Project Number: EE-DC-9601

XUDP: A Real-Time Multimedia Networking Protocol

A Major Qualifying Project Report:
submitted to the Faculty
of the
Worcester Polytechnic Institute

in partial fulfillment of the requirements for the
Degree of Bachelor of Science

by

Michael J. Andrews
Date: March 19, 1997

Approved:

1.internetworking
2.multimedia
3.protocol

Professor David Cyganski, Advisor

Abstract

A study of chronological advancements in TCP/IP combines with advanced topics in multimedia communications to produce a functional implementation of a real-time, multimedia internetworking data transmission protocol. This new IP-based protocol, named XUDP, uses the concept of timed-obsolescence and the lessons learned from the development of TCP/IP, yielding an end product that efficiently utilizes the available network bandwidth to transmit real-time multimedia data in a manner that maintains the end user's perception of the linear progression of timed sequential data.

Contents

List of Figures

List of Tables

Chapter 1

Executive Summary

This MQP began with the simple intention of developing an improved method for transporting real-time, multimedia data over an internet. To this end, a research effort was spawned to investigate the suitability of the existing TCP/IP Internet Protocol suite for the task of delivering potentially high-bandwidth, real-time multimedia data. This research led to two important early conclusions regarding both transport layer protocols in the TCP/IP protocol suite, the *Transmission Control Protocol (TCP)* and the *User Datagram Protocol (UDP):*

1. TCP, the fully-reliable, stream-based data transport protocol used in such applications as `telnet` and `FTP` where reliability is an important consideration, has a number of parameters accessible at both the application layer and inside the actual kernel implementation that can be *tuned* to yield optimum performance. Quite often, implementations of TCP are left in an unoptimized state to allow for successful operation over nearly any type of networking medium; given specific information about the underlying network architecture that a particular TCP implementation is to operate over, a programmer could tune it to meet these requirements.

2. UDP, the best-effort, datagram delivery service in the transport layer of the Internet Protocol suite, is primarily an interface to the internet-layer Internet Protocol, and as such, has very few parameters that would require tuning. Recently, UDP has become the protocol of choice for many multimedia application developers due to its lack of a flow control mechanism, placing the burden of flow control back at the application. Network testing performed by myself and graduate student Brian Hazzard while supported by a grant from Lockheed Martin Government Electronic Systems indicated that a UDP-based application left to run without flow control will soon overload local buffers and drop packets before they even reach the physical network. Multimedia applications that utilize UDP must necessarily lack the robustness of TCP's carefully implemented flow control algorithm, and suffer from less-than-desirable performance.

Clearly, the method by which improved transfer of multimedia data can be reached must lie at a point somewhere near the middle of the spectrum formed by the nearly diametrically opposed implementations of TCP and UDP.

In the fall of 1997, a second MQP was started to develop an internet video-phone that would make use of a unique set of simple, but effective compression algorithms. During that project's preliminary talks, a common-sense idea was presented: THE MOST IMPORTANT FACTOR IN REAL-TIME, MULTIMEDIA DATA IS THE LINEAR PROGRESSION OF THE TIME-SENSITIVE DATA AND THIS

SHOULD BE PRESERVED AT ALL COSTS. If an application using TCP to transmit a video stream encounters network congestion, the video stream that, in a linear world, is five minutes long could take six or seven minutes to view with variations in the time-wise stretching of the play back, a very nonlinear experience! In contrast, a protocol that made every effort to preserve sequential time linearity in the face of network congestion, if given a five minute video stream would provide a video stream that lasted exactly five minutes, but with, at times, a lowered quality than that of the original. In the nonlinear case, an audience member could close her eyes, and the video would pick up where she left off; in the linear case, the video appears to keep running while her eyes are closed and she simply misses some data.

Our proposed solution to the successful transport of real-time multimedia data was then formulated. The phrase *timed-obsolescence* was coined to indicate that multimedia data should have a window of usefulness associated with it, a sort of expiration date for video and audio. Once a set of data has exceeded that window, it should be discarded without further attempts at delivery.

This MQP's newly defined end product would be a robust, fully functional internetworking transport layer protocol that implemented the above concept of timed-obsolescence as its main objective.

The protocol needed to have a fully-functional flow control mechanism, similar on many levels to that employed by TCP. With good flow control, the protocol would be able to dynamically adjust to network congestion and retransmit lost packets in a timely manner. This necessitated that, at the very least, the protocol under development would use a sliding window scheme to control the amount of data in the network at any given time. In fact, it was discovered that many of the well documented optimizations to TCP in the 1980's and early 90's would be absolutely necessary for this multimedia-specific endeavor, including: slow start, round trip time variance estimation, exponential retransmit timer back-off, dynamic congestion window sizing and per-packet time-stamping. It also became obvious that the timed-obsolescence feature demanded selective acknowledgment and retransmit, a subject that TCP developers have been avoiding for some time now.

Months of development went in to the resulting 4,000 lines of C source code that now comprise the unique transport protocol developed to satisfy the goal of timely and semi-reliable transmission of real-time multimedia data. The protocol's name, the eXtended User Datagram Protocol (XUDP) is intended to convey that it is actually built on top of the services of UDP, that it utilizes a UDP datagram-like entity for its fundamental unit of data transfer and that it is an extension to these services in that it behaves much more like TCP than like UDP. As evidenced by the results (chapter 8) and conclusions (chapter 9) sections, *XUDP met the goal of preserving the linear progression of timed sequential data in every case.*

Although some work remains to be done to optimize critical algorithms in XUDP, it is, at the completion of this project, a viable communications protocol with great promise for use in future real-time multimedia applications.

Chapter 2

Introduction

Over the last fifteen years, the number of computers that comprise the Internet has grown *exponentially,* making an incredible amount of information easily accessible and forever altering the way in which we communicate ideas and conduct business. With the development of the World Wide Web (WWW) and newer video conferencing applications, users have begun to experiment with *multimedia* data transmission, incorporating images and audio alongside textual data. Until recently, the Internet was primarily used for non-interactive electronic mail or file transmission, with a few low-bandwidth interactive applications, such as `telnet.` Wherever images, video or audio are concerned, users understandably expect quality approaching or exceeding that which they already have in telephones and television, a requirement that is often not met by the Internet.

Although there is little that a single user can do to improve the physical network conditions between two points on the Internet, there is still room to improve the communication protocols used to transport the data over the underlying network. At the heart of the Internet is the TCP/IP protocol suite, comprised primarily of the Internet Protocol (IP), the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP.) The use of TCP in combination with IP essentially guarantees reliable delivery of data from a process running on a computer connected to an internet over the underlying network architecture to a process on another host anywhere in that internet. IP defines the addressing scheme which the Internet uses, often represented in "dotted quad notation" (i.e. WPI's current WWW server: 130.215.24.209), and provides a best-effort, connectionless datagram delivery service. The Transmission Control Protocol builds a reliable "stream" service with flow control on top of the services provided by IP. Since 1983, every computer on the Internet has communicated using TCP/IP, as mandated by the Office of the Secretary of Defense [9, p.6].

However, the recent trend toward real-time multimedia data transmission demands a type of service that TCP/IP alone, by design, cannot provide: *semi-reliability.* The notion of semi-reliability stems from the nature of multimedia data streams such as audio and video, where the primary interest is that of preserving the integrity of the timed progression of sequential data and the sense of "linear time." Given the large amount of data required to present a realistic motion video stream to the user, it is inevitable that in a networked video application, transmitted packets will be lost either through collisions or in overrun buffers when the necessary transmission bandwidth exceeds the available network bandwidth. If TCP is being used to transmit the data, it guarantees that all data transmitted will be received at the other end of the connection, and resorts to retransmitting lost segments of the stream, contributing to the overall congestion on the network. If no buffering methods were in place in the application, this behavior would result in delays in the presented video stream - an effect analogous to the user closing his eyes, and having the world "pick up where it left off" when he reopens them! Naturally, we'd prefer to have the world keep turning, even though our eyes are closed.

Ideally, multimedia applications would like to have a certain amount of control over what data needs to get through, and what data can be discarded if network congestion occurs. Recently, real-time multimedia applications have been making use of UDP as an alternative to the more robust TCP. UDP functions primarily as an interface to the best-effort delivery service of IP, and as such offers no guarantees to the reliability of its data transmission service. In fact, UDP offers no sort of flow control whatsoever, frequently resulting in less-than-desirable performance on fast computers where applications are able to overload local networking buffers, losing packets before they even enter the physical network.

This MQP report documents the research and development of a *fast multimedia protocol* built to fill a "middle layer" between the application and the User Datagram Protocol. It attempts to fit the need for *semi-reliability* by making use of concepts and innovations from standard existing protocols such as the Transmission Control Protocol and recent multimedia-specific protocols such as the eXpress Transport Protocol (XTP). Particular attention was paid to the well-documented historical development of TCP.

Chapter 3

Literature Review

3.1 Introduction to Internetworking

Good designs imitate life. The rise in popularity of *internetworking* as a means of interconnecting computers is largely due the naturalness and work-ability it embodies, because it mirrors our own, human methods of communication in social hierarchies. Internetworking is loosely based three things:

1. Clearly defined, standard communication interfaces.
2. Minimal or no knowledge of the workings of external systems.
3. Good faith.

Many systems created by our culture exhibit these characteristics as a means of getting things done efficiently. Writing a bank check is one such example. The end user has a standard communication interface, the check, that he knows how to work with; he need not know the details of the system that handles the checks or the people involved, and yet he sends it away freely. As Douglas Comer indicates in the introduction to his *Internetworking with TCP/IP - Volume I: Principles, Protocols and Architecture* [9], a key facet of internetworking is its ability to "hide the details" of other systems:

> In the past 15 years, a new technology has evolved that makes it possible to interconnect many disparate physical networks and make them function as a coordinated unit. The technology, called *internetworking,* accommodates multiple, diverse underlying hardware technologies by providing a way to interconnect heterogeneous networks and a set of communication conventions. The internet technology hides the details of network hardware and permits computers to communicate independent of their physical network connections.[9, p.1]

Although a full-fledged introduction to internetworking is beyond the scope of this paper, the subject is extremely well documented; an excellent example is Douglas Comer's book mentioned above. Other books such as Parker's *Teach Yourself TCP/IP in 14 Days* [36] and similar texts with mass-market appeal can focus greatly on an end product of learning appropriate terminology, rather than learning the terminology through the process of understanding networking concepts, and should be avoided. Texts by Comer, W. Richard Stevens[42] and Andrew S. Tannenbaum[44] can appear daunting, but are expertly written.

Regardless, there are four important items that cannot be ignored in any presentation of internetworking: *the client/server model, protocol layering and encapsulation, TCP/IP* and *data transport units.*

3.1.1 Data Transport Units

There is a good deal of terminology that is used to refer to certain groupings of data commonly found associated with networking. Figure 3.1 attempts to illustrate the differences graphically, it is followed by descriptions of many of the usual data units found on the Internet.

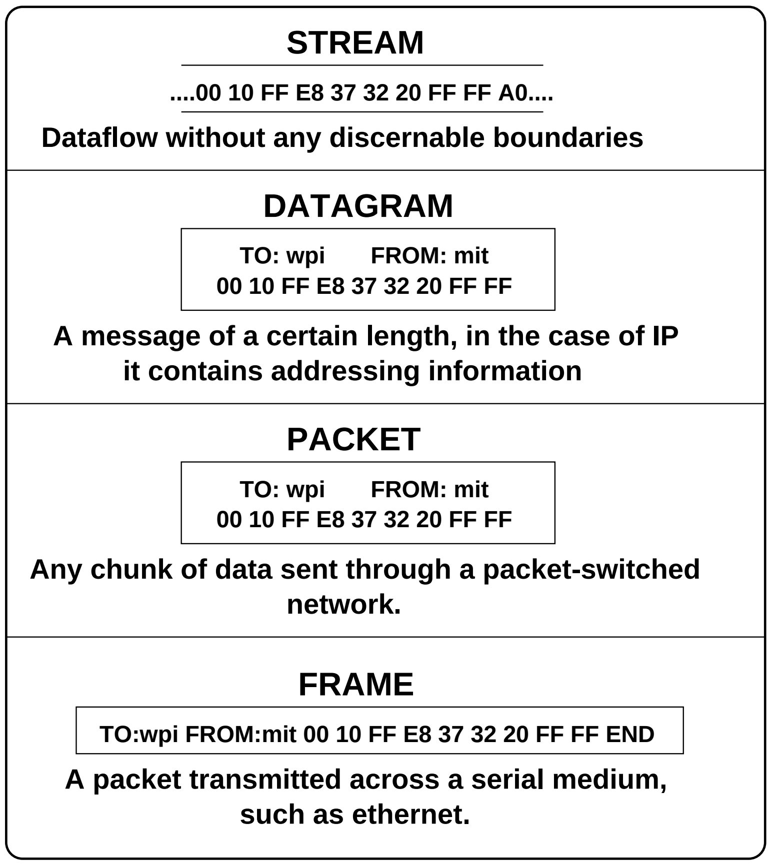

Figure 3.1: Data Transport Units

- A *stream* is used to refer to data that doesn't contain any message or record boundaries. Most programmers are used to dealing with disk-based files as streams, where an arbitrary number of bytes can be read or written at will.

- *Datagrams* are messages of a defined length that are usually packaged with source and destination addresses of some kind. The term "datagram" is appropriately similar to "telegram," both are arbitrary length and contained units of data transfer.

- Internetworks are fundamentally packet-switched networks. A *packet* is the fundamental unit of data transfer for these networks and is simply a small chunk of data. Packets typically range from the tens of bytes to a few kilobytes in size. A packet can be a datagram.

- Serial media such as Ethernet and RS-232 transmit data one bit at a time. Jargon held over from days of old maintains that these protocols transmit "start frame" and "end frame" messages. Recently, a frame is used to refer to any packet when it is transferred over a physical medium.

3.1.2 The TCP/IP Internet Protocol Suite

TCP/IP is the de-facto standard for communications on the *Internet* today. It arose out of research conducted by the *Advanced Research Projects Agency (ARPA)*[1] and the *National Science Foundation (NSF).* These research efforts led to the creation of a physical backbone which used TCP/IP as the mandated standard for communications. The ARPAnet and the NSFnet were the seed for the entity that would become known as *the Internet.* (The Internet is the largest existing example of *an internet,* and will be referred to with as previously, Internet with a capital 'I.')

Although the two acronyms that form TCP/IP resolve to *Transmission Control Protocol* and *Internet Protocol,* the TCP/IP Internet Protocol suite is actually composed of a number of protocols and pieces of software. Of particular importance to this paper is that TCP/IP is fundamentally composed of three protocols: the previously mentioned TCP and IP, as well as the *User Datagram Protocol (UDP).* As will be discussed to greater extent in the next section, IP defines the addressing scheme used on the Internet and both UDP and TCP interface with IP to wrap their data with information necessary for transport across the Internet. Programmers can choose to utilize either UDP or TCP to transfer data, but both ultimately interface to IP. Kale and Socolofsky offer an excellent practical discussion of the TCP/IP protocols in RFC1180[31].

Briefly, the salient points of the three protocols are summarized below:

Internet Protocol A best-effort, connectionless datagram delivery service, implementing the 32-bit addressing scheme used by the Internet.

User Datagram Protocol A best-effort, connectionless datagram delivery service that acts primarily as an interface to IP.

Transmission Control Protocol A reliable, connection-oriented, stream transport service.

3.1.3 Protocol Layering and Encapsulation

TCP/IP is based on four conceptual layers of implementation. Each layer is composed of a system or systems with a defined task and standardized communication interface to the surrounding layers. Figure 3.2 shows the four layers: application, transport, internet and network interface.

The layers are defined as follows:

Application Usually implemented in the user-space, a program that utilizes the underlying network architecture in some way. This can be a user application or an existing one such as `telnet` or `FTP`. A common interface to the TCP/IP protocols is Berkeley Sockets, found in networking kernels derived from source code developed at Lawrence Berkeley Laboratories.

Transport Both UDP and TCP reside in the transport layer, responsible for bridging the gap between the application and the Internet Protocol. Transport protocols communicate with streams or messages to the application layer and with packets to the Internet layer.

[1] Renamed the *Defense Advanced Research Projects Agency* for a period in the 1980's[9, p.2], and then again in the 1990's.

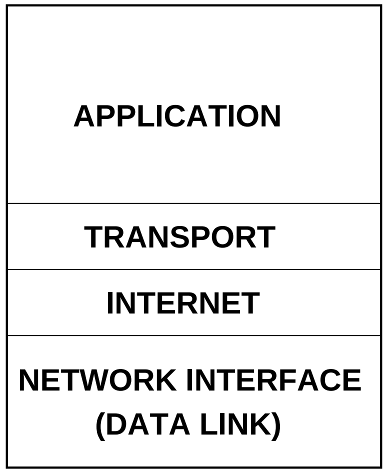

Figure 3.2: The TCP/IP conceptual layers

Internet IP defines the Internet layer. It handles the addressing and proper encapsulation and fragmentation of packets from the transport layer protocols. IP generates datagrams appropriate for the Network Interface.

Network Interface Drivers for physical networking devices can be found in this layer. They are responsible for proper communication of the incoming IP datagrams to the underlying physical network.

The four conceptual layers of TCP/IP are actually derived from a standard developed by the *International Organization for Standardization (ISO)* referred to as the *Reference Model of Open System Interconnection.* ISO's model has 7 layers that bear only historical importance to protocols in use today, as the protocols based on these specifications were never as widely accepted as TCP/IP. The ISO 7-layer model is reproduced in figure 3.3.

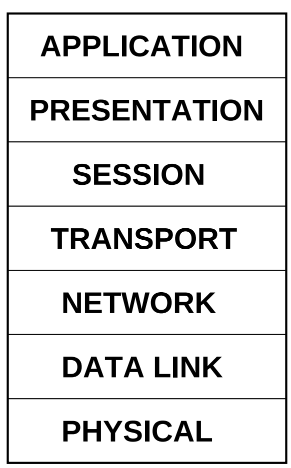

Figure 3.3: The ISO 7-layer reference model

Layering is the heart of the Internet. Thanks to properly documented and publicly available standards of interconnection, layering makes the Internet "work" by allowing communication between, but hiding the details of all kinds of different manufacturers' systems.

3.1.4 The Client/Server Model

The client/server model for communication is one that, again, mimics a great number of existing human systems. Typically, a *server* is a provider of a service that a *client* wishes to utilize in some manner. A few examples of this interaction are tabulated in table 3.1

Client	**Server**
hungry person	restaurant
web-browser	web-server
student	college
word processing program	print spooler
cable-TV box	cable-TV provider
couch potato	cable-TV provider
`finger`	finger daemon: `fingerd`

Table 3.1: Examples of the client/server model.

The connection between client and server is abstracted to one as simple as that in figure 3.4.

Figure 3.4: Simple client to server connection.

Many servers are capable of dealing with more than a single client at the same time, either by multiplexing or through creating multiple execution instances (for example by using `fork()` in C programs or a similar command.) Figure 3.5 shows multiple clients connecting to a single server.

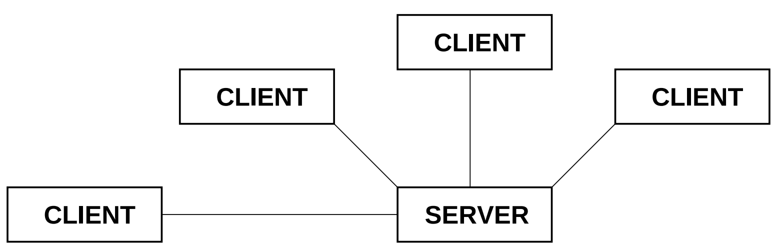

Figure 3.5: Many clients to single server.

Figure 3.6 reveals the details of the connection that have been hidden from us through appropriate layering.

The client/server model is an important conceptual tool at many levels of internetworking technology, and is relied on greatly by the networking community.

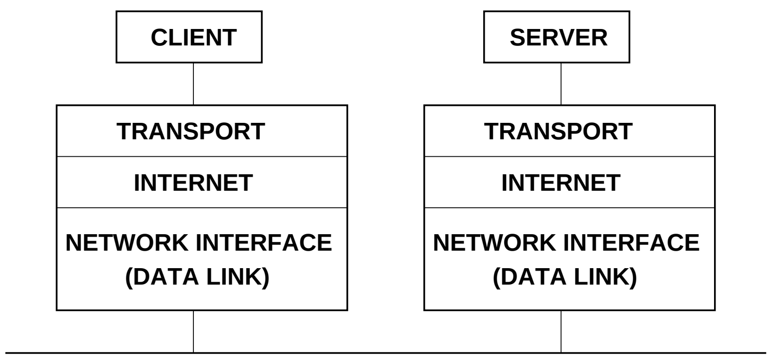

Figure 3.6: Actual client to server connection.

3.2 The Development of TCP/IP

This section will trace the development of TCP/IP through specific enhancements made to the Transmission Control Protocol. Version numbers of the networking code released by Berkeley Software Distribution will be used as milestones in TCP/IP's development, as most implementations of UNIX have tracked Berkeley's improvements. Figure 3.7 shows TCP/IP's timeline and the revision increments that will be considered here.

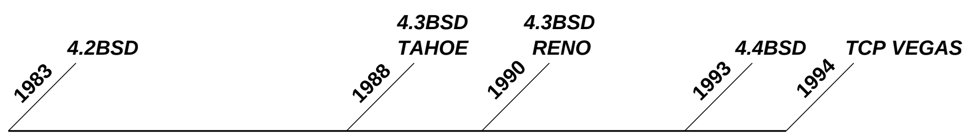

Figure 3.7: The TCP/IP Version Timeline

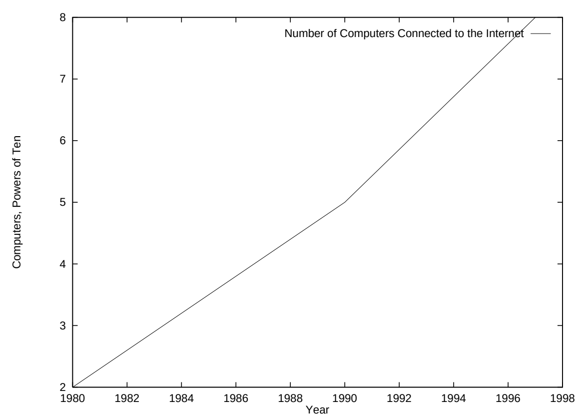

Figure 3.8: Number of computers connected to the Internet.

As Internet usage increased *exponentially,* so did the congestion problems. Comer relates the exponentiality of the expansion in chart form, the numbers are reproduced here as the logarithmic plot in figure 3.8[2]. This growth manifested itself as periods of lowered bandwidth and increased delays, and caused researchers at Lawrence Berkeley Laboratories (LBL) to question the mechanism employed by the current implementation of TCP to combat congestion. Most of the improvements made to TCP were instigated by this group, led by Van Jacobson, who has achieved nearly deity status in networking circles. As Jacobson and Michael Karels relate in *Congestion Avoidance and Control:*

> In October of '86, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL [Lawrence Berkeley Laboratories] to UC Berkeley (sites separated by 400 yards and two IMP hops) dropped from 32 Kbps to 40bps. We were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. In particular, we wondered if the 4.3BSD (Berkeley UNIX) TCP was mis-behaving or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions was "yes".[19]

[2]Numbers are from [9, p.13], which was published in 1995, indicating the data for 1997 is a projection

1983 *4.2BSD* Many of the implementations of TCP/IP at the time were pulled together to create the first widely available (and used) version of TCP/IP.

1988 *4.3BSD* TAHOE Jacobson's landmark 1996 paper, *Congestion Avoidance and Control* spawned a number of congestion-related improvements:

- Slow-start
- Better RTT variance estimation
- Exponential retransmit timer back-off
- Dynamic congestion window sizing

1990 *4.3BSD* RENO Improvements aimed at reducing CPU load and stack times.

1993 *4.4BSD* Modifications for long, fat networks and gigabit throughput rates.

- Window scaling, allowing in excess of 64KBytes of data in a network at one time.
- Protect Against Wrapped Sequences, prevented sequence number wrapping on connections such as satellite links with long delays but high bandwidth capacity.
- Round Trip Time (RTT) measurement with timestamps.

1994 *TCP "VEGAS"* based upon 4.3BSD RENO source, it utilizes congestion *anticipation* to achieve a reported 40% to 70% improvement in throughput over RENO.

Table 3.2: Improvements in TCP

The specific improvements made to TCP that are important within the context of this document are listed in chronological order in table 3.2.

3.2.1 Early TCP: 4.2BSD

The Transmission Control Protocol and the Internet Protocol in their earliest incarnations are defined in two Internet Request For Comments (RFCs), RFC793[41] and RFC791[40], respectively. From the introduction of RFC793, the Transmission Control Protocol:

> The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks.[41, p.1]

And, more specifically:

> TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. The TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. Very few assumptions are made as to the reliability of the communication protocols below the TCP layer. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, the TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit-switched networks.[41, p.1]

TCP uses layering to hide the details of its reliable, stream-based service from the application layer. It can effectively create a virtual point to point link between two separate processes, either running on the same machine or on separate machines, connected through a physical network. This connection is so reliable that it is often treated, from a programming perspective, as if it were between an application and a file residing on a local disk drive.

TCP works at the byte-level of data. It divides an incoming data stream into packets called *segments*; similar to datagrams in that each segment is packaged with a full header that is interpretable by TCP, but subtly different in that segments do not create discernible record boundaries, they merely represent sections of a continuous stream of data.

Reliability is insured by a sliding-window acknowledgement and retransmission mechanism. All data sent by TCP must be acknowledged by the receiver at the other end of the connection. TCP maintains a variable-sized *window* of data that is allowed to exist unacknowledged in the network at any given time. If this limit is reached, no data is sent until an acknowledgement is received. If no data has been acknowledged in a certain amount of time, as detected by the expiration of the Retransmission Time Out (RTO) timer, TCP assumes that the data has become somehow lost or destroyed, and retransmits all of the data in the window.

The original method for calculating the RTO time was rather crude, and could adapt to network loads of at most 30% (equations 3.1, 3.2 and 3.3):

$$RTT_{average} = \alpha RTT_{average} + (1 - \alpha) RTT_{measured} \tag{3.1}$$

$$RTT_{average} = RTT_{measured} + \alpha(RTT_{average} - RTT_{measured}) \tag{3.2}$$

$$RTO = \beta RTT_{average} \tag{3.3}$$

In the above equations, $RTT_{average}$ is calculated by passing the measured round trip times (RTT) through what is essentially a low pass filter, rejecting quick changes in the measured round trip times. With α usually defaulting to 0.9 and β taking its suggested value of 2, 4.2BSD's RTO calculation mechanism left much to be desired during the congestion collapses of the mid 1980's.

3.2.2 4.3BSD Tahoe

4.3BSD Tahoe grew out of optimizations suggested in Jacobson's *Congestion Avoidance and Control.*[19] This paper set the standard for protocol optimization, and influenced a great deal of subsequent internetworking protocols, including the one developed as part of this MQP. The fundamental points are itemized below:

- The TCP connection is a Self-Clocking system, formed by the transmission of data and the reception of acknowledgements.
- The problem of congestion needs to be approached from a "Conservation of Packets" principle:
  - New packets aren't put into the network until the old ones leave.
- There are only three ways in which packet conservation can fail:
  1. The connection stream never reaches equilibrium.
  2. A new packet enters the network before an old one has left.
  3. Equilibrium isn't attained because of dynamic congestion problems.

Slow-Start

Much of the time, earlier implementations of TCP would end up initially "shocking" a network by sending too much data right off the start. Taking a cue from a standard practice in electrical equipment design where the in-rush of electrical current into a device is desirably limited and slowly increased during a turn-on phase, Jacobson and his fellow researchers created a slow-start mode for TCP that exponentially increases the size of a congestion window. The exponential increase was enough to let TCP feel-out a connection's bandwidth and delay statistics while allowing the network to acclimate itself to the increase in load. Figure 3.2.2 shows Tahoe's slow-start mechanism theoretical throughput vs. time in comparison to 4.2BSD's.

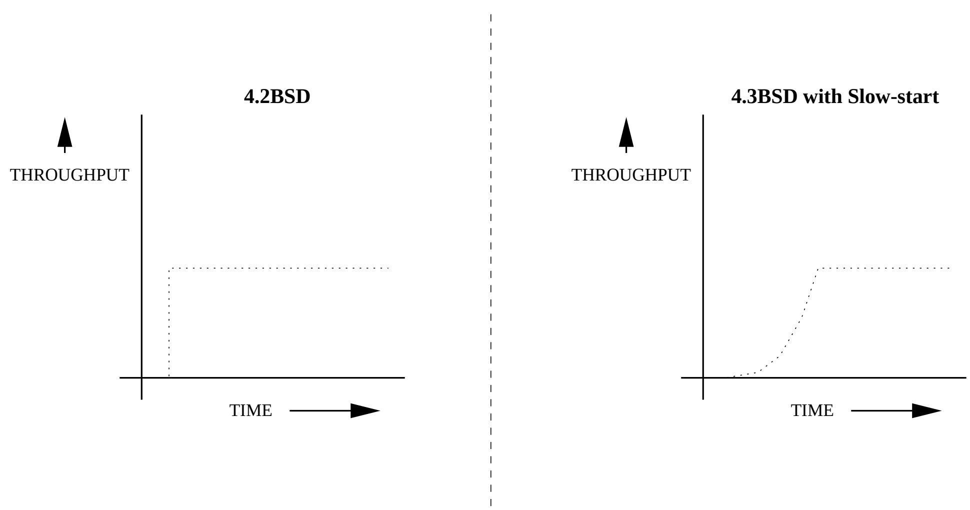

Figure 3.9: The slow-start algorithm of 4.3BSD Tahoe TCP

Retransmission Time Out Estimator

Jacobson's group also made changes to the Retransmission Time Out (RTO) estimation algorithms. The modified algorithm takes RTT variance ($\sigma_{RTT_{average}}$) into account instead of using a constant β value as indicated in the previous section (EQ: 3.3):

$$RTO = RTT_{average} + 4\sigma(RTT_{average}) \tag{3.4}$$

The new RTO estimator is capable of adapting to much greater loads. In addition, the entire algorithm was tweaked to utilize integer arithmetic based on computation with scaled powers of two, instead of fractional floating point arithmetic, further reducing CPU load.

Karn's Exponential RTO Back-off Algorithm

Quite often, TCP implementations would worsen network congestion due to multiple, successive retransmits. The RTO estimator is only updated upon receipt of an acknowledgement; if data is continually lost due to high network loading, the data will be retransmitted at constant intervals. Phil Karn, in his implementation of TCP/IP for extremely low-bandwidth, high-delay amateur packet radio usage, developed a strategy that handles multiple retransmits with an exponential back-off of the RTO timer for each successive retransmit.

Dynamic Congestion Window Sizing

TAHOE offered a means of combating congestion through dynamically altering the size of the protocol's sliding window. The algorithm followed three simple rules:

1. If the RTO expires, set the congestion window size to half of its current size.
2. Increase the congestion window size by 1 segment for each acknowledgement.
3. Send the minimum of the receiver's advertised window and congestion window.

The performance of 4.3BSD TAHOE was outstanding. In a Usenet news message, Jacobson expressed the details:

> I don't know what would constitute an "official" confirmation but maybe I can put some rumors to bed. We have done a TCP that gets 8Mbps between Sun 3/50s (the lowest we've seen is 7Mbps, the highest 9Mbps – when using 100Ethernet exponential back-off makes throughput very sensitive to the competing traffic distribution.) The throughput limit seemed to be the Lance chip on the Sun – the CPU was showing 10-15time. I don't believe the idle time number (I want to really measure the idle time with a uprocessor analyzer) but the interactive response on the machines was pretty good even while they were shoving 1MB/s at each other so I know there was some CPU left over.[Article 167, comp.protocols.tcp-ip, March 1988][21]

3.2.3 4.3BSD Reno

4.3BSD RENO implements both fast recovery/retransmit and header prediction, improvements arguably specific to TCP/IP but bearing little relevance to other communications protocols. Study of these will be avoided here, since the design of this MQP's protocol isn't affected by either enhancement.

