1 and KEY1:

1 – if IP1 and KEY1 already exist in the ATM table, then this is a duplicate FIRST packet:

a) if there is no ATM Timer active, then the FIRST packet is a duplicate of a *current* connection. As per XTP Specification section 4.2.1 step 1, respond to the SREQ/DREQ bit if set.

b) If there is an ATM Timer active, then the FIRST packet is a duplicate of a previous connection. Silently drop the packet.

2 – if IP1 and KEY1 do not exist in the ATM, then proceed with the FPM algorithm.

The ATM Timer delays the deletion of an IP-KEY entry in the ATM table. It is set to 2xMSL. The ATM Timer is activated when an existing entry is about to de deleted from the ATM table. Upon the ATM Timer expiration, the entry is removed from the ATM table.

The XTP traces in Figure 20 show an example of the FIRST Packet Hazard, and Figure 21 shows how XTP handles the same scenario using the ATM Timer.

In Figure 21, line #9, the duplicate FIRST packet is received by the host "server" and it responds with a TCNTL packet. When the host "client" receives the TCNTL packet it is still in the ZOMBIE state and therefore responds with a CNTL packet with the END bit set.

However the host "server" has already processed the FIRST packet. In Figure 21, the duplicate FIRST packet is spotted by the ATM Timer and is silently dropped.

1	951693271.096953 client > server: F ky3 in0 dl24 sy1 sq90000 SRQ| al16 ad9216 at256 dp9999 dh:192.200.200.200 sp1284 sh:192.100.100.100 tl8 sv4 tl0
2	951693271.098131 server > client: T ky3 in0+RTN dl40 sy0 sq0 rq24 al90024 ec1 xk35 tl8 sv4 tl0
3	951693271.098547 client > server: D ky35 in0+RTN dl1000 sy2 sq24 EOM|SRQ|
4	951693271.102558 server > client: C ky3 in0+RTN dl20 sy0 sq0 rq1024 al91024 ec2
5	951693271.102816 client > server: C ky35 in0+RTN dl20 sy3 sq1024 WCL|SRQ| rq0 al90000 ec0
6	951693271.103764 server > client: C ky3 in0+RTN dl20 sy0 sq0 RCL| rq1024 al91024 ec3
7	951693271.103992 server > client: C ky3 in0+RTN dl20 sy1 sq0 WCL|RCL|SRQ| rq1024 al91024 ec3
8	951693271.104036 client > server: C ky35 in0+RTN dl20 sy3 sq1024 END|WCL|RCL| rq0 al90000 ec1
9	951693277.096505 client > server: F ky3 in0 dl24 sy1 sq90000 SRQ| al16 ad9216 at256 dp9999 dh:192.200.200.200 sp1284 sh:192.100.100.100 tl8 sv4 tl0
10	951693277.097619 server > client: T ky3 in0+RTN dl40 sy0 sq0 rq24 al90024 ec1 xk36 tl8 sv4 tl0
11	951693277.097703 client > server: C ky35 in0+RTN dl20 sy3 sq1024 END|WCL|RCL| rq0 al90000 ec0

Figure 20 XTP FIRST Packet Hazard

1	951693479.222417 client > server: F ky5 in0 dl24 sy1 sq90000 SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp1796 sh:192.100.100.100 t18 sv4 tf0
2	951693479.223664 server > client: T ky5 in0+RTN dl40 sy0 sq0 rq24 al90024 ec1 xk41 t18 sv4 tf0
3	951693479.224562 client > server: D ky41 in0+RTN dl1000 sy2 sq24 EOM|SRQ|
4	951693479.228578 server > client: C ky5 in0+RTN dl20 sy0 sq0 rq1024 al91024 ec2
5	951693479.229023 client > server: C ky41 in0+RTN dl20 sy3 sq1024 WCL|SRQ| rq0 al90000 ec0
6	951693479.230008 server > client: C ky5 in0+RTN dl20 sy0 sq0 RCL| rq1024 al91024 ec3
7	951693479.230233 server > client: C ky5 in0+RTN dl20 sy1 sq0 WCL|RCL|SRQ| rq1024 al91024 ec3
8	951693479.230281 client > server: C ky41 in0+RTN dl20 sy3 sq1024 END|WCL|RCL| rq0 al90000 ec1
9	951693485.216496 client > server: F ky5 in0 dl24 sy1 sq90000 SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp1796 sh:192.100.100.100 t18 sv4 tf0

Figure 21 XTP with ATM Timer and Duplicate FIRST packet

5.3 Congestion Management

Congestion management is the management of the packet transmission rate of a transport protocol. The objective is to balance the performance requirements of the application layer with the bandwidth availability of the network layer.

As pointed out in section 4.2.2, the lack of an explicit congestion management policy in XTP is detrimental to its stability and performance in a switched IP environment and hinders its acceptance by the Internet community. With a clear need for a well behaved congestion management policy for XTP we propose a set of congestion management components tailored for XTP/IP.

In this section we are concerned with Round Trip Time (RTT) measurements and estimates, Packet Acknowledgement Frequency, and Congestion Control algorithms.

5.3.1 Background

Transport protocols try to transmit user data as quickly and efficiently as possible. Due to limited network resources and other competing transport sessions, network congestion occurs, requiring a protocol to adjust its transmission rate accordingly so that all participating hosts can fairly utilize the network resources and so that a network congestion collapse is averted. A key component to congestion management is determining the available bandwidth between the two communicating hosts. This is a challenge because the effective bandwidth is dynamic. Therefore the detection of drastic changes in the existing level of effective bandwidth is crucial. With a given change in bandwidth a protocol has 3 options:

1 - increase transmission rate

2- decrease transmission rate

3 - maintain current transmission rate

The set of algorithms and polices a protocol uses to manage this constitutes the congestion management.

It is important to understand the dynamics of congestion management for a transport session since the flow of packets significantly impacts the performance of long data transfers and can impact the stability of the Internet. Given the objectives of optimal performance and equitable sharing of resources, there are two issues of concern in congestion management. The first is how quickly should a protocol increase the transmission rate in order to maximize performance, and the second is how should a protocol respond to congestion indicators. Due to the inherent nature of the IP layer, there is no a priori knowledge of available bandwidth or of existing congestion. Therefore transport protocols "probe" the network to determine the existing level of bandwidth by continually increasing the rate of packet transmission. If all packets are successfully acknowledged, then this is interpreted as there being more bandwidth available. If data is lost, then this is interpreted as a congestion indicator.

A protocol has 5 choices for managing the packet transmission rate. Each choice relates to a given network bandwidth indicator. If a protocol detects undersaturated bandwidth, it increases the transmission rate aggressively. If it detects or suspects the transmission rate is nearing the bandwidth saturation point, it slows down the aggressive increase and begins increasing the transmission rate very slowly. If moderate congestion is detected the protocol backs off and decreases the transmission rate. If it detects severe congestion it decreases its transmission rate drastically to avert further worsening of the

congestion condition. If the protocol detects no or little bandwidth change it performs no increase or decrease to the transmission rate.

We briefly review the existing congestion management of TCP to illustrate these principles. TCP has a well defined and developed congestion management. To probe the initial bandwidth availability, TCP performs "slow-start". This increases the packet transmission rate exponentially. At a predefined level TCP performs "congestion avoidance", which increases the transmission rate linearly. TCP continues to increase the transmission rate until it detects congestion via a packet loss, either by receiving duplicate ACK packets or a packet timeout. If a packet timeout occurs, TCP drastically reduces its transmission rate with the assumption that severe network congestion has caused the packet timeout. If TCP detects duplicate ACKs, it performs a smaller reduction in its transmission rate, assuming that the cause of the duplicate ACKs is due to minor or temporary network congestion. It is worth noting that TCP's congestion management has evolved over a period of time. It is from actual Internet experience that TCP's designers have modified and improved TCP's transmission algorithms and procedures.

Before we introduce XTP's proposed congestion control algorithms, we examine the roles of Round Trip Time estimates and Data Acknowledgement Frequency. Both of these have a strong influence on congestion management.

5.3.2 Round Trip Time Estimates

A critical component of congestion control management is the Round Trip Time (RTT) estimation algorithm. The RTT is used for two purposes: 1) as the basis for determining a packet retransmission timeout, and 2) as an indicator of network congestion. In section 5.2 we illustrated that XTP minimizes its dependency on retransmission timeouts for packet loss

detection. However, XTP does rely on packet timeouts to detect network congestion, so the accuracy of the RTT algorithm is a critical component of congestion management.

While XTP's RTT estimation algorithm is not dictated by the XTP Specification, Van Jacobson's algorithm [JK88] is strongly recommended. It already serves as TCP's RTT estimation algorithm. A way to improve the accuracy of Van Jacobson's algorithm is to increase the sampling of RTT measurements. For this reason, we lean towards configuring XTP's Acknowledgement Frequency policy to generate as many RTT measurements as possible during a transmission.

From a protocol design point of view, XTP is well suited for generating and processing a high number of RTT measurements during a transmission. A strong feature of XTP is its ability to identify an incoming acknowledgement packet with its outbound SREQ packet. This identification is based on the SYNC/ECHO field values. This feature enables XTP to accurately measure and process RTTs. This is in contrast to TCP which has no explicit method to match an acknowledgement packet with a data packet. As a result TCP has a well known ambiguity problem with RTT estimates and retransmitted packets [Zha86, KP87]. The issue of how many acknowledgement packets to generate for RTT purposes is dealt with later in section 5.3.3. From our experiences with LinuxXTP, we have found two anomalies with RTT measurements: 1) RTT Starvation and 2) Association Type Dependency.

5.3.2.1 RTT Starvation

Although XTP does not suffer from RTT ambiguity, it can under certain scenarios suffer from a type of "measurement starvation". This is a result of a subtle dependency between the WTIMER algorithm and the setting of the SREQ bit. Since RTT measurements are critical

in establishing the packet timeout timer, WTIMER, we investigate this further to develop solutions for XTP's RTT startvation.

In XTP, the Round Trip Time is measured by setting the SREQ bit on an outbound packet and time stamping its SYNC value. When the packet is acknowledged with a Control packet, XTP uses the ECHO field to identify the corresponding SYNC value. A time delta of the SYNC's timestamp and the ECHO's arrival time generates an RTT measurement. The XTP Specification requires a WTIMER to be started for every outbound packet that has the SREQ bit set. It does not specify however, if there should be multiple WTIMERs in-progress when multiple SREQ-bearing packets are transmitted, or if there should be only 1 active WTIMER for the most recently transmitted SREQ-bearing packet. Section 4.3 of the XTP Specification identifies this behaviour as an implementation choice. In this paper we will refer to this choice as the N-Concurrent-WTIMER Policy, where n is the number of in-progress WTIMERs. An implementation that keeps only one WTIMER active at any given time is referred to as a 1-Concurrent-WTIMER Policy. For LinuxXTP we implement a 1-Concurrent-WTIMER Policy.