3.2.4 4.4BSD with Extensions for High Performance

Internet Request For Comments number 1323 (RFC1323)[28] specified a set of extensions to TCP that attempt to extend its usability on networks with large bandwidth capability and long delay times. Below are listed the relevant suggested optimizations:

- Window scale option, allows windows $> 2^{16} = 65KBytes$.
- Round Trip Time Measurements with Timestamps
- Protect Against Wrapped Sequences (PAWS) for the case where the sequence number, a 32-bit integer, could "wrap" around while the data is delayed in queues in the network.

The optimizations are presented with the idea that TCP's performance relies on the product of the utilized bandwidth and the round trip delay time, yielding a measure of the amount of data present in the network at any given time. This calculation (equation 3.5) is elucidated in RFC1323:

> TCP performance depends not upon the transfer rate itself, but rather upon the product of the transfer rate and the round-trip delay. This "$bandwidth * delayproduct$" measures the amount of data that would "fill the pipe;" it is the buffer space required at sender and receiver to obtain maximum throughput on the TCP connection over the path, i.e., the amount of unacknowledged data that TCP must handle in order to keep the pipeline full. TCP performance problems arise when the $bandwidth * delay$ product is large. We refer to an Internet path operating in this region as a "long, fat pipe," and a network containing this path as an "LFN" (pronounced "elephan(t)"). [28, p.2]

$$bandwidth(\frac{bits}{second}) \cdot delay(seconds) = data(bits) \tag{3.5}$$

Window Scaling Option

TCP currently uses a 16-bit unsigned integer in the header of each TCP segment to communicate the size of the receive buffer to the sender. With this limitation, the maximum amount of data allowed in the network's buffers at any one time is 64KBytes; up until recently this hasn't had any detrimental effect on TCP's performance. Even utilizing the full bandwidth of a 10Mbps Ethernet, the round trip time required to fill the window would be in excess of $50ms$, an event rarely witnessed on a LAN. However, recent advancements in physical-layer networking technology have made gigabit throughput rates realizable. In the case of a full-bandwidth, 1Gbps transfer rate, TCP's window size would necessitate a round trip time no longer than $500\mu s$; clearly, the effective bandwidth achievable with TCP is theoretically limited by the round trip time delay as $2^{16}(bytes)/RTT(seconds)$. Thus, with a low $1ms$ RTT, TCP's ultimate achievable bandwidth is 524Kbps.

The TCP window scaling extends the window size to 32 bits, and is predicted to yield acceptable performance for quite some time into the future, tolerating a 34 second round trip time on a 1Gbps link.

Round Trip Time Measurement (RTTM)

Originally, the designers of TCP discarded the idea of per-sequence timestamps because of the compute power required to retrieve data from the system clock. The difficulty in obtaining accurate measurement of round trip time intervals necessary for the determination of the retransmission time was compounded by this lack of timestamps. These days, CPU power has grown to the point where time-stamping is cheap. The implementation of RTTM for TCP adds increased accuracy with very little impact on the compute time budget.

Protect Against Wrapped Sequences (PAWS)

Links with large *bandwidth·delay* products utilizing the TCP window scaling option could generate windows so large that there is potential to wrap the 32-bit per-segment sequence number around while data is still waiting in network queues. If this happened, two different segments, both numbered *zero,* would exist on the network at the same time, making proper acknowledgement impossible.

PAWS makes use of the new TCP timestamps to take both the sequence number and the departure time of segments into account, weeding out duplicate sequence numbers.

3.2.5 The Future: TCP Vegas?

TCP VEGAS [3] is an implementation of TCP developed at the University of Arizona from 4.3BSD RENO source code. It promises increased throughput over that of the existing TCP implementations largely due to its congestion *anticipation* mechanism. VEGAS'S congestion anticipation algorithm dynamically varies the size of the congestion window by monitoring the utilized bandwidth for small changes. This method contrasts with RENO'S congestion *avoidance* mechanism that needs to create losses to vary the congestion window size, VEGAS attempts to *predict* congestion.

Listed below are brief major implementation details:

- VEGAS claims to offer 40-70% increased throughput over RENO.
- VEGAS doesn't change the TCP standard, and only makes modifications to the sending side of the RENO source that include:
    1. More "timely" retransmission of lost segments (A *faster* fast retransmit)
    2. Congestion *anticipation*, with the ability to slowly drop the congestion window, instead of the abrupt *halving* that TCP TAHOE implemented.
    3. A *slower* slow-start algorithm.

3.3 Multimedia Protocols

The demand for appropriate handling of multimedia data transmission on the Internet has spawned several attempts at multimedia-specific protocols. The most promising of them, the *eXpress Transport Protocol (XTP)* will be examined here to provide impetus for later decisions regarding protocol implementation issues.

3.3.1 The eXpress Transport Protocol (XTP)

XTP is an amazingly complex communications protocol, capable of providing many of the much-needed services for which network programmers have been crying out. The system-level details of

[3] Its features are documented in detail in Brakmo, O'Malley and Peterson's *TCP* VEGAS: *New Techniques for Congestion Detection and Avoidance.*[3]

its necessarily massive implementation are covered in Alfred C. Weaver's *Xpress Transport Protocol Version 4*[45]. From the opening statement:

> The Xpress Transport Protocol (XTP) has been designed to support a variety of applications ranging from real-time embedded systems to multimedia distribution to applications distributed over a wide area network. In a single protocol it provides all the classic functionality of TCP, UDP, and TP4 [TP4 is the OSI standard transport protocol] , plus new services such as transport multicast, multicast group management, transport layer priorities, traffic descriptions for quality-of-service negotiation, rate and burst control, and selectable error and flow control mechanisms. XTP has the same interconnectivity as TCP/UDP/TP4 because it operates over any network layer (IP, CLNP), any data-link layer (LLC, MAC), or directly on top of the AAL of ATM. In general, XTP avoids coupling policy with mechanism; XTP offers services but the user's application defines what communications paradigm is most appropriate for its particular environment. XTP is a high performance protocol, and can sustain high throughput (92Mbits/s over FDDI between a pair of IBM RS/6000 model 370s) and low latency ($350\mu s$ to move 100 bytes from user memory to user memory on two 50 MHz PCs connected by FDDI). Since XTP can run in parallel with all other transport protocols, and can run over whatever network layer (if any) is provided, it represents a low-risk way to exploit the increased functionality required for distributed applications without sacrificing connectness or interoperability.[45, p.1]

A detailed investigation into XTP is outside the scope of this report, briefly, XUDP provides the following interesting features:

- XTP offers "levels" of flow, error and retransmission control. Instead of either a connection-oriented reliable stream (TCP) or a connectionless best-effort datagram delivery service (UDP), we can have a service in the middle, such as a best-effort stream service for video.
- It was specifically designed with multicasting in mind (unlike TCP/IP in which multicasting is at best an afterthought.) Multicasting is essential in a distributed computing environment.
- XTP will support Quality of Service (QoS) through traffic descriptor fields in the header.

The only implementation that I was able to locate was developed at Sandia Labs (*SandiaXTP*). It was packaged with something called the *Meta Transport Library* which it required for operation, and both were written using the object-oriented paradigm of C++. It is worth keeping in mind that Dave Clark stated in RFC817[7], submitted back in 1982, that layering and modularity may be excellent visualization techniques for protocols, but are far from desirable in the actual implementation. This affected my initial reaction to the protocol, but further simple testing against TCP's reliable stream proved that it handled the task adequately.

Chapter 4

Problem Documentation and Project Goals

Real-time multimedia data not only has high bandwidth requirements, but can easily be disturbed by varying delays in the communication channel. For years, people the world over have become used to viewing high bandwidth and constant delay video and audio on televisions. Attempting to packetize that same data with intent to distribute it over delay-varying communication channels such as the Internet has usually failed miserably. Recent attempts to implement multimedia specific enhancements to networking technology such as *Quality of Service* and *streaming* have been only mildly successful, with the latter arguably worsening the growing congestion problem on the Internet.

A solution will be presented here that fuses a concept from everyday life with existing network transmission protocol ideas to produce a internetworked multimedia communication channel that yields acceptable results with a lower bandwidth requirement than that necessitated by conventional streaming methods.

4.1 The Nature of Multimedia Data

Real-time multimedia data is expensive to transport and must be received in a timely manner, with utmost importance placed on preserving the linear progression of sequential timed data at the user's end.

Two major obstacles stand in the way of high-fidelity, real-time multimedia data transport over an internet: limited available network bandwidth and dynamically varying transmission delays. Understandably, the larger problem of the two is the high bandwidth requirement necessitated by real time audio and video. By way of example, a full-motion video and audio stream with television resolution (512x512 pixels) at 256 colors per pixel (8 bits per pixel) and 30 frames per second, and with 8-bit 22.05 kHz sampled audio has the following bandwidth requirements for the video signal (EQ 4.1):

$$\begin{aligned} x \cdot y \cdot bpp \cdot fps &= bits/second \\ 512 \cdot 512 \cdot 8 \cdot 30 &= 62{,}914{,}560 \ bits/second \end{aligned} \tag{4.1}$$

and for the audio signal (EQ 4.2):

$$\begin{aligned} sampling\ rate \cdot bits/sample &= bits/second \\ 22050 \cdot 8 &= 176,400\ bits/second \end{aligned} \tag{4.2}$$

Yielding a total far greater than that which we'd want to even think about transferring over the Internet: 63,090,960 bits/second - over six conventional (thinnet) Ethernets running in parallel. A more realistic data stream would be that found in a conventional teleconferencing system: 160x120, 6-bit gray scale, 10 frames per second and better than telephone quality 8-bit, 11.05 kHz audio: total required bandwidth just over 1 megabit/s. Steep, but not unreasonable on a LAN. If transmission over the actual Internet is required, then some amount of compression will be necessary on the data stream to obtain low enough rates.

Once the difficulties associated with bandwidth limitations have been overcome, any good transmission system needs to be able to deal with the finite and possibly dynamic delay between sender and receiver. In the case of the most familiar of multimedia data streams, the television signal, engineers needed only to worry about a small, fixed delay due to the distance between the transmitter and the television set to which the television signal in the form of an electro-magnetic wave propagates at the speed of light (as in figure 4.1.)

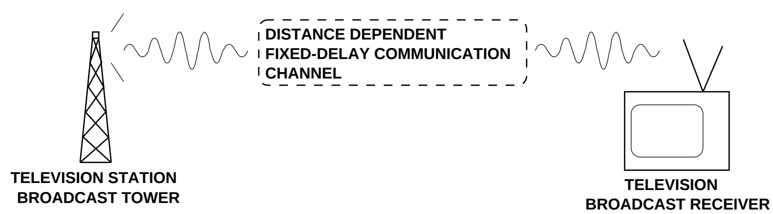

Figure 4.1: The way we're used to accessing multimedia data: The communication channel in a television transmitter and receiver system.

For a modern packet switching internetwork, the delay is not fixed, and dynamically varies with the bandwidth utilization that the network is currently enduring (as well as with other variables.) When networks become congested, extreme delays many orders of magnitude greater than that of the average expected delay can result. With a conventional networked connection, the ideal, timely delivered multimedia data stream (as depicted in Figure 4.2, representing a set of images comprising a movie of a bouncing ball) can undergo delays, distorting the presentation of timed sequential data beyond what is acceptable (Figure 4.3.)

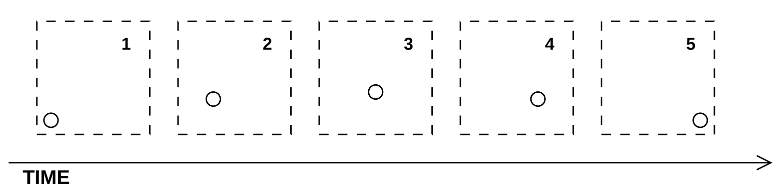

Figure 4.2: Frames in the complete video stream of a bouncing ball.

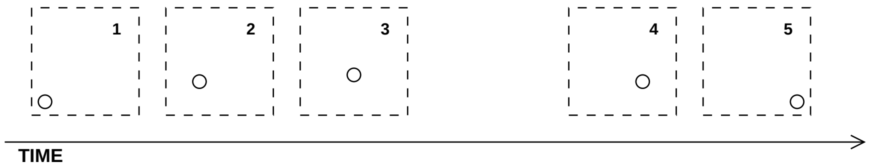

Figure 4.3: Frames in a typical damaged video stream that doesn't preserve the linear time relationships.

The motion depicted by the images in figure 4.3 defies our common sense about what happens when a ball bounces, and is different than figure 4.4, in which we close our eyes for a period of one frame. In the second case, the world with linear time keeps moving while we ignore it; in the first, time stops, and picks up again where it left off!

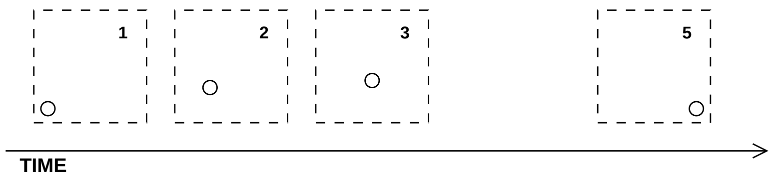

Figure 4.4: Frames in a damaged but linear time relationship preserving video stream of a bouncing ball.

For this reason, multimedia streaming was developed to attempt to overcome, or at least stave off temporarily the effects of varying transmission delays.

Multimedia streaming works by buffering a good amount of the data before presenting it to the user, much like a typical household water heater heats a cache of water before anybody actually needs it, rather than heating it on demand. Figure 4.5 illustrates the concept of data flow in a multimedia stream through this "water heater" analogy. The rate of data output is independent of the input rate, as long as there is enough data (or water) in the cache to source the required amount of output. If the input rate begins to lag behind the output rate, eventually there will not be enough data in the cache to support the high output rate, and our stream runs dry (and we don't have any hot water!)

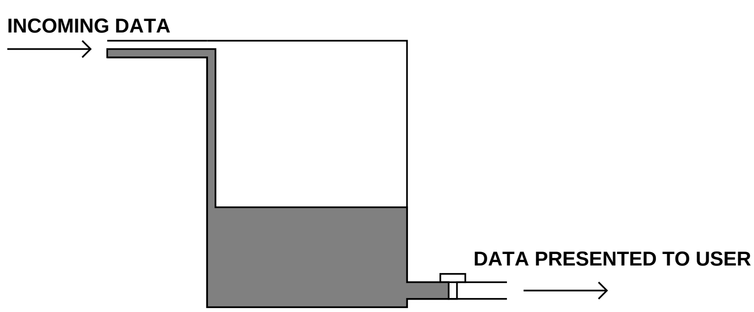

Figure 4.5: Multimedia streaming.

Depending on the speed of the network that will be used to transport multimedia data, streaming applications may buffer from a few seconds of data to a few minutes of data. The stream is a reservoir, providing an illusion of continuous delivery rather than the actual delivery rate that the network can provide. Designers of streaming applications hope that the reservoir is never emptied because the output rate was in excess of the input rate; they wish that the reservoir only dries up when the end of the multimedia stream has been reached.

Although this method of multimedia data presentation offers high-fidelity output at the user's end, it can tend to utilize quite a bit more of the available network bandwidth than is acceptable for a "nicely behaved application." If the streaming protocol is built on top of a fully reliable protocol such as TCP, segments of data that are lost due to congestion will be retransmitted until they have been acknowledged properly, adding to the congestion of the network. What's worse, if difficulty is encountered in transmitting a particular segment, the protocol will not attempt to bypass it, and the transmission will become stuck on that segment. Often, these difficulties result in the depletion of the multimedia reservoir, forcing the incoming network data to completely source the presented data stream.

Reliable data transmission protocols such as TCP were not designed to handle the special requirements of real-time applications. They were designed for low bandwidth interactive applications such as `telnet` and potentially high bandwidth non-interactive applications such as electronic mail handling and `ftp`. The best-effort transport services inherent to UDP are more suited to delivering multimedia data payloads.

The User Datagram Protocol is becoming an important player in the realm of multimedia protocols. Because it is essentially an interface to the low-level Internet Protocol, and because it offers a speedy checksum and I/O multiplexing through Berkeley sockets, it is an ideal choice for applications that do not wish to be constrained to the flow control mechanism in TCP. However, operation without any flow control in place will quickly fill the local socket-level buffers and UDP datagrams will be discarded *before they even reach the physical network.*

Work performed by myself and graduate student Brian Hazzard at WPI while under contract from Lockheed Martin led to the creation of a network testing system with a quickly discovered ability to overload local buffers. We found, when operating with no flow control, save for an adjustable time delay between datagrams, exceeding an experimentally determinable maximum transmission rate would degrade network throughput as well as load the CPU tremendously; in effect, the application could create a much worse performance situation by sending out data "as fast as it could" rather than by limiting itself to a maximum transmission rate. An excerpt from Hazzard's *Evaluation of Optimal Socket Buffer Sizes for the Internet Protocol Suite (draft)* [15] illustrates this experimentally derived fact:

> ... applications [] that generate datagrams faster than the kernel can handle the data, result in poor utilization of CPU time and degradations in performance will be observed, resulting in larger Round Trip Times (RTTs) [in the case of our RTT measuring network testing system] and slower network throughputs.

In effect, protocols such as TCP, with a highly refined flow control mechanism, attempt to dynamically center in on that optimum transmission rate through the feedback loop formed by data transmission and subsequent data acknowledgment. Given this observation, operation without flow control has become out of the question for high-bandwidth multimedia applications.

A last important consideration for the transport of multimedia data remains, and that is unique to this project. With earlier protocol efforts, all data was assumed to be important and therefore was delivered reliably; there would be very little chance that a transferred file would be acceptable if corrupt or incomplete in any way. With real-time multimedia data, the primary concern is for preserving the linear presentation of timed sequential data. Other aspects of the end product's quality will be allowed to suffer, as long as the remaining data is delivered so as not to disturb the linear time of the presentation.

4.2 The Notion of Timed Obsolescence

Figure 4.6 shows a great way of visualizing the "packaging" of a typical internetworked datagram delivery service. The data to be sent over the network is wrapped by a layer that is recognized by the delivery service that will be used, and contains at least a "To:" address field for the destination and most likely a "From:" address field to let the recipient know who sent the package.

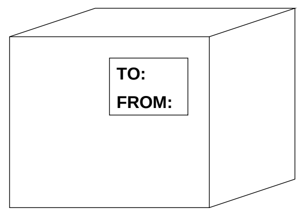

Figure 4.6: A typical datagram, from the post office point of view.

A reliable delivery service will keep attempting to deliver a package until the recipient successfully receives it. This can sometimes take a long time and use up quite a bit of system resources, especially if the receiving end has gone on vacation! Postal services have a number of ways of dealing with this situation, eventually resulting in the undeliverable package ending up in the dead letter office or back at the sender's door. The Internet Protocols, as they stand, don't have a happy medium between fully reliable delivery and unreliable delivery:

UDP offers a *best-effort* delivery service. While the "best effort" of most parcel shipping services involves several attempts at delivery and very few lost or untraceable packages, UDP's best effort gives delivery *one* chance and upon failure, discards the data entirely.

TCP employs a *fully reliable* delivery service. It ensures that all data is received at the destination, barring complete disconnection of service.

A *semi-reliable* data transmission system is needed fill the gap between the reliable and the unreliable. Much like the postal services, the system should know when to "give up" and stop trying to deliver certain packages. But, unlike the postal services, this transmission system could treat each parcel of data differently, rather than under a single policy that covers all packages.

WHAT IF DATA COULD EXPIRE IF NOT SUCCESSFULLY DELIVERED WITHIN A CERTAIN AMOUNT OF TIME?

Such would be the case for a parcel of fruit or other food - it would have an expiration or "sell by" date, as in Figure 4.7. If the recipient receives the fruit after the expiration date, the fruit is simply discarded. Better yet, if the sender knows that the fruit will not reach the recipient in time, the fruit will never be sent in the first place. *Timed obsolescence* lets applications label data with an expiration date, quantitatively assigning a time at which that data will become obsolete.

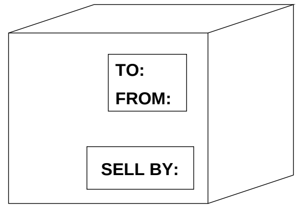

Figure 4.7: A datagram with a “sell by” date!

The earlier example of the “bouncing ball” video stream (reprinted in figure 4.8,) managed to preserve the time relationship of the video frames. It can now be seen that this behavior is congruent to what would be exhibited by a system implementing *timed obsolescence.* In this specific case, for whatever reason, frame four of the video stream could not be delivered in its entirety before its *time to obsolescence* or “expiration date.” Instead of sending now useless data in an attempt to fill in lost components of the parcel, the sending and receiving sides discard what data they have remaining and/or unacknowledged and continue on with the transfer.

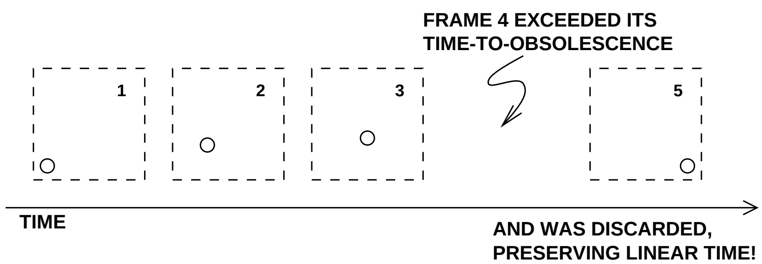

Figure 4.8: Frames in a damaged but linear time relationship preserving video stream of a bouncing ball

In summary, an internetworked multimedia data transmission system with the ability to timestamp parcels of data with expiration dates could not only preserve the linear progression of timed sequential data at the user’s end in the face of network congestion related data loss and delay, but could reduce network bandwidth significantly by not attempting to transmit or retransmit data that has outlived its usefulness.

4.3 The eXtended User Datagram Protocol (XUDP)

The primary goal of this MQP was to develop a method by which real-time multimedia data could be optimally transferred over the Internet. After determining that neither the Transmission Control Protocol nor the User Datagram Protocol were up to the task, common-sense reasoning on the nature of multimedia data prevailed, providing an optimal method, timed obsolescence. Timed obsolescence, attempts to preserve, above all, the linear time relationships of timed sequential data. As it stands, this concept necessitates some amount of lossiness or semi-reliability in the delivery mechanism - a rarely expressed quality of an internetworking transport layer protocol.

A thorough implementation of timed obsolescence would require an entirely new transport layer protocol that combines the best-effort delivery capability of UDP with the optimized flow control mechanisms of TCP. The protocol could be implemented on top of UDP, utilizing UDP as an interface to IP, with a quick checksum and I/O multiplexing.

Four thousand lines of C code and several months later, a fully-functional robust implementation of timed obsolescence was produced as an extension to UDP; the *eXtended User Datagram Protocol* or *XUDP* is entirely capable of preserving the linear time relationships inherent in multimedia data.

Much of XUDP's functionality lies in the adoption of a new unit of internetworking data transfer named *the parcel.* A parcel is a single unit of inseparable data that is defined by the user's application, such as a frame in a video, a one second burst of audio or any number of imaginable constructs that require delivery as a whole. The size of each parcel is set at the time that the parcel is sent from the application; XUDP currently supports parcel sizes from 1 byte to 32KBytes.

Because of the unknowable conglomeration of hardware and software that make up the network layer, a possibility exists that an XUDP parcel will not be able to pass atomically through the network. The actual transmitted unit of data on an internetwork is referred to as a packet. If a parcel is small enough, it may fit within a single packet of data (typically around 512 bytes). However, if, in the likely case that the parcel is larger than the maximum allowable size of a packet, the parcel is *fragmented* into many separate packets that contain position indicators (sequence numbers) unique to that particular parcel. The receiver guarantees that parcels will be reassembled *completely* and *in order* from the string of packets that comprise it.

In the case that packets become lost in the network, retransmission will take place selectively at the packet level, until they have been successfully acknowledged allowing XUDP the capability of full reliability if so required by the application and requested as the mode of service from XUDP. XUDP is capable of sending the parcel in one of three modes:

1. **Full reliability:** parcels are guaranteed to be received at the remote application in a timely manner. XUDP's ability to retransmit packets guarantees that all of the data in a parcel will be received by the application.

2. **Semi-reliability (timed obsolescence:)** parcels are stamped with a time-to-obsolescence (sell-by time or expiration time). If the XUDP server detects that the entire parcel will not be received before the time expires, all data related to the parcel, whether acknowledged or unacknowledged, is removed. This allows XUDP to conserve network bandwidth for the purpose of sending data that is still *useful.*

3. **Best-effort, one shot delivery attempt:** Similar to the service provided by UDP, this delivery method offers no chance of retransmit.

Research has shown that there is not any product in existence at the completion of this project that provides, or attempts to provide any service similar to timed obsolescence. Therefore, the product developed during the course of this project is able to provide a unique, unmatched service to the networking community.

Chapter 5 contains a detailed description of the functionality and implementation of XUDP. This is followed by a sample application developed to test XUDP, presented in chapter 6 and by an exhaustive presentation of XUDP's source code in chapter 7.

Chapter 5

High-Level System Design and Specifications

In this section, the design of the protocol implementation will be examined through a top-down process. The system's presentation will be divided into two layers, similar to those used in the previous description of TCP: the *application* layer (primarily the application programmer's interface) and the *transport* layer.

5.1 System Specifications Per Layer

1. **Application Layer**

   (a) XUDP runs on a variety of UNIX variants.

   (b) XUDP is implemented in user space (rather than kernel space) as daemon.

   (c) The protocol's functions run as a process separate from the application.

   (d) IPC between the protocol daemon and the application is handled through a Berkeley sockets-like API, transparent to the application layer

   (e) The basic unit of data transfer is the *parcel,* an entity similar to a datagram, with the following characteristics:

      i. Parcels are not necessarily part of a continuous stream of data, and thus preserve record boundaries.

      ii. Parcels will be of arbitrary length, determined at the time of sending.

      iii. Each parcel can be labeled with a finite expiration time, a one-shot delivery or as fully-reliable delivery.

      iv. Parcels will be received one at a time, preserving parcel boundaries.

      v. Due to the timed-obsolescence feature, parcels are not guaranteed to be received in-order, however, order will be preserved for the data forming a parcel.

   (f) All information related to the connection will be stored in a simple C structure, usually referenced with a structure pointer.

2. **Transport layer**

   (a) XUDP is connection-oriented.

(b) UDP is used as the delivery mechanism, one layer below.

(c) UDP's payload checksum is used to ensure error-free delivery.

(d) Parcels are fragmented into packets of a given, maximum allowable size.

(e) Flow control of the connection is achieved through algorithms similar to those in place in TCP:

    i. The connection is sliding window based.

    ii. Flow control is not based off of individual bytes, as in TCP, but on packets, regardless of their length.

    iii. There is a slow-start on connection initiation that ramps up the number of packets in the data window according to some function.

    iv. Packet round trip time variance estimation and per-packet time-stamping is used to aid in retransmission timer timeout prediction.

    v. Congestion anticipation is implemented by scaling the size of the window through close monitoring of RTT's and comparisons with recent RTT's.

    vi. There is an exponential retransmit timer backoff on successive retransmits.

    vii. Selective acknowledgment is used at both the parcel and packet level.

    viii. Selective retransmit is used at the packet level.

    ix. Fine-grained timers (1ms period) with 32-bit timestamps are used, no signals will be generated in the current implementation.

5.2 Application Layer

The application layer consists of the Berkeley sockets-like interface in the user's application to the underlying XUDP architecture.

5.2.1 Layering

XUDP is designed to provide a lightweight transport mechanism for multimedia data through a Berkeley sockets-like API. As evidenced in earlier sections, the Berkeley sockets interface has successfully removed the complexities of the lower levels of network programming from the programmer's list of worries. Figure 5.1 shows the ideal final implementation, where all the details of network transmissions are completely hidden from the programmer's perspective, allowing the programmer to concentrate on developing a quality application, rather than building low-level internetworking routines.

Figure 5.2 displays to a greater extent the detail involved in the layering of the networking system. Found in the "kernel level" are the implementations of the UDP and IP layers, as well as the Network Interface Card drivers. UDP will provide an interface to IP, as well as a packet checksum and a port multiplexing function. IP provides a best-effort datagram delivery service and address handling for transport over the Internet. The NIC and physical layer are abstracted to include FDDI, Ethernet, ATM with AAL 5 or any physical networking medium.

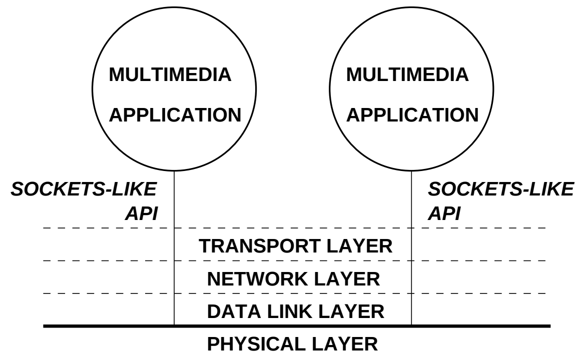

Figure 5.1: XUDP From the API Programmer's Perspective

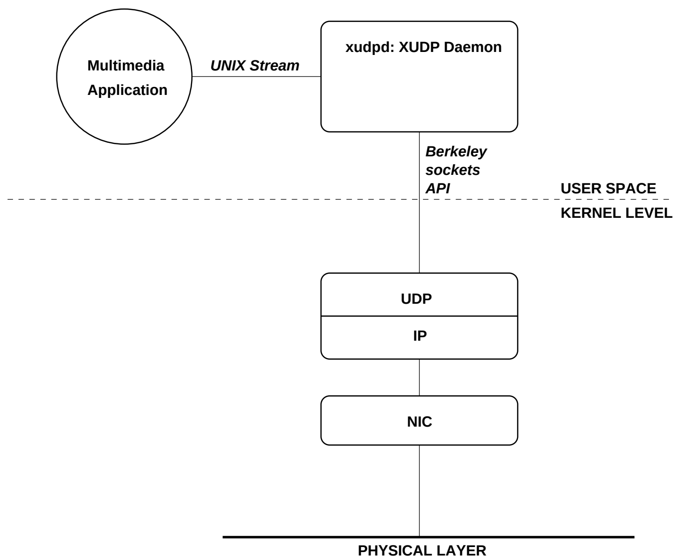

Figure 5.2: Protocol Layers of XUDP

5.2.2 The XUDP daemon: `xudpd`

The application layer needs to be concerned mainly with the presentation of transmitted and received data to the user. Packets will need to be sent, received and acknowledged independently from reads and writes to and from the user's application, but it is not consistent with layered protocol design techniques to require detailed input/output multiplexing on the part of the application layer. Therefore, in order for the flow control algorithms to function asynchronously without intervention from the application program, they must exist as a separate process.

Utilizing a separate process for the protocol implementation preserves the layered programming paradigm, leaving only those functions related to the API as a part of the user's process. However, this separation comes at the cost of an additional connection to move the data between the protocol's daemon process and the user's application. Fortunately, the connection exists only between two processes on the same machine, so it can be implemented with UNIX streams - yielding much lower system load than a TCP/IP implementation!

5.2.3 Application Programmer's Interface (API)

From the API programmer's perspective, XUDP is accessed through a series of high-level sockets-like functions: `xudpSocket`, `xudpBind`, `xudpListen`, `xudpAccept`, `xudpConnect`, `xudpSend`, `xudpRecv` and `xudpClose`. Figure 5.3 demonstrates how similar this interface is to the existing Berkeley sockets API (the diagram is purposely similar to Stevens' *Figure 6.2[42, p.261]*).

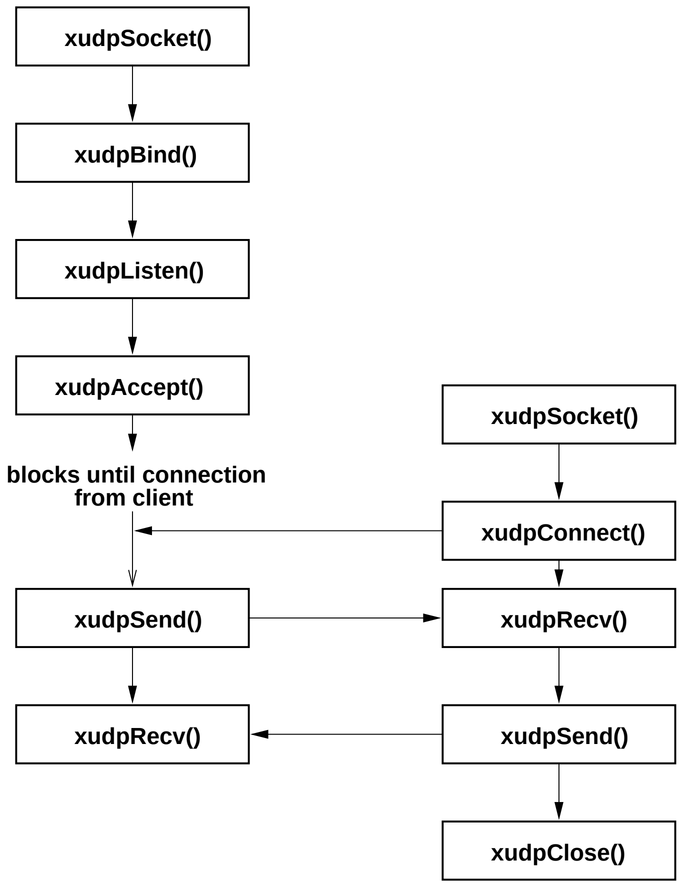

Figure 5.3: XUDP Socket API Calls

Below, the API functionality is discussed briefly, for specific implementation details, refer to section 7.3.

Registering: `xudpSocket`

`xudpSocket` connects the user process to the locally running XUDP server, informing it of its presence. For the sake of programming ease and abstraction, three separate UNIX stream connections will be formed between the user application and the XUDP daemon to distribute control information, audio data and video data. Figure 5.4 shows the stream connections graphically. Among other improvements, this will allow for easier multiplexing between the control information and the data to be sent to the foreign host.

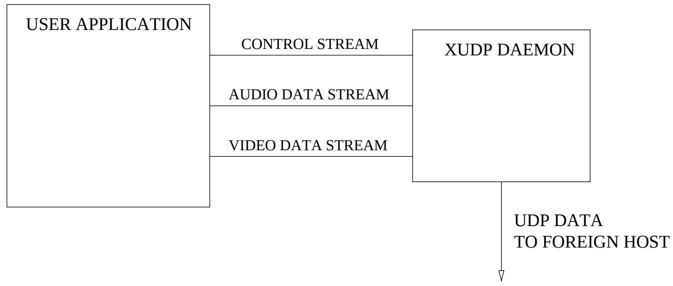

Figure 5.4: XUDP Daemon Stream Connections

Connecting: `xudpConnect`, `xudpBind`, `xudpListen` and `xudpAccept`

XUDP is a connection-oriented protocol. A typical client/server pair developed using the Berkeley sockets API utilizes calls to `socket`, `connect`, `bind`, `listen` and `accept`. In a similar manner, programs developed specifically for XUDP will use `xudpSocket`, `xudpConnect`, `xudpBind`, `xudpListen` and `xudpAccept`. In fact, the XUDP "socket" functions are nearly drop in replacements for their representative Berkeley sockets functions.

In the case of a conventional client side application, after a XUDP socket is created with `xudpSocket`, the application will typically attempt to connect with a server application by using `xudpConnect`.

If the application is a server, it may change the UDP port number that the XUDP server utilizes to receive incoming connections with `xudpBind`. It can then proceed to instruct XUDP to listen for incoming connection requests with `xudpListen` and to block until a connection is accepted with `xudpAccept`.

In the operation of a complete point-to-point communications system, XUDP daemon protocol servers will need to be started on each host involved. A user application will "place a call" with `xudpConnect` XUDP protocol server on a foreign host, where the foreign server will connect the call if a user application has registered with it (`xudpSocket`) and is prepared to accept callers (`xudpBind`→`xudpListen`→`xudpAccept`.) Figure 5.5 shows this case with all layers present for our typical example.

5.2.4 The Parcel

The basic unit of data transfer implemented by XUDP is the *parcel.* Bearing a good deal of resemblance to an IP datagram, the parcel is a variable-length block of data that is sent as a whole from the sending application and received as a whole by the receiving application. If the parcel is not marked as ***`RELIABLE`***, and it expires before all of its data is completely received, the parcel is never delivered and the incomplete data is discarded. Parcels will never be delivered incompletely, either the entire parcel or, if timed-obsolescence is invoked, nothing will be received at the application layer.

5.2.5 `struct XUDPHost *` Host Data Structure

Shown below is the C code that forms the host data structure that XUDP uses to identify connections (defined in `xudp.h`, instead of an integer file descriptor.

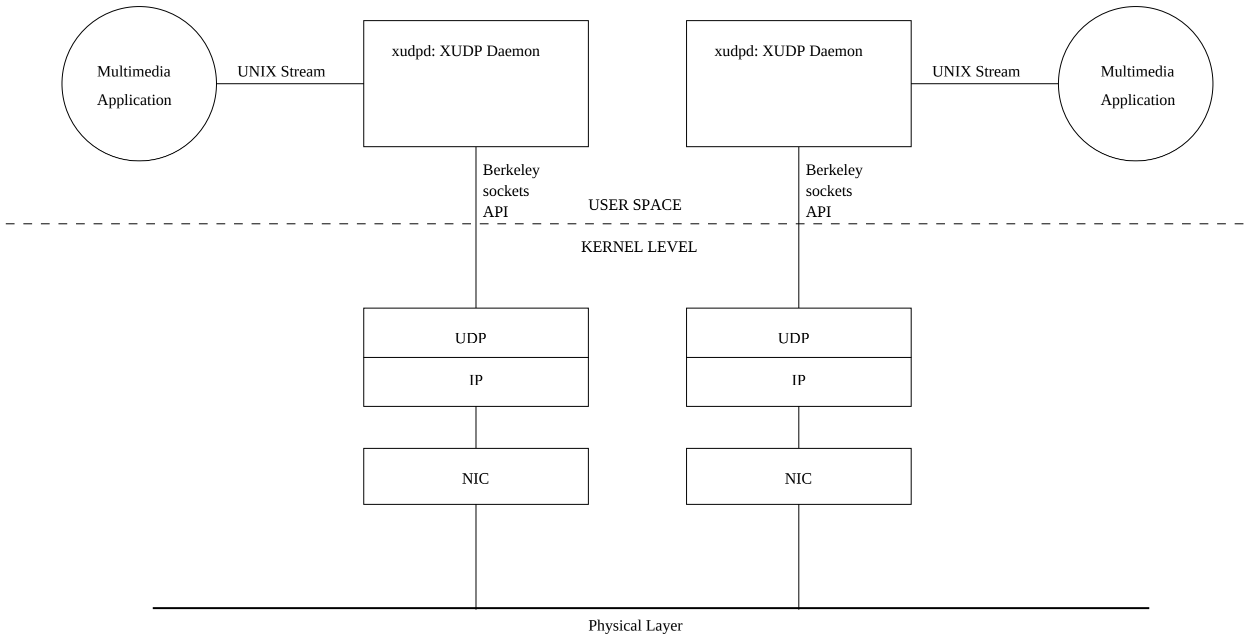

Figure 5.5: Two XUDP Hosts

```
struct XUDPHost
{
    int                         fdAudio;    /* File descriptor of UNIX Stream */
    int                         fdVideo;    /* File descriptor of UNIX Stream */
    int                         fdControl;  /* File descriptor of UNIX
  Stream */
    struct sockaddr_in          hostaddr;
    struct hostent              *pHostent;  /* hostent structure for client */
};
```

`XUDPHost` structure.

XUDP maintains the application layer `struct XUDPHost` through calls to its API functions. The host data structure is intended provide a convenient storage facility for miscellaneous data pertinent to the connection. It keeps the three UNIX stream file descriptors handy for communication with the local XUDP server. The `struct hostent *` structure is defined in `/usr/include/netdb.h` and appears as follows on an OSF/1 V3.2 system:

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
#define h_addr  h_addr_list[0]  /* address, for backward compatiblity */
```

`hostent` structure.

Of particular interest to the conventional application programmer is the storage herein of the official hostname (in C style *NULL* terminated string form) for the connected endpoint. The `hostaddr` entry in the `XUDPHost` structure similarly holds the numerical IP and addressing information related to the connection endpoint.

5.3 Transport Layer

The transport layer consists of the entire XUDP server implementation, from reception of data from the application layer to delivery of fragmented data to the network layer.

5.3.1 Interface to UDP

UDP will act as the delivery service for XUDP data. Although XUDP could have been implemented directly on top of IP, UDP relieves the burden of a few difficult low level operations from XUDP:

- UDP's highly optimized payload checksum will be used to ensure error-free delivery.
- UDP provides the port multiplexing feature, essential if a single XUDP server is to handle multiple connections.
- UDP (through Berkeley sockets) will automatically discard data received from a point other than the specified connection endpoint.

XUDP won't rely on UDP for fragmentation of datagrams. Instead, XUDP will supply UDP with data intended to pass atomically onto the network, with a typical maximum packet size of 512 bytes.

5.3.2 XUDP's Queues

XUDP liberally uses doubly-linked lists to manage data internally at various points in the transmission process. The queues are presented graphically in figure 5.6.

Outgoing Parcel Queue

The outgoing parcel queue holds parcels after they have been received from the application process. The parcel fragmentation process strips data from the beginning of the parcel, one packet at a time, until the end of the parcel has been reached. The parcel is then dequeued.

Incoming Parcel Queue

The incoming parcel queue holds completely reassembled parcels for a short time until they are read by the application layer.

Fragmented Parcel Acknowledgment Wait Queue

The fragmented parcel acknowledgment wait queue holds fragmented parcels until all of the packets that comprise the parcel have been acknowledged.

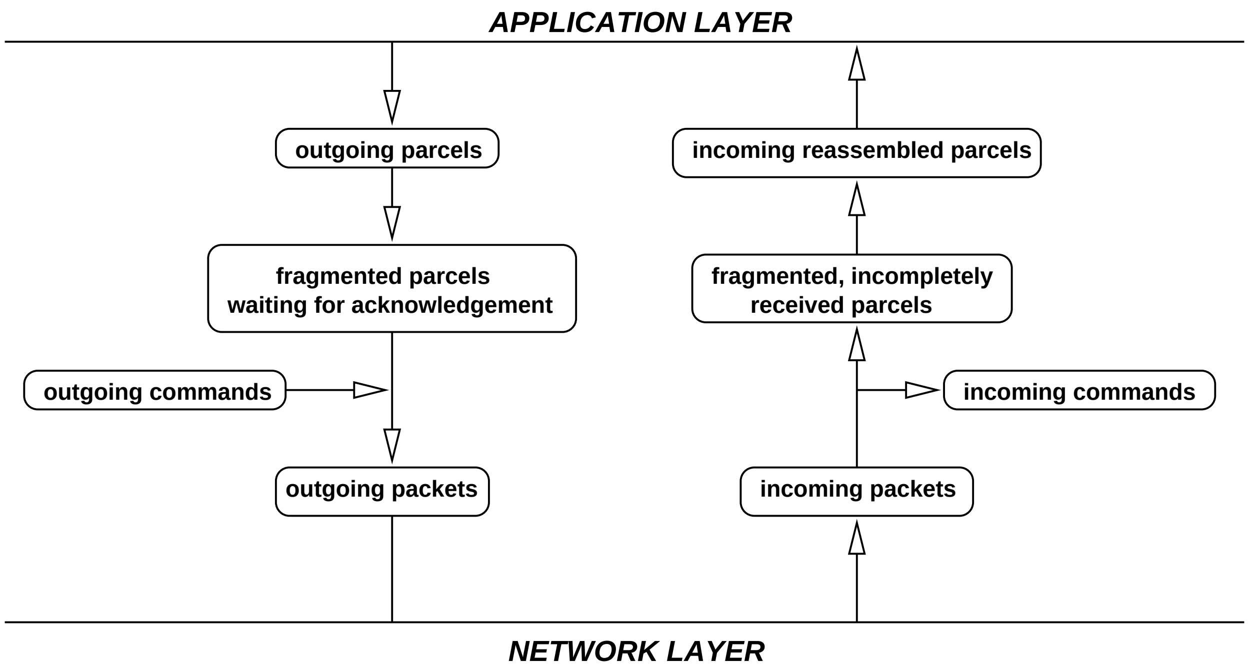

Figure 5.6: XUDP's Queues

Partially Received Fragmented Parcel Queue

The incoming fragment parcel queue holds onto partially formed parcels until they are completed.

Outgoing Command Queue

The outgoing command queue holds commands intended for the connection endpoint's XUDP server that will be inserted at the end of the header in the next available packet. Commands are currently only different types of acknowledgments for received packets and parcels.

Incoming Command Queue

This queue holds incoming commands that have been received from the network and stripped out of their packets. The incoming command queue holds onto the commands until they can be processed by their respective functions.

Outgoing Packet Queue

The outgoing list of packets is maintained until the packets can be transmitted onto the network. Packets on this list are considered to be in the network already by the flow control algorithms.

Incoming Packet Queue

The incoming packet queue is maintained for packets that have been received but not placed on a partially assembled parcel in the received, fragmented parcel queue.

5.3.3 Flow Control

XUDP implements a number of flow control related algorithms based on the ones developed for the Transmission Control Protocol.

Sliding Window

The connection is sliding window based. Flow control is not based upon individual bytes, as in TCP, but on packets, regardless of their length.

Slow Start

There is a slow-start on connection initiation that ramps up the number of packets in the data window linearly.

RTT Variance Estimation

Round trip time variance estimation uses the same method described in section 3.2.2.

Congestion Anticipation

Congestion anticipation is implemented by scaling the size of the window through close monitoring of RTT's and comparisons with recent RTT's.

Exponential Backoff Upon Successive Retransmits

There is an exponential retransmit timer backoff on successive retransmits.

Selective Acknowledgment and Retransmit

Selective acknowledgment is used at both the parcel and packet level. Selective retransmit will be used at the packet level.

Chapter 6

Application Layer Sample Program

Before delving into the intricacies of XUDP's implementation, some application layer examples are warranted to introduce the programmer to the process of generating an application for XUDP.

The following example assumes a properly installed, currently running **xudpd** in the user space. UNIX streams will be used for communication with the daemon, and those streams may create files in the installation directory.

6.1 Simple Stream-oriented Client/Server

For much of the initial testing of XUDP, I used a client/server pair that simply acted as receiver and sender of a parcelized stream of data. One hundred, 10240 byte parcels were transmitted one at a time with full reliability in this example.

6.1.1 Client

The source code for the sending side client is described below.

```
/*
 * Testing platform for XUDP
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "xudp.h"
#include "xudp_socket.h"

#define  DATAFILE "test.txt"

void main(int argc, char *argv[])
{
    struct XUDPHost *xudphost;
    FILE            *file;
    char            *buffer;
    int             i;
```

```
19
20      if (argc<2)
21      {
22       printf("Whoa! No hostname!\n");
23       exit(0);
24      }
25
26      if ((buffer=malloc(10240))==0) exit(0);
```

Top of the stream-based sending side client.

The client requires that the hostname of a machine on which a XUDP server is running, is specified on the command line. Lines 20-24 check for the presence of this command line argument. Line 26 allocates enough memory to hold one parcel's worth of data.

Lines 27 and 28, below, instruct XUDP in the connection process and are purposefully similar to the Berkeley sockets API. `xudphost` is a `struct XUDPHost *` that is filled in upon invoking `xudpSocket` with file descriptors for the UNIX streams used in communications with the XUDP server. `xudpSocket` actually opens this connection with the server at this time, informing the server of the presence of this application. `xudpConnect` sends a message through the UNIX streams to the XUDP server, instructing it to attempt a connection with the hostname given by the string specified in argument 2 of the command. The final argument of `xudpConnect` is used to specify an alternate port number to use that the foreign XUDP server will be listening on, if set to zero, the default is selected.

The remaining lines below open a simple text file and load the parcel buffer with 10,240 bytes of data from the file.

```
27      xudphost=xudpSocket();
28      xudpConnect(xudphost, argv[1], 0);
29      printf("Client connected to %s", xudphost->pHostent->h_name);
30      fflush(stdout);
31
32      file=fopen(DATAFILE, "r");
33      fread(buffer, 10240, 1, file);
34      i=0;
```

Initialization procedure in the sending side client.

Below is the main loop of the code for the client side that focuses on sending the 100 parcels. Most importanly, line 37 contains the `xudpSend` command, used to transmit a parcel through an existing connection. The format of `xudpSend` is as follows:

```
int xudpSend(struct XUDPHost *xudphost, char *buffer, u_short bytes, u_char type, u_long
timeout);
```

`xudpSend` is intended to act as an extended version of `send`, the prototype of which is shown below:

`int send(int s, const void *msg, int len, unsigned int flags);`

`xudphost` is the host address structure, referred to before, acting in place of the socket file descriptor. `buffer` is a pointer to a character buffer and `bytes` is the length in bytes of the buffer. The `type` field is utilized as a means of future expansion, currently only *`VIDEO`* is defined.

Timed obsolescence is implemented with the last argument, `timeout`. `timeout` specifies the parcel's expiration time in milliseconds from complete receipt of the parcel at the local XUDP server. If this timeout expires or will expire before the next round trip time, and the parcel hasn't been completely acknowledged, the parcel is deleted from the queue because it can't possibly reach the destination in time to be useful. However, in the case of this parcel, *`RELIABLE`* has been specified, indicating to the XUDP server that it should implement full reliability in transporting this parcel.

```
    while (i<100)
    {
        if (xudpSend(xudphost, buffer, 10240, VIDEO, RELIABLE)>0)
        {
            i++;
            printf("sent %d\n", i*10240);
            fflush(stdout);
        }
        else
        {
            printf(".");
            fflush(stdout);
        }
    }

    sleep(30);
    xudpClose(xudphost);
}
```

Send loop of the client.

At the end of the process, the connection is closed with `xudpClose`.

6.2 Server

The top of the server side is similar to the client:

```
/*
 * Testing platform for XUDP
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
 8 #include "xudp.h"
 9 #include "xudp_socket.h"
10
11 void main()
12 {
13     struct XUDPHost *xudphost;
14     char            *buffer;
15     FILE            *out;
16     u_short         length;
17     int             total=0;
18     char            type;
19     int             i;
20
21     if ((buffer=malloc(10240))==0) exit(0);
```

Top of the stream-based receiving side server.

Again, the XUDP API follows a process very similar to the Berkeley Sockets API. Below, line 22 starts with `xudpSocket` forming a connection to the local XUDP server (this time on the application layer server-side machine.) Line 23, `xudpBind` instructs the XUDP server to bind the UDP port that it will use to receive incoming communication from the network to the specified port number (its zero, so XUDP uses the default port). `xudpListen` tells the XUDP server to begin listening for incoming connection requests on the previously bound UDP port. Line 25, `xudpAccept` blocks until a foreign XUDP server client forms a successful connection with us.

```
22     xudphost=xudpSocket();
23     xudpBind(xudphost, 0);
24     xudpListen(xudphost);
25     xudpAccept(xudphost);
26     printf("Server connected to %s\n", xudphost->pHostent->h_name);
27     fflush(stdout);
28
29     out=fopen("parcel_recv.log","w");
```

Initialization of the stream-based receiving side server.

The receive loop of the server is markedly similar to that of the client. As was the case with `xudpSend`, `xudpRecv` is intended to act similarly to `recv`. The argument formatting for `xudpRecv` is listed below:

```
int xudpRecv(struct XUDPHost *xudphost, char *buffer, u_short *bytes, u_char *type);
```

Again, `type` is provided only as a means for future expansion.

```
30     for (i=0; i<100; i++)
31     {
```

```
        xudpRecv(xudphost, buffer, &length, &type);
        fwrite(buffer, length, 1, out);
        total+=length;
        printf("received %d\n", total);
        fflush(stdout);
    }

    xudpClose(xudphost);
    fclose(out);
    sleep(15);
}
```

Receive loop of the stream-based receiving side server.

Chapter 7

Source Code Presentation

7.1 Conventions

Properly documenting even the smallest engineering endeavor requires adherence to a set of typographical conventions aimed both at reducing the probability of misinterpretations and keeping the documenter honest. XUDP is just over 4,000 lines of source code. Rather than include the entire block of source code as an appendix, it will be documented and included piecemeal as required throughout this section. In the interest of maintaining a standard, many of the conventions used in this report follow those used in the *TCP/IP Illustrated* series by Wright and Stevens [43].

Short pieces of C code will be listed in a fixed-width font figure seperated from the main text by two horizontal rules as illustrated below. The `typedefs` declared here will be used throughout the XUDP source code.

```
typedef u_short Sequence;   /* 16-bit Sequence numbering */
typedef u_int   Timestamp;  /* 32-bit Timestampping */
typedef u_char  Options;    /* 8-bit  Options field */
typedef u_short Window;     /* 16-bit Window size */
typedef u_char  Boolean;    /* 8-bit  Boolean, why not? */
```

XUDP Specific `typedef` types.

A fixed-width font is used for most variable or function names (`xudp_sendmsg()`) and a slanted, fixed width font for constants (*MAXPAYLOAD*.)

> Brief asides or notes will be offered in this indented, smaller font size format.

7.2 Data Structures

7.2.1 `XUDPHost` structure

```
struct XUDPHost
{
    int                     fdAudio;    /* File descriptor of UNIX Stream */
```

```
    int                    fdVideo;    /* File descriptor of UNIX Stream */
    int                    fdControl;  /* File descriptor of UNIX
  Stream */
    struct sockaddr_in     hostaddr;
    struct hostent         *pHostent;  /* hostent structure for client */
};
```

XUDPHost structure.

7.2.2 XUDPPacket structure

```
struct XUDPPacket
{
    struct XUDPHeader header;              /* Header information */
    u_short           length;              /* Length in bytes of
      payload + header */
    Boolean           ack;                 /* Has packet been ACKed? */
    u_char            payload[MAXPAYLOAD]; /* Duplicates Header with
      Network byte ordering */
    struct XUDPPacket *prev;
    struct XUDPPacket *next;
};
```

XUDPPacket structure.

7.2.3 XUDPHeader structure

```
struct XUDPHeader
{
    Options    options;    /* Options field */
    u_char     cmds;       /* Number of commands present at end of */
   /* header */
    Window     window;     /* Advertised window size */

    Timestamp  timestamp;  /* Packet timestamp field */
    Timestamp  tto;        /* Time to obsolescence timestamp field */

    Sequence   sequence;   /* Packet in-parcel sequence number */
    Sequence   parcel;     /* Parcel sequence number */

    u_short    length;     /* Length of payload */
};
```

XUDPHeader structure.

7.2.4 XUDPParcel structure

```
\begin{center}
struct XUDPParcel
{
    Sequence             parcel;   /* Sequence number of parcel */
    Options              options;  /* High-level options field */
    Timestamp            tto;      /* Time to obsolescence timestamp */
   /* field */
    Timestamp            timestamp;  /* Parcel creation timestamp */
    u_short              length;   /* Length of data buffer */
    char                 *buffer;  /* Pointer to data buffer */
    char                 *pos;     /* Current position in buffer */
    struct XUDPParcel *prev;
    struct XUDPParcel *next;

};
```

XUDPParcel structure.

7.2.5 XUDPFragParcel structure

```
struct XUDPFragParcel
{
    Sequence          parcel;          /* Sequence number of the parcel */
    Options           options;         /* High-level options field */
    Timestamp         tto;             /* Time to obsolescence timestamp */
      /* field */
    Timestamp         timestamp;       /* Parcel creation timestamp */
    Sequence          end;             /* Ending packet number */
    Boolean           ack;             /* Has parcel been ACKed */
    struct XUDPPacket *headPacket;     /* Beginning of packet list */
    struct XUDPFragParcel *prev;
    struct XUDPFragParcel *next;
};
```

XUDPFragParcel structure.

7.2.6 XUDPCmd structure

```
struct XUDPCmd
{
    u_char          cmd;            /* Command ID */
    char            length;         /* Length in bytes of argbuffer */
    Timestamp       timestamp;
    char            *argbuffer;     /* Command argument data */
    struct XUDPCmd *next;
    struct XUDPCmd *prev;
};
```

XUDPCmd structure.

7.2.7 XUDPCB "Control Block" structure

```
struct XUDPCB
{
    int                    fdCli;            /* Socket fd for UDP Client */
    struct sockaddr_in     hostaddr;         /* Address structure for remote */

    struct XUDPFragParcel *pAckWaitQueue;  /* head of parcels waiting */
   /* for ACKS*/
    struct XUDPParcel     *pOutHead;        /* head of outgoing queue */
    struct XUDPFragParcel *pRecvWaitQueue; /* head of received parcels */
    struct XUDPParcel     *pInHead;         /* head of assembled parcels */
    struct XUDPPacket     *pXmitQueue;      /* Outgoing packets */
    struct XUDPPacket     *pRecvQueue;      /* Incoming packets */
    struct XUDPCmd        *pCmdQueue;       /* head of queued up
      commands to be
      transmitted over the
      network */
    struct XUDPCmd        *pCmdInQueue;     /* head of incoming commands */

    struct XUDPHost    *localclient;        /* Local application */

    struct sockaddr_un addrAudio_un;        /* Address structures for */
    struct sockaddr_un addrVideo_un;        /* UNIX streams used      */
    struct sockaddr_un addrControl_un;

    enum   XUDPState   state;               /* State of the XUDP */
   /* connection */

    Boolean            push;                /* XUDP PUSH flag */

    u_short            mps;               /* Current MPS */
    Window             winAdvertised;     /* Local advertised window */
```

```
    Window              winCongestion;     /* Local congestion window */
    float               winFCongestion;    /* Floating point version of */
  /* congestion window */
    Window              winRemote;         /* Remote advertised window */
    Window              packetsInPipe;     /* Packets in network from us */
    Window              packetsInPipeRmt;  /* Packets in network from */
  /* them, we perceive */
    Timestamp           rtt;               /* Current Round Trip Time */
    float               rttavg;            /* Average RTT */
    float               rttvar;            /* Current RTT Variance */
    float               rttmax;            /* Decaying Max RTT */
    float               rttmin;            /* Decaying Min RTT */
    Timestamp           rto;               /* Current Retransmission */
  /* Time Out Timer */

    unsigned long int   bytesSent;         /* Bytes sent during this */
  /* session */
    unsigned long int   bytesACKed;        /* Bytes ACKed during this */
  /* session */
    Timestamp           timeElapsed;       /* Elapsed time for this */
  /* session */
    unsigned int        bandwidth;         /* Calculated average bandwidth */
  /* usage */
    float               windowdelta;       /* Indicates the rate of */
  /* change of the congestion */
  /* window */

    Boolean             usedparcels[MAXPARCEL]; /* Keeps track of used */
       /* parcel numbers */
    Timestamp           closing;           /* Gets set upon invocation */
  /* of XUDP_CLOSE_WAIT state */
};
```

XUDPCB "control block" structure.

7.2.8 XUDPSCB "Server Control Block" structure

```
struct XUDPSCB
{
    struct sockaddr_in addrServ_in;        /* Local UDP Server */
    int                fdServ;             /* Socket fd for UDP Server */
    int                fdServAudio;        /* Socket fd for UNIX Audio */
   /* Stream */

    int                fdServVideo;        /* UNIX Video stream */
```

```
    int                fdServControl;     /* UNIX Control data stream */

    struct sockaddr_un addrServAudio_un;   /* UNIX Address structure for the
      audio stream server */
    struct sockaddr_un addrServVideo_un;   /* Video stream server */
    struct sockaddr_un addrServControl_un; /* Control stream server */
};
```

XUDPSCB "server control block" structure.

7.3 Application Programmers Interface: XUDP Sockets

7.3.1 xudpSocket function

`struct XUDPHost *xudpSocket(void);`

```
struct XUDPHost *xudpSocket(void)
{
    struct XUDPHost     *xudphost;
    int                 un_len, temp;
    struct sockaddr_un  xudp_addr;

    if ( !(xudphost=malloc(sizeof(struct XUDPHost))))
        xudpError("xudpSocket: Allocating RAM");

    bzero((char *) &xudp_addr, sizeof(xudp_addr));
    if ( (xudphost->fdControl=socket(AF_UNIX, SOCK_STREAM, 0)) <0)
        xudpError("xudp: socket(UNIX)");   /* Opens a UNIX Stream socket */
    if ( (xudphost->fdAudio=socket(AF_UNIX, SOCK_STREAM, 0)) <0)
        xudpError("xudp: socket(UNIX)");   /* Opens a UNIX Stream socket */
    if ( (xudphost->fdVideo=socket(AF_UNIX, SOCK_STREAM, 0)) <0)
        xudpError("xudp: socket(UNIX)");   /* Opens a UNIX Stream socket */

                      /* Fill out address structure for server */

    xudp_addr.sun_family=AF_UNIX;         /* AF_UNIX is the UNIX Domain
                                             Protocol Family.*/

    /*** Control Port **********/
    strcpy(xudp_addr.sun_path, CONTROL_UNIX); /* Server stream path */

    un_len=strlen(xudp_addr.sun_path) + sizeof(xudp_addr.sun_family);

    if (connect(xudphost->fdControl, (struct sockaddr *) &xudp_addr,
                un_len) < 0)
        xudpError("xudpSocket: Could not connect");
```

```

    /*** Audio Port **********/
    strcpy(xudp_addr.sun_path, AUDIO_UNIX); /* Server stream path */

    un_len=strlen(xudp_addr.sun_path) + sizeof(xudp_addr.sun_family);

    if (connect(xudphost->fdAudio, (struct sockaddr *) &xudp_addr,
                un_len) < 0)
        xudpError("xudpSocket: Could not connect");

    /*** Video Port **********/
    strcpy(xudp_addr.sun_path, VIDEO_UNIX); /* Server stream path */

    un_len=strlen(xudp_addr.sun_path) + sizeof(xudp_addr.sun_family);

    if (connect(xudphost->fdVideo, (struct sockaddr *) &xudp_addr,
                un_len) < 0)
        xudpError("xudpSocket: Could not connect");

    /*** Socket buffer sizes *************/

    temp=SOCKBUFSIZE;  /* Set up maximum socket-level buffers */
    if (setsockopt(xudphost->fdControl, SOL_SOCKET, SO_SNDBUF,
                   &temp, sizeof(int))<0) xudpError("xudpSocket");
    if (setsockopt(xudphost->fdControl, SOL_SOCKET, SO_RCVBUF,
                   &temp, sizeof(int))<0) xudpError("xudpSocket");
    if (setsockopt(xudphost->fdAudio, SOL_SOCKET, SO_SNDBUF,
                   &temp, sizeof(int))<0) xudpError("xudpSocket");
    if (setsockopt(xudphost->fdAudio, SOL_SOCKET, SO_RCVBUF,
                   &temp, sizeof(int))<0) xudpError("xudpSocket");
    if (setsockopt(xudphost->fdVideo, SOL_SOCKET, SO_SNDBUF,
                   &temp, sizeof(int))<0) xudpError("xudpSocket");
    if (setsockopt(xudphost->fdVideo, SOL_SOCKET, SO_RCVBUF,
                   &temp, sizeof(int))<0) xudpError("xudpSocket");

    return xudphost;
}
```

7.3.2 xudpBind function

`int xudpBind(struct XUDPHost *xudphost, u_short port);`

```
int xudpBind(struct XUDPHost *xudphost, u_short port)
{
    xudpSendFunction(xudphost, F_BIND, (char *) &port);

```

```
    return 0;
}
```

7.3.3 `xudpListen` function

`int xudpListen(struct XUDPHost *xudphost);`

```
int xudpListen(struct XUDPHost *xudphost)
{
    xudpSendFunction(xudphost, F_LISTEN, NULL);
    return 0;
}
```

7.3.4 `xudpAccept` function

`int xudpAccept(struct XUDPHost *xudphost);`