Under certain conditions a measurement starvation exists which prevents an XTP Context from obtaining any RTT measurements. To illustrate this scenario, we assume an implementation that has a 1-Concurrent-WTIMER. This means that every time the SREQ bit is transmitted, any existing WTIMER is deleted and a new WTIMER is started. If we select an Acknowledgement Frequency Policy of 1 (ack every Data packet) and have a non-zero RTT (which is almost always), then every WTIMER will be cancelled except for the last outbound SREQ-bearing packet, (usually the last DATA packet in the Allocation window).

We instrumented the XTP code in LinuxXTP to track the number of RTT estimates performed (**rtt_updates**) for a given connection and obtained this information via a getsockopt() system call. We also tracked the total number of SREQ bits transmitted (**Sync**) and the number of SREQ transmitted with a DATA packet (**Data_sreq_sent**). Figures 22 to 24 are metrics obtained from a transmission of 100 000 bytes. We varied the Ack Freq with values of 1, 5, and 99.

From figures 22 to 24 we note that when the Acknowledgement Frequency is high (Ack Freq=1), the RTT estimate is updated much less frequently. Out of 70 SREQs transmitted (Sync), only 3 RTT estimates were performed. Ideally there should be a one to one ratio of SREQs transmitted to number of RTT updates. For Ack Freq=99, we achieve this because the Ack Freq setting is set to such a high value that a SREQ is sent only when absolutely required by the protocol for protocol correctness, which is usually less than 1 SREQ per round trip.

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=70 T=1 E=0;  Total=71
Congestion Management:
    FCwaits=19  RCwaits=0
    Sync=70  Data_sreq_sent=69  Rtt_updates=3  Ack_freq=1
    EdgeBit Set=0  RTT_cache_size=1
    Initial_trainsize=1448  Final_trainsize=36200 Cong_trainsize=36200
    Linear_threshold=32000  Linear_count=112
    Exp Inc=22  Exp Dec=0  Lin Inc=49  Lin Dec=0
XTP Timers:
    wtimerTMO=8  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 22 RTT Updates and ACK FREQ=1

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=27 T=1 E=0;  Total=28
Congestion Management:
    FCwaits=13  RCwaits=0
    Sync=27  Data_sreq_sent=26  Rtt_updates=4  Ack_freq=5
    EdgeBit Set=0  RTT_cache_size=1
    Initial_trainsize=1448  Final_trainsize=34752 Conq_trainsize=34752
    Linear_threshold=32000  Linear_count=33416
    Exp Inc=9  Exp Dec=0  Lin Inc=19  Lin Dec=0
XTP Timers:
    wtimerTMO=8  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 23 RTT Updates and ACK FREQ=5

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=7 T=1 E=0;  Total=8
Congestion Management:
    FCwaits=4  RCwaits=0
    Sync=7  Data_sreq_sent=6  Rtt_updates=7  Ack_freq=99
    EdgeBit Set=0  RTT_cache_size=1
    Initial_trainsize=1448  Final_trainsize=47784 Conq_trainsize=47784
    Linear_threshold=32000  Linear_count=8800
    Exp Inc=5  Exp Dec=0  Lin Inc=3  Lin Dec=0
XTP Timers:
    wtimerTMO=9  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 24 RTT Updates and ACK FREQ=99

5.3.2.2 Solutions for RTT Starvation

There are two techniques to avoid RTT starvation in XTP. The objective is to maximize the number of RTT updates such that each SREQ bit setting generates a valid RTT estimate.

The first method is to use a cache of timestamps-SYNC tuples of recently transmitted SREQ packets. With such a cache, even if the WTIMER for a given packet is cancelled, the reply packet may find its ECHO/SYNC value in the cache. If a cache lookup is successful, then a RTT measurement can still be performed.

For LinuxXTP, we implemented such a cache and refer to it as the RTT Cache. By default it has a cache size of 10 entries (maximum size), but for testing purposes we can vary its size on a per association basis. To illustrate the algorithm for the RTT Cache, we define 6 variables:

CS – RTT cache size

SYNC - sync value of the outbound SREQ bearing packet

Stm - timestamp of the SREQ bearing packet

ECHO - echo value of an inbound Control packet

Etm - timestamp of an inbound Control packet

INDEX – index into the RTT cache [(ECHO % CS) or (SYNC % CS), where % is the modulo function]

The RTT Cache algorithm is outlined below:

1 – Transmit a SREQ-bearing packet according to the XTP Specification.

2 – Calculate the INDEX value (SYNC % CS) and store

rttcache[INDEX].tm=Stm

rttcache[INDEX].sync=SYNC

3 – When a Control Packet arrives, calculate the INDEX value (ECHO % CS) and look up rttcache[INDEX].

A) If rttcache[INDEX].sync == ECHO, then calculate RTT measurement as

Etm – rttcache[INDEX].Stm. Then clear rttcache[INDEX].sync and rttcache[INDEX].Stm.

B) If rttcache[INDEX].sync != ECHO, then skip rtt measurement.

There is a limit to the effectiveness of the RTT Cache technique. Starvation will still occur if the following condition is not met:

Condition A: SREQ Freq < = RTT / CS ,for 1-Concurrent-WTIMER Policy

Condition B: SREQ Freq <= (RTT / CS) * N , for N-Concurrent-WTIMER Policy

To test the effectiveness of this technique, we run the same 3 data transmission tests from figures 22-24 using a RTT Cache size of 10 slots. Figures 25 to 27 show the results. The number of transmitted SREQ remains the same as before but there are significantly more RTT updates. In Figure 26 (ACK FREQ=5), we achieve a 1 to 1 ratio of SREQ to RTT updates where we previously did not (see Figure 23 ACK FREQ=5).

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=70 T=1 E=0;  Total=71
Congestion Management:
    FCwaits=20  RCwaits=0
    Sync=70  Data_sreq_sent=69  Rtt_updates=46  Ack_freq=1
    EdgeBit Set=0  RTT_cache_size=10
    Initial_trainsize=1448  Final_trainsize=36200 Cong_trainsize=36200
    Linear_threshold=32000  Linear_count=112
    Exp Inc=22  Exp Dec=0  Lin Inc=49  Lin Dec=0
XTP Timers:
    wtimerTMO=7  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 25 RTT Updates with RTT-Cache, ACK FREQ=1

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=26 T=1 E=0;  Total=27
Congestion Management:
    FCwaits=12  RCwaits=0
    Sync=26  Data_sreq_sent=25  Rtt_updates=26  Ack_freq=5
    EdgeBit Set=0  RTT_cache_size=10
    Initial_trainsize=1448  Final_trainsize=34752 Conq_trainsize=34752
    Linear_threshold=32000  Linear_count=33416
    Exp Inc=9  Exp Dec=0  Lin Inc=18  Lin Dec=0
XTP Timers:
    wtimerTMO=6  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 26 RTT Updates with RTT-Cache, ACK FREQ=5

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=7 T=1 E=0;  Total=8
Congestion Management:
    FCwaits=4  RCwaits=0
    Sync=7  Data_sreq_sent=6  Rtt_updates=7  Ack_freq=99
    EdgeBit Set=0  RTT_cache_size=10
    Initial_trainsize=1448  Final_trainsize=47784 Conq_trainsize=47784
    Linear_threshold=32000  Linear_count=8800
    Exp Inc=5  Exp Dec=0  Lin Inc=3  Lin Dec=0
XTP Timers:
    wtimerTMO=11  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 27 RTT Updates with RTT-Cache, ACK FREQ=99

The second technique is to decouple the setting of the SREQ bit from the Acknowledgement Frequency by using a combination of SREQ and EDGE bits. XTP can generate an acknowledgement packet with either an SREQ bit or with a change in the EDGE bit. The SREQ bit always has a WTIMER, the EDGE bit does not. We can decouple the

Acknowledgement Frequency from the SREQ-bit Frequency by selectively using the EDGE bit to generate an Acknowledgement Packet. The algorithm for a Selective SREQ-bit Frequency for a given Acknowledgement Frequency Policy is as follows:

1) Send a SREQ whenever required for protocol correctness (section 5.2).

2) Send a SREQ according to the Acknowledgement Frequency Policy unless Condition A fails.

3) If Condition A fails, use the EDGE bit to trigger an Acknowledgement Packet.

We test the effectiveness of this technique independently of the RTT Cache technique. In LinuxXTP we can enable or disable the SREQ-Ack Freq coupling on a per association basis. Figures 28 – 30 show the results of running the same three tests as before. We note the number of RTT updates do not improve with SREQ-Ack Freq decoupling. However, the number of SREQ transmitted is reduced since the EDGE bit is being used. Thus with SREQ-Ack Freq decoupling, the efficiency of LinuxXTP's RTT estimates improve (the ratio of SREQ to RTT updates) but the effectiveness remains the same.

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=70 T=1 E=0;  Total=71
Congestion Management:
    FCwaits=18  RCwaits=0
    Sync=23  Data_sreq_sent=22  Rtt_updates=3  Ack_freq=1
    EdgeBit Set=65  RTT_cache_size=1
    Initial_trainsize=1448  Final_trainsize=36200 Cong_trainsize=36200
    Linear_threshold=32000  Linear_count=112
    Exp Inc=22  Exp Dec=0  Lin Inc=49  Lin Dec=0
XTP Timers:
    wtimerTMO=8  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 28 RTT Updates with Decoupled Ack Freq, ACK FREQ=1

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=26 T=1 E=0;  Total=27
Congestion Management:
    FCwaits=10  RCwaits=0
    Sync=15  Data_sreq_sent=14  Rtt_updates=4  Ack_freq=5
    EdgeBit Set=11  RTT_cache_size=1
    Initial_trainsize=1448  Final_trainsize=34752 Cong_trainsize=34752
    Linear_threshold=32000  Linear_count=33416
    Exp Inc=9  Exp Dec=0  Lin Inc=18  Lin Dec=0
XTP Timers:
    wtimerTMO=8  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 29 RTT Updates with Decoupled Ack Freq, ACK FREQ=5

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=7 T=1 E=0;  Total=8
Congestion Management:
    FCwaits=4  RCwaits=0
    Sync=7  Data_sreq_sent=6  Rtt_updates=7  Ack_freq=99
    EdgeBit Set=0  RTT_cache_size=1
    Initial_trainsize=1448  Final_trainsize=47784 Cong_trainsize=47784
    Linear_threshold=32000  Linear_count=8800
    Exp Inc=5  Exp Dec=0  Lin Inc=3  Lin Dec=0
XTP Timers:
    wtimerTMO=12  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 30 RTT Updates with Decoupled Ack Freq, ACK FREQ=99

Since these two techniques operate independently, we combine them to maximize their benefit. We run the same three tests as before and present the results in Figures 31 to 33.