```
int xudpAccept(struct XUDPHost *xudphost)
{
    xudpSendFunction(xudphost, F_ACCEPT, NULL);

    if (xudpGetFunction(xudphost)==F_ERROR)
        return xudpGetFunction(xudphost);

    streamRead(xudphost->fdControl, (char *) &xudphost->hostaddr,
               sizeof(struct sockaddr_in));

    xudphost->pHostent=gethostbyaddr((char *)&xudphost->hostaddr.sin_addr,
                                      sizeof(struct in_addr), AF_INET);

    return 0;
}
```

7.3.5 `xudpConnect` function

`int xudpConnect(struct XUDPHost *xudphost, char *fqdn, u_short port);`

```
int xudpConnect(struct XUDPHost *xudphost, char *fqdn, u_short port)
{
    xudphost->pHostent=gethostbyname(fqdn);
                                       /* Given any DNS or /etc/hosts
                                          registered hostname, gethostbyname()
```

```
                                         will return useful information,
                                         including the 32-bit IP that we need
                                         */
    if (xudphost->pHostent==NULL)
        xudpError("Trying to resolve hostname");
                                 /* Make sure our hostname is valid */

    memcpy(&xudphost->hostaddr.sin_addr, xudphost->pHostent->h_addr_list[0],
           sizeof(struct in_addr));
    xudphost->hostaddr.sin_family=AF_INET;
    if (port==0) xudphost->hostaddr.sin_port=htons(SERVER_PORT);
    else xudphost->hostaddr.sin_port=htons(port);

    xudpSendFunction(xudphost, F_CONNECT, NULL);

    if (xudpGetFunction(xudphost)==F_ERROR)
        return xudpGetFunction(xudphost);
    streamRead(xudphost->fdControl, (char *) &xudphost->hostaddr,
               sizeof(struct sockaddr_in));

    return 0;
}
```

7.3.6 `xudpSend` function

`int xudpSend(struct XUDPHost *xudphost, char *buffer, u_short bytes, u_char type, u_long timeout);`

```
int xudpSend(struct XUDPHost *xudphost, char *buffer, u_short bytes,
             u_char type, u_long timeout)
{
    Timestamp tto;
    int       result;

    if ((timeout/GRANULARITY)>RELIABLE) tto=RELIABLE;
    else tto=timeout/GRANULARITY;

    /* CAUTION: Improve Error Handling! */
    switch (type)
    {
/*    case AUDIO:
        xudpSendFunction(xudphost, F_SENDAUDIO, (char *)&bytes);
        if (streamWrite(xudphost->fdControl, (char *)&tto,
                        sizeof(tto))!=sizeof(tto))
            xudpError("UNIX Stream Failed");
        if (streamWrite(xudphost->fdAudio, buffer, bytes)!=bytes)
```

```
            xudpError("UNIX Stream Failed");
        break;*/
    case VIDEO:
        if ((result=send(xudphost->fdVideo, buffer, bytes, 0))>0)
        {
            if (result!=bytes)
                xudpError("UNIX Stream Failed");
            xudpSendFunction(xudphost, F_SENDVIDEO, (char *)&bytes);
            if (streamWrite(xudphost->fdControl, (char *)&tto,
                            sizeof(tto))!=sizeof(tto))
                xudpError("UNIX Stream Failed");
        }
        else return -1;

        break;
    default: return 0; break;
    }


    return bytes;
}
```

7.3.7 `xudpRecv` function

`int xudpRecv(struct XUDPHost *xudphost, char *buffer, u_short *bytes, u_char *type);`

```
int xudpRecv(struct XUDPHost *xudphost, char *buffer, u_short *bytes,
             u_char *type)
{
    char    cmd;
    u_short length;

    cmd=xudpGetFunction(xudphost);
    switch (cmd)
    {
    case F_RECVAUDIO: break;
    case F_RECVVIDEO:
        streamRead(xudphost->fdControl, (char *)&length, sizeof(length));
        streamRead(xudphost->fdVideo, buffer, length);
        memcpy(bytes, &length, sizeof(u_short));
        *type=VIDEO;
        break;
    default:    return 0; break;
    }
    return length;
}
```

7.3.8 `xudpClose` function

`int xudpClose(struct XUDPHost *xudphost);`

```
int xudpClose(struct XUDPHost *xudphost)
{
    xudpSendFunction(xudphost, F_CLOSE, NULL);
    xudpGetFunction(xudphost);
    close(xudphost->fdControl);
    close(xudphost->fdAudio);
    close(xudphost->fdVideo);
    free(xudphost);
    return 0;
}
```

7.4 User Space Implementation: XUDP Daemon

7.4.1 `main()`

```
void main()
{
    struct XUDPCB  *xudpcb;
    struct XUDPSCB *xudpscb;
    struct XUDPCmd *pCmd;
    int            pidChild;
    fd_set         setFD;
    int            result;
    struct timeval timestuff;
    Timestamp      ts;
    Boolean        dontwait=FALSE;
#ifdef STATLOG
    Timestamp      statts=0;
#endif
    printf("xudpd: eXtended User Datagram Protocol handler\n");
    printf("[Initializing...");
    fflush(stdout);

    if ( !(xudpscb=malloc(sizeof(struct XUDPSCB))))
        xudpError("xudpd: Allocating RAM");

    xudp_init(xudpscb);

```

```
    printf("ok]\n");
    fflush(stdout);

    while(1) /* Loop "forever" */
    {
        if ( !(xudpcb=malloc(sizeof(struct XUDPCB))))
            xudpError("xudpd: Allocating RAM");

        printf("[Waiting for client...");
        fflush(stdout);

        xudp_getnewuser(xudpcb, xudpscb);

        printf("accepted client]\n");
        fflush(stdout);

        /* FORK BELOW *****************************************************/

#ifdef DONTFORK
/* DISABLE FORKING */
        pidChild=0;
#endif
#ifndef DONTFORK
        if ((pidChild=fork())<0) xudpError("Trying to fork");
        else
#endif

        if (pidChild==0)
        {
            /***************************************************************
             * CONNECT Loop:  Executes until either:
             *       1) A foreign "xudpd" connects with us through
             *          the Internet
             *       2) Our local user application wishes to place a
             *          call to a foreign host.
             */
            while(!xudpcb->state)
            {
                xudp_handlecontrol(xudpcb, xudpscb);
            }

#ifdef DEBUG
            show_sockaddr_in(&xudpcb->hostaddr);
#endif

            printf("[Default socket buffer size is 32K]\n");
            fflush(stdout);
            xudp_socketinit(xudpcb);
```

```
            xudpcb->state=XUDP_SOFT_START;

            while(xudpcb->state) /* Main protocol loop ******************/
            {
#ifdef DEBUG
                show_status(xudpcb);
#endif
#ifdef DELAY
                sleep(1);
#endif
                ts=makeTimestamp();
                if (ts-statts>=20)
                {
                    statts=ts;
/*                  xudp_windowchange(xudpcb); */
#ifdef STATLOG
                    fprintf(xudpcb->bytlog, "%u %lu\n",
                             xudpcb->timeElapsed, xudpcb->bytesACKed);
/*                  fflush(xudpcb->bytlog);*/
                    fprintf(xudpcb->rttlog, "%u %u %f\n",
                            xudpcb->timeElapsed,
                            xudpcb->rtt, xudpcb->rttavg);
/*                  fflush(xudpcb->rttlog);*/
                    fprintf(xudpcb->paclog, "%u %u\n",
                            xudpcb->timeElapsed,xudpcb->packetsInPipe);
/*                  fflush(xudpcb->paclog);*/
#endif
                 }
                 xudpcb->timeElapsed=ts;

                 /** Just poll if the parcel outgoing queue has data */
                 /** waiting in it AND there's room on the net for it */

                 if (xudpcb->packetsInPipe<xudpcb->winRemote)
                     if (xudpcb->pOutHead->next!=NULL)
                         dontwait=TRUE;

                 /** Just poll if there's packets waiting to go out */

                 if (xudpcb->pXmitQueue->next!=NULL) dontwait=TRUE;

                 pCmd=xudpcb->pCmdQueue->next;
                 while (pCmd!=NULL)
                 {
                     /* Make a command parcel if we're really waiting
                        around, flushes out commands that might be held in
                        our queue */
```

```
    if ((ts-xudpcb->pCmdQueue->next->timestamp)>=TIMEOUT)
    {
        xudp_makecommandparcel(xudpcb);
        xudpcb->push=TRUE;
        pCmd=NULL;
    }

    if (pCmd!=NULL) pCmd=pCmd->next;
}

/** Wait a reasonably long time for something on */
/** either the UNIX stream or Internet UDP sockets - */
/** saves system load, unless we're pushing a parcel */
/** or just polling  */

if (!(xudpcb->push) && !(dontwait))
{
    result=0;
    FD_ZERO(&setFD);
    FD_SET(xudpcb->localclient->fdControl, &setFD);
    FD_SET(xudpcb->fdCli, &setFD);
    timestuff.tv_usec=1000*GRANULARITY*TIMEOUT;
    timestuff.tv_sec=0;
    if ((result=select(max(xudpcb->localclient->fdControl,
                           xudpcb->fdCli)+1, &setFD, NULL,
                       NULL, &timestuff))<0)
        xudpError("select");
}
else /* PUSH or DONTWAIT is set, do a quick poll */
{
    result=0;
    FD_ZERO(&setFD);
    FD_SET(xudpcb->localclient->fdControl, &setFD);
    FD_SET(xudpcb->fdCli, &setFD);
    timestuff.tv_usec=0;
    timestuff.tv_sec=0;
    if ((result=select(max(xudpcb->localclient->fdControl,
                           xudpcb->fdCli)+1, &setFD, NULL,
                       NULL, &timestuff))<0)
        xudpError("select");
}
dontwait=FALSE;

/** See if we're trying to close, if we are, close */
/** after we've waited for everybody. */

if (xudpcb->state==XUDP_CLOSE_WAIT)
```

```
                if (makeTimestamp()-(xudpcb->closing)>=TIMECLOSE)
                    xudpcb->state=0;
            if (xudpcb->state)
            {
                /****** Incoming Commands on UNIX Stream ***********/
                if (FD_ISSET(xudpcb->localclient->fdControl, &setFD))
                    xudp_handlecontrol(xudpcb, xudpscb);

                /****** Incoming packets from the network **********/
                /*** Queue on cb->pRecvQueue */
                if (FD_ISSET(xudpcb->fdCli, &setFD))
                    xudp_recv(xudpcb);

                /*** Move packets in cb->pRecvQueue to appropriate */
                /*** parcels in cb->pRecvWaitQueue and order them */
                xudp_organizeincoming(xudpcb);

                /*** Mark ending packet sequence numbers on parcels */
                xudp_endparcel(xudpcb);

                /*** Set the ACK flag on parcels that we've */
                /*** received an ACK for */
                xudp_recvackparcels(xudpcb);

                /*** Set the ACK flag on individual packets that */
                /*** we've received an ACK for */
                xudp_recvackpackets(xudpcb);

                /*** Queue up commands to ACKnowledge parcels that */
                /*** we've  received in their entirety */
                xudp_ackparcels(xudpcb);

                /*** Queue up commands to ACKnowledge individual */
                /*** packets based on various flow control */
                /*** algorithms */
                xudp_ackpackets(xudpcb);

                /*** Reassemble fragmented RAW parcels to */
                /*** cb->pInHead formed by completed parcels in */
                /*** cb->pRecvWaitQueue */
                xudp_reassembleparcel(xudpcb);

                /*** Fragment and queue up packets on */
                /*** cb->pXmitQueue to fill up our advertised */
                /*** window.  Take packets from queued RAW parcels */
                /*** in cb->pOutHead.  Insert waiting commands to */
                /*** be sent from cb->pCmdQueue */
                xudp_fragmentparcel(xudpcb);
```

```

                        /*** Send packets queued on cb->pXmitQueue and */
                        /*** properly queue those packets on the list of */
                        /*** packets waiting to receive acknowledgements */
                        /*** (cb->pAckWaitQueue) */
                        xudp_xmit(xudpcb);

                        /*** Blow completed RAW parcels on the cb->pInHead */
                        /*** out to our XUDP application */
                        xudp_dequeueparcel(xudpcb);

                        /*** Remove obsoleted parcels from the queues */
                        xudp_removeobsolete(xudpcb);

                        /*** Retransmit packets that have exceeded their */
                        /*** RTO's */
                        xudp_retransmit(xudpcb);
                    }
                }

                /** Reset client connections, clean up the mess we've made */
                /** of RAM and prepare for next client. */
                xudp_loseuser(xudpcb);

                /** Quit out of this child process */
                exit(0);
            } /* end Child Process */

        } /* end "Loop Forever" */

        close(xudpscb->fdServ);
        close(xudpscb->fdServAudio);
        close(xudpscb->fdServVideo);
        close(xudpscb->fdServControl);
        free(xudpscb);
}
```

7.4.2 `xudp_init` function

`int xudp_init(struct XUDPSCB *scb);`

```
int xudp_init(struct XUDPSCB *scb)
{
    /* xudpd handles two connection points!
       ---------------------------------------
       1. It will bind itself to UDP/IP port SERVER_PORT as defined in
```

```
       xudp.h.  It will accept incoming connections, presumably from
       foreign xudpd's.

       2. It will bind itself to UNIX Streams CONTROL_UNIX, AUDIO_UNIX and
       VIDEO_UNIX as defined in xudp.h.  It will accept incoming connections,
       presumably from local XUDP applications.*/

    /* Internet server side *********************************************/

    if ( (scb->fdServ=socket(AF_INET, SOCK_DGRAM, 0)) <0)
        xudpError("xudp_init: socket(UDP)");     /* Opens a UDP/IP socket */

                    /* Fill out address structure for server */

    scb->addrServ_in.sin_family=AF_INET;         /* AF_INET is the Internet
                                                    Protocol Family.  */
    scb->addrServ_in.sin_addr.s_addr=htonl(INADDR_ANY);
                                               /* Will accept incoming
                                                  connections from any
                                                  address */

    /* Localhost UNIX Protocol server side *****************************/

    /* makeServer_un creates the UNIX stream sockets, binds them to the
       specified pathnames and starts listening for connections */

    scb->fdServAudio=makeServer_un((struct sockaddr_un
                                    *)&scb->addrServAudio_un, AUDIO_UNIX);
    scb->fdServVideo=makeServer_un((struct sockaddr_un
                                    *)&scb->addrServVideo_un, VIDEO_UNIX);
    scb->fdServControl=makeServer_un((struct sockaddr_un
                                      *)&scb->addrServControl_un,
                                     CONTROL_UNIX);
    return 0;
}
```

7.4.3 xudp_getnewuser function

int xudp_getnewuser(struct XUDPCB *cb, struct XUDPSCB *scb);

```
int xudp_getnewuser(struct XUDPCB *cb, struct XUDPSCB *scb)
{
    int un_len, i;

    if ( !(cb->localclient=malloc(sizeof(struct XUDPHost))))
        xudpError("xudp_getnewuser: Allocating RAM");
```

```
/******* Wait here until a local XUDP application registers with us */

un_len=sizeof(cb->addrControl_un);

cb->localclient->fdControl=accept(scb->fdServControl,
                                  (struct sockaddr *)&cb->addrControl_un,
                                  &un_len);
                          /* Block until we accept (or we error
                             out) on an incoming UNIX connection.
                             The addressing information for the
                             client will be stored in addrControl_un */
if (cb->localclient->fdControl<0)
    xudpError("xudpd: Accepting connection (UNIX)");
                          /* Check for errors */

un_len=sizeof(cb->addrAudio_un);
cb->localclient->fdAudio=accept(scb->fdServAudio,
                                (struct sockaddr *)&cb->addrAudio_un,
                                &un_len);
if (cb->localclient->fdAudio<0)
    xudpError("xudpd: Accepting connection (UNIX)");

un_len=sizeof(cb->addrVideo_un);
cb->localclient->fdVideo=accept(scb->fdServVideo,
                                (struct sockaddr *)&cb->addrVideo_un,
                                &un_len);
if (cb->localclient->fdVideo<0)
    xudpError("xudpd: Accepting connection (UNIX)");

/*** Initialize the variables in the control block for this new */
/*** application */

cb->pOutHead=makeParcel();
cb->pInHead=makeParcel();

cb->pAckWaitQueue=makeFragParcel();
cb->pRecvWaitQueue=makeFragParcel();

cb->pCmdQueue=makeCmd();
cb->pCmdInQueue=makeCmd();

cb->pXmitQueue=makePacket();
cb->pRecvQueue=makePacket();

cb->pOutHead->parcel=0;
cb->pInHead->parcel=0;
cb->pAckWaitQueue->parcel=0;
```

```
    cb->pRecvWaitQueue->parcel=0;

    cb->mps=MPS;
    cb->push=0;
    cb->winAdvertised=MAXWINDOW;
    cb->winCongestion=1;
    cb->winRemote=MAXWINDOW;
    cb->packetsInPipe=0;
    cb->packetsInPipeRmt=0;
    cb->rtt=1;
    cb->rttavg=10.0;
    cb->rttvar=20.0;
    cb->rttmax=20.0;
    cb->rttmin=0.0;
    cb->rto=20;
    cb->windowdelta=0.0;

    for (i=0; i<MAXPARCEL; i++)
        cb->usedparcels[i]=FALSE;

#ifdef SRLOG
    cb->sendlog=fopen("send.log", "w");
    cb->recvlog=fopen("recv.log", "w");
#endif

#ifdef STATLOG
    cb->bytlog=fopen("stats.byt", "w");
    cb->rttlog=fopen("stats.rtt", "w");
    cb->paclog=fopen("stats.pac", "w");
#endif
    return 0;
}
```

7.4.4 `xudp_socketinit` function

`int xudp_socketinit(struct XUDPCB *cb);`

```
int xudp_socketinit(struct XUDPCB *cb)
{
    int temp;

    temp=SOCKBUFSIZE;  /* Set up maximum socket-level buffers */
    if (setsockopt(cb->localclient->fdControl, SOL_SOCKET, SO_SNDBUF,
                   &temp, sizeof(int))<0) xudpError("xudp_socketinit");
    if (setsockopt(cb->localclient->fdControl, SOL_SOCKET, SO_RCVBUF,
                   &temp, sizeof(int))<0) xudpError("xudp_socketinit");
```

```
    if (setsockopt(cb->localclient->fdAudio, SOL_SOCKET, SO_SNDBUF,
                   &temp, sizeof(int))<0) xudpError("xudp_socketinit");
    if (setsockopt(cb->localclient->fdAudio, SOL_SOCKET, SO_RCVBUF,
                   &temp, sizeof(int))<0) xudpError("xudp_socketinit");
    if (setsockopt(cb->localclient->fdVideo, SOL_SOCKET, SO_SNDBUF,
                   &temp, sizeof(int))<0) xudpError("xudp_socketinit");
    if (setsockopt(cb->localclient->fdVideo, SOL_SOCKET, SO_RCVBUF,
                   &temp, sizeof(int))<0) xudpError("xudp_socketinit");
    if (setsockopt(cb->fdCli, SOL_SOCKET, SO_SNDBUF, &temp,
                   sizeof(int))<0) xudpError("xudp_socketinit");
    if (setsockopt(cb->fdCli, SOL_SOCKET, SO_RCVBUF, &temp,
                   sizeof(int))<0) xudpError("xudp_socketinit");

    if (fcntl(cb->fdCli, F_SETFL, FNDELAY)<0)
        xudpError("xudp_socketinit");

    return 0;
}
```

7.4.5 xudp_loseuser function

`int xudp_loseuser(struct XUDPCB *cb);`

```
int xudp_loseuser(struct XUDPCB *cb)
{
    xudpSendFunction(cb->localclient, F_CLOSE, NULL);
    sleep(1);
    close(cb->fdCli);
    close(cb->localclient->fdControl);
    close(cb->localclient->fdAudio);
    close(cb->localclient->fdVideo);
    free(cb->localclient);

    while (cb->pAckWaitQueue->next!=NULL)
      deleteFragParcel(cb->pAckWaitQueue->next);
    deleteFragParcel(cb->pAckWaitQueue);

    while (cb->pOutHead->next!=NULL)
      deleteParcel(cb->pOutHead->next);
    deleteParcel(cb->pOutHead);

    while (cb->pRecvWaitQueue->next!=NULL)
      deleteFragParcel(cb->pRecvWaitQueue->next);
    deleteFragParcel(cb->pRecvWaitQueue);

    while (cb->pInHead->next!=NULL)
```

```
      deleteParcel(cb->pInHead->next);
    deleteParcel(cb->pInHead);

    while (cb->pXmitQueue->next!=NULL)
      deletePacket(cb->pXmitQueue->next);
    deletePacket(cb->pXmitQueue);

    while (cb->pRecvQueue->next!=NULL)
      deletePacket(cb->pRecvQueue->next);
    deletePacket(cb->pRecvQueue);

    while (cb->pCmdQueue->next!=NULL)
      deleteCmd(cb->pCmdQueue->next);
    deleteCmd(cb->pCmdQueue);

    while (cb->pCmdInQueue->next!=NULL)
      deleteCmd(cb->pCmdInQueue->next);
    deleteCmd(cb->pCmdInQueue);

#ifdef SRLOG
    fclose(cb->sendlog);
    fclose(cb->recvlog);
#endif
#ifdef STATLOG
    fclose(cb->bytlog);
    fclose(cb->rttlog);
    fclose(cb->paclog);
#endif

    free(cb);
    return 0;
}
```

7.4.6 xudp_handlecontrol function

int xudp_handlecontrol(struct XUDPCB *cb, struct XUDPSCB *scb);

```
int xudp_handlecontrol(struct XUDPCB *cb, struct XUDPSCB *scb)
{
    int     fcn=0;
    u_short temp;
    int     in_len;
    char    buffer[25];
    struct sockaddr_in remoteaddr, localaddr;
    fd_set  setFD;
    char    t2;
```

```
struct XUDPCmd *pCmd;

fcn=xudpGetFunction(cb->localclient);
switch (fcn)
{
case F_CONNECT:
  printf("[Received connection request]\n");
  fflush(stdout);
  /* Opens a UDP/IP socket */
  if ((cb->fdCli=socket(AF_INET, SOCK_DGRAM, 0))<0)
    xudpError("xudpd: socket(UDP)");
  localaddr.sin_family=AF_INET;
  localaddr.sin_port=htons(0);
  localaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  if ((bind(cb->fdCli, (struct sockaddr *)&localaddr,
     sizeof(struct sockaddr_in)))<0)
    xudpError("xudpd: bind(UDP)");  /* Sets up a unique port
                                       number */

  streamRead(cb->localclient->fdControl, (char *) &cb->hostaddr,
        sizeof(struct sockaddr_in));
  temp=0;
  while (++temp<5)
  {
    if (sendto(cb->fdCli, "HELLO XUDP", 10, 0,
          (struct sockaddr *)&cb->hostaddr,
          sizeof(struct sockaddr_in))<0)
      xudpError("sendto");
    sleep(1);
    in_len=sizeof(struct sockaddr_in);
    if (recvfrom(cb->fdCli, buffer, 10, 0,
           (struct sockaddr *)&remoteaddr, &in_len)>0)
    {
      if (memcmp("HELLO XUDP", buffer, 10)==0)
      {
        temp=101;
        /*** Grab the new port number */
        memcpy(&cb->hostaddr, &remoteaddr,
            sizeof(struct sockaddr_in));
        /*** Connect with remote xudpd to gain system
         level features */
        if (connect(cb->fdCli,
              (struct sockaddr *)&cb->hostaddr,
              sizeof(struct sockaddr_in))<0)
          xudpError("connect");
      }
    }
    else xudpError("recvfrom");
```

```
        }
        if (temp<100) /* No connect for 5 seconds */
        {
            t2=ERR_NOCONNECT;
            xudpSendFunction(cb->localclient, F_ERROR, (char *)&t2);
            printf("[Aborted, no connect for 5 seconds]\n");
            close(cb->fdCli);
            fflush(stdout);
        }
        else
        {
            printf("[Connected successfully]\n");
            fflush(stdout);
            cb->state=XUDP_ESTABLISHED;
            xudpSendFunction(cb->localclient, F_CONNECTED,
                             (char *)&cb->hostaddr);
            sleep(1);
        }
        break;
    case F_ACCEPT:
        printf("[Waiting for foreign clients]\n");
        fflush(stdout);

        /*** Wait for foreign xudpd to wake us up */
        FD_ZERO(&setFD);
        FD_SET(scb->fdServ, &setFD);

        /* Wait indefinitely for something on the UDP server socket */
        while(!cb->state)
        {
            temp=0;
            if (select(scb->fdServ+1, &setFD, NULL, NULL, NULL)<0)
                xudpError("select");
            in_len=sizeof(struct sockaddr_in);
            if (recvfrom(scb->fdServ, buffer, 24, 0,
                         (struct sockaddr *)&remoteaddr, &in_len)>0)
            {
                if (memcmp("HELLO XUDP", buffer, 10)==0)
                {
                    /* Opens a UDP/IP socket */
                    if ((cb->fdCli=socket(AF_INET, SOCK_DGRAM, 0))<0)
                        xudpError("xudpd: socket(UDP)");
                     localaddr.sin_family=AF_INET;
                     localaddr.sin_port=htons(0);
                     localaddr.sin_addr.s_addr=htonl(INADDR_ANY);
                     if (bind(cb->fdCli, (struct sockaddr *)&localaddr,
                              sizeof(struct sockaddr_in))<0)
                         xudpError("xudpd: bind");
```

```
                    /* Sets up a unique port number */
                    memcpy(&cb->hostaddr, &remoteaddr,
                           sizeof(struct sockaddr_in));
                    if (sendto(cb->fdCli, "HELLO XUDP", 10, 0,
                               (struct sockaddr *)&cb->hostaddr,
                               sizeof(struct sockaddr_in))<0)
                        xudpError("sendto");
                    if (connect(cb->fdCli,
                                (struct sockaddr *)&cb->hostaddr,
                                sizeof(struct sockaddr_in))<0)
                        xudpError("xudpd: connect");
                    xudpSendFunction(cb->localclient, F_CONNECTED,
                                     (char *)&cb->hostaddr);
                    cb->state=XUDP_ESTABLISHED;
                }
            }
        }
        break;
    case F_BIND:
        printf("[Binding local address]\n");
        fflush(stdout);
        streamRead(cb->localclient->fdControl, (char *) &temp,
                   sizeof(u_short));
        if (temp==0) scb->addrServ_in.sin_port=htons(SERVER_PORT);
        else scb->addrServ_in.sin_port=htons(temp);

        in_len=sizeof(struct sockaddr_in);

        if (bind(scb->fdServ, (struct sockaddr *)&(scb->addrServ_in),
                 in_len)<0 )
            xudpError("xudpd: bind()");/* We "bind" the server
                                          to the information
                                          in xudpserver->Addr */
        break;
    case F_LISTEN:
        printf("[Listening for foreign clients]\n");
        fflush(stdout);
        listen(scb->fdServ, 1);/* Listen for clients, holding at most 1
                                  in the queue */
        break;
    case F_CLOSE:
        cb->state=XUDP_CLOSE_WAIT;
        cb->closing=makeTimestamp();
        pCmd=makeCmd();
        pCmd->cmd=C_CLOSE;
        cb->push=TRUE;
        break;
    case F_SENDVIDEO:
```

```
        xudp_queueparcel(cb, VIDEO);
        break;
    case F_SENDAUDIO:
        xudp_queueparcel(cb, AUDIO);
        break;
    default: break;
    }
    return 0;
}
```

7.5 Parcel Sublayer

7.5.1 `xudp_queueparcel` function

`int xudp_queueparcel(struct XUDPCB *cb, u_char type);`

```
int xudp_queueparcel(struct XUDPCB *cb, u_char type)
{

    u_short                 wDataWaiting;
    char                    *buffer;
    struct XUDPParcel       *pParcel;
    Timestamp               tto;

    /** Read in the number of bytes in the parcel waiting on the data */
    /** socket */

    streamRead(cb->localclient->fdControl, (char *)&wDataWaiting,
               sizeof(u_short));

    /* Read in time to obsolence of parcel */

    streamRead(cb->localclient->fdControl, (char *) &tto,
               sizeof(Timestamp));

    if ( !(buffer=malloc(wDataWaiting)))
        xudpError("xudp_queuevideo: Allocating RAM");

    /** Read in the parcel's data */

    streamRead(cb->localclient->fdVideo, buffer, wDataWaiting);

    /** Set it up and queue it on the queue of parcels waiting to be */
    /** fragmented and sent */

    pParcel=makeParcel();
```

```
    pParcel->length=wDataWaiting;
    pParcel->options=type;
    pParcel->tto=tto;
    pParcel->buffer=buffer;
    pParcel->pos=buffer;
    addParcel(cb->pOutHead, pParcel);

    return 0;
}
```

7.5.2 `xudp_dequeueparcel` function

`int xudp_dequeueparcel(struct XUDPCB *cb);`

```
int xudp_dequeueparcel(struct XUDPCB *cb)
{
    struct XUDPParcel *pParcel, *pParcelTmp;
    int err;

    /** Traverse list of assembled parcels received from the network */

    pParcel=cb->pInHead->next;
    while (pParcel!=NULL)
    {
        pParcelTmp=pParcel->next;

        /** Hits the earliest parcel first! */
        /** Write the parcel's data to the appropriate socket stream */

        err=write(cb->localclient->fdVideo, pParcel->buffer,
                  pParcel->length);

        /** Send the appropriate command and size over the control */
        /** socket connection */

        xudpSendFunction(cb->localclient, F_RECVVIDEO,
                         (char *)&pParcel->length);


        /** Remove the parcel from the queue, if it was sent ok. */
        /**  If it wasn't, return immediately */

        if (err==pParcel->length)
            deleteParcel(pParcel);
        else return -1;
        pParcel=pParcelTmp;
```

```
    }
    return 0;
}
```

7.5.3 xudp_makecommandparcel function

int xudp_makecommandparcel(struct XUDPCB *cb);

```
int xudp_makecommandparcel(struct XUDPCB *cb)
{
    struct XUDPParcel *pParcel;

    /** Set up a parcel for a RELIABLE, COMMAND payload */

    pParcel=makeParcel();
    pParcel->length=0;
    pParcel->options=COMMAND;
    pParcel->tto=RELIABLE;
    pParcel->buffer=NULL;
    pParcel->pos=NULL;

    /** Queue parcel on the list of parcels waiting to be fragmented */
    /** and sent on the network */

    addParcel(cb->pOutHead, pParcel);
    cb->push=TRUE;

    return 0;
}
```

7.5.4 xudp_fragmentparcel function

int xudp_fragmentparcel(struct XUDPCB *cb);

```
int xudp_fragmentparcel(struct XUDPCB *cb)
{
    static Sequence      sequence=0;
    struct XUDPParcel *pParcel, *pParcelTmp;
    struct XUDPPacket    *pHead, *pPacket;
    struct XUDPCmd       *pCmd, *pCmdTmp;
    u_char               *pPayload;
    u_short              cmds=0;
    u_short              temp;
    u_short              length=0;
```

```
    u_short             datalen=0;

    pHead=cb->pXmitQueue;
    pPacket=NULL;

    pParcel=cb->pOutHead->next;
    pCmd=cb->pCmdQueue->next;

    /** We're only going to fragment parcels to create packets while */
    /** there is room available on the network */
    /***  PARCEL LEVEL: Traverses the list of parcels waiting to be */
    /***                fragmented */

    while ((pParcel!=NULL) &&
           (cb->packetsInPipe < (min(cb->winRemote, cb->winCongestion))
            || cb->push))
    {
        pParcelTmp=pParcel->next;

        /** Test again to make sure we still have room on the net. */
        /** This may create a packet with data from this parcel, with */
        /** data from commands that are queued up to be sent (even a */
        /** zero-length payload with only commands is okay). */
        /*** PACKET LEVEL: Creates a new packet with either commands, */
        /***               payload or both */

        while (((pParcel->length) || (pCmd!=NULL)) &&
               ((cb->packetsInPipe < min(cb->winRemote, cb->winCongestion)) ||
                cb->push))
        {
            cmds=0;
            length=0;
            datalen=0;

            pPacket=makePacket();
            addPacket(pHead, pPacket);  /* Tack this new packet onto the */
                                        /* Xmit queue */

            pPayload=(pPacket->payload)+HEADERSIZE; /* Skip over the space */
                                                    /* where the header goes */

            if (pParcel->buffer==pParcel->pos) sequence=0;
                                  /* if we're at the start of a parcel, reset */
                                  /* packet sequence */