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=70 T=1 E=0;  Total=71
Congestion Management:
    FCwaits=19  RCwaits=0
    Sync=54  Data_sreq_sent=53  Rtt_updates=53  Ack_freq=1
    EdgeBit Set=18  RTT_cache_size=10
    Initial_trainsize=1448  Final_trainsize=36200 Cong_trainsize=36200
    Linear_threshold=32000  Linear_count=112
    Exp Inc=22  Exp Dec=0  Lin Inc=49  Lin Dec=0
XTP Timers:
    wtimerTMO=17  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 31 RTT Updates with RTT-Cache and Decouple Ack Freq, ACK FREQ=1

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=26 T=1 E=0;  Total=27
Congestion Management:
    FCwaits=10  RCwaits=0
    Sync=26  Data_sreq_sent=25  Rtt_updates=26  Ack_freq=5
    EdgeBit Set=0  RTT_cache_size=10
    Initial_trainsize=1448  Final_trainsize=34752 Cong_trainsize=34752
    Linear_threshold=32000  Linear_count=33416
    Exp Inc=9  Exp Dec=0  Lin Inc=18  Lin Dec=0
XTP Timers:
    wtimerTMO=11  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 32 RTT Updates with RTT-Cache and Decouple Ack Freq, ACK FREQ=5

```
Association Info:
    Assoc_type=1 Teardown_method=1 Service=4
    Pk sent: F=1 D=69 DG=0 C=2 T=0 E=0;  Total=72     (Re-tranxmit:DpkOut=0)
    Pk recv: F=0 D=0 DG=0 C=7 T=1 E=0;  Total=8
Congestion Management:
    FCwaits=4  RCwaits=0
    Sync=7  Data_sreq_sent=6  Rtt_updates=7  Ack_freq=99
    EdgeBit Set=0  RTT_cache_size=10
    Initial_trainsize=1448  Final_trainsize=47784 Cong_trainsize=47784
    Linear_threshold=32000  Linear_count=8800
    Exp Inc=5  Exp Dec=0  Lin Inc=3  Lin Dec=0
XTP Timers:
    wtimerTMO=12  ctimerTMO=60000  ctimeoutTMO=42000  zombieTMO=1000
    HandShake=0
```

Figure 33 RTT Updates with RTT-Cache and Decouple Ack Freq, ACK FREQ=99

Comparing the initial three tests runs (Figures 22 - 24) and Figures 31 - 33, we find a much higher number of RTT updates and the ideal 1-1 ratio of SREQ transmitted to RTT updates.

Implementing both techniques improves the efficiency (due to Ack Freq decoupling) and the effectiveness (due to RTT Cache) of XTP's RTT measurements.

5.3.2.3 RTT Estimation Function

Another key component to XTP's congestion management is the RTT estimator function used to determine the WTIMER value. The XTP Specification recommends Van Jacobson's estimator function. This function has since been updated to include a higher 'additive' factor to the formula and is now used in most TCP implementation.

Initial experience in LinuxXTP with the updated estimator function generated peculiar packet timeouts. The problem occurs when separate Setup and Data Transfer Phases are used in a network environment that is sensitive to propagation delays (i.e., size of packet influences latency). Under such conditions, the First Packet/Traffic Packet exchange sets an incorrect value for the initial RTT.

We examine in detail two traces of a 5000 byte transfer in a CD1 Network. Figure 34 displays the packet trace of an SDT association type and Figure 35 an S_D_T association type. In Figure 35, line #5 a Control packet is sent by the sender due to a WTIMER timeout. By comparing the RTT of the First Packet/Traffic Packet exchanges from the two traces we can determine the cause of the WTIMER timeout. The RTTs for the First/Traffic Packet exchange and the first Data/Control Packet exchange in SDT are 0.159 secs and 0.303 secs. For S_D_T, they are 0.017 secs and 0.169 secs. The increase for SDT is a factor of 2, which is within the WTIMER timeout range. For S_D_T the increase is a factor of 10, which triggers a timeout. The non-data bearing First Packet in S_D_T generates a false timeout value for the maxdata size Data Packet that follows.

To resolve this problem we re-evaluated Van Jacobson's formula and modified the 'additive' factor. The LinuxXTP RTT estimator functions doubles the timeout value from Van Jacobson's original formula. We implement this modification for three reasons:

1) XTP relies less heavily on timeouts for packet loss detection than other protocols, thus affording XTP the luxury of a higher timeout value.
2) the cost of a false timeout in XTP is at least 1 RTT.
3) the Linux implementation of TCP's RTT estimator function also adds a higher (but different) 'additive' factor (see Appendix E).

1	21:01:54.599009 client > server: F ky73 in0 dl1448 sy1 sq90000 SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp4 sh:192.100.100.100 tl8 sv4 tf0
2	21:01:54.758015 server > client: T ky73 in0+RTN dl40 sy0 sq0 | rq1448 al91448 ec1 xk75 tl8 sv4 tf0
3	21:01:54.759402 client > server: D ky75 in0+RTN dl1448 sy1 sq1448 |
4	21:01:54.759965 client > server: D ky75 in0+RTN dl1448 sy2 sq2896 SRQ|
5	21:01:55.062002 server > client: C ky73 in0+RTN dl20 sy0 sq0 | rq4344 al94344 ec2
6	21:01:55.063099 client > server: D ky75 in0+RTN dl680 sy3 sq4344 EOM|WCL|SRQ|
7	21:01:55.143715 server > client: C ky73 in0+RTN dl20 sy0 sq0 RCL| rq5024 al95024 ec3
8	21:01:55.150691 server > client: C ky73 in0+RTN dl20 sy1 sq0 WCL|RCL|SRQ| rq5024 al95024 ec3
9	21:01:55.150742 client > server: C ky75 in0+RTN dl20 sy3 sq5024 END|WCL|RCL| rq0 al90000 ec1

Figure 34 Trace of Normal RTT with SDT for 5000 Bytes

1	21:01:41.343380 client > server: F ky72 in0 dl24 sy1 sq90000 SRQ| al16 ad9216 af256 dp9999 dh:192.200.200.200 sp4 sh:192.100.100.100 tl8 sv4 tf0
2	21:01:41.360788 server > client: T ky72 in0+RTN dl40 sy0 sq0 | rq24 al90024 ec1 xk74 tl8 sv4 tf0
3	21:01:41.363254 client > server: D ky74 in0+RTN dl1448 sy1 sq24 |
4	21:01:41.363851 client > server: D ky74 in0+RTN dl24 sy2 sq1472 SRQ|
5	21:01:41.457050 client > server: C ky74 in0+RTN dl20 sy3 sq1496 SRQ| rq0 al90000 ec0
6	21:01:41.532595 server > client: C ky72 in0+RTN dl20 sy0 sq0 | rq1496 al91496 ec2
7	21:01:41.546539 server > client: C ky72 in0+RTN dl20 sy0 sq0 | rq1496 al91496 ec3
8	21:01:41.547319 client > server: D ky74 in0+RTN dl1424 sy3 sq1496 |
9	21:01:41.547874 client > server: D ky74 in0+RTN dl1448 sy4 sq2920 SRQ|
10	21:01:41.549078 client > server: D ky74 in0+RTN dl656 sy5 sq4368 EOM|SRQ|
11	21:01:41.817034 client > server: C ky74 in0+RTN dl20 sy6 sq5024 SRQ| rq0 al90000 ec0
12	21:01:41.922499 server > client: C ky72 in0+RTN dl20 sy0 sq0 | rq4368 al94368 ec4
13	21:01:41.936536 server > client: C ky72 in0+RTN dl20 sy0 sq0 | rq5024 al95024 ec5
14	21:01:41.943514 server > client: C ky72 in0+RTN dl20 sy0 sq0 | rq5024 al95024 ec6
15	21:01:41.944372 client > server: C ky74 in0+RTN dl20 sy7 sq5024 WCL|SRQ| rq0 al90000 ec0
16	21:01:41.959281 server > client: C ky72 in0+RTN dl20 sy0 sq0 RCL| rq5024 al95024 ec7
17	21:01:41.966259 server > client: C ky72 in0+RTN dl20 sy1 sq0 WCL|RCL|SRQ| rq5024 al95024 ec7
18	21:01:41.966310 client > server: C ky74 in0+RTN dl20 sy7 sq5024 END|WCL|RCL| rq0 al90000 ec1

Figure 35 Trace of RTT Anomaly with S_D_T for 5000 Bytes

5.3.3 Data Acknowledgement Frequency Policies

The data acknowledgement frequency policy determines how often a transmitter receives acknowledgement packets from the receiving host. Acknowledgement packets return status information from the receiver about received data bytes and out of sequence data bytes. XTP utilizes data acknowledgements to 1) detect lost data, 2) update its congestion management information, and 3) determine round trip times.

Unlike most other protocols, the sender in a XTP association determines all aspects of the data transmission including how often the receiver is to transmit data acknowledgement packets. In TCP for example, it is the receiver that determines the acknowledgement frequency. This is a fixed policy outlined by [Brad89]. TCP sends an acknowledgement for every second full sized packet or if more than 500ms has elapsed since the last acknowledgement was transmitted.

In contrast to this fixed policy, XTP allows the sender to determine when and how often it is to receive data acknowledgements. Selecting a data acknowledgement frequency is non-trivial. We investigate what effect acknowledgement frequency policies may have on performance for XTP/IP.

Table 7 Acknowledgement Frequency Performance - LAN

Ack Freq	Data Bytes Transferred	Number of Packets Exchanged	Number of SREQ bits Transmitted	Transfer Time (s)	Percentage Difference
999	100000	79.0	8.0	0.1187	0%
4	100000	96.8	25.8	0.1222	2%
5	100000	95.4	24.4	0.1247	5%
3	100000	101.6	30.6	0.1251	5%
2	100000	108.5	37.3	0.1285	8%
1	100000	142.0	54.8	0.1338	12%

Table 8 Acknowledgement Frequency Performance – FP1

Ack Freq	Data Bytes Transferred	Number of Packets Exchanged	Number of SREQ bits Transmitted	Transfer Time (s)	Percentage Difference
999	100000	79.0	8.0	1.5531	0%
5	100000	100.0	29.0	1.9557	25%
4	100000	101.0	30.0	1.9672	26%
3	100000	103.0	32.0	2.0102	29%
2	100000	109.0	38.0	2.1322	37%
1	100000	142.0	62.0	2.7606	77%

Table 9 Acknowledgement Frequency Performance – FP2

Ack Freq	Data Bytes Transferred	Number of Packets Exchanged	Number of SREQ bits Transmitted	Transfer Time (s)	Percentage Difference
999	100000	79.0	8.0	2.9598	0%
5	100000	100.0	29.0	3.7422	26%
4	100000	101.0	30.0	3.7705	27%
3	100000	103.0	32.0	3.8543	30%
2	100000	109.0	38.0	4.0808	37%
1	100000	142.0	62.0	5.3105	79%

Table 10 Acknowledgement Frequency Performance – CD1

Ack Freq	Data Bytes Transferred	Number of Packets Exchanged	Number of SREQ bits Transmitted	Transfer Time (s)	Percentage Difference
999	100000	79.0	8.0	5.1553	0%
5	100000	100.0	29.0	5.2256	1%
4	100000	101.0	30.0	5.2286	1%
3	100000	104.0	33.0	5.2372	1%
2	100000	109.0	38.0	5.2569	1%
1	100000	142.0	62.0	5.3828	4%

For the LAN and CD network environment, the Ack Freq has negligible impact on performance. For Fat Pipe networks (FP1, FP2) the Ack Freq has significant impact, up to 79% in the best/worst case. Due to the simulation characteristics of FP (see section 6.1), FP1 and FP2 cause latency equally on every packet regardless of packet size (unlike CD). It

follows that the greater the number of packets in the network pipe the greater the latency for the connection.

5.3.4 Packet-based vs Byte-Counting Algorithm

Typically, congestion control algorithms use a congestion window to manage the number of packets allowed for transmission [JK88, Stev97]. As the inbound acknowledgement packets arrive the congestion window is adjusted accordingly. For packet-based algorithms, the rate of inbound acknowledgements determines the growth of the congestion window. This is acceptable for protocols that have a fixed packet acknowledgement frequency policy, such as TCP [Brad89]. However, for XTP, this is problematic. The setting of XTP's Acknowledgement Frequency policy can have a profound effect on a packet-based congestion control algorithm. Since the acknowledgement frequency can vary on a per association basis, a packet-based algorithm will generate unpredictable behaviour. The transmission rate would then be a function of the congestion control algorithm and the acknowledgement frequency policy. We can decouple this dependency by basing the congestion control algorithm on acknowledged data bytes rather than on the number of acknowledged packets [Allm98]. In [Allm98], the author investigates various acknowledgement generation and utilization techniques for improving TCP's performance, especially for TCP's slow start phase. One such method is the use of a byte-count approach for increasing the congestion window.

For XTP, we utilize the byte-counting approach to decouple the dependency between the Acknowledgement Frequency and the transmission rate. For this reason we use a byte-counting-based algorithm in our proposed congestion control algorithm for XTP.

5.3.5 Congestion Control Policies for XTP

For XTP/IP it is necessary to explicitly define a congestion control management that uses existing XTP mechanisms and that is responsive to network indicators of congestion. We propose a Simple Congestion Control (SCC) algorithm to ensure stability and provide a baseline for developing other congestion control algorithms.

We note that the area of congestion management for transport protocols is complex due to the constant dynamic nature of network bandwidth availability. TCP has undergone numerous congestion management revisions and continues to do so. We envision a similar process for XTP. Although we can learn much from the development of TCP's congestion control management we note that we simply cannot "port" its algorithms to XTP. There are significant differences between these two protocols.

The primary difference is that TCP automatically retransmits a data packet upon a packet timeout. This can trigger a flood of data packet retransmissions which further congest an already congested network link. XTP suspends its data transmission upon a packet timeout and goes into a HandShake Procedure to verify the availability of its peer. In doing so, there is no danger of XTP flooding a congested network with more data packets. Also, through the HandShake procedure, the sending context can obtain a more accurate RTT measurements, which may diminish future packet timeouts. There are other differences as well, such as explicit data loss notification via the Error Control packet and sender-controlled acknowledgement frequency, that differentiate XTP's congestion control mechanisms from TCP.

5.3.5.1 Simple Congestion Control Algorithm

The proposed Simple Congestion Control algorithm adds congestion responsiveness to the XTP transmission mechanisms. Before we outline the congestion control algorithm, we identify the state variables that are a part of SCC. Two new variables are introduced, Bytes-In-Flight Window (BIFW) and Linear Growth Threshold (LGT).

Overview of SCC Algorithm

We outline an overview of SCC algorithm and provide component details afterwards.

1 – At the beginning of an association, XTP probes the network for available bandwidth by employing exponential growth mode.

2 – Barring any WTIMER timeout or Error packet, XTP continues with exponential growth mode until the BIFW reaches LGT. At this point, XTP probes the network for available bandwidth by employing Linear Growth Mode.

3 – Barring any WTIMER timeout or Error packet, XTP remains in Linear Growth Mode until the end of data transmission.

4 – If during data transmission, a WTIMER timeout occurs, XTP moves into No Growth Mode, and initiates the HandShake procedure. At the termination of the HandShake procedure, XTP implements a Moderate Decrease Adjustment if no packet loss was detected during the HandShake procedure. If packet loss was detected, then XTP implements High Decrease Adjustment.

5 – If during data transmission, packet loss is detected via the Error packet, then XTP implements High Decrease Adjustment.

SCC Component Details

Exponential Growth Mode:

Objective: ramp up the bytes-in-flight window as quickly as possible without packet loss.

Starting Condition(s):

BIFW= 1 maxdata size (1 packet)

LGT= 320000 bytes (arbitrarily set, subject to experimentation)

Algorithm:

Increase BIFW by maxdata bytes for every maxdata bytes acknowledged until we reach the LGT value.

Linear Growth Mode:

Objective: slow down the increase of bytes in the bytes-in-flight window to avoid packet loss and/or network congestion.

Starting Condition(s):

BIFW >= LGT

Algorithm:

Increase BIFW by 1/BIFW for every maxdata bytes acknowledged.

No Growth Mode:

Objective: suspend any increase or decrease of BIFW when there is uncertainty about the condition of network bandwidth availability.

Starting Condition(s):

Context State == SYNC

Algorithm:

No change to BIFW or LGT

Moderate Decrease Adjustment:

Objective: reduce the rate of transmission after detection of slight congestion.

Starting Condition(s):

the HandShake has ended and K=2, (no Control packet has been lost)

Algorithm:

If in Exponential Growth Mode:

LGT=BIFW/2

BIFW=LGT

If in Linear Growth Mode:

LGT=LGT/2

BIFW=LGT

High Decrease Adjustment:

Objective: drastically reduce the transmission rate after packet loss detection. This is interpreted as heavy network congestion

Starting Condition(s):

In HandShake Procedure with K>2 or arrival of Error Control Packet

Algorithm:

BIFW=1

LGT=LGT/2

5.4 XTP Traffic Service Framework

To alleviate the need for an application programmer to have an extensive network background, we introduce Transport Traffic Services for XTP. By providing these Transport Traffic Services, a programmer will only require knowledge and understanding of the traffic pattern requirements for the application. The Transport Traffic Services are a "packaged" configuration of XTP's mechanisms and protocol parameters for a given traffic pattern.

5.4.1 XTP Transport Traffic Service

For XTP/IP we configure certain XTP parameters as "fixed" for an XTP/IP environment. We list 9 fixed configurations. These are drawn from sections 5.2 and 5.3. The 9 fixed configurations are:

1 – Selective Data Retransmission

2 – ATMAP Timer enabled

3 – RTT Cache Size greater than 1

4 – Selective SREQ/EDGE bit setting

5 - Simple Congestion Control Algorithm based on Byte-Counting method

6 – Explicit SREQ setting for Packet Loss Detection for:

i) FIRST Packet

ii) Last Data packet in a packet "burst"

iii) Control packets

7 – Full Packet Checksum

8 – NULL Traffic Format

9 – Reliable Delivery semantic

These configurable parameters in XTP should not be modified from the application layer.

The only parameter that should be configured from the application layer is the Transport Traffic Service. We define 3 Transport Traffic Services for LinuxXTP. A Traffic Service is a set of pre-configured XTP settings that is tailored for a specific traffic pattern.

1) DEFAULT_SERVICE

This is the default service for XTP/IP. It is a general purpose traffic service.

It is pre-configured as:

1 - Association Type = S_D_T

2 - Teardown Method = AbbrevClose

3 - Acknowledgement Frequency = 2

2) FILETRANSFER_SERVICE

This is tailored for uni-directional file transfers. We exploit the fact that a file transfer is in one direction only by setting the RCLOSE in the FIRST packet. For short file transfers this helps the teardown process. We note that this service is strictly for a uni-directional transfer of a file, which is different from the FTP program that uses the FTP protocol for transferring files.

It is pre-configured as:

1 - Association Type = SDT

2 - Teardown Method = AbbrevClose

3 - Acknowledgement Frequency = 2

4 - RCLOSE bit set on FIRST packet

3) REQRESP_SERVICE

This is tailored for request/response traffic. Due to the tight interaction between legacy TCP/IP programs and the TCP semantics, the XTP service is required to utilize S_D_T semantics. This reduces the benefit of XTP's features, yet it is required so as to not break the existing code. As a result it is identical to the default Service.

It is pre-configured as:

1 - Association Type = S_D_T

2 - Teardown Method = AbbrevClose

3 - Acknowledgement Frequency = 2

By selecting one of these Transport Traffic Services, an application programmer is freed from knowing intimate protocol details and network characteristics.

Chapter 6 Evaluation of XTP/IP

In this chapter we evaluate the XTP/IP implementation in LinuxXTP. We use a controlled and isolated test environment to eliminate any outside influences. Comparisons are made with TCP to highlight the strengths and weaknesses of XTP. We first examine the packet exchange sequence of XTP and TCP for 4 common Internet traffic patterns. Then latency tests are performed to measure and evaluate XTP's performance.

6.1 Test Environment

To evaluate XTP/IP performance we set up a controlled network environment. We use two local Ethernet networks, each 10 Mb/s, connected by a linux-based router (see Figure 36).