            /* Go through the list of queued up commands, and add up all */
            /* the commands and lengths of arguments */

```

```
            pCmdTmp=pCmd;
            temp=0;
            while (pCmdTmp!=NULL)
            {
                temp+=1+pCmdTmp->length;
                pCmdTmp=pCmdTmp->next;
            }
            /** "temp" now holds the total length of all of the */
            /** commands in the command queue */

            /* Sees if this is going to be the last packet in the parcel, */
            /* if so, queues up a EOP command for it */
            /** Commands actually take precedence to data here, so all */
            /** commands that are queued up when xudp_fragmentparcel */
            /** is called get sent before any data */

            if ((temp+HEADERSIZE+pParcel->length+1)<=cb->mps)
            {
                pCmdTmp=makeCmd();
                pCmdTmp->cmd=C_EOP;
                pCmdTmp->length=0;
                addCmd(cb->pCmdQueue, pCmdTmp);
                pCmd=cb->pCmdQueue->next;
            }

            /** COMMAND LEVEL: Traverse list of commands waiting to be */
            /**                sent, preparing and de-queueing them */
            /**   Add commands to packet until we've reached the end */
            /**   of the list or we've reached the maximum allowable */
            /**   commands in a packet.  If the packet is a COMMAND */
            /**   packet, then there isn't a limit. */

            while ((pCmd!=NULL) && ((cmds<=MAXCMD) ||
                                    (pParcel->options==COMMAND)))
            {
                pCmdTmp=pCmd;

                /** Make sure we have space left in the packet! */

                if (length <= (cb->mps - HEADERSIZE - pCmd->length))
                {
                     *(pPayload++)=pCmd->cmd;
                     if (pCmd->length>0)
                         memcpy(pPayload, pCmd->argbuffer, pCmd->length);
                     pPayload+=pCmd->length;
                     length+=(pCmd->length)+1;
                     cmds++;
                     pCmd=pCmd->next;
```

```
          deleteCmd(pCmdTmp); /* Delete old commands */
        }
        else pCmd=pCmd->next;
      }

      /** Finish setting up the packet headers and stuff */
      /*** "temp" is now the minimum of the maximum remaining */
      /*** payload capacity of this packet and the remaining */
      /*** bytes in the current parcel */

      temp=min((cb->mps - HEADERSIZE - length), pParcel->length);
      length+=temp;
      datalen+=temp;

      /** Copy the payload out of the parcel into the packet */

      if (temp) memcpy(pPayload, pParcel->pos, temp);

      pPacket->header.sequence=sequence;
      pPacket->header.parcel=pParcel->parcel;
      pPacket->header.length=datalen; /* Header length field only */
                                      /* represents the data */
                                      /* payload (no commands) */
      pPacket->header.window=cb->winAdvertised;
      pPacket->header.timestamp=makeTimestamp();
      pPacket->header.options=pParcel->options;
      pPacket->header.tto=pParcel->tto;
      pPacket->header.cmds=cmds;
      pPacket->length=length+HEADERSIZE;  /* Packet length field */
                                          /* represents both */
                                          /* header and payload */

      /** Pop data off the top of the parcel's payload */

      pParcel->length-=datalen;
      pParcel->pos+=datalen;

      if (++sequence==65535) sequence=0; /* Wrap sequence numbers */

      cb->packetsInPipe++;
      cb->push=FALSE;

#ifdef DEBUG2
      show_status(cb);
#endif
    }

    if (pParcel->length==0) deleteParcel(pParcel);
```

```

        pParcel=pParcelTmp;
    }

    return 0;
}
```

7.5.5 xudp_reassembleparcel function

int xudp_reassembleparcel(struct XUDPCB *cb);

```
int xudp_reassembleparcel(struct XUDPCB *cb)
{
    struct XUDPParcel        *pParcel;
    struct XUDPFragParcel    *pFragParcel, *pFragParcelTmp;
    struct XUDPPacket        *pPacket, *pPacketTmp;
    char                     *buffer;

    /** Traverse list of fragmented parcels that have been received */
    /** from the network */

    pFragParcel=cb->pRecvWaitQueue->next;
    while(pFragParcel!=NULL)
    {
        pFragParcelTmp=pFragParcel->next;

        /** Only consider the ones that are complete and have sent an */
        /** acknowledgement to the sender */

        if (pFragParcel->ack)
        {
            /** Only attempt to reassemble something if its not a command */
            /** parcel! */

            if (pFragParcel->headPacket->header.length>0)
            {
                /** Create a new raw parcel to reassemble into */

                pParcel=makeParcel();
                pParcel->options=pFragParcel->options;
                pParcel->tto=pFragParcel->tto;
                pParcel->parcel=pFragParcel->parcel;
                pParcel->length=0;

                /** Traverse list of packets that form our parcel to */
                /** determine the size of buffer to allocate for the raw */
```

```
                /** parcel */

                pPacket=pFragParcel->headPacket;
                while (pPacket!=NULL)
                {
                    pParcel->length+=pPacket->header.length;
                    pPacket=pPacket->next;
                }

                if ( !(pParcel->buffer=malloc(pParcel->length)) )
                    xudpError("xudp_reassembleparcel: Allocating RAM");
                buffer=pParcel->buffer;
                pParcel->pos=buffer;

                /** Traverse the list of packets that form our parcel */
                /** and tack their payloads onto our parcel's buffer */

                pPacket=pFragParcel->headPacket;
                while (pPacket!=NULL)
                {
                    pPacketTmp=pPacket->next;
                    memcpy(buffer, pPacket->payload, pPacket->header.length);
                    buffer+=pPacket->header.length;
                    deletePacket(pPacket);
                    pPacket=pPacketTmp;
                }

                /** Queue up our reassembled RAW parcel on the list of */
                /** parcels waiting to be read by the application */

                addParcel(cb->pInHead, pParcel);
            }

            /** Record that this parcel number has been used */
            cb->usedparcels[pFragParcel->parcel]=TRUE;

            /** Wipe the fragmented version */
            deleteFragParcel(pFragParcel);

#ifdef DEBUG2
            show_status(cb);
#endif
        }
        pFragParcel=pFragParcelTmp;
    }

    return 0;
}
```

7.5.6 `xudp_recvackparcels` function

`int xudp_recvackparcels(struct XUDPCB *cb);`

```
int xudp_recvackparcels(struct XUDPCB *cb)
{
    struct XUDPCmd    *pCmd, *pCmdTmp;
    struct XUDPFragParcel *pFragParcel, *pFragParcelTmp;
    struct XUDPPacket *pPacket, *pPacketTmp;
    Sequence          parcel;
    Boolean           eflag=FALSE, anyacks=FALSE;
    Timestamp         ts;

    pCmd=cb->pCmdInQueue->next;
    while (pCmd!=NULL)
    {
        pCmdTmp=pCmd->next;
        if (pCmd->cmd==C_ACKPARCEL)
        {
            memcpy(&parcel, pCmd->argbuffer, sizeof(Sequence));
            parcel=ntohs(parcel);
            eflag=FALSE;
            pFragParcel=cb->pAckWaitQueue->next;
            while ((pFragParcel!=NULL)&&!eflag)
            {
                pFragParcelTmp=pFragParcel->next;
                if (pFragParcel->parcel==parcel)
                {
                    eflag=TRUE;
                    pPacket=pFragParcel->headPacket;
                    while (pPacket!=NULL)
                    {
                        pPacketTmp=pPacket->next;
                        if (!pPacket->ack)
                        {
                            anyacks=TRUE;
                            if (cb->packetsInPipe>0) cb->packetsInPipe--;
                            ts=makeTimestamp();
                            cb->timeElapsed=ts;
                            cb->bytesACKed=cb->bytesACKed+pPacket->length;
                            cb->rtt=ts-pPacket->header.timestamp;
                            xudp_rttmath(cb);
                        }

                        deletePacket(pPacket);
```

```
                        pPacket=pPacketTmp;
                    }
                    deleteFragParcel(pFragParcel);
                }
                pFragParcel=pFragParcelTmp;
            }
            deleteCmd(pCmd);
#ifdef DEBUG2
            show_status(cb);
#endif
        }

        pCmd=pCmdTmp;
    }
    if (anyacks) xudp_windowchange(cb);

    return 0;
}
```

7.5.7 `xudp_ackparcels` function

`int xudp_ackparcels(struct XUDPCB *cb);`

```
int xudp_ackparcels(struct XUDPCB *cb)
{
    struct XUDPCmd    *pCmd;
    struct XUDPFragParcel *pFragParcel;
    struct XUDPPacket *pPacket;
    Sequence          sequence;
    Boolean           eflag=FALSE;

    /** Traverse list of received fragmented parcels */

    pFragParcel=cb->pRecvWaitQueue->next;
    while (pFragParcel!=NULL)
    {
        /** Continue if an End-Of-Parcel marker has been received (parcel's */
        /** end marker is less than 65535) and parcel hasn't already been */
        /** acknowledged. */

        if ((pFragParcel->end<65535) && (!pFragParcel->ack))
        {
            sequence=0;

            /** Traverse list of packets that form our fragmented parcel, */
            /** exiting by setting "eflag" if we find an hole in the */
```

```
/** parcel, or by reaching the end of the fragments */
/** successfully */

pPacket=pFragParcel->headPacket;
eflag=FALSE;
while (!eflag&&(pPacket!=NULL)&&(sequence!=pFragParcel->end))
{
    if (pPacket->header.sequence==sequence)
    {
        sequence++;
        pPacket=pPacket->next;
    }
    else eflag=TRUE;
}

/** Do this if we actually have the entire parcel */

if (!eflag && (pPacket!=NULL) && (sequence==pFragParcel->end))
{
    if (pPacket->header.sequence==sequence)
    {
        pCmd=makeCmd();

        /** Acknowledge as either a regular parcel or a */
        /** command only parcel */

        if (pPacket->header.length==0)
            pCmd->cmd=C_ACKCMDPCL;
        else pCmd->cmd=C_ACKPARCEL;

        /** Queue up an ACKPARCEL for this parcel and mark it */
        /** as such. */

        pCmd->length=sizeof(Sequence);
        sequence=pFragParcel->parcel;
        sequence=htons(sequence);
        memcpy(pCmd->argbuffer, &sequence, sizeof(Sequence));
        addCmd(cb->pCmdQueue, pCmd);
        pFragParcel->ack=TRUE;

        /** Traverse through the packets that form this parcel */
        /** and adjust the remote packetsInPipe size to reflect */
        /** the ones that haven't been individually ACKed. */

        pPacket=pFragParcel->headPacket;
        while (pPacket!=NULL)
        {
            if (!(pPacket->ack))
```

```
                            if (cb->packetsInPipeRmt>0)
                                cb->packetsInPipeRmt--;
                        pPacket=pPacket->next;
                    }
                }
            }
#ifdef DEBUG2
            show_status(cb);
#endif
        }

        pFragParcel=pFragParcel->next;
    }

    return 0;
}
```

7.5.8 xudp_endparcels function

`int xudp_endparcel(struct XUDPCB *cb);`

```
int xudp_endparcel(struct XUDPCB *cb)
{
    struct XUDPFragParcel *pFragParcel;
    struct XUDPCmd        *pCmd, *pCmdTmp;
    Sequence              sequence;
    Boolean               eflag;


    /*** Scan through the incoming commands queue for an End Of Parcel. */
    /*** If we find it, locate the appropriate parcel in the Receive */
    /*** Wait Queue and mark its ending packet. */

    /** Traverse the list of commands that have been dequeued from */
    /** packets received from the network */

    pCmd=cb->pCmdInQueue->next;
    while (pCmd!=NULL)
    {
        pCmdTmp=pCmd->next;

        /** Go until we find an End Of Parcel command */

        if (pCmd->cmd==C_EOP)
        {
            memcpy(&sequence, pCmd->argbuffer, sizeof(Sequence));
```

```

            /** Traverse the list of parcels that are in the process of */
            /** receiving packets from the network */

            eflag=FALSE;
            pFragParcel=cb->pRecvWaitQueue->next;
            while ((pFragParcel!=NULL)&&!eflag)
            {
                /** If we find a parcel that matches the one we have an */
                /** EOP command for, or we reach the end, exit the loop. */

                if (pFragParcel->parcel!=sequence)
                    pFragParcel=pFragParcel->next;
                else eflag=TRUE;
            }

            /** Set the end packet sequence number on the parcel */

            if (pFragParcel!=NULL)
                memcpy(&pFragParcel->end,
                       pCmd->argbuffer+sizeof(Sequence),
                       sizeof(Sequence));

            /** De-queue the command, as it was either used or refers */
            /** to an erroneous parcel that we don't have any data for */

            deleteCmd(pCmd);
        }
        pCmd=pCmdTmp;
    }
    return 0;
}
```

7.5.9 xudp_removeobsolete function

int xudp_removeobsolete(struct XUDPCB *cb);

```
int xudp_removeobsolete(struct XUDPCB *cb)
{
    struct XUDPParcel     *pParcel, *pParcelTmp;
    struct XUDPFragParcel *pFragParcel, *pFragParcelTmp;
    struct XUDPPacket     *pPacket;
    Timestamp             ts;
    int                   count;

    /** Get the current time */
```

```
ts=makeTimestamp();

/** Add 1/2 the recent average RTT to it (the delivery time) */
ts+=(int)(0.5*cb->rttavg);

/** Traverse through the outgoing raw parcel queue */

pParcel=cb->pOutHead->next;
while (pParcel!=NULL)
{
    pParcelTmp=pParcel->next;

    /** Test if we've exceeded the "sell by" time on this parcel, if */
    /** we have, delete it. */

    if (pParcel->tto<=(ts-pParcel->timestamp))
        deleteParcel(pParcel);

    pParcel=pParcelTmp;
}

/** Traverse the Outgoing ACK wait queue */

pFragParcel=cb->pAckWaitQueue->next;
while (pFragParcel!=NULL)
{
    pFragParcelTmp=pFragParcel->next;

    /** Test to see if we've exceeded the "sell by" time on this */
    /** parcel */

    if (pFragParcel->tto<=(ts-pFragParcel->timestamp))
    {
        /** Traverse the list of packets that form the fragmented */
        /** parcel */

        count=0;
        pPacket=pFragParcel->headPacket;
        while (pPacket!=NULL)
        {
            /** Keep the accounting straight by incrementing a */
            /** counter variable every time we find a packet that */
            /** hasn't been acknowledged.  (i.e. we still think */
            /** its in the network */

            if (!(pPacket->ack)) count++;

            pPacket=pPacket->next;
```

```
            }

            /** Delete the parcel */

            deleteFragParcel(pFragParcel);

            /** Realize that these packets can't be in the network */
            /** anymore, and that they have lived out their */
            /** usefullness.  Remove them from the accounting stuff */

            cb->packetsInPipe-=count;
        }
        pFragParcel=pFragParcelTmp;
    }

    /** Traverse the incoming queue of received fragmented parcels */

    pFragParcel=cb->pRecvWaitQueue->next;
    while (pFragParcel!=NULL)
    {
        pFragParcelTmp=pFragParcel->next;

        /** Test to see if we've exceeded the "sell by" time on this */
        /** parcel, if we have, delete it now. */

        if (pFragParcel->tto<=(ts-pFragParcel->timestamp))
            deleteFragParcel(pFragParcel);

        pFragParcel=pFragParcelTmp;
    }

    return 0;
}
```

7.6 Packet Sublayer

The packet sublayer in XUDP is responsible for bridging the gap between the parcel layer and the network layer. It is at this level that all flow control is accomplished and from where most of the CPU load will come. Duties include sending and receiving packets, retransmitting packets, acknowledging packets, adjusting round trip time statistics, dynamic congestion window sizing and insuring in-order delivery of packets on a per-parcel basis.

7.6.1 `xudp_sendmsg` function

`xudp_sendmsg` attempts to transmit a packet to the next lower layer, currently via `send` through a UDP socket. Verifies quickly that the socket is available for writing, sends the packet and increments the total number of packets in the network. Command syntax is:

`int xudp_sendmsg(struct XUDPCB *cb, struct XUDPPacket *pPacket);`

- `cb` is a pointer to the control block.
- `pPacket` is a pointer to a `struct XUDPPacket` that is the packet to be sent.

```
1  int xudp_sendmsg(struct XUDPCB *cb, struct XUDPPacket *pPacket)
2  {
3      fd_set              setFD;
4      struct timeval      timestuff;
5
6      /** Some UNIX variants may require that the socket be checked for */
7      /** writeability IMMEDIATELY before writing to it.  Here, select() */
8      /** is used to conduct a one-shot poll of the socket */
9
10     timestuff.tv_sec=0;
11     timestuff.tv_usec=0;
12     FD_ZERO(&setFD);
13     FD_SET(cb->fdCli, &setFD);
14     if (select((cb->fdCli+1), NULL, &setFD, NULL, &timestuff)<0)
15         xudpError("xudp_sendmsg");
16     if (FD_ISSET(cb->fdCli, &setFD))
17     {
```

Top of `xudp_sendmsg` function.

Lines 10-16 perform a poll using `select` to test whether the socket file descriptor is available for writing. Line 23 invokes `send` to write the payload of this packet to the socket. In this case, the `connect` function was previously used to inform the sockets layer of the endpoint's address, thereby allowing the use of `send` rather than `sendto`, even though a UDP socket is inherently connectionless.

```
18         /** Send the packet to the network.  We're able to use send */
19         /** because we connect()ed the UDP sockets earlier, send() */
20         /** wraps around sendto(), but stores the connected endpoint's */
21         /** address to make it easy on us */
22
23         if (send(cb->fdCli, pPacket->payload, pPacket->length, 0)<0)
24         {
25             /** We've hit an error sending the packet. */
26             /** If the error is acceptable, just return a fail code, */
27
28             return -1;
```

```
        }

        cb->bytesSent+=pPacket->length;

    }
    else return -1;

    return 0;
}
```

Bottom of **xudp_sendmsg** function.

7.6.2 `xudp_recvmsg` function

`xudp_recvmsg` attempts to receive a packet from the next lower layer, currently via `recv` from a UDP socket. Function syntax is:

`int xudp_recvmsg(struct XUDPCB *cb);`

- `cb` is a pointer to the control block.

Operation of the function is similar to **xudp_sendmsg**, lines 12-19 testing if data is available to be read on the socket and line 25 reading in the payload with `recv`. Due to the prior `connect` function call, sockets automatically discards incoming packets from an address that differs from the set endpoint, and it is not important to use `recvfrom`.

Line 40 increments the count of the number of packets that it has received from the network, interpreted here as the number of packets XUDP believes have been placed in the network by the foreign host as of this time. `packetsInPipeRmt` can be used in determining whether to send an acknowledgement upon this value reaching the size of the local host's advertised window.

```
 1 struct XUDPPacket *xudp_recvmsg(struct XUDPCB *cb)
 2 {
 3     struct XUDPPacket  *pPacket;
 4     int                result;
 5     fd_set             setFD;
 6     struct timeval     timestuff;
 7
 8     /** Some UNIX variants may require that the socket be checked for */
 9     /** readability IMMEDIATELY before reading from it.  Here, select() */
10     /** is used to conduct a one-shot poll of the socket */
11
12     pPacket=NULL;
13     timestuff.tv_sec=0;
14     timestuff.tv_usec=0;
15     FD_ZERO(&setFD);
16     FD_SET(cb->fdCli, &setFD);
17     if (select((cb->fdCli+1), &setFD, NULL, NULL, &timestuff)<0)
```

```
        xudpError("xudp_recvmsg");
       if (FD_ISSET(cb->fdCli, &setFD))
       {
        /** Create an empty packet and recv() from the network into the */
        /** payload.  See xudp_sendmsg() for a discussion on connect() */

        pPacket=makePacket();
        if ((result=recv(cb->fdCli, pPacket->payload, cb->mps, 0))<=0)
        {
            /** We received nothing.  Delete the packet we just */
            /** made. */

            deletePacket(pPacket);
            pPacket=NULL;
        }
        else
        {
            /** Set up our packet's length field and increase our */
            /** perceived number of packets in the network from the */
            /** sending host */

            pPacket->length=result;
            cb->packetsInPipeRmt++;
        }

       }

       return pPacket;
}
```

xudp_recvmsg function.

7.6.3 xudp_queueackwait function

xudp_queueackwait queues up a recently sent packet on the ordered list of packets that comprise this packet's fragmented parcel where it will wait for an acknowledgement from the receiving end. Its syntax is:

```
int xudp_queueackwait(struct XUDPCB *cb, struct XUDPPacket *pPacket);
```

- cb is a pointer to the control block.
- pPacket is the packet to queue on the acknowledgement wait queue.

```
int xudp_queueackwait(struct XUDPCB *cb, struct XUDPPacket *pPacket)
{
```

```
3      struct XUDPFragParcel *pFragParcel;
4      struct XUDPPacket     *pPacketTmp;
5      Boolean               eflag=FALSE;
6
7      if (pPacket->next!=NULL)
8       pPacket->next->prev=pPacket->prev;   /* Seperate packet from list */
9      if (pPacket->prev!=NULL)
10      pPacket->prev->next=pPacket->next;   /* Without deleting it */
11     pPacket->next=NULL;
12     pPacket->prev=NULL;
13
14     /** Traverse list of parcels that have been wholly or partially */
15     /** sent to the network and are waiting for acknowledgements */
16
17     pFragParcel=cb->pAckWaitQueue->next;
18     while ((pFragParcel!=NULL)&&!eflag)
19     {
20      /** Descend list until we reach the end or the correct parcel */
21
22      if (pFragParcel->parcel!=pPacket->header.parcel)
23          pFragParcel=pFragParcel->next;
24
25      else eflag=TRUE;
26     }
```

Top of `xudp_queueackwait` function.

As it is likely that `pPacket` could be a member in a doubly-linked list, lines 7-12 prune the packet out of the list, redirecting the pointers of the previous and next nodes.

Lines 17-26 traverse through the list of packets waiting to receive acknowledgements, `cb->pAckWaitQueue` until the end of the list is reached (indicating this is the first packet in a new parcel) or the correct parcel is reached.

> POTENTIAL OPTIMIZATION: rather than traversing through linked lists as is the case here, creation of hash tables might reduce not only CPU load, but stack times as well!

If the end of the list was reached, lines 34-39 create a new fragmented parcel list taking relevant information from the packet's header. The packet is set at the head of the fragmented parcel's list of packets. Otherwise, the correct fragmented parcel was located and `pFragParcel` is left pointing at it.

Lines 47-57 verify that the packet is not already in the queue, as would happen with a packet that has been retransmitted. Line 67 deletes the packet under this condition. Line 62 adds the packet to the end of the queue, if it isn't in it already. Since this is the sending side, we are assured that packets are in order, as they are being sent that way, so no insertion mechanism is necessary.

```
28     if (pFragParcel==NULL)  /* New parcel */
29     {
```

```
     /** There weren't any references to this parcel number in the */
     /** ACK Wait list, so we'll create a new parcel and tack it */
     /** onto the end of the list */

     pFragParcel=makeFragParcel();
     pFragParcel->parcel=pPacket->header.parcel;
     pFragParcel->options=pPacket->header.options;
     pFragParcel->tto=pPacket->header.tto;
     pFragParcel->headPacket=pPacket;
     addFragParcel(cb->pAckWaitQueue, pFragParcel);
    }
    else
    {
     /** Make sure the packet isn't already in the queue, (i.e. its a */
     /** retransmit) */

     /** Traverse the list of packets that form the parcel */
     pPacketTmp=pFragParcel->headPacket;
     eflag=FALSE;
     while ((pPacketTmp!=NULL) && !eflag)
     {
         /** If the sequence number of this packet matches ours, jump out */
         /** immediately, setting eflag */

         if (pPacketTmp->header.sequence==pPacket->header.sequence)
             eflag=TRUE;
         else pPacketTmp=pPacketTmp->next;
     }

     if (!eflag)
     {
         /** Add the packet to the end of the queue. */
         addPacket(pFragParcel->headPacket, pPacket);
     }
     else
     {
         /** Packet was a retransmit copy, blow it away. */
         deletePacket(pPacket);
     }
    }

    return 0;
}
```

Bottom of `xudp_queueackwait` function.

7.6.4 `xudp_organizeincoming` function

`xudp_organizeincoming` queues up incoming packets on their appropriate fragmented parcel's packet queue, insuring in-order delivery.

```
int xudp_organizeincoming(struct XUDPCB *cb);
```

- `cb` is a pointer to the control block.

The first fifty lines of code are nearly identical to those in `xudp_queueackwait`: pruning the packet out of the incoming queue, locating its parcel on the receive queue, creating a new parcel when the first packet in that parcel arrives. This purposefully mimics the sending side operation on the receiving side. If this was in fact a new parcel creation, this function stops there.

```
int xudp_organizeincoming(struct XUDPCB *cb)
{
    struct XUDPFragParcel *pFragParcel;
    struct XUDPPacket *pPacket, *pPacketTmp, *pPacketSearch;
    Boolean           eflag=FALSE;

    /** Traverse the list of packets received from the network but not */
    /** ordered and placed under fragmented parcels */

    pPacket=cb->pRecvQueue->next;
    while (pPacket!=NULL)
    {
     pPacketTmp=pPacket->next;

     /** Seperate the packet from the list without deleting it */

     if (pPacket->next!=NULL)
         pPacket->next->prev=pPacket->prev;
     if (pPacket->prev!=NULL)
         pPacket->prev->next=pPacket->next;
     pPacket->next=NULL;
     pPacket->prev=NULL;

     /** Traverse the list of parcels coming in from the network */

     pFragParcel=cb->pRecvWaitQueue->next;
     eflag=FALSE;
     while ((pFragParcel!=NULL)&&!eflag)
     {
         /** Descend list until we reach the end or the correct */
         /** parcel */

         if (pFragParcel->parcel!=pPacket->header.parcel)
             pFragParcel=pFragParcel->next;
```

```
35          else eflag=TRUE;
36      }
37      if (pFragParcel==NULL)  /* New parcel */
38      {
39          /** No reference was made to this parcel number in the */
40          /** list, so we're going to make a new parcel and tack */
41          /** it on the end of the list */
42
43          pFragParcel=makeFragParcel();
44          pFragParcel->parcel=pPacket->header.parcel;
45          pFragParcel->options=pPacket->header.options;
46          pFragParcel->tto=pPacket->header.tto;
47          pFragParcel->headPacket=pPacket;
48          addFragParcel(cb->pRecvWaitQueue, pFragParcel);
49      }
```

Top of `xudp_organizeincoming` function.

The code below is reached when the correct parcel is found. This is XUDP's in-order delivery mechanism. Lines 58-70 form a search loop that exits under two conditions: first, if the loop arrives at the end of the queue, second, if the packet currently being placed has a sequence number lesser than the packet the algorithm is examining in the list. The second condition is sufficient to insure that the list orders itself as it is being built.

```
50      else
51      {
52          /** Search list of packets in parcel and insert in the */
53          /** appropriate place, ordering the packets by */
54          /** ascending sequence number */
55
56          /*** Traverse the list of packets that form this parcel */
57
58          pPacketSearch=pFragParcel->headPacket;
59          eflag=FALSE;
60          while ((pPacketSearch!=NULL)&&!eflag)
61          {
62              /** Bail out if the sequence number of our packet */
63              /** is ever less than that of the current list */
64              /** entry or we reach the end of the list */
65
66              if (pPacketSearch->header.sequence <
67                  pPacket->header.sequence)
68                  pPacketSearch=pPacketSearch->next;
69              else eflag=TRUE;
70          }
```

Lines 50-70 of the `xudp_organizeincoming` function.

Lines 71-77 handle the case when the end of the list of packets is reached and the new packet's sequence number is still higher - the packet should fall at the end of the current queue. Line 76 adds the packet to the end of the queue.

Two cases remain and are handled in lines 83-95. If the packet's sequence number numerically places it at the head of the list, line 93 takes the entire list and adds it to the back of the new packet, then line 94 sets the packet as the head of the list. Lines 83-84 remain to insert the packet between its lower and upper sequence number neighbors.

```
71          if (pPacketSearch==NULL)
72          {
73              /** We're at the end of the list, so just put the */
74              /** packet there */
75
76              addPacket(pFragParcel->headPacket, pPacket);
77          }
78          else
79          {
80              /** We need to insert the packet between this list */
81              /** member and the previous */
82
83              if (pPacketSearch->prev!=NULL)
84                  insertPacket(pPacketSearch->prev, pPacket);
85              else
86              {
87                  /** This packet comes before every other one */
88                  /** Insert the packet at the beginning of the */
89                  /** list Add current list of packets in parcel */
90                  /** to the end of our single packet and set */
91                  /** parcel to start on that packet. */
92
93                  addPacket(pPacket, pFragParcel->headPacket);
94                  pFragParcel->headPacket=pPacket;
95              }
96
97          }
98      }
99      pPacket=pPacketTmp;
100     }
101
102     return 0;
103 }
```

Bottom of the`xudp_organizeincoming` function.

7.6.5 xudp_xmit function

`xudp_xmit` dequeues packets waiting in the outgoing packet queue, assembles their transmittable headers in the packet payload and transmits the packets through `xudp_sendmsg`.

```
int xudp_xmit(struct XUDPCB *cb);
```

- `cb` is a pointer to the control block.

`xudp_xmit` Acts primarily as an interface to the `xudp_send` function. Lines 10-13 instruct the loop to grab packets from the list of packets waiting to be sent to the network. Lines 19-42 prepare the header for transport in the payload, creating the header exactly as seen in figure 7.1.

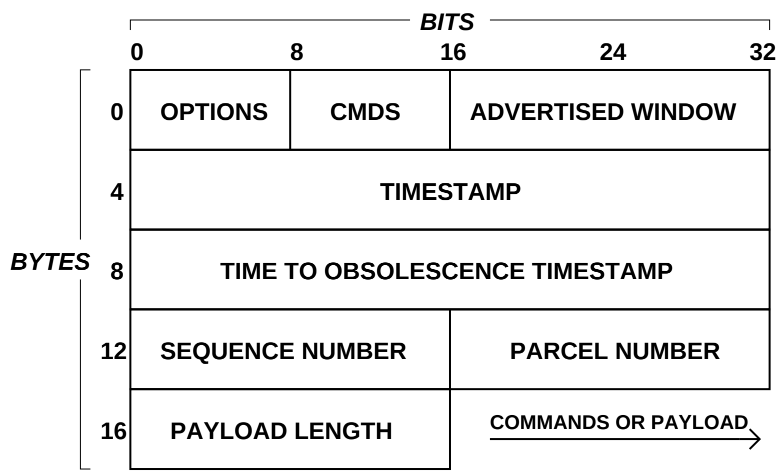

Figure 7.1: Transmitted XUDP Header

As evidenced by the comments on lines 15-17, I'm still doubtful that this is the best way to generate a header of this sort. But, with byte ordering and alignment varying from machine to machine, this may in fact be the only way. This area needs optimization.

```
 1 int xudp_xmit(struct XUDPCB *cb)
 2 {
 3     struct XUDPPacket *pPacket, *pPacketTmp;
 4     char              *payload;
 5     u_short           temp;
 6     u_int             tempint;
 7
 8     /** Traverse the list of packets waiting to be transmitted */
 9
10     pPacket=cb->pXmitQueue->next;
11     while (pPacket!=NULL)
12     {
13      pPacketTmp=pPacket->next;
```

```
14
15      /** WARNING!!!!! NOTE, THIS is SLOW, dumb code. */
16      /** Although it looks like it, this will not work for */
17      /** different typesizes than the default! */
18
19      payload=pPacket->payload;
20      *(payload++)=pPacket->header.options;
21      *(payload++)=pPacket->header.cmds;
22      temp=htons(pPacket->header.window);
23      memcpy(payload, &temp, 2);
24      payload+=2;
25
26      tempint=htonl(pPacket->header.timestamp);
27      memcpy(payload, &tempint, 4);
28      payload+=4;
29      tempint=htonl(pPacket->header.tto);
30      memcpy(payload, &tempint, 4);
31      payload+=4;
32
33      temp=htons(pPacket->header.sequence);
34      memcpy(payload, &temp, 2);
35      payload+=2;
36      temp=htons(pPacket->header.parcel);
37      memcpy(payload, &temp, 2);
38      payload+=2;
39
40      temp=htons(pPacket->header.length);
41      memcpy(payload, &temp, 2);
42      payload+=2;
43
```

Top of `xudp_xmit` function.

The remaining portion of `xudp_xmit` handles sending the packet (line 46) and subsequently queueing it on the list of packets waiting to receive acknowledgements with `xudp_queueackwait`. It will not attempt to queue a packet with a time to obsolescence of zero, as that packet is a one shot and can never be retransmitted.

```
44      /** Send the packet */
45
46      if (xudp_sendmsg(cb, pPacket)>=0)
47      {
48          if (pPacket->header.tto)
49          {
50              /** Queue up the packet to wait for an acknowledgement */
51
```

```
            xudp_queueackwait(cb, pPacket);
            pPacket=pPacketTmp;
        }
        else
        {
            /** Delete packet if it's a ONESHOT */

            deletePacket(pPacket);
            pPacket=pPacketTmp;
        }

     } else pPacket=NULL;

    }
    return 0;
}
```

Bottom of `xudp_xmit` function.

7.6.6 `xudp_recv` function

`xudp_recv` receives a packet incoming from the network, making use of the `xudp_recvmsg` function, and fills out a `struct XUDPPacket` with the appropriate payload and header information, insuring that the packet delivered to the XUDP protocol stack is in the correct form.

```
int xudp_recv(struct XUDPCB *cb);
```

- `cb` is a pointer to the control block.

The function is necessarily complicated due to the undetermined length of the optional commands present in the packet header as well as the chance that the packet received could be a retransmitted duplicate. Below, the functionality is enumerated:

1. Receive a packet from the network with `xudp_recvmsg`
2. Discard duplicate packets, detecting them with one of three ways:
   (a) Locating the packet sequence number on the correct parcel on the queue of received parcels.
   (b) Detecting that the packet belongs to a parcel that XUDP has already reassembled by searching the list of used parcels.
   (c) Finding the packet in the queue of received packets waiting to be placed in parcels.
3. Set the size of the receiver's advertised window in the control block.
4. Process commands out of the header's payload and queue them on the incoming command queue.
5. Move the packet's data payload back to the start of the payload buffer.

6. Queue the packet on the list of received packets, increasing the number of packets that we perceive to exist in the network as sent by the remote host.

```
int xudp_recv(struct XUDPCB *cb)
{
    struct XUDPFragParcel *pFragParcel;
    struct XUDPPacket     *pPacket, *pPacketTmp;
    struct XUDPCmd        *pCmd;
    char                  *payload, cmd;
    u_short               temp;
    u_int                 tempint;
    int                   i=0;
    Boolean               eflag;

    /** Attempt to receive packets from the network.  Continue doing so */
    /** until unsuccessful. */

    while ((pPacket=xudp_recvmsg(cb))!=NULL)
    {
     i++;

     /** WARNING!!!! */
     /** Although it looks like it, this will not work for */
     /** different typesizes than the default! */

     payload=pPacket->payload;
     pPacket->header.options=*(payload++);
     pPacket->header.cmds=*(payload++);
     memcpy(&temp, payload, 2);
     pPacket->header.window=ntohs(temp);

     payload+=2;
     memcpy(&tempint, payload, 4);
     pPacket->header.timestamp=ntohl(tempint);
     payload+=4;
     memcpy(&tempint, payload, 4);
     pPacket->header.tto=ntohl(tempint);

     payload+=4;
     memcpy(&temp, payload, 2);
     pPacket->header.sequence=ntohs(temp);
     payload+=2;
     memcpy(&temp, payload, 2);
     pPacket->header.parcel=ntohs(temp);

     payload+=2;
```

```
44      memcpy(&temp, payload, 2);
45      pPacket->header.length=ntohs(temp);
46
47      payload+=2;
```

Top of `xudp_recv` function, reassembling the packet's header structure.

Lines 1-47 receive a packet from the network with `xudp_recvmsg` and move through the packet payload, reassembling the header one element at a time, accounting for network-byte-ordered integers.

Below, lines 48-79 constitute part 1 of the duplication detection process, examining the queue of received fragmented parcels. The searching algorithm consists of two nested `while` loops: an external loop that traverses through the queue of parcels until either the parcel that matches this packet's parcel is found, or the end of the queue is reached and an internal loop that descends the list of packets in that parcel, exiting upon detecting a matching packet sequence number or reaching the end of the list.

If at any point in this routine or the following routines, `eflag` gets set, the packet has been determined to be a duplicate.

```
48      /*** Find out NOW if we've received a duplicate packet.  We've */
49      /*** got to go through both the immediate packet-level receive */
50      /*** queue as well as the fragmented parvel recv-wait queue */
51
52      /** Traverse list of received, fragmented parcels */
53
54      pFragParcel=cb->pRecvWaitQueue->next;
55      eflag=FALSE;
56      while ((pFragParcel!=NULL)&&(!eflag))
57      {
58          /** See if we're on the right parcel */
59
60          if (pFragParcel->parcel==pPacket->header.parcel)
61          {
62              /** Traverse list of packets that form the fragmented */
63              /** parcel */
64
65              pPacketTmp=pFragParcel->headPacket;
66              while ((pPacketTmp!=NULL)&&(!eflag))
67              {
68                  /** If we find a match for the sequence number of */
69                  /** the packet, mark eflag as true (we have a */
70                  /** dupe) */
71
72                  if (pPacket->header.sequence ==
73                      pPacketTmp->header.sequence) eflag=TRUE;
74                  pPacketTmp=pPacketTmp->next;
```

```
75              }
76
77          }
78          pFragParcel=pFragParcel->next;
79      }
```

`xudp_recv` function, detecting duplicate packets in the received parcel queue.

Lines 80-84 handle the second characteristic of a duplicate packet, one that is a member of a parcel that has already been completely received and reassembled, and therefore isn't present in the queue of received, fragmented parcels. `usedparcels[]` is a simple `Boolean` array that is updated in `xudp_reassembleparcel` by setting the array element that is equal to the parcel's sequence number to a nonzero value.

```
80
81      /** If the packet belongs to a parcel that we've already */
82      /** reassembled, then it's a delayed duplicate and we'll get rid */
83      /** of it */
84      if (cb->usedparcels[pPacket->header.parcel]) eflag=TRUE;
```

`xudp_recv` function, detecting duplicate packet by virtue of its parcel having been entirely received and reassembled.

The section below (85-104) detects duplicate packets in the queue of received packets. It follows a similar format to before, setting `eflag` if it matches both packet sequence numbers and parcel sequence numbers.

```
85
86      /** Do unless we already have detected a duplicate packet */
87
88      if (!eflag)
89      {
90          /** Traverse list of received packets */
91
92          pPacketTmp=cb->pRecvQueue->next;
93          while ((pPacketTmp!=NULL) && (!eflag))
94          {
95              /** If both the parcel and sequence numbers match, we */
96              /** have a dupe. */
97              if ((pPacketTmp->header.parcel ==
98                    pPacket->header.parcel) &&
99                   (pPacketTmp->header.sequence ==
100                   pPacket->header.sequence)) eflag=TRUE;
101             pPacketTmp=pPacketTmp->next;
102         }
103
104     }
```

`xudp_recv` function, detecting duplicates in the received packets queue.

Lines 105-179 wrap up the `xudp_recv` function, primarily stripping the variable length commands out of the payload unless the packet has been determined to be a duplicate (`eflag` is set) line 108 deletes it and the rest of the function is skipped.

First, the receiver's advertised window is updated to that contained in the packet header (line 114.)

The `while` loop from line 118 to line 166 executes while commands remain to be processed in the payload immediately following the packet's header. Figure 7.2 displays the organization of the commands stored in the payload; the size and number of arguments to each command depend on the command itself. The loop uses the `payload` character pointer to maintain the current position in the payload buffer.

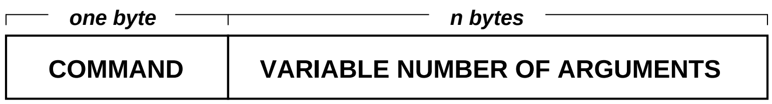

Figure 7.2: Layout of a post-header command stored in a packet's payload.

Currently, five commands are defined, table 7.1 contains the command names as well as the size of the argument buffer.

Command	Arguments
C_ACK	4 bytes
C_EOP	no arguments
C_ACKPARCEL	2 bytes
C_ACKCMDPCL	2 bytes
C_CLOSE	no arguments

Table 7.1: Inter-packet commands and corresponding argument buffer sizes.

The processed commands are stored in `struct XUDPCmd` structures, as described in chapter 7.2. Line 164 adds the command onto the end of the incoming command queue.

Once the packet has been appropriately handled, line 173 adds the packet to the end of the queue of received packets (`cb->pRecvQueue`) and line 174 increases our the number of packets we perceive that the remote host has sent to the network.

```
105     /** Delete the packet if it is a duplicate or else continue */
106     /** normal unwrapping. */
107
108     if (eflag) deletePacket(pPacket);
109     else
110     {
111         /** Set the control block's Remote Window size to the */
112         /** Advertised Window of this packet */
```

```
        cb->winRemote=pPacket->header.window;

        /** Strip all of the commands out of the header */

        while(pPacket->header.cmds)
        {
            /** We're using "payload" to point at the current */
            /** position in the payload of the packet */

            cmd=*payload;
            payload++;
            pCmd=makeCmd();
            pCmd->cmd=cmd;

            switch(cmd)
            {
            case C_ACK:
                pCmd->length=sizeof(Sequence)*2;
                memcpy(pCmd->argbuffer, payload, pCmd->length);
                payload+=pCmd->length;
                break;
            case C_ACKPARCEL:
                pCmd->length=sizeof(Sequence);
                memcpy(pCmd->argbuffer, payload, pCmd->length);
                payload+=pCmd->length;
                break;
            case C_EOP:
                pCmd->length=sizeof(Sequence)*2;
                memcpy(pCmd->argbuffer, &(pPacket->header.parcel),
                       sizeof(Sequence));
                memcpy(pCmd->argbuffer+sizeof(Sequence),
                       &(pPacket->header.sequence), sizeof(Sequence));
                break;
            case C_ACKCMDPCL:
                pCmd->cmd=C_ACKPARCEL;  /* Treat as if regular parcel */
                pCmd->length=sizeof(Sequence);
                memcpy(pCmd->argbuffer, payload, pCmd->length);
                payload+=pCmd->length;
                break;
            case C_CLOSE:
                deleteCmd(pCmd);
                pCmd=NULL;
                cb->state=XUDP_CLOSE_WAIT;
                cb->closing=makeTimestamp();
                break;
            default: break;
            }
```

```
            /** Queue the commands on the list of incoming commands */

            if (pCmd!=NULL) addCmd(cb->pCmdInQueue, pCmd);
            pPacket->header.cmds--;
        }

        if (pPacket->header.length)
            memmove(pPacket->payload, payload, pPacket->header.length);

        /** Queue packet on list of received packets */

        addPacket(cb->pRecvQueue, pPacket);
        cb->packetsInPipeRmt++;

    }
  }
  return i;
}
```

Bottom of `xudp_recv` function.

7.6.7 `xudp_recvackpackets` function

`xudp_recvackpackets` wades through the queue of incoming commands and interprets *C_ACK* commands - received acknowledgements of packets sent by the local host. It sets `ack` on the acknowledged packets waiting on the queue of fragmented parcels waiting for acknowledgements.

`int xudp_recvackpackets(struct XUDPCB *cb);`

- `cb` is a pointer to the control block.

The function consists of a `while` loop that searches through the list of incoming commands (`pCmdInQueue`) invoking the code within the `if` statment on line 18 when a command is a *C_ACK*.

Lines 20-24 parse the argument buffer as two network-byte-ordered 16-bit unsigned integers; the first is the number of the parcel, the second the packet sequence number.

Lines 26-33 constitute a loop structure that attempts to locate the fragmented parcel in `pAckWaitQueue` with a matching parcel sequence number. The loop exits with `pFragParcel` pointing to the parcel and `eflag` set, or `pFragParcel = NULL`, indicating that the end was reached without a match.

```
 1 int xudp_recvackpackets(struct XUDPCB *cb)
 2 {
 3     struct XUDPCmd    *pCmd, *pCmdTmp;
 4     struct XUDPFragParcel *pFragParcel;
 5     struct XUDPPacket *pPacket;
 6     Sequence          sequence, parcel, temp;
 7     Timestamp         ts, maxrtt=0;
```

```
8     Boolean           eflag=FALSE, anyacks=FALSE;
9
10
11    /** Traverse list of incoming commands */
12
13    pCmd=cb->pCmdInQueue->next;
14    while (pCmd!=NULL)
15    {
16        pCmdTmp=pCmd->next;
17
18        if (pCmd->cmd==C_ACK)
19        {
20            memcpy(&temp, pCmd->argbuffer, sizeof(Sequence));
21            parcel=ntohs(temp);
22            memcpy(&temp, (pCmd->argbuffer)+sizeof(Sequence),
23                   sizeof(Sequence));
24            sequence=ntohs(temp);
25
26            pFragParcel=cb->pAckWaitQueue->next;
27            eflag=FALSE;
28            while((pFragParcel!=NULL)&&!eflag)
29            {
30                if (pFragParcel->parcel!=parcel)
31                    pFragParcel=pFragParcel->next;
32                else eflag=TRUE;
33            }
```

Top of the `xudp_recvackpackets` function.

Lines 35-58 are executed if `pFragParcel` is non-null, pointing to the matched fragmented parcel. Lines 36-43 attempt to locate within the queue of packets that comprise the fragmented parcel a packet that matches the acknowledged packet sequence number. On success, `pPacket` points at the packet and `eflag` is set; on failure, `pPacket = NULL`.

Lines 47-55 are executed if the acknowledged packet is indeed found. The following tasks are accomplished:

- Line 48 sets the `ack` flag.
- Line 49 removes the acknowledgement command from the incoming command queue.
- The number of packets placed in the network by the local host is decreased by one on line 50, keeping our books straight.
- The number of bytes acknowledged is updated on line 51 with the total length (header+payload) of this packet.
- (lines 52-54) the round trip time for this packet is determined by subtracting the send time of the packet from the current time.

- `xudp_rttmath` is called to update the statistics based on this packet's round trip time.

```
            if (pFragParcel!=NULL)
            {
                pPacket=pFragParcel->headPacket;
                eflag=FALSE;
                while((pPacket!=NULL)&&!eflag)
                {
                    if (pPacket->header.sequence!=sequence)
                        pPacket=pPacket->next;
                    else eflag=TRUE;
                }

                if (pPacket!=NULL)
                {
                    anyacks=TRUE;
                    pPacket->ack=TRUE;
                    deleteCmd(pCmd);
                    if (cb->packetsInPipe>0) cb->packetsInPipe--;
                    cb->bytesACKed=cb->bytesACKed+pPacket->length;
                    ts=makeTimestamp();
                    cb->timeElapsed=ts;
                    cb->rtt=ts-pPacket->header.timestamp;
                    xudp_rttmath(cb);
                }

            }
        }
        pCmd=pCmdTmp;
    }

    return 0;
}
```

Bottom of the `xudp_recvackpackets` function.

7.6.8 `xudp_ackpackets` function

`xudp_ackpackets` creates acknowledgement commands on the outgoing command queue (`cb->pCmdQueue`) for packets that comprise the queue of fragmented parcels that have been received from the network that haven't generated an acknowledgement yet.

```
int xudp_ackpackets(struct XUDPCB *cb);
```

- `cb` is a pointer to the control block.

```
int xudp_ackpackets(struct XUDPCB *cb)
{
    struct XUDPCmd        *pCmd;
    struct XUDPFragParcel *pFragParcel;
    struct XUDPPacket     *pPacket;
    Sequence              sequence;

    pFragParcel=cb->pRecvWaitQueue->next;

    /*** Generate ACKs for each packet */

    while (pFragParcel!=NULL)
    {
        if (!pFragParcel->ack)
        {
            pPacket=pFragParcel->headPacket;
            while (pPacket!=NULL)
            {
                if (!pPacket->ack)
                {
                    pCmd=makeCmd();
                    pCmd->cmd=C_ACK;
                    pCmd->length=2*sizeof(Sequence);
                    sequence=htons(pPacket->header.parcel);
                    memcpy(pCmd->argbuffer, &sequence,
                           sizeof(Sequence));
                    sequence=htons(pPacket->header.sequence);
                    memcpy(pCmd->argbuffer+sizeof(Sequence),
                           &sequence, sizeof(Sequence));
                    addCmd(cb->pCmdQueue, pCmd);
                    pPacket->ack=TRUE;
                    if (cb->packetsInPipeRmt>0) cb->packetsInPipeRmt--;
                }
                pPacket=pPacket->next;

            }
        }
        pFragParcel=pFragParcel->next;
    }

    return 0;
}
```

`xudp_ackpackets` function.

Lines 12-39 are a `while` loop that traverses through the list of fragmented parcels on the receive wait queue until the end of the queue is reached. Inside, lines 16-37 are executed if the fragmented parcel hasn't already been acknowledged in its entirety.

A similar loop to find unacknowledged packets exists inside this loop over lines 16-36. If an unacknowledged packet is found, lines 21-29 create a new `struct XUDPCmd,` initializing it as a *C_ACK* command, with a four byte argument buffer consisting of two network-byte-ordered 16-bit unsigned integers: the parcel number and sequence number, respectively. Line 30 adds the command to the end of the outgoing commands queue (`cb->pCmdQueue`.) Line 31 marks this packet as having generated an acknowledgement. Line 32 decreases the number of packets perceived by the local host to have been placed in the network by the remote host.

7.6.9 `xudp_retransmit` function

`xudp_retransmit` tromps through the queue of fragmented parcels waiting for acknowledgement, examining the packets that form them. If it detects that a packet has exceeded the Retransmission Time Out timer (RTO), it queues that packet up for retransmission.

`int xudp_retransmit(struct XUDPCB *cb);`

- `cb` is a pointer to the control block.

```
 1 int xudp_retransmit(struct XUDPCB *cb)
 2 {
 3     struct XUDPPacket     *pPacket, *pPacketTmp, *pPacketNew;
 4     struct XUDPFragParcel *pFragParcel;
 5     Timestamp             ts;
 6
 7     ts=makeTimestamp();
 8
 9     /** Traverse the list of fragmented parcels waiting to be */
10     /** acknowledged */
11
12     pFragParcel=cb->pAckWaitQueue->next;
13     while(pFragParcel!=NULL)
14     {
15         /** Traverse the list of packets that form this parcel that */
16         /** are waiting to be acknowledged. */
17
18         pPacket=pFragParcel->headPacket;
19         while(pPacket!=NULL)
20         {
21             pPacketTmp=pPacket->next;
22
23             /** Only consider the packets that haven't been */
24             /** acknowledged yet */
25
26             if (!(pPacket->ack))
27             {
```

xudp_ackpackets function.

```
        /** See if we've exceeded the RTO */

        if ((ts-pPacket->header.timestamp)>=cb->rto)
        {
          /** Packet needs to be retransmit. */
          /** Put a new timestamp in */

          pPacket->header.timestamp=ts;

          /** Create a copy to be queued up for transmission, but */
          /** castrate the copy so it's not pointing at this queue */

          pPacketNew=makePacket();
          memcpy((void *)pPacketNew, (void *)pPacket,
                 sizeof(struct XUDPPacket));
          pPacketNew->next=NULL;
          pPacketNew->prev=NULL;

          /** Insert it at the head of the outgoing packets */
          /** queue.  It will be sent and requeued on the */
          /** Acknowledgement Wait queue.  We're not creating */
          /** a new packet here, just retransmitting an old */
          /** one.  **No need to adjust the packetsInPipe */
          /** counter because we assume the packet has left the */
          /** network, and we're just going to put it back in.***/

          insertPacket(cb->pXmitQueue, pPacketNew);

          /** Increase RTO exponentially, RTO will only get */
          /** set by the _rttmath algorithm, so we'll keep */
          /** backing off until we ACK something. */

          cb->rto=cb->rto<<1;
        }
      }
      pPacket=pPacketTmp;
    }
    pFragParcel=pFragParcel->next;
  }
  return 0;
}
```

xudp_ackpackets function.

7.6.10 `xudp_rttmath` function

`int xudp_rttmath(struct XUDPCB *cb);`

- `cb` is a pointer to the control block.

```
int xudp_rttmath(struct XUDPCB *cb)
{
    float      error;

    /** Retransmit Time Out timer calculations, straight out of */
    /** _Congestion Avoidance and Control_ */

    /*** The error between the measured RTT and the average */
    error=((float)cb->rtt)-(cb->rttavg);

    /*** Calculate the new average taking into account a bit of the */
    /*** error */
    cb->rttavg+=(RTTGAIN*error);

    /*** Calculate the variance of the RTT estimation */
    if (error<0) error=-error;
    cb->rttvar+=RTTGAIN*(error-(cb->rttvar));

    /*** RTO is average RTT plus four times the variance */
    cb->rto=(int)(cb->rttavg+4*cb->rttvar);

    cb->bandwidth=(cb->bytesACKed*1000)/(cb->timeElapsed);

    if (cb->rtt > cb->rttmax) cb->rttmax=cb->rtt;
    if (cb->rtt < cb->rttmin) cb->rttmin=cb->rtt;

    cb->rttmax-=0.01*(cb->rttmax-cb->rttavg);
    cb->rttmin+=0.01*(cb->rttavg-cb->rttmin);

    xudp_windowchange(cb);

    return 0;
}
```

`xudp_ackpackets` function.

7.6.11 `xudp_windowchange` function

`int xudp_rttmath(struct XUDPCB *cb);`

- `cb` is a pointer to the control block.

```
int xudp_windowchange(struct XUDPCB *cb)
{
    if (cb->state==XUDP_SOFT_START)
    {
        if ((float)cb->rtt > cb->rttavg+(0.1*cb->rttavg))
            cb->state=XUDP_CONG_ANTICIPATION;
/*      cb->windowdelta+=WINDSTEP;*/
        cb->windowdelta=0.25;
    }

    if (cb->state==XUDP_CONG_ANTICIPATION)
    {
        if ((float)cb->rtt < ((cb->rttmin+cb->rttmax)/2))
            cb->state=XUDP_NORMAL;
        cb->windowdelta=-1;
    }

    if (cb->state==XUDP_NORMAL)
    {
        if ((float)cb->rtt > ((cb->rttmin+cb->rttmax)/2))
            cb->state=XUDP_CONG_ANTICIPATION;
/*      cb->windowdelta+=WINDSTEP;*/
        cb->windowdelta=0.125;

    }

    /** Our congestion window is the minimum of the max Remote */
    /** advertised window size and the current window adjusted by */
    /** windowdelta, but always at least 1. */

    if (cb->windowdelta>MAXWIND) cb->windowdelta=MAXWIND;
    if (cb->windowdelta<-MAXWIND) cb->windowdelta=-MAXWIND;
    cb->winFCongestion+=cb->windowdelta;
    if (cb->winFCongestion>(float)cb->winRemote)
        cb->winFCongestion=(float)cb->winRemote;
    if (cb->winFCongestion<1.0) cb->winFCongestion=1.0;
    cb->winCongestion=(int)cb->winFCongestion;
    cb->winCongestion=1;

    return 0;
}
```

`xudp_ackpackets` function.

7.7 Support Routines

7.7.1 `makeFragParcel` function

`struct XUDPFragParcel *makeFragParcel(void);`

```
struct XUDPFragParcel *makeFragParcel(void)
{
    static Sequence   parcel=0;
    struct XUDPFragParcel *pParcel;

    if ( !(pParcel=malloc(sizeof(struct XUDPFragParcel))))
        xudpError("makeFragParcel: Allocating RAM");

    pParcel->parcel=parcel++;
    pParcel->next=NULL;
    pParcel->prev=NULL;
    pParcel->headPacket=NULL;
    pParcel->end=65535;
    pParcel->tto=0;
    pParcel->ack=0;
    pParcel->timestamp=makeTimestamp();

    return pParcel;
}
```

7.7.2 `makeCmd` function

`struct XUDPCmd *makeCmd(void);`

```
struct XUDPCmd *makeCmd(void)
{
    struct XUDPCmd *pCmd;

    if ( !(pCmd=malloc(sizeof(struct XUDPCmd))))
        xudpError("makeCmd: Allocating RAM");

    pCmd->prev=NULL;
    pCmd->next=NULL;
    pCmd->cmd=0;
    pCmd->timestamp=makeTimestamp();

    if ( !(pCmd->argbuffer=malloc(MAXARG)) )
        xudpError("makeCmd: Allocating RAM");

    pCmd->length=0;
```

```c

    return pCmd;
}
```

7.7.3 `makeParcel` function

`struct XUDPParcel *makeParcel(void);`

```c
struct XUDPParcel *makeParcel(void)
{
    static Sequence      parcel=0;
    struct XUDPParcel *pParcel;

    if ( !(pParcel=malloc(sizeof(struct XUDPParcel))))
        xudpError("makeParcel: Allocating RAM");

    pParcel->parcel=parcel++;
    pParcel->tto=0;
    pParcel->next=NULL;
    pParcel->prev=NULL;
    pParcel->buffer=NULL;
    pParcel->pos=NULL;
    pParcel->timestamp=makeTimestamp();

    return pParcel;
}
```

7.7.4 `makePacket` function

`struct XUDPPacket *makePacket(void);`

```c
struct XUDPPacket *makePacket(void)
{
    struct XUDPPacket *pPacket;

    if ( !(pPacket=malloc(sizeof(struct XUDPPacket))))
        xudpError("makePacket: Allocating RAM");

    pPacket->header.sequence=0;
    pPacket->header.tto=0;
    pPacket->length=0;
    pPacket->next=NULL;
    pPacket->prev=NULL;
    pPacket->ack=0;
```

```c

    return pPacket;
}
```

7.7.5 insertFragParcel function

`int insertFragParcel(struct XUDPFragParcel *pInsertAfter, struct XUDPFragParcel *pParcel);`

```c
int insertFragParcel(struct XUDPFragParcel *pInsertAfter, struct XUDPFragParcel *pParcel)
{
    pParcel->next=pInsertAfter->next;
    pParcel->prev=pInsertAfter;
    pInsertAfter->next=pParcel;
    return 0;
}
```

7.7.6 insertParcel function

`int insertParcel(struct XUDPParcel *pInsertAfter, struct XUDPParcel *pParcel);`

```c
int insertParcel(struct XUDPParcel *pInsertAfter, struct
                    XUDPParcel *pParcel)
{
    pParcel->next=pInsertAfter->next;
    pParcel->prev=pInsertAfter;
    pInsertAfter->next=pParcel;
    return 0;
}
```

7.7.7 insertPacket function

`int insertPacket(struct XUDPPacket *pInsertAfter, struct XUDPPacket *pPacket);`

```c
int insertPacket(struct XUDPPacket *pInsertAfter, struct XUDPPacket *pPacket)
{
    pPacket->next=pInsertAfter->next;
    pPacket->prev=pInsertAfter;
    pInsertAfter->next=pPacket;
    return 0;
}
```

7.7.8 deleteFragParcel function

`int deleteFragParcel(struct XUDPFragParcel *pParcel);`

```
int deleteFragParcel(struct XUDPFragParcel *pParcel)
{
    if (pParcel->prev!=NULL) (pParcel->prev)->next=pParcel->next;
    if (pParcel->next!=NULL) (pParcel->next)->prev=pParcel->prev;
    free(pParcel);
    return 0;
}
```

7.7.9 deleteCmd function

`int deleteCmd(struct XUDPCmd *pCmd);`

```
int deleteCmd(struct XUDPCmd *pCmd)
{
    if (pCmd->prev!=NULL) (pCmd->prev)->next=pCmd->next;
    if (pCmd->next!=NULL) (pCmd->next)->prev=pCmd->prev;
    free(pCmd->argbuffer);
    free(pCmd);
    return 0;
}
```

7.7.10 deleteParcel function

`int deleteParcel(struct XUDPParcel *pParcel);`

```
int deleteParcel(struct XUDPParcel *pParcel)
{
    if (pParcel->prev!=NULL) (pParcel->prev)->next=pParcel->next;
    if (pParcel->next!=NULL) (pParcel->next)->prev=pParcel->prev;
    free(pParcel->buffer);
    free(pParcel);
    return 0;
}
```

7.7.11 deletePacket function

`int deletePacket(struct XUDPPacket *pPacket);`

```
int deletePacket(struct XUDPPacket *pPacket)
{
    if (pPacket->prev!=NULL) (pPacket->prev)->next=pPacket->next;
    if (pPacket->next!=NULL) (pPacket->next)->prev=pPacket->prev;
    free(pPacket);
    return 0;
}
```

7.7.12 `addFragParcel` function

`int addFragParcel(struct XUDPFragParcel *pHead, struct XUDPFragParcel *pParcel);`

```
int addFragParcel(struct XUDPFragParcel *pHead, struct XUDPFragParcel *pParcel)
{
    struct XUDPFragParcel *pTail;
    pTail=pHead;

    while(pTail->next!=NULL) pTail=pTail->next;

    pTail->next=pParcel;
    pParcel->prev=pTail;
    return 0;
}
```

7.7.13 `addParcel` function

`int addParcel(struct XUDPParcel *pHead, struct XUDPParcel *pParcel);`

```
int addParcel(struct XUDPParcel *pHead, struct XUDPParcel *pParcel)
{
    struct XUDPParcel *pTail;
    pTail=pHead;

    while(pTail->next!=NULL) pTail=pTail->next;

    pTail->next=pParcel;
    pParcel->prev=pTail;
    return 0;
}
```

7.7.14 `addCmd` function

`int addCmd(struct XUDPCmdxs *pHead, struct XUDPCmd *pCmd);`

```
int addCmd(struct XUDPCmd *pHead, struct XUDPCmd *pCmd)
{
    struct XUDPCmd *pTail;
    pTail=pHead;

    while(pTail->next!=NULL) pTail=pTail->next;

    pTail->next=pCmd;
    pCmd->prev=pTail;

    return 0;
}
```

7.7.15 `addPacket` function

`int addPacket(struct XUDPPacket *pHead, struct XUDPPacket *pPacket);`

7.7.16 `xudpError` function

`void xudpError(char *errtext);`

```
int addPacket(struct XUDPPacket *pHead, struct XUDPPacket *pPacket)
{
    struct XUDPPacket *pTail;
    pTail=pHead;

    while(pTail->next!=NULL) pTail=pTail->next;

    pTail->next=pPacket;
    pPacket->prev=pTail;
    return 0;
}
```

7.7.17 `xudpGetFunction` function

`int xudpGetFunction(struct XUDPHost *xudphost);`

```
int xudpGetFunction(struct XUDPHost *xudphost)
{
    char function;

    if (recv(xudphost->fdControl, &function, 1, 0)!=1)
     return -1;

    return function;
}
```

7.7.18 `xudpSendFunction` function

`int xudpSendFunction(struct XUDPHost *xudphost, char fcn, char *data);`

```
int xudpSendFunction(struct XUDPHost *xudphost, char fcn, char *data)
{
    int   bytes=0;
    int   result;

    if (send(xudphost->fdControl, (char *) &fcn, 1, 0)==1)
    {
     switch (fcn)   /* Set up the data sizes for all of the functions */
     {
     case F_CONNECTED: bytes=sizeof(struct sockaddr_in); break;
     case F_ERROR: bytes=1; break;
     case F_READY: break;
     case F_BIND: bytes=sizeof(u_short); break;
     case F_LISTEN: break;
     case F_ACCEPT: break;
     case F_CONNECT:
         bytes=sizeof(struct sockaddr_in);
         data=(char *)&xudphost->hostaddr;
         break;
     case F_SENDAUDIO: bytes=sizeof(u_short); break;
     case F_SENDVIDEO: bytes=sizeof(u_short); break;
     case F_RECVVIDEO: bytes=sizeof(u_short); break;
     case F_RECVAUDIO: bytes=sizeof(u_short); break;
     case F_CLOSE: break;
     default: break;
     }

     if (bytes>0)
```

```
     {
         if ((result=streamWrite(xudphost->fdControl, data,
                                bytes))!=bytes)
             xudpError("UNIX Stream Failed");
     }
    }
    else return -1;

    return bytes;
}
```

7.7.19 streamRead function

int streamRead(int fd, char *buffer, u_int num);