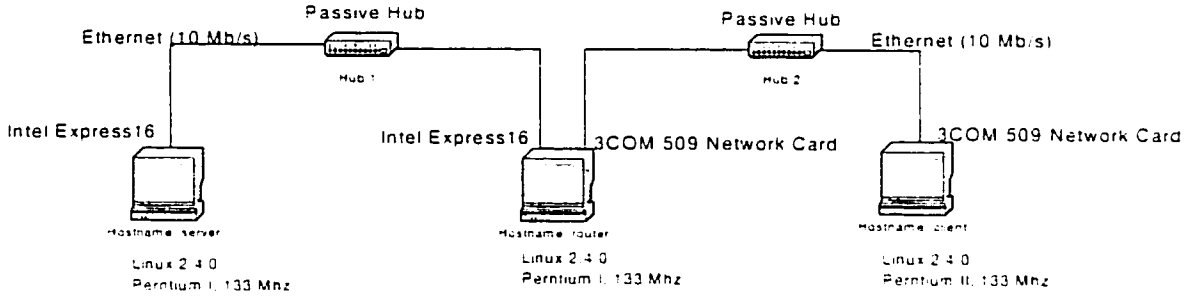

Figure 36 Physical Diagram

We modified the Linux router software to simulate different network characteristics. The simulation is to subject the packet traffic to network characteristics other than Ethernet LAN. Through software simulation at the router, we simulate Congestion Delay (CD) by

multiplying the size of an inbound IP packet with a delay factor. This causes network latency based on the size of the IP packet. We simulate a network with a High Bandwidth*Delay factor or Fat Pipe (FP), by simply delaying each inbound IP packet factor by a user-defined delay constant. This introduces latency for every packet uniformly, regardless of packet size. We note that these simulations are 'simplistic' and are only meant to add variation to the network layer characteristics for the purpose of this work. Since all of this simulation is at the IP level, there is no advantage to either TCP or XTP traffic. Figure 37 provides a logical view of the network environment.

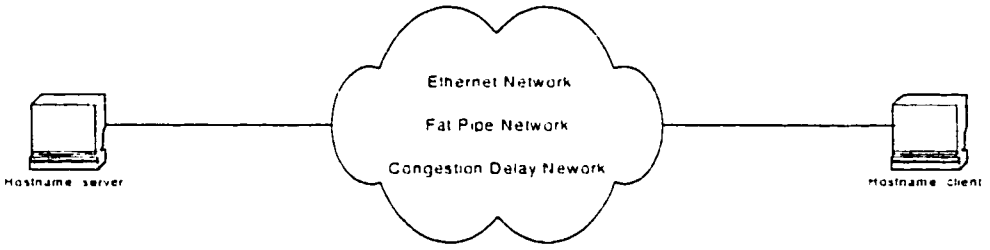

Figure 37 Logical/Simulated Diagram

6.1.1 Network Layer Characteristics

To better understand the network characteristics of the test environment, we perform two latency tests with the 'ping' command. We use the ping command to test the latency of the simulated environment using different delay factors. We use the standard packet size of 64 bytes and a full transport-layer packet size of 1456 bytes (1448 + 8 bytes of ping overhead). The ping command sends an IP packet once every second and calculates the average response time (see Table 11).

We also use the burst option of 'ping' to transmit the first 5 packets as quickly as possible and then revert to 1 packet / second. We graph the individual ping latencies in Figures 38 to 40 to illustrate the effects of the network environment. The major distinguishing feature is that after a burst of maxdata size packets, all traffic patterns return to a normal steady state. The exception to this is the congestion delay simulations where in two cases the delay factor is so high that the ping burst never recover from the burst (CD7) or remain in a heavy congested state (CD6).

With this data, we determine not to use Congestion Delay factors that are too high. For most of the tests in this paper we utilize CD1 and FP1.

Table 11 Ping Latency Tests

Network Type	Ping Latency (ms) [64 bytes]	Ping Latency (ms) [1456 bytes]
LAN	001.1	0009.5
FP1	037.2	0045.7
FP2	073.3	0081.8
FP3	362.4	0370.9
CD1	009.1	0151.8
CD2	017.2	0294.0
CD3	033.4	0578.4
CD4	049.6	0862.9
CD5	053.7	0934.0
CD6	057.7	1057.7
CD7	061.7	3370.1
CD8	065.8	5783.8

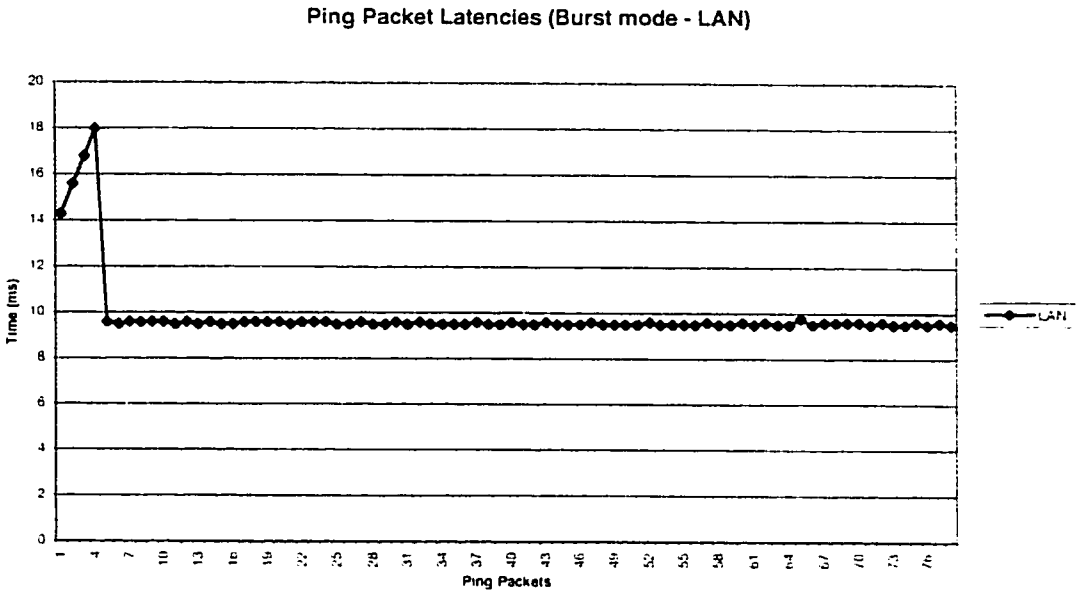

Figure 38 Ping Latency – LAN

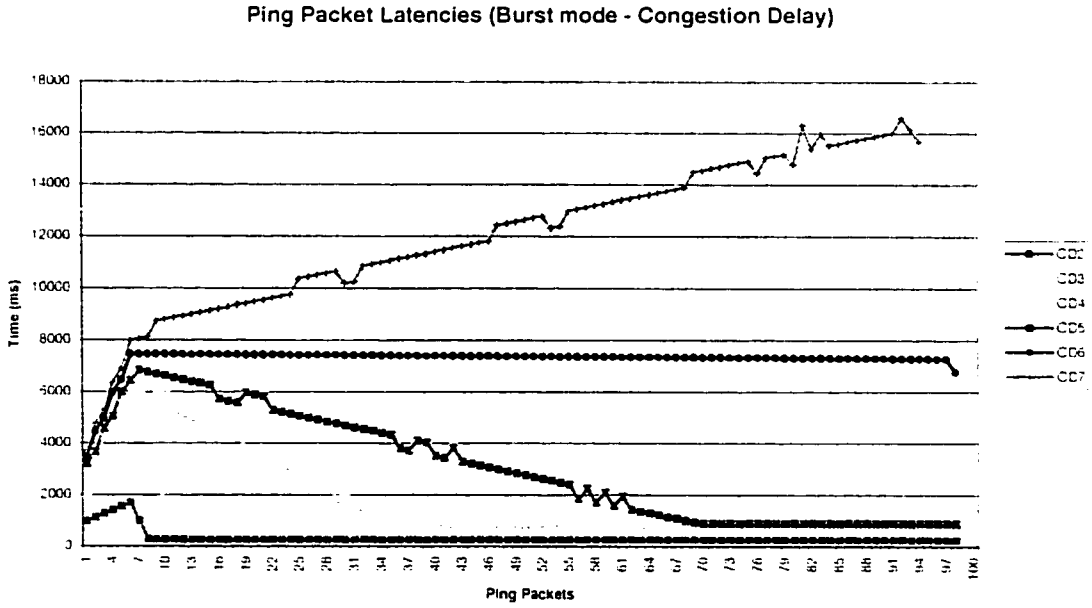

Figure 39 Ping Latency - CD

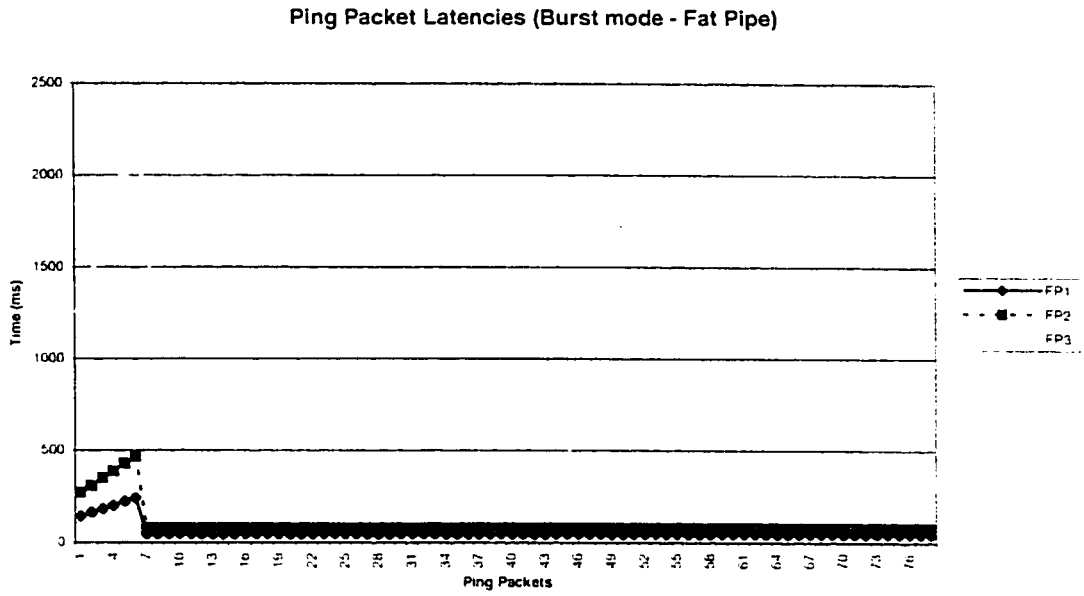

Figure 40 Ping Latency - FP

6.1.2 Application Layer Characteristics

To generate traffic we use the Internet and custom-developed programs described in Appendix D. We use these programs to generate 4 types of traffic patterns:

1 - Short Request-Response (SRR) Traffic

We utilize the 'finger' program to generate SRR type traffic. Both the request and response data can be transmitted in one to two packets. The performance of short request response traffic is primarily affected by the connection setup and teardown phases. This pattern is representative of Internet traffic for HTTP, DNS, RPC, and SNMP.