```
int streamRead(int fd, char *buffer, u_int num)
{
    return(recv(fd, buffer, num, 0));
}
```

7.7.20 streamWrite function

int streamWrite(int fd, char *buffer, u_int num);

```
int streamWrite(int fd, char *buffer, u_int num)
{
    return(send(fd, buffer, num, 0));
}
```

7.7.21 makeServer_un function

int makeServer_un(struct sockaddr_un *un_addr, char *path);

```
int makeServer_un(struct sockaddr_un *un_addr, char *path)
{
    int fdSocket, un_len;

    bzero((char *)un_addr, sizeof(struct sockaddr_un));

    if ( (fdSocket=socket(AF_UNIX, SOCK_STREAM, 0)) <0)
     xudpError("xudp: socket(UNIX)");   /* Opens a UNIX Stream socket */

```

```
                                /* Fill out address structure for server */

    un_addr->sun_family=AF_UNIX;         /* AF_UNIX is the UNIX Domain
                                              Protocol Family.*/

    strcpy(un_addr->sun_path, path); /* Server stream path */

    un_len=strlen(un_addr->sun_path) + sizeof(un_addr->sun_family);

    if (bind(fdSocket, (struct sockaddr *)un_addr, un_len)<0)
     xudpError("xudpd: Can't bind address");

    listen(fdSocket, 5);

    return fdSocket;
}
```

7.7.22 `max` function

`int max(int x, int y);`

```
int max(int x, int y)
{
    if (x>y) return x;
    else return y;
}
```

7.7.23 `min` function

`int min(int x, int y);`

```
int min(int x, int y)
{
    if (x<y) return x;
    else return y;
}
```

7.7.24 `makeTimestamp` function

`Timestamp makeTimestamp(void);`

```
Timestamp makeTimestamp(void)
{
    Timestamp        timestamp;
    static long      startsec=0, startusec=0;
    struct timeval   timestuff;

    gettimeofday(&timestuff, (struct timezone *) 0);
    if (startsec==0)
    {
     startsec=timestuff.tv_sec;
     startusec=timestuff.tv_usec;
    }

    timestuff.tv_sec-=startsec;
    timestuff.tv_usec-=startusec;

    timestamp=((timestuff.tv_usec / (GRANULARITY*1000)) +
            ((timestuff.tv_sec*1000)/GRANULARITY));