2 - Multiple Request-Responce (MRR) Traffic

We utilize the 'ftp' program to generate MRR traffic. Although the ftp program is used to transfer files, an FTP session has 2 transport layer connections, one for control information

and the other for data information. In addition to file transfers, FTP can provide directory listings. By using these features we simulate MRR type traffic. This pattern is representative of Internet traffic for HTTP, and FTP.

3 - Short File Transfer Traffic

For short data transfers we utilize the 'xft' program to send a file. This traffic is uni-directional. The performance of short file transfer is primarily determined by the connection setup and teardown phases. SMTP is an example of Internet traffic that reflects this pattern.

4 - Long File Transfer Traffic

For long data transfers we utilize the 'xft' program to send a file. This traffic is uni-directional. The performance is primarily influenced by the data transfer phase.

6.2 Analytical Comparison of XTP and TCP

We examine the traces of 2 traffic type for XTP and TCP. In some cases we find a strong dependency between the programs code and TCP's semantics, such that if we implement XTP's semantics it breaks the program. An example of this is trying to use combined SD phases. In most cases we must implement a S_D in order not to break the code (or recode the existing application). We examine two traffic types, Short File Transfer and Short Request-Response Traffic.

6.2.1 Short Request - Response Traffic

The 'request' portion of the finger program is actually transmitted in two packets even though the request can fit into one packet. The second packet delineates the end of the

request for the finger daemon process. The finger program breaks if we use an optimal SD phase for XTP, so we utilize a S_D semantic.

Figure 41 displays a TCP trace of the finger program, and Figure 42 displays an XTP trace. In XTP's trace we see a lack of any combined phases. Although the teardown is not combined with the Data Transfer, we do use an Abbreviated Close method to minimize packet exchange.

6.2.2 Short File Transfer Traffic

Figure 43 and 44 show the traces of a 5000 byte data transfer for TCP and XTP respectively. For XTP we are able to fully exploit the combined phases. As a result it takes XTP only 8 packets (figure 44) to transmit 5000 bytes while TCP takes 12 packets (figure 43). In Figure 45, XTP takes only 3 packets to transmit 1000 bytes. This illustrates XTP's combined Setup, Data Transfer, and Teardown with the Abbreviated Close Teardown method.

setup	request	984758065.808283 client.1159 > server.finger: S 953516800:953516800(0) win 5840 <mss 1460,sackOK,timestamp 515277[|tcp]>
		984758065.809295 server.finger > client.1159: S 4278898884:4278898884(0) ack 953516801 win 5792 <mss 1460,sackOK,timestamp 519038[|tcp]>
		984758065.809421 client.1159 > server.finger: . ack 1 win 5840 <nop,nop,timestamp 515277 519038>
data transfer		984758065.810384 client.1159 > server.finger: P 1:4(3) ack 1 win 5840 <nop,nop,timestamp 515277 519038>
		984758065.811248 server.finger > client.1159: . ack 4 win 5792 <nop,nop,timestamp 519039 515277>
		984758065.811376 client.1159 > server.finger: P 4:6(2) ack 1 win 5840 <nop,nop,timestamp 515277 519039>
		984758065.812216 server.finger > client.1159: . ack 6 win 5792 <nop,nop,timestamp 519039 515277>
	response	984758065.853573 server.finger > client.1159: P 1:1402(1401) ack 6 win 5792 <nop,nop,timestamp 519042 515277>
teardown		984758065.853602 server.finger > client.1159: F 1402:1402(0) ack 6 win 5792 <nop,nop,timestamp 519042 515277>
		984758065.853699 client.1159 > server.finger: . ack 1402 win 8406 <nop,nop,timestamp 515282 519042>
		984758065.854516 client.1159 > server.finger: F 6:6(0) ack 1403 win 8406 <nop,nop,timestamp 515282 519042>
		984758065.855359 server.finger > client.1159: . ack 7 win 5792 <nop,nop,timestamp 519043 515282>

Figure 41 Finger – Short Request Response Transfer – TCP Packet Trace

setup	reqeust	984927279.073994 client > server: F ky103 in0 dl24 sy1 sq90000 SRQ| al16 ad9216 af256 dp79 dh:192.200.200.200 sp4 sh:192.100.100.100 tl8 sv4 tf0
		984927279.075429 server > client: T ky103 in0+RTN dl40 sy0 sq0 | rq24 al90024 ec1 xk106 tl8 sv4 tf0
data transfer		984927279.075855 client > server: D ky106 in0+RTN dl3 sy2 sq24 EOM|SRQ|
		984927279.076783 server > client: C ky103 in0+RTN dl20 sy0 sq0 | rq27 al90027 ec2
		984927279.076930 client > server: D ky106 in0+RTN dl2 sy3 sq27 EOM|SRQ|
		984927279.077866 server > client: C ky103 in0+RTN dl20 sy0 sq0 | rq29 al90029 ec3
	response	984927279.107150 server > client: D ky103 in0+RTN dl1414 sy1 sq0 EOM|SRQ|
		984927279.107196 client > server: C ky106 in0+RTN dl20 sy3 sq29 | rq1414 al91414 ec1
teardown		984927279.108532 server > client: C ky103 in0+RTN dl20 sy2 sq1414 WCL|SRQ| rq29 al90029 ec3
		984927279.108572 client > server: C ky106 in0+RTN dl20 sy3 sq29 END|RCL| rq1414 al91414 ec2

Figure 42 Finger – Short Request Response Transfer – XTP Packet Trace

setup	984755470.597230 client.1128 > server.9998: S 2532583094:2532583094(0) win 5840 <mss 1460,sackOK,timestamp 255756[|tcp]>
	984755470.604150 server.9998 > client.1128: S 1533902635:1533902635(0) ack 2532583095 win 5792 <mss 1460,sackOK,timestamp 259527[|tcp]>
	984755470.604277 client.1128 > server.9998: . ack 1 win 5840 <nop,nop,timestamp 255757 259527>
data transfer	984755470.604741 client.1128 > server.9998: . 1:1449(1448) ack 1 win 5840 <nop,nop,timestamp 255757 259527>
	984755470.605309 client.1128 > server.9998: . 1449:2897(1448) ack 1 win 5840 <nop,nop,timestamp 255757 259527>
	984755470.606530 client.1128 > server.9998: P 2897:4345(1448) ack 1 win 5840 <nop,nop,timestamp 255757 259527>
	984755470.607756 client.1128 > server.9998: FP 4345:5001(656) ack 1 win 5840 <nop,nop,timestamp 255757 259527>
	984755470.610924 server.9998 > client.1128: . ack 1449 win 8688 <nop,nop,timestamp 259527 255757>
	984755470.611961 server.9998 > client.1128: . ack 2897 win 11584 <nop,nop,timestamp 259528 255757>
	984755470.613201 server.9998 > client.1128: . ack 4345 win 14480 <nop,nop,timestamp 259528 255757>
teardown	984755470.613483 server.9998 > client.1128: F 1:1(0) ack 5002 win 17376 <nop,nop,timestamp 259528 255757>
	984755470.613530 client.1128 > server.9998: . ack 2 win 5840 <nop,nop,timestamp 255758 259528>

Figure 43 XFT – Short File Transfer – TCP Packet Trace 5000 bytes

setup - data transfer	984755431.628482 client > server: F ky248 in0 dl1448 sy1 sq0 RCL|SRQ| al16 ad9216 af256 dp9998 dh:192.200.200.200 sp4 sh:192.100.100.100 tl8 sv4 tf0
	984755431.634115 server > client: T ky248 in0+RTN dl40 sy0 sq0 WCL| rq1448 al91448 ec1 xk148 tl8 sv4 tf0
data transfer - teardown	984755431.634325 client > server: D ky148 in0+RTN dl1448 sy1 sq1448 RCL|
	984755431.634886 client > server: D ky148 in0+RTN dl1448 sy2 sq2896 RCL|SRQ|
	984755431.640943 server > client: C ky248 in0+RTN dl20 sy0 sq0 WCL| rq4344 al94344 ec2
	984755431.641063 client > server: D ky148 in0+RTN dl680 sy3 sq4344 EOM|WCL|RCL|SRQ|
teardown	984755431.644052 server > client: C ky248 in0+RTN dl20 sy0 sq0 WCL|RCL| rq5024 al95024 ec3
	984755431.644191 client > server: C ky148 in0+RTN dl20 sy3 sq5024 END|WCL|RCL| rq0 al90000 ec0

Figure 44 XFT – Short File Transfer – XTP Packet Trace 5000 bytes

setup - data transfer - teardown	984884141.501590 client > server: F ky306 in0 dl1024 sy1 sq0 EOM|WCL|RCL|SRQ| al16 ad9216 af256 dp9998 dh:192.200.200.200 sp4 sh:192.100.100.100 tl8 sv4 tf0
	984884141.505968 server > client: T ky406 in0+RTN dl40 sy0 sq0 WCL|RCL| rq1024 al91024 ec1 xk206 tl8 sv4 tf0
	984884141.506184 client > server: C ky206 in0+RTN dl20 sy1 sq1024 END|WCL|RCL| rq0 al90000 ec0

Figure 45 XFT – Short File Transfer – XTP Packet Trace 1000 bytes

6.3 Performance Comparison of XTP and TCP

For all performance tests we execute each test 100 times and take the average transfer time. We use 3 network environments: LAN, CD1, and FP1.

6.3.1 Short Request - Response Traffic

We run 2 series of tests for SRR traffic. In both series, the total request byte size is the same. The request is composed of 2 packets, each 3 bytes and 2 bytes long. The response size for Test Run #1 is about 1400 bytes and fits into 1 packet. The response size for Test Run #2 is about 2800 bytes and requires 2 packets. The response portion in 'finger' varies slightly by a few bytes.

In Table 12, there is a clear performance advantage by XTP in the LAN network and a slight advantage in FP. The Congested Delay network reveals no advantage.

Table 12 SRR Traffic Performance Test

Number of Response Packets	Transfer Time (secs)	
	XTP	TCP
LAN		
1	0.0342	0.0468
2	0.0424	0.0515
CD1		
1	0.4014	0.4085
2	0.7019	0.7014
FP1		
1	0.1702	0.2238
2	0.2131	0.2619

6.3.2 Multiple Request-Response Traffic

The MRR performance test consisted of several very short transfers and two short 5000 byte transfers. In addition to the traffic generated by the FTP protocol (one control and one data connections) we created 3 very short transfers by requesting a directory listings separated by two short file transfers (put and get) of 5000 bytes.

From the performance test results in Table 13, we see that XTP has a slight advantage over XTP for LAN networks, and TCP has an advantage over XTP for FP1 networks. For congested delay networks, both protocols had similar performance.

Table 13 MRR Traffic Performance Test

Network Type	Transfer Time (secs)	
	XTP	TCP
LAN	0.2309	0.3092
CD1	5.4163	5.4698
FP1	2.9349	2.7890

6.3.3 Short File Transfer Traffic

We performed two sets of tests for short file transfers. The first was a data transfer of 1000 bytes, which fits into one packet, and the second was 10 000 bytes.

Table 14 shows that XTP has a definite edge when transferring short data sizes. All 1000 byte transfers are faster than TCP. At 10 000 bytes, TCP has an advantage over XTP in the LAN environment.

Table 14 SFT Traffic Performance Test

Data Transferred (bytes)	Transfer Time (secs)	
	XTP	TCP
LAN		
1000	0.0045	0.0066
10 000	0.0220	0.0184
CD1		
1000	0.2273	0.2712
10 000	2.0618	2.1198
FP2		
1000	0.0402	0.1296
10 000	0.2191	0.3307

6.3.4 Long File Transfer Traffic

The long file transfers consist of a uni-directional transfer of 1 000 000 bytes. Table 15 shows that both XTP and TCP are comparable in performance.

Table 15 LFT Traffic Performance Test

Data Transferred (bytes)	Transfer Time (secs)	
	XTP	TCP
LAN		
1 000 000	1.2388	1.1559
CD1		
1 000 000	206.3546	208.5621
FP2		
1 000 000	21.8379	21.4650

Chapter 7 Conclusion

Implementing XTP Revision 4.0 in an IP environment requires enhancements and strong implementation guidelines. From the work presented in this thesis, we make the following recommendations for implementing XTP over IP in an Internet-like environment:

1 - Implement protection against FIRST Packet Replay Hazard by adding an expiration timer to delay the deletion of an Address Translation Map table entry.

2 – Use a RTT Cache to improve the effectiveness of RTT measurements.

3 – Decouple the Packet Acknowledgement Frequency from the SREQ-setting frequency to improve the efficiency of RTT measurements.

4 – Implement a Byte-Counting-based Congestion Algorithm to avoid influence from the Packet Acknowledgement Frequency.

5 – Implement an 'Internet-Friendly' Congestion Control Algorithm that responds to network congestion.

6 – Use a pre-defined packaged Transport Traffic Service for configuring XTP's parameters to alleviate programmer's need to be network knowledgeable.

By implementing these enhancements, we remove potential hindering factors and show that XTP is well suited for Internet Traffic patterns.

In examining XTP's behaviour through LinuxXTP, we demonstrated the effects that association types, teardown methods, and acknowledgement frequency have on packet

exchanges and performance. We showed that XTP has better performance than TCP for short live data transfers (1000 bytes, 10 000 bytes) and is comparable to TCP for long data transfers (1 000 000 bytes).

We also showed how a predefined package approach to XTP's configuration can be used in legacy TCP/IP programs. However, we noted that due to a tight dependencies between the application and TCP's semantics, recoding is necessary to fully exploit XTP's features.

7.1 Future Work

We hope that the work presented in this thesis further enables research and development of XTP. The following are some areas of potential research work:

1) Multicast: investigate the issues and implication in implementing multicast capabilities in LinuxXTP.

2) Rate Control: investigate using Rate Control as an alternative congestion control algorithm, or explore the possibility of using a combination of SCC for short data transfers and Rate Control for long data transfers.

3) Telnet Traffic: investigate what is needed to enable XTP to support user interactive traffic.

4) Multimedia Traffic Service: develop a Transport Traffic Service tailored for multimedia application requirements.

Bibliography

[AC93] J. W. Atwood, G. Chung, *Error Control in the Xpress Transfer Protocol*, Conference on Local Computer Networks, Sept. 1993

[ACFS96] J. W. Atwood, O. Catrina, J. Fenton, T. W. Strayer, *Reliable Multicasting in the Xpress Transport Protocol*, in Proceedings of the 21st Local Computer Networks Conf., Oct. 1996

[Allm98] M. Allman, *On The Generation and Use of TCP Acknowledgements*, ACM Computer Communication Review, Oct. 1998.

[BCCD98] B. Braden, D. Clark, J. Crowcroft, B. Davie, et al, *Recommendations on Queue Management and Congestion Avoidance*, 1998.

[BD92a] Yves Baguette, Andre Danthine, *Comparison of TP4, TCP, and XTP, Part 1: Connection Management Mechanisms*, 1992

[BD92b] Yves Baguette, Andre Danthine, *Comparison of TP4, TCP, and XTP, Part 2: Data Transfer Mechanisms*, 1992

[BFF95] T. Berners-Lee, R. Fielding, H. Frystyk, *HyperText Transfer Protocol – HTTP/1.0*, RFC 1945, May 1995.

[Brad89] R. Braden, *Requirements for Internet Host – Communication Layers*, RFC 1122, Oct. 1989.

[Brad92a] R. Braden, *T/TCP—TCP Extensions for Transactions Functional Specification*, RFC 1644, July 1992.

[Brad92b] R. Braden, *TCP for Transaction—Concepts*, RFC 1379, Nov. 1992.

[CA90] J. Chen, J. W. Atwood, *Performance of the Xpress Transfer Protocol in an Ethernet Environment*, 15th Conf on Local Computer Networks, Oct. 1990.

[Cher88] D. Cheriton, *VMTP: Versatile Message Transaction Protocol: Protocol Specification*, RFC 1045, Feb. 1988.

[Chess89] Greg Chesson, *XTP/PE Design Considerations*, Protocols for High-Speed Networks, May 1989.

[Chess91] Greg Chesson, *The Evolution of XTP*, Proceedings of the 3rd International Conf. On High Speed Networking, 1991.

[Chess92] Greg Chesson et al, *Xpress Transfer Protocol Definition Revision 3.6*, Protocol Engines Inc., Santa Barbara, CA, January 1992.

[CLZ87] D. D. Clark, M. L. Lambert, L. Zhang, *NETBLT: A Bulk Data Transfer Protocol*, RFC998, March 1987.

[Deer89] S. Deering, *Host Extensions for IP Multicasting*, RFC1112 1989.

[DLW94] B. Dempsey, M. Lucas, A. Weaver, *High Quality Video Distribution using XTP Reliable Multicast*, in Proceedings of the Second International Workshop on Advanced Communications and Applications for High Speed Networks (IWACA), Sept. 1994.

[DSW] B. Dempsey, T. Strayer, A. Weaver, *Adaptive Error Control for Multimedia Data Transfer*.

[FB90] D. C. Feldmeier, E. W. Biersack, *Comparison of Error Control Protocols for High Bandwidth-Delay Product Networks*, Proceedings of the IFIP Workshop on Protocols for High Speed Networks, 1990.

[FF96] S. Floyd, Kevin Fall, *Simulation-based Comparisons of Tahoe, Reno, and SACK TCP*, Lawrence Berkeley National Laboratory, 1996.

[FF98] S. Floyd, K. Fall, *Promoting the Use of End-to-End Congestion Control in the Internet*, Feb. 1998.

[FFBGM96] R. Fielding, H. Frystyk, T. Berners-Lee, J. Gettys, J. Mogul, *Hypertext transfer protocol—HTTP/1.1*, RFC draft-ietf-http-v11-spec-04.txt, 1996.

[Fox89] R. Fox, *TCP Big Window and NAK Options*, RFC1106, June 1989.

[Heid97] J. Heidemann, *Performance Interactions Between P-HTTP and TCP Implementations*, Feb. 1997

[HOT96] J. Heidemann, K. Obraczka, J. Touch, *Modeling the Performance of HTTP Over Several Transport Protocols*, Nov. 1996

[HV97] J. Heideemann, V. Visweswaraiah, *Improving Restart of Idle TCP Connections*, Nov. 1997.

[JBB92] V. Jacobson, R. Braden, D. Borman, *TCP Extensions for High Performance*, RFC1323, May 1992.

[JK88] Van Jacobson, Michael Karels, *Congestion Avoidance and Control*, In Proceedings of SIGCOMM '88, ACM

[KB96] Espen Klovning, Olivier Bonaventure, *Behaviour of XTPX in the European ATM Pilot*, European Transactions on Telecommunications, Vol 7, No. 5, September/October 1996, pp.445-454.

[KP87] Phil Karn, Craig Partridge, *Improving Round-Trip Time Estimates in Reliable Transport Protocol*, In Proceedings of SIGCOMM '87, ACM

[Mapp95] Glenford Mapp, *Preliminary Performance Evaluation of SandiaXTP on ATM at ORL*, PROMS '95, 2nd Workshop on Protocols for Multimedia Systems, Salzburg, Austria, October 9-12, 1995.

[Ment98] Mentat Inc., *Xpress Transport Protocol Specification Revision 4.0b*, July 1998.

[MF97] J. Mahdavi, S. Floyd, *TCP-Friendly Unicast Rate-Based Flow Control*, 1997.

[MMFR96] M. Mathis, J. Mahdavi, S. Flyod, A. Romanow, *TCP Selective Acknowledgement Options*, RFC 2018, Oct. 1996.

[MN95] R. Mechler, G. W. Neufeld, *XTP Application Programming Interface*, University of British Columbia, Dept. of Computer Science, Technical Report TR-95-17, July 1995.

[Mogu95] J. C. Mogul, *The case for persistent-connection HTTP*, in Proceedings of the SIGCOMM '95, pp. 299-313, ACM, Aug. 1995.

[Nagl84] J. Nagle, *Congestion control in IP/TCP Internetworks*, RFC 896, 1984.

[PK98] V. Padmanabhan, R. Katz, *TCP Fast Start: A Technique For Speeding Up Web Transfer*, 1998.

[PM94] V. N. Padmanabhan, J. C. Mogul, *Improving HTTP Latency*, in Proceedings of the Second International WWW Conference, Oct. 1994.

[Post80] J. Postel, *User Datagram Protocol*, RFC768, 1980.

[Post81] J. Postel, *Transmission Control Protocol*, RFC793, 1981.