    return timestamp;
}
```

Chapter 8

Results and Analysis

To facilitate the testing of XUDP, three log files were written by the XUDP server itself to record, at twenty millisecond intervals *bytes of data acknowledged, round trip time* and the *sliding window size (packets in network).* For comparison tests between TCP and XUDP, a fourth log file was maintained by the testing application on the receiver side that recorded *parcel sequence numbers received* and the time that each reached the application layer.

Each test is documented both in graphical form and as a table of statistics. Comparisons between sets of data are drawn where applicable. All data was transmitted across the same piece of 10base-2 Ethernet between two Digital Alphas running OSF/1 V3.2.

8.1 Early Results: Problems with XUDP's "Nagle" Algorithm

This section documents a test conducted while XUDP's congestion avoidance and retransmission algorithms were still in their infancy.

The test utilized the simple data streaming application described in chapter 6, sending 1MB of data as a continuous stream of 10,240 byte `RELIABLE` parcels. Table 8.1 lists the statistical results.

Measurement	**Result**
Total Data Acknowledged	516.100 KBytes
Total Transmission Time	54.215 seconds
Network Bandwidth Utilization	9.520 KBytes/second
Average Round Trip Time	103.6 milliseconds
Average Window Size	3.9 packets

Table 8.1: Test C, Feb 26, 1997

The total data acknowledged for this connection was 516.1KBytes only due to a timeout that had been set to end the transfer after almost a minute; witness that the total transmission time is 54 seconds. As should be evident by now, these results indicate a truly shoddy performance, with a round trip time of over 100ms and an average bandwidth utilization of 9.5KBytes/second. In fact, this session used under 1% of the total available 10Mbps bandwidth.

Figure 8.1 shows a plot of the total bytes of data acknowledged versus time in milliseconds. If this graph were representative of a perfect transfer, it would consist of a solid line that begins

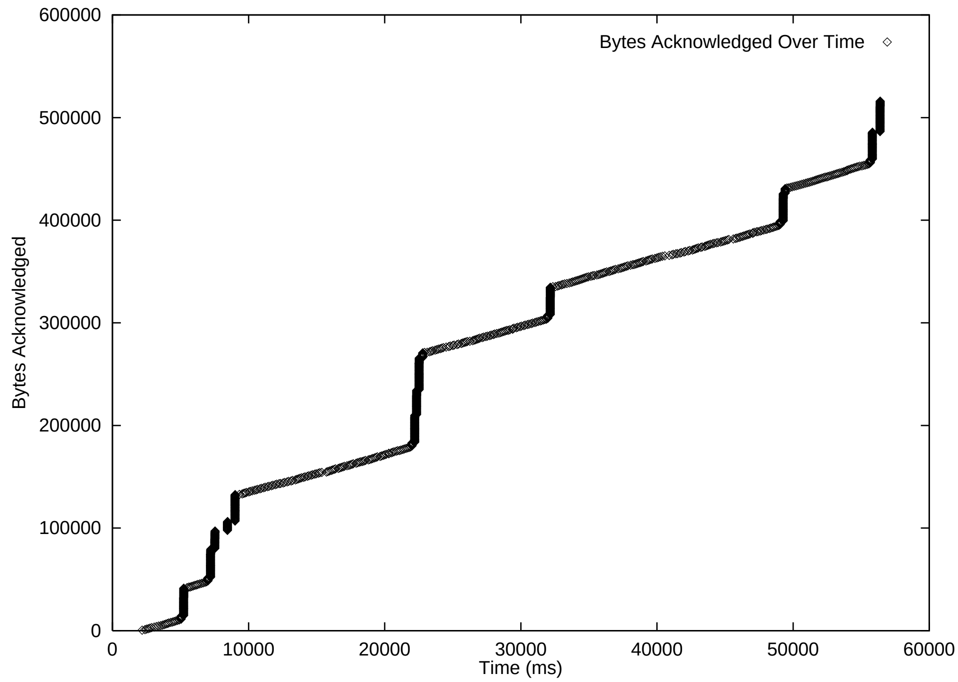

Figure 8.1: Bytes Acknowledged ('C' test 2/26/97)

at the origin and extends to the top, right hand corner with a decidedly shorter time axis. This particular graph is obviously departing a lot from the ideal case.

To understand this graph, consider a case where a file transfer only totals three bytes in length. If the first byte is acknowledged at $t = 1ms$, the second byte, for a total of two, at $t = 2ms$ and the third, for a total of three, at $t = 3ms$, the graph would consists of three points tracing out a diagonal straight line as the previous paragraph explained.

Interestingly, on this type of graph, the instantaneous bandwidth utilization can be determined as the derivative; the average bandwidth over a given unit of time can be found from the slope. More vertical slopes correspond to higher bandwidth utilization.

To understand the jagged nature of figure 8.1 we need to look toward the next graph of round trip times, figure 8.2.

The round trip time graph has time on both axes. Each point on the graph is the time between a packet leaving the sending side and the reception of an acknowledgment for that packet at the sending side on the Y axis and the approximate time (quantized to 20ms) that it took the packet's acknowledgment to arrive from the beginning of the entire session shown on the X axis. The ideal graph here is a flat distribution of points as close to the bottom (X axis) as possible, indicating similar and low round trip times for all the packets. For example, if a packet's acknowledgment arrived at the sender 100ms from the start of the connection and the round trip time for the packet was 10ms, the packet would be plotted as a dot at (100ms, 10ms).

Figure 8.2 shows a surprising plateau at approximately 100ms, an extremely high RTT for 10Mbps Ethernet. After much searching, the culprit was determined to be a faulty "Nagle" (in this case a delayed acknowledgment) algorithm. In the Transmission Control Protocol, to avoid sending small amounts of data, the protocol would store the data and try to clump additional pieces of outgoing data onto it until a maximum-sized packet was formed; if an acknowledgment

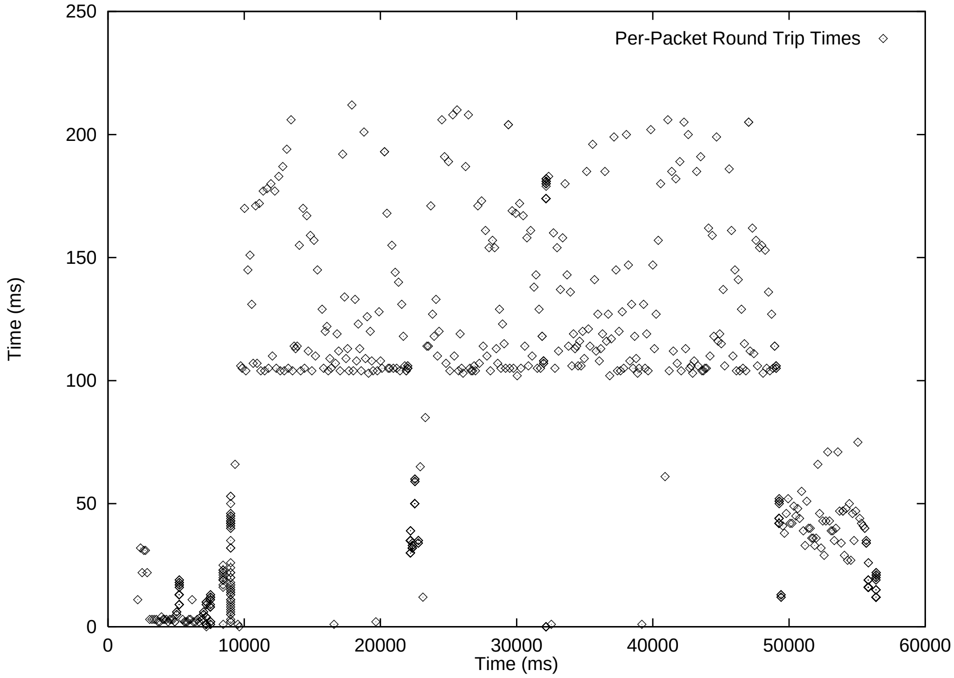

Figure 8.2: Round Trip Times ('C' test 2/26/97)

for something is received, the algorithm transmitted all of the stored data. Earlier versions of this algorithm utilized a fixed time delay for clumping data together, if the time expired, the data was sent. In the case of XUDP's algorithm, it implemented a fixed, 100ms delay. Packet and parcel acknowledgments would build up in the queues and only get sent every 100ms, accounting for the slightly greater than 100ms round trip time. This algorithm was subsequently removed from the implementation.

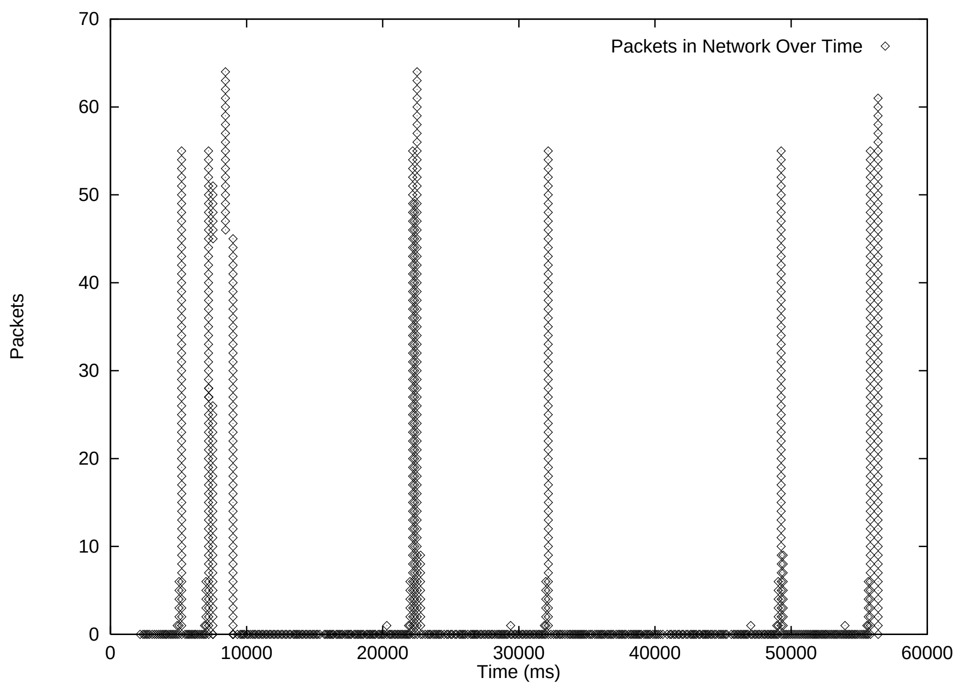

Figure 8.3: Packets In Network ('C' test 2/26/97)

The final graph, figure 8.3, is interesting at best, and shows the extreme flaws in this version of the XUDP server. Plotted here are the number of packets in the network, versus time, as predicted by the size of the congestion window. This graph, if perfect, would show a gently sloping line up to a maximum, where it would stay unless congestion avoidance kicks in. What the graph of figure 8.3 shows indicates that the network is getting filled in quick bursts, followed by long periods of time when only one packet exists in the network. This is far from optimal.

8.2 Early Results: "Tweaked" Congestion Avoidance Mechanism

The following day, after a great deal of tweaking all of the congestion avoidance algorithms, I ran the following test. Again, it transmitted a fully-reliable 1MB stream of 10,240 byte parcels.

Measurement	**Result**
Total Data Acknowledged	1068.766 KBytes
Total Transmission Time	5.175 seconds
Network Bandwidth Utilization	206.525 KBytes/second
Average Round Trip Time	16.9 milliseconds
Average Window Size	10.7 packets

Table 8.2: Test C, Feb 27, 1997

Table 8.2 shows the improvement here. This session sustained an average network bandwidth utilization of 206KBytes/second. Figure 8.4 even indicates, based on the slope of the curve, that the bandwidth utilization at peak hit 750KBytes/second!

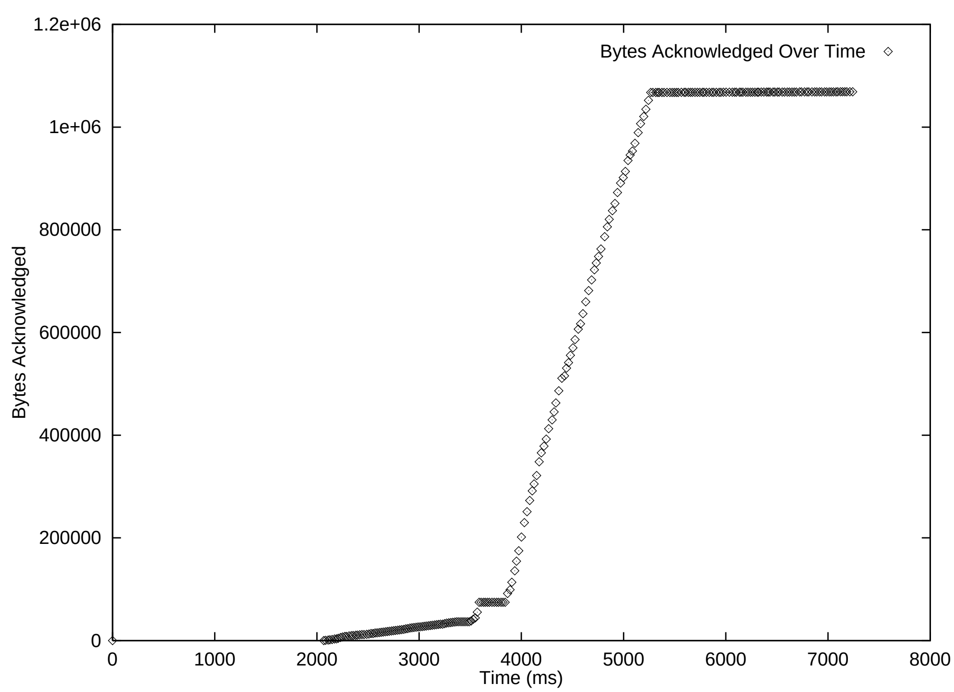

Figure 8.4: Bytes Acknowledged ('C' test 2/27/97)

Figures 8.5 and 8.6 indicate that round trip times were a bit high and varied greatly, and that the window opened too quickly.

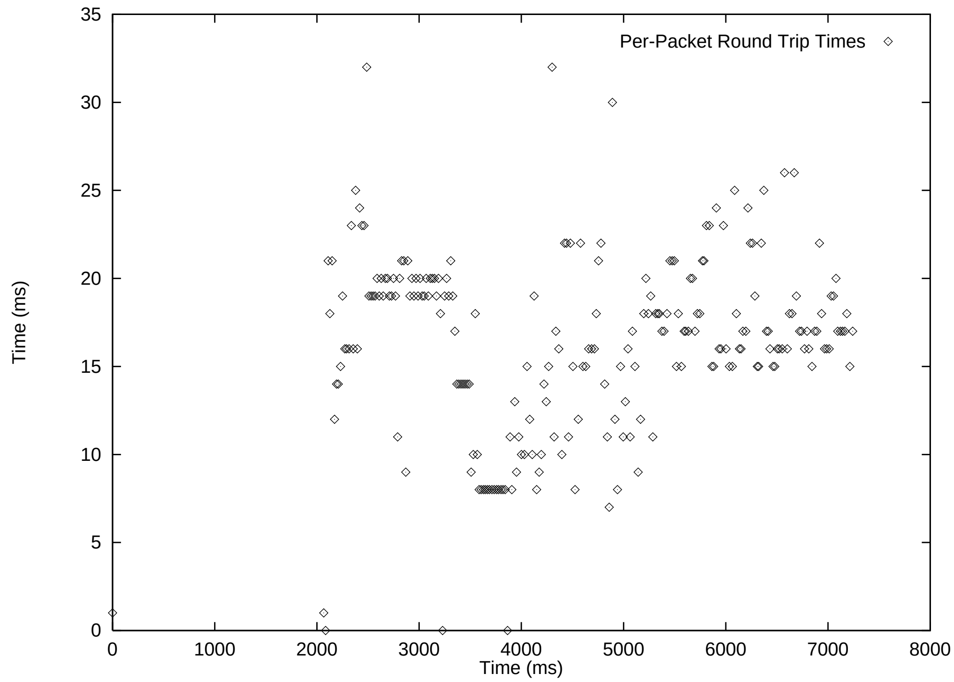

Figure 8.5: Round Trip Times ('C' test 2/27/97)

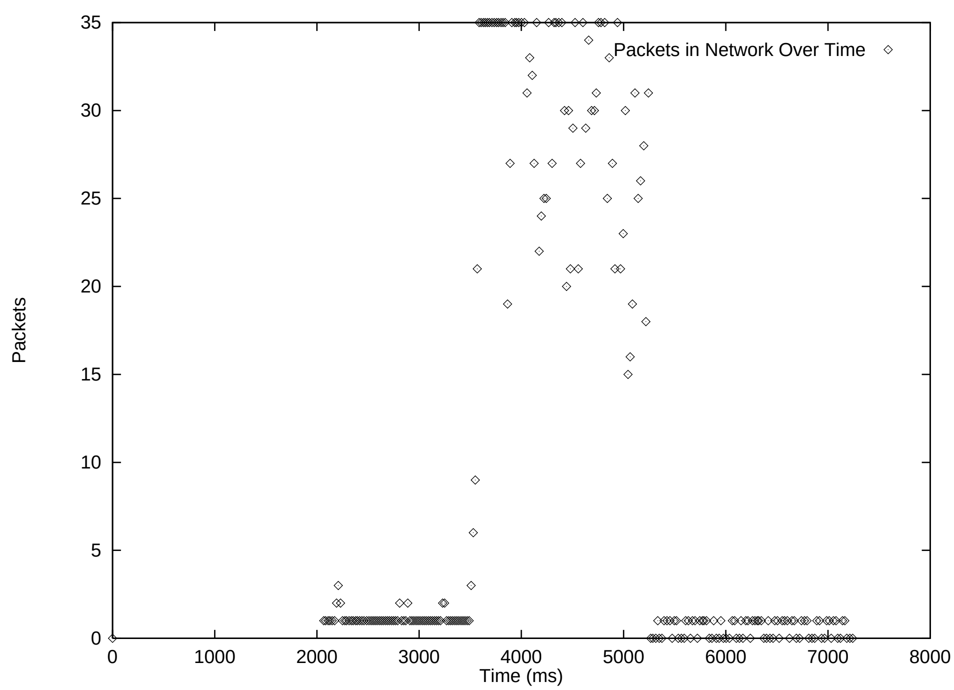

Figure 8.6: Packets In Network ('C' test 2/27/97)

8.3 Comparison of Early Results

This section addresses briefly the performance improvement gained by having better optimized flow control algorithms, based on the data from the previous two test runs.

Table 8.3 juxtaposes the statistics of both tests, and including a measure of the factor of increase for the second test. Note especially the 21.7 factor increase in bandwidth (that's a 2000% improvement!)

Measurement	**2/26 Result**	**2/27 Result**	**Factor of Increase**
Total Data Acknowledged	516.100 KBytes	1068.766 KBytes	2.071
Total Transmission Time	54.215 seconds	5.175 seconds	0.095
Network Bandwidth Utilization	9.520 KBytes/second	206.525 KBytes/second	21.69
Average Round Trip Time	103.6 milliseconds	16.9 milliseconds	0.163
Average Window Size	3.9 packets	10.7 packets	2.744

Table 8.3: Comparison between 'C' tests of 2/26/97 and 2/27/97

To avoid any confusion regarding the differences in scale between the two tests, figure 8.7 shows both test's bytes acknowledged versus time curves on the same plot.

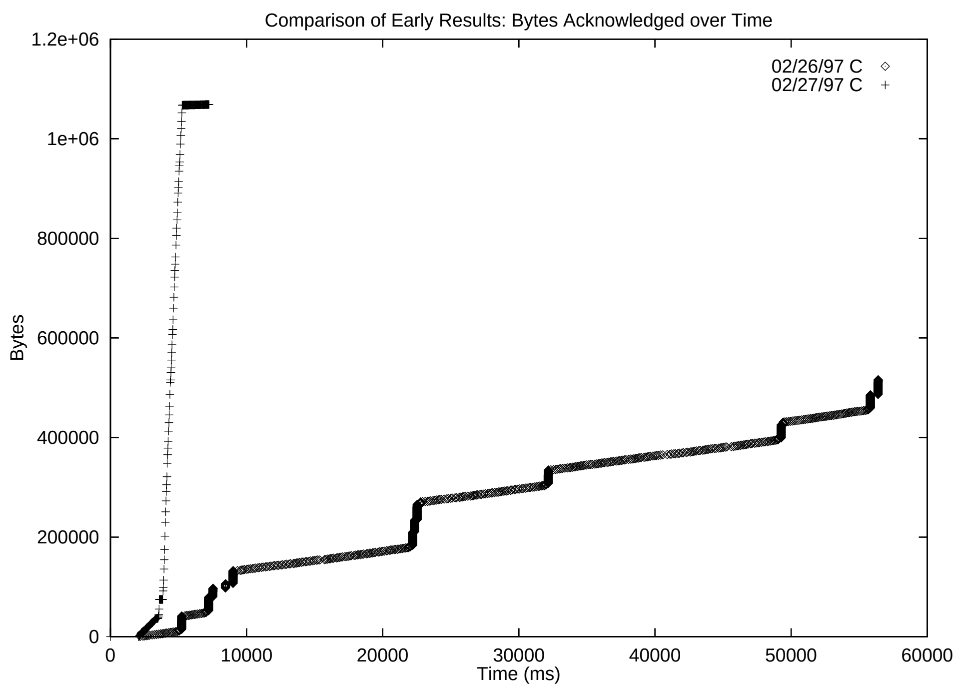

Figure 8.7: Comparison between 'C' tests of 2/26/97 and 2/27/97 for Bytes Acknowledged

8.4 Fully-reliable Data Stream

After the server had been much updated (approximately 18 days later) the remainder of the tests presented here were conducted. The test in this section is the last of the fully-reliable data stream tests, with the same parameters as before.

Measurement	**Result**
Total Data Acknowledged	1067.017 KBytes
Total Transmission Time	6.084 seconds
Network Bandwidth Utilization	175.381 KBytes/second
Average Round Trip Time	22.0 milliseconds
Average Window Size	10.2 packets

Table 8.4: Test 20:29, March 16, 1997

Although table 8.4 indicates slower general results than the test of 18 days ago, the server here is quite a bit more well behaved. The bytes acknowledged graph (figure 8.8) shows a more ideal curve than previously seen; the tail is caused by residual acknowledgments during the closing of the connection.

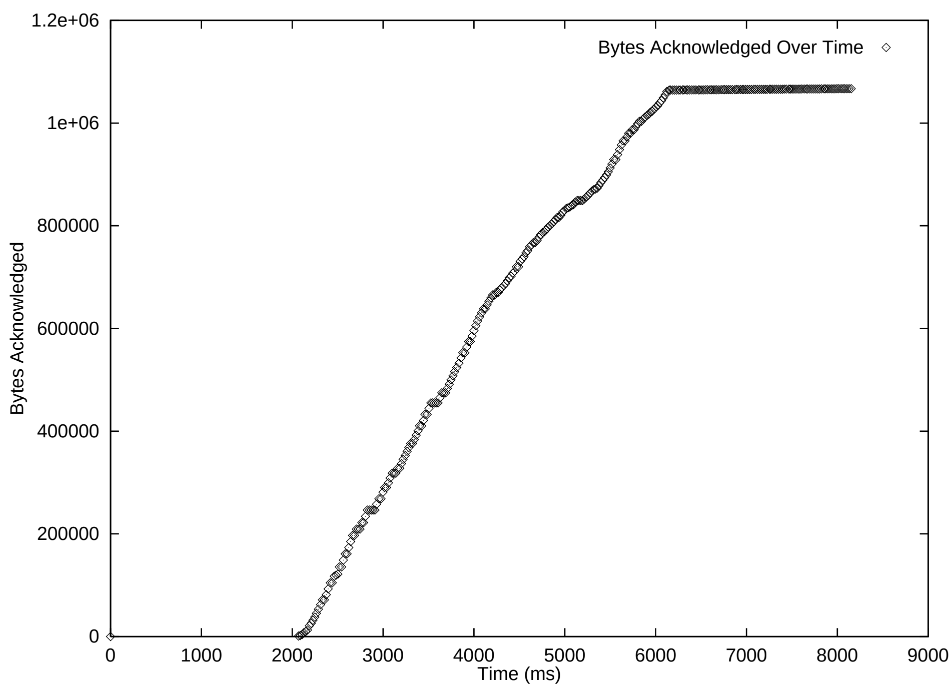

Figure 8.8: Bytes Acknowledged (Test 20:29 3/16/97)

The real performance improvements made here affected the window sizing (figure 8.10) which still scales too quickly, but is nonetheless an improvement over the previous test's plot. The RTT distribution (figure 8.9) is still all over the map.

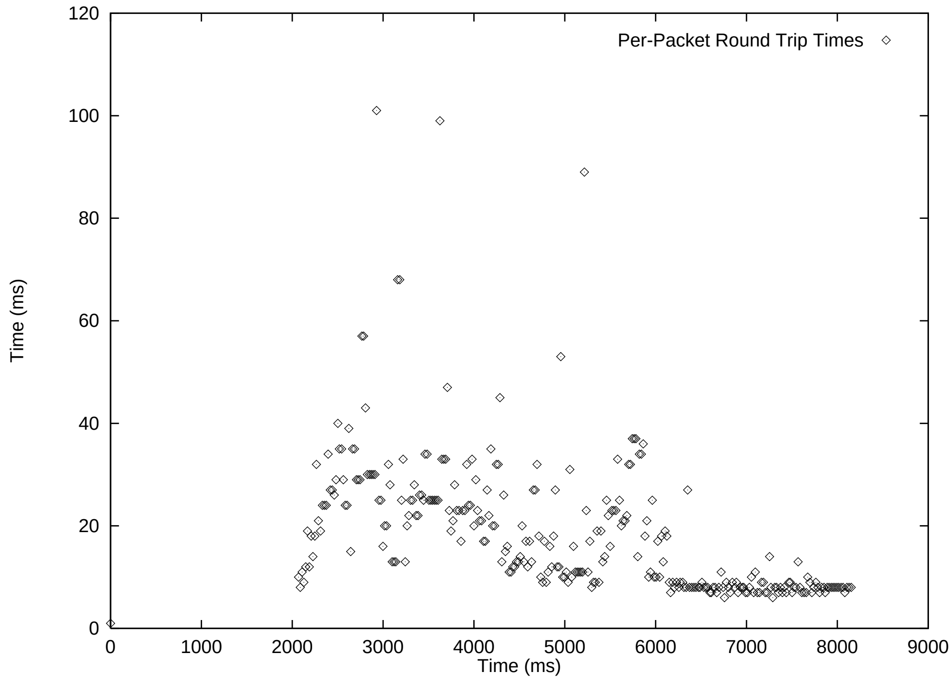

Figure 8.9: Round Trip Times (Test 20:29 3/16/97)

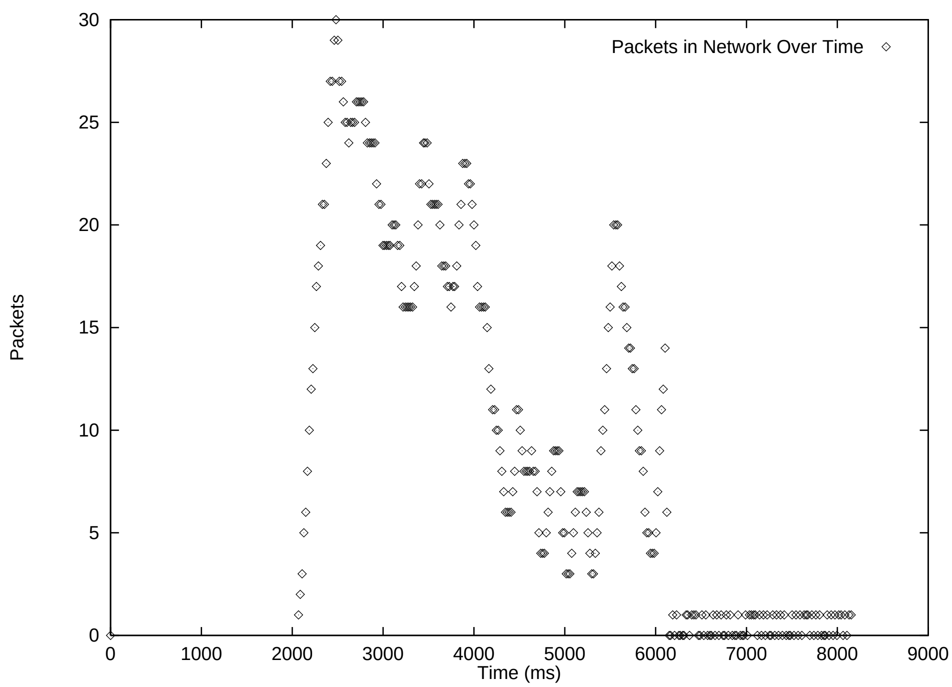

Figure 8.10: Packets In Network (Test 20:29 3/16/97)

8.5 Simulated 10fps Real-Time Uncompressed Video Data

Starting with this section, a new type of test designed to utilize timed obsolescence was used to produce the results. A more complex testing application than that described previously in chapter 6 was necessary to effectively simulate a real-time, 30 second, uncompressed video stream. Instead of continuously sending 10,240 byte parcels as in the prior cases, this test paces itself. At the beginning of every second, 10 parcels are sent to the XUDP server; these parcels are 14,400 bytes long, corresponding to 160x120, 6-bit grey scale images. For XUDP, the first parcel in every second is labeled as ***RELIABLE*** and every subsequent parcel gets a time to obsolescence of two seconds.

The tests are also each conducted using TCP as the transport layer, so that the impact of timed obsolescence can be properly evaluated.

8.5.1 XUDP

Table 8.5 shows the statistical results reported by the XUDP server (top) and the receiving-side application (bottom.) The frame rate at the receiver (9.8fps) nearly matched the theoretical sending frame rate (10fps) and XUDP didn't need to resort to timed obsolescence as no parcels were dropped.

Measurement	**Result**
Total Data Acknowledged	4391.134 KBytes
Total Transmission Time	29.779 seconds
Network Bandwidth Utilization	147.457 KBytes/second
Average Round Trip Time	13.8 milliseconds
Average Window Size	10.2 packets
Total Parcels Received	291 parcels
Total Transmission Time	29.683 seconds
Total Parcels Skipped	0 parcels
Network Bandwidth Utilization (14.4 KBytes/parcel)	141.172 KBytes/second
Average Parcel Reception Frequency	9.804 parcels/second

Table 8.5: XUDP Test 01:02, March 17, 1997

Figure 8.11 is slightly different than the graph of bytes acknowledged in previous sections. This graph is recorded in the application layer by simply writing the parcel sequence number of the just received parcel and the time at which it was received to a log file. The peculiar stair-stepping is due to the sending side sending the 10 parcels at the beginning of every second, rather than evenly spacing them.

An interesting RTT distribution can be found in figure 8.13. Although instances of high RTT's occur, most of the hits fall just under 10ms.

The congestion window sizing manifested as the number of packets in the network in figure 8.14 is not a pretty sight. A large percentage of the time the window is either 1 or even zero packets in length. Future study will hopefully deal with this situation.

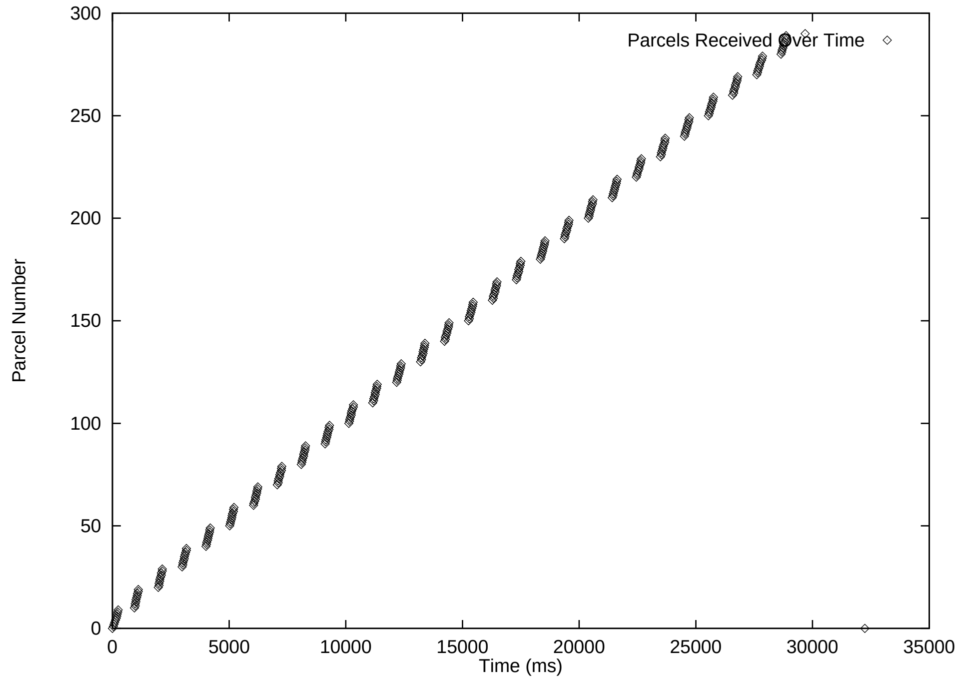

Figure 8.11: Parcels Received (XUDP Test 01:02 3/17/97)

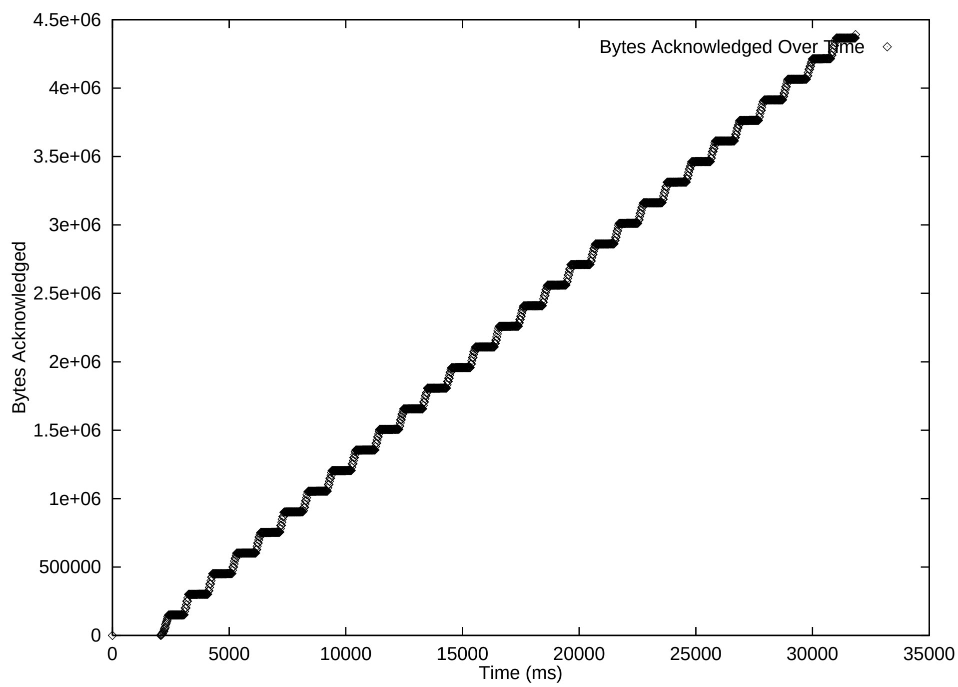

Figure 8.12: Bytes Acknowledged (XUDP Test 01:02 3/17/97)

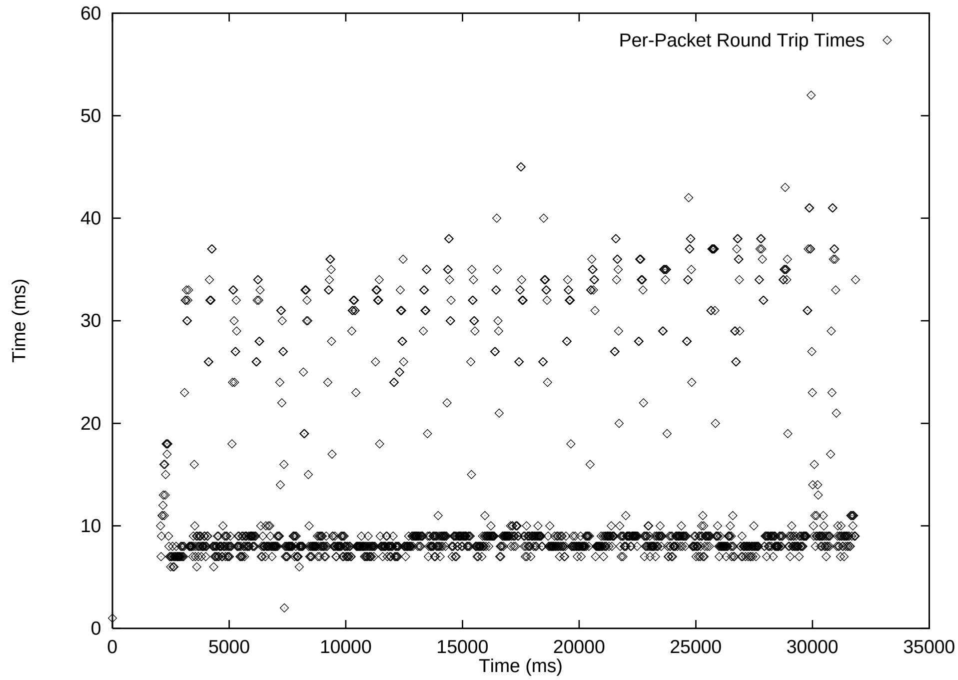

Figure 8.13: Round Trip Times (XUDP Test 01:02 3/17/97)

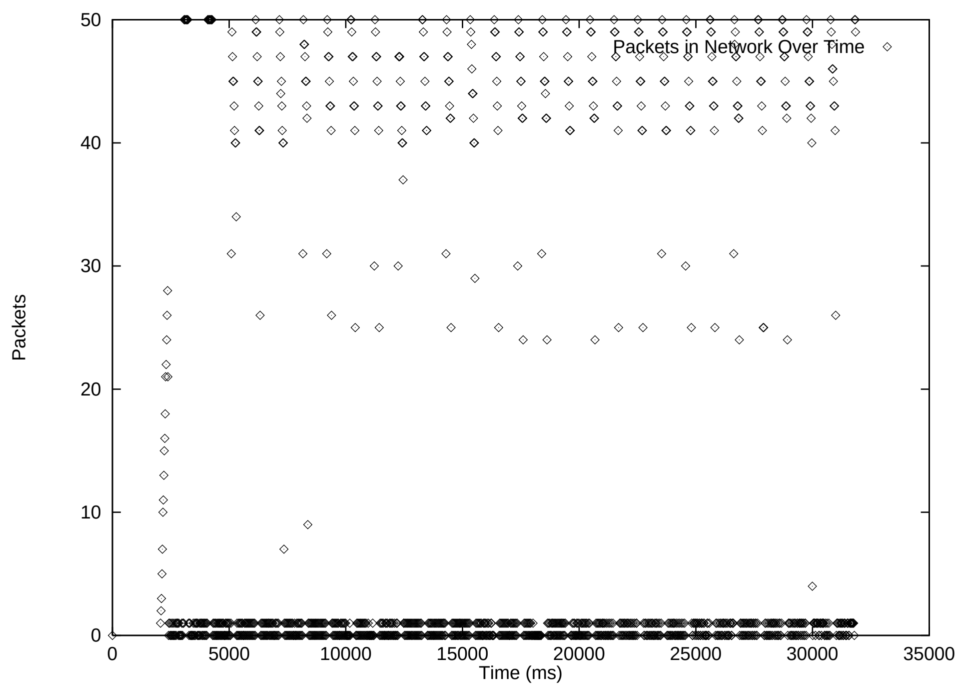

Figure 8.14: Packets In Network (XUDP Test 01:02 3/17/97)

8.5.2 TCP

This brief section covers the TCP version of the previous test. TCP doesn't implement timed obsolescence, so all the parcels sent to TCP will be delivered.

Measurement	Result
Total Parcels Received	291 parcels
Total Transmission Time	32.244 seconds
Total Parcels Skipped	0 parcels
Network Bandwidth Utilization (14.4 KBytes/parcel)	129.959 KBytes/second
Average Parcel Reception Frequency	9.025 parcels/second

Table 8.6: TCP Test 01:04, March 17, 1997

Table 8.6 shows that TCP finished two seconds later than XUDP, that the throughput was lower and only 9fps was realized at the receiver. Figure 8.15 shows the graph of parcels received over time.

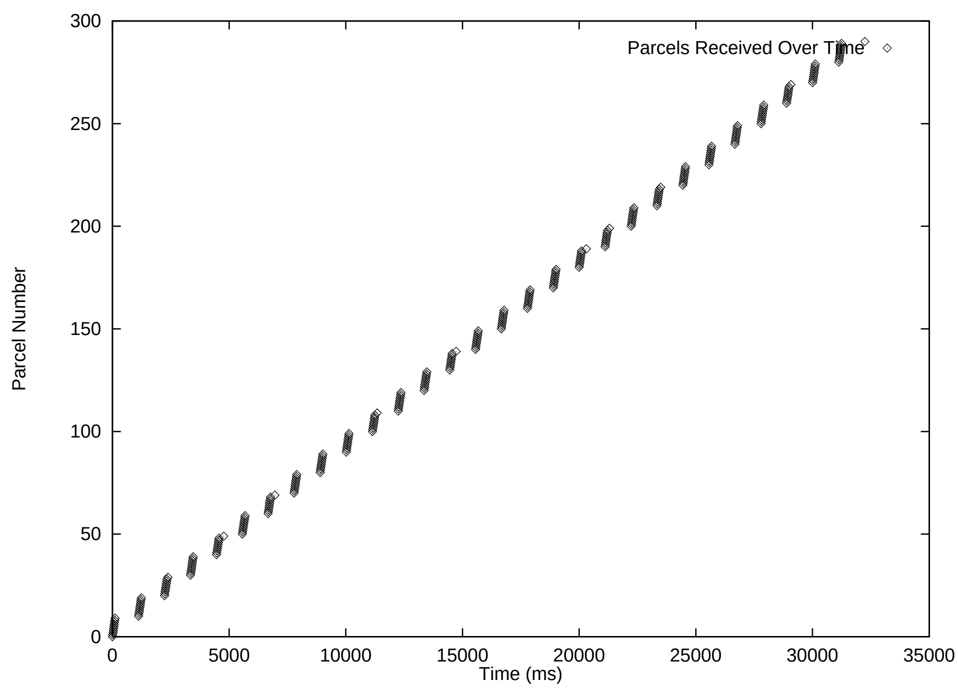

Figure 8.15: Parcels Received (TCP Test 01:04 3/17/97)

8.5.3 Comparison

A comparison of the results for XUDP and TCP is displayed in figure 8.16 that shows XUDP edging ahead of TCP, XUDP ending on time (30 seconds) and TCP falling two seconds behind.

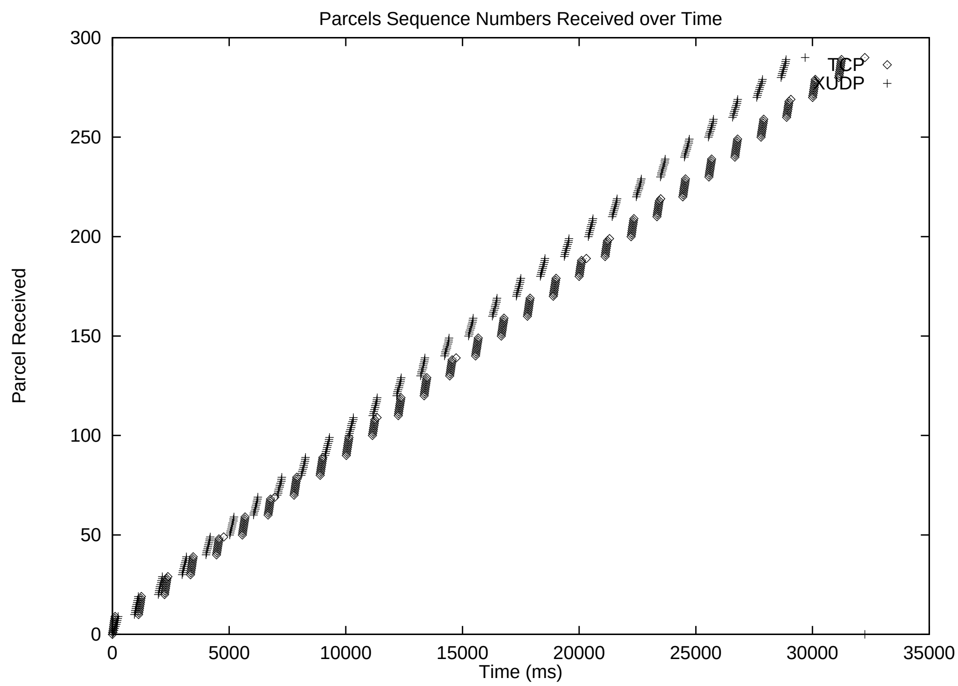

Figure 8.16: Comparison of TCP vs. XUDP Parcels Received (Tests 01:04 and 01:02 3/17/97)

8.6 Simulated 20fps Uncompressed Real-Time Video Data

For the case of 20fps video, the testing application generated twenty 14,400 byte parcels at the start each second, again with the first one in the second marked as ***RELIABLE*** and each subsequent parcel marked with a two second time-to-obsolescence.

8.6.1 XUDP

Twenty frames per second was enough of a transmission burden to force XUDP into invoking its timed obsolescence mode. According to table 8.7, out of 581 total parcels, 448 were received and 133 were discarded due to timed obsolescence. The transfer finishes in 32.4 seconds, the length of the transfer plus two seconds for the timed obsolescence window.

Measurement	**Result**
Total Data Acknowledged	6962.400 KBytes
Total Transmission Time	32.650 seconds
Network Bandwidth Utilization	213.243 KBytes/second
Average Round Trip Time	19.5 milliseconds
Average Window Size	11.0 packets
Total Parcels Received	448 parcels
Total Transmission Time	32.351 seconds
Total Parcels Skipped	133 parcels
Network Bandwidth Utilization (14.4 KBytes/parcel)	199.413 KBytes/second
Average Parcel Reception Frequency	13.848 parcels/second

Table 8.7: XUDP Test 23:41 3/16/97

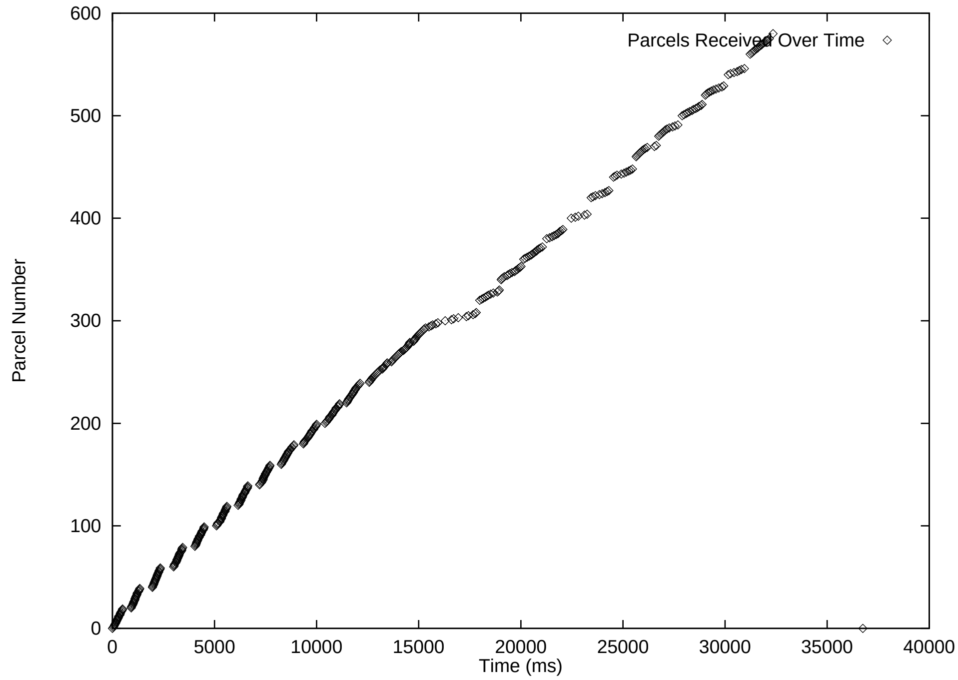

Figure 8.17: Parcels Received (XUDP Test 23:41 3/16/97)

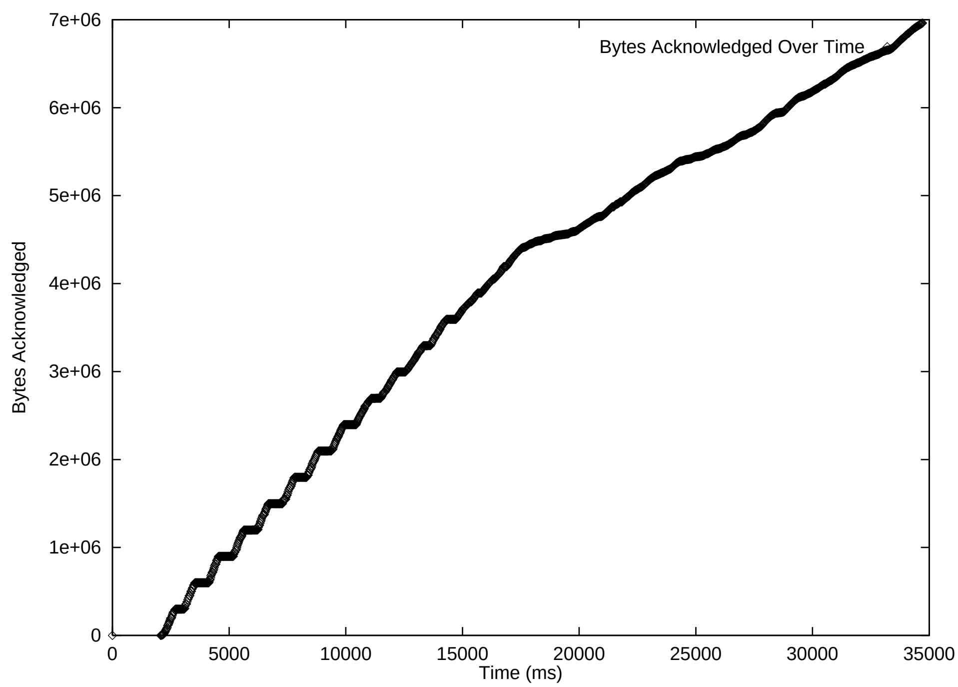

Figure 8.18: Bytes Acknowledged (XUDP Test 23:41 3/16/97)

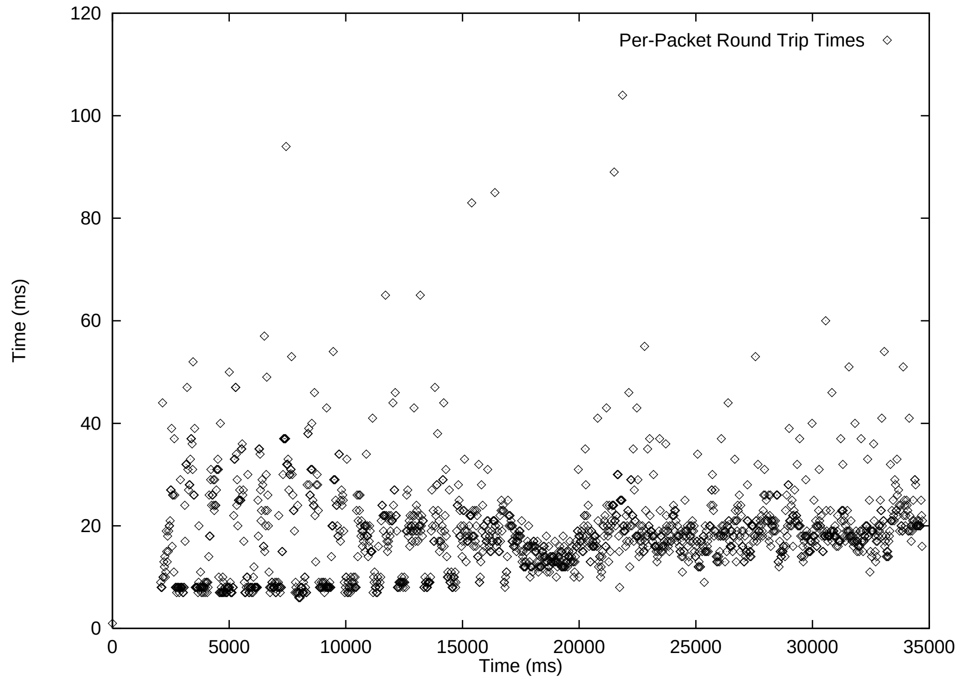

Figure 8.19: Round Trip Times (XUDP Test 23:41 3/16/97)

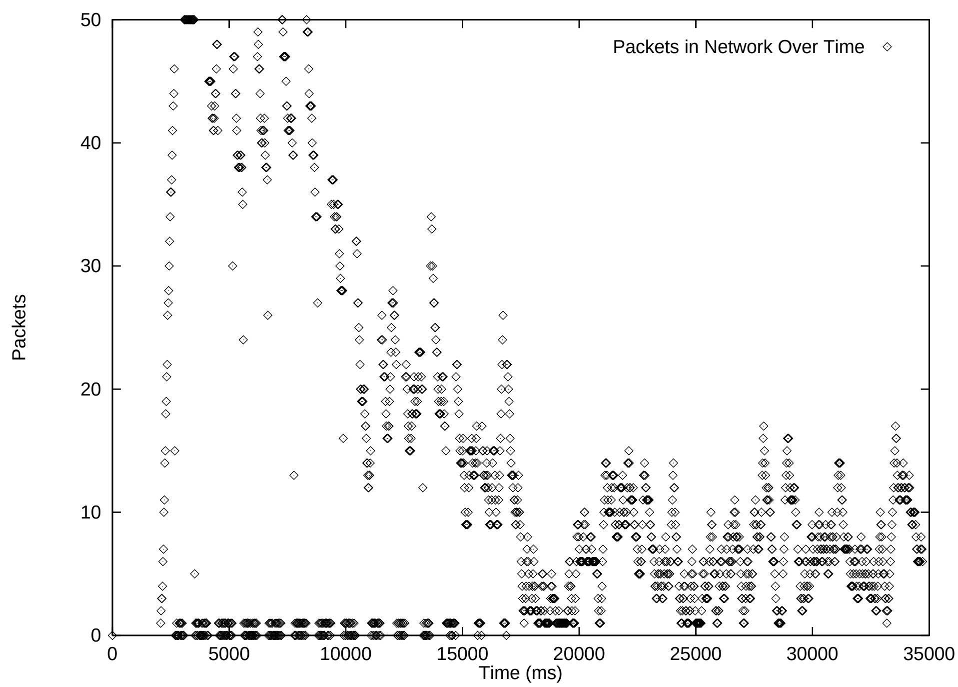

Figure 8.20: Packets In Network (XUDP Test 23:41 3/16/97)

8.6.2 TCP

The 20fps attempted TCP transfer, of course, received all 581 parcels (table 8.8, but at the expense of an extra four seconds on the transmission time yielded in the XUDP test. TCP was able to handle the bandwidth requirement, transmitting at 228KBytes/second.

Measurement	**Result**
Total Parcels Received	581 parcels
Total Transmission Time	36.753 seconds
Total Parcels Skipped	0 parcels
Network Bandwidth Utilization (14.4 KBytes/parcel)	227.639 KBytes/second
Average Parcel Reception Frequency	15.808 parcels/second

Table 8.8: TCP Test 23:39, March 16, 1997

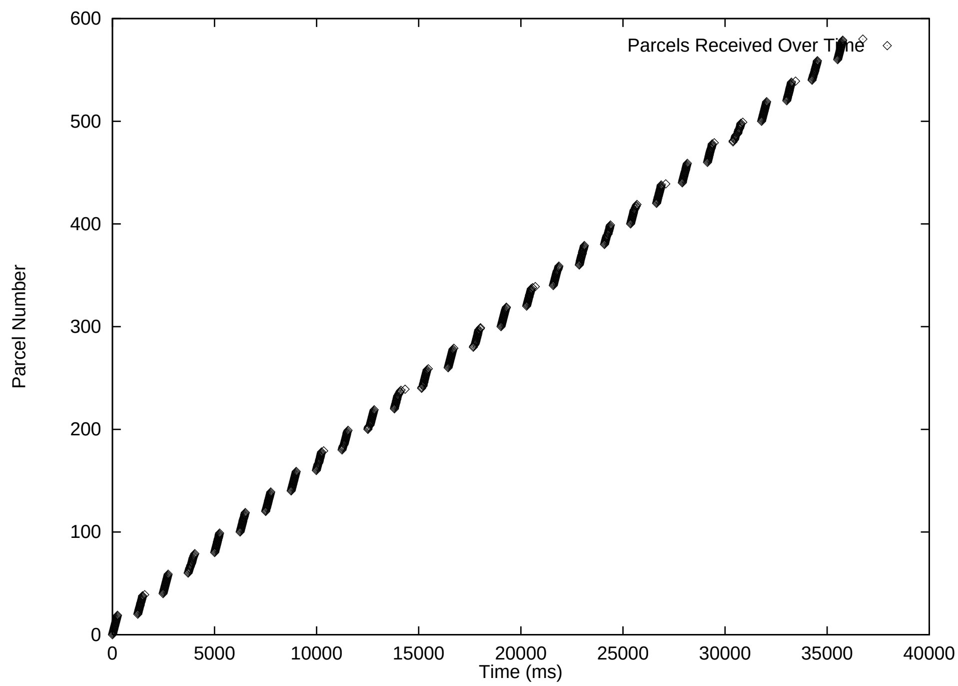

Figure 8.21: Parcels Received (TCP Test 23:39 3/16/97)

8.6.3 Comparison

Below is figure 8.22, a plot of both the parcel sequence numbers received over time for the 20fps tests conducted on XUDP and TCP. XUDP's timed obsolescence mode performed well, receiving parcels on time. The bottleneck here appears to be XUDP's rather unoptimized architecture; bandwidth requirements approaching 200KBytes/second are pushing the limit of XUDP's resources.

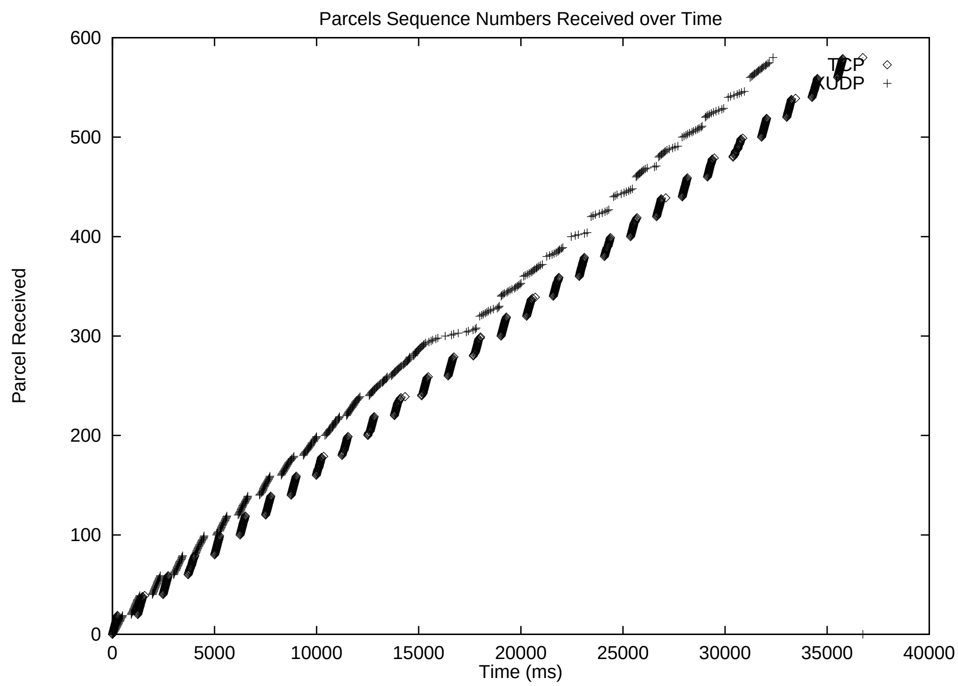

Figure 8.22: Comparison of TCP vs. XUDP Parcels Received (Tests 23:39 and 23:41 3/17/97)

8.7 Simulated 20fps Uncompressed Video Data Under Severe Network Loading

To determine how XUDP and TCP fared when faced with severe network congestion, I developed a simple client/server pair to run in parallel with the test with the singular goal of generating lots of network load. The actual 20fps tests are identical to those in the preceding section.

8.7.1 XUDP

XUDP was worse off this time, only receiving 198 parcels out of 581 (table 8.9.) The graph of figure 8.23 indicates that after approximately 15 seconds, XUDP resorted to only sending the `RELIABLE` parcels. XUDP fell 7 seconds behind due to the required reliable data transfer of 1 parcel