[SDW92] W.T. Strayer, B.J. Dempsey, A.C. Weaver, *XTP: The Xpress Transfer Protocol*, Addison-Wesley, 1992.

[SGC94] W. T. Strayer, S. Gray, R. E. Cline, *An Object-Oriented Implementation of the Xpress Transfer Protocol*, in Proceedings of the Second International Workshop on Advanced Communications and Applications for High Speed Networks (IWACA), Sept. 1994.

[SLC94] W. T. Strayer, M. J. Lewis, R. E. Cline, *XTP as a Transport Protocol for Distributed Parallel Processing*, Proceedings of the USENIX Symposium on High Speed Networking, Aug. 1994.

[Sper95] S. Spero, *Analysis of HTTP Performance Problems*, http://sunsite.unc.edu/mdma-release/http-prob.html, 1995.

[Stev97] Richard Stevens, *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*, RFC 2001, Jan. 1997.

[Stra91] W. Strayer, Communication Services for Real-Time Systems: An Examination of ARTS over XTP, University of Virginia, Nov. 1991.

[Stra95] W.T. Strayer, editor, *Xpress Transport Protocol 4.0 Specification*, XTP Forum Inc., 1394 Greenworth Place, Santa Barbara, CA 93108 USA., 1995.

[Stra95b] W. T. Strayer, *END Game: Closing Strategies in SandiaXTP*, 1995.

[SW88] W. Strayer, A. Weaver, *Evaluation of Transport Protocols for Real-Time Communication*, CS Report TR-88-18, University of Virginia, June 1988.

[SW92] W. Strayer, A. Weaver, *Is XTP Suitable for Distribute Real-Time Systems*, Proceedings of the IWACA, March 1992.

[THO96] J. Touch, J. Heidemann, K. Obraczka, *Analysis of HTTP Performance*, Aug. 1996.

[Weav1] A.C. Weaver, *Xpress Transport Protocol Version 4.0*, Dept. of Computer Science, University of Virginia.

[Weav2] A.C. Weaver, *What is the Xpress Transport Protocol?*, Network Xpress Inc., 700 Harris Street Suite 101, Charlottesville, VA 22903.

[VB88] V. Jacobson, R. Braden, *TCP Extensions for Long-Delay Paths*, RFC1072, Oct. 1988.

[VBZ90] V. Jacobson, R. Braden, L. Zhang, *TCP Extensions for High Speed Paths*, RFC1185, Oct. 1990.

[Zha86] Lixia Zhang, *Why TCP Timers Don't Work Well*, In Proceedings of SIGCOMM '86, ACM.

Appendix A Setsockopt Parameters

In table 16, we list the socket options that are available in LinuxXTP. These socket options can be utilized in the setsockopt() and getsockopt() system calls.

Table 16 XTP Setsockopt parameters

SETSOCKOPT PARAMETER	Description
XTPSOCK_ASSOCTYPE	association type
XTPSOCK_RATECONTROL	rate control
XTPSOCK_TTS	transport traffic service
XTPSOCK_MAXDATA	max data size
XTPSOCK_LN_THRESH	linear growth threshold
XTPSOCK_INIT_TRAINSIZE	initial bytes-in-flight-window
XTPSOCK_TEARDOWNMETHOD	teardown method
XTPSOCK_SERVICEPROFILE	service profile
XTPSOCK_SREQFREQ	packet acknowledgement frequency
XTPSOCK_NB_CONNECT	non-blocking connect
XTPSOCK_RTT_NODECOUPLE	disable ack freq/sreq freq decoupling
XTPSOCK_RTT_CACHE_SIZE	rtt cache size
XTPSOCK_WTIMER_TMO	wtimer timeout
XTPSOCK_CTIMER_TMO	connection timeout
XTPSOCK_ZOMBIE_TMO	zombie timeout
XTPSOCK_CTIMEOUT_TMO	context timeout
XTPSOCK_NO_ATMAP_TIMER	disable addr trans map timer
XTPSOCK_NOCHECK	disable ipsum over whole packet
XTPSOCK_MIBSTAT	retrieve xtp metrics/statistics

Appendix B XTP Metrics Output

The socket option XTPSOCK_MIBSTAT returns a data structure with configuration and performance information. Table 17 lists the variables tracked by the LinuxXTP implementation and available with the getsockopt() system call.

Table 17 XTP Metrics

VARIABLE
FpkOut
DpkOut
CpkOut
TpkOut
EpkOut
DGpkOut
FpkIn
DpkIn
CpkIn
TpkIn
EpkIn
DGpkIn
data_retransmits
FCwait
RCwait
sync
datasreq_sent
edgebit_set
rtt_updates
lin_inc
lin_dec
xp_inc
xp_dec
sreq_freq
init_trainsize
trainsize
cong_trainsize
linear_thresh
linear_count
assoc_type
teardown_type
profile_type
wtimerTMO
ctimerTMO
ctimeoutTMO
zombieTMO
rtt_cache_sz

Appendix C TCPDUMP Legend

In order to capture XTP packets from LinuxXTP, we modified the existing tcpdump program to capture and display XTP packets. Tables 18 to 20 list the abbreviations used in the XTP packet traces.

Table 18 Packet Type Abbreviations

Packet Type	Abbreviation
First	F
Data	D
Diag	G
Common Control	C
Traffic Control	E
Error Control	T

Table 19 Packet Field Abbreviations

Packet Field	Abbreviation
Key	ky
Index	in
Sync	sy
Seq	sq
Sort	not shown
Check	not shown
Dlen	dl
Address Segment Length	al
Address Domain	ad
Address Format	af
Destination Port	dp
Destination Host	dh
Source Port	sp
Source Host	sh
Traffic Segment Length	tl
Service Type	sv
Traffic Format	tf
Rseq	rq
Alloc	al
Echo	ec
XKey	xk

Table 20 Command Bits

Command Option	Abbreviation
NOFLOW	NFL
NOERR	NER
END	END
EOM	EOM
SREQ	SRQ
DREQ	DRQ
FASTNAK	FSN
WCLOSE	WCL
RCLOSE	RCL
EDGE	EDG
NOCHECK	NCK
BTAG	BTG
RES	RES
MULTI	MUL
SORT	SOR

Appendix D XTP Programs

The following are descriptions of XTP enabled programs created and used in this thesis work.

INETD Server

The Inetd server is a daemon process that listens for connections on well known Internet ports. When a connection request arrives Inetd determines if there is a listening service for that request. If there is one then it spawns the appropriate application (telnet, ftp, finger, etc.) to handle the connection request. By default, the Inetd server listens for TCP and UDP connection requests only. We modified the source code of the Inetd server to also listen for XTP connection requests. The XTP enabled Inetd server will redirect a FIRST packet to the appropriate XTP enabled application. We have two XTP enabled applications that will work with Inetd, ftpdx and fingerx.

FTPX/FTPDX

The ftp application is the implementation of the File Transfer Protocol specification. The ftp implementation consists of two components, a server daemon (ftpd) and a client application (ftp). We modified both to create XTP enabled ftpdx and ftpx. These differ from the XTP enabled Inetd application in that we completely replaced the TCP/IP code with XTP/IP, thus they exclusively work with the XTP protocol. The ftpx application initiates a FTP session by sending a FIRST packet with port number 23. The ftpdx process gets spawned by Inetd

when a FIRST packet with port number 23 arrives. After the initial packet, all future packet exchanges are between the respective ftpx and ftpdx, just as in the traditional TCP/IP implementation.

The port of ftp to ftpx was straight forward. We replaced all TCP/IP references with XTP/IP. The port of ftpd to ftpdx was not as straight forward as ftp because of a particular behaviour for the BSD systems call connect(). For TCP, the BSD semantic of connect() is that the system call returns immediately to the user application while TCP is in progress of establishing the connection to the remote host. For XTP, the semantic for connect() is that it blocks until the remote context responds successfully before returning to the user application. Using the default blocking behaviour of connect() breaks the ftp code. We had the option of either changing the ftp code to accommodate the new connect() call semantic, or add a new semantic to the connect() call. To minimize recoding, we opted to implement a non-blocking semantic for connect(). We provide a non-blocking semantic via a setsockopt() parameter. XTP_NB_CONNECT. We utilize this only for ftp. This is an example where we interpret this particular behaviour of a system call as an 'application service requirement' and implement a workaround in the XTP implementation rather than implement a workaround in the application layer. With this simple addition, the ftpd code works as is.

Fingerx

The finger application displays a user's login information, mostly taken from the /etc/passwd file. The finger application can display user information from local login accounts, which does not require any network protocol, or from remote login accounts, which does require network protocols. The server process, fingerd, contains no network code since it operates

independently of a request's origin. The client application finger does contain network code for requesting remote user login information. We replaced the existing TCP/IP code with XTP/IP in finger to create fingerx.

xft/xftd

Since ftp does not generate simple file transfer traffic, we created xft, XTP File Transfer, which does only file transfers. There are two components, a server daemon process and a client application. The xft client transfers a file to another host that has the xftd process running. This generates a simplex connection which terminates at the end of the file transfer. We designed xft to use either the XTP or TCP transport protocol for comparison purposes.

xtpsend/xtprecv

During the development of LinuxXTP and the XTP Transport Traffic Service, we needed to test various combinations of XTP policies and mechanisms. The xtpsend and xtprecv programs enable us to configure the many XTP features from the command line. The command line arguments correlate to the XTP parameters that can be set with the setsockopt() call.

Appendix E RTT Estimator Functions

The Linux source code describes modifications made to the RTT estimator function for TCP.

This is from the tcp_input.c module.

```
static __inline__ void tcp_set_rto(struct tcp_opt *tp)
{
      tp->rto = (tp->srtt >> 3) + tp->mdev;
      /* I am not enough educated to understand this magic.
       * However, it smells bad. snd_cwnd>31 is common case.
       */
      /* OK, I found comment in 2.0 source tree, it deserves
       * to be reproduced:
       * ====
       * Note: Jacobson's algorithm is fine on BSD which has a 1/2 second
       * granularity clock, but with our 1/100 second granularity clock we
       * become too sensitive to minor changes in the round trip time.
       * We add in two compensating factors. First we multiply by 5/4.
       * For large congestion windows this allows us to tolerate burst
       * traffic delaying up to 1/4 of our packets. We also add in
       * a rtt / cong_window term. For small congestion windows this allows
       * a single packet delay, but has negligible effect
       * on the compensation for large windows.
       */
      tp->rto += (tp->rto >> 2) + (tp->rto >> (tp->snd_cwnd-1));
}
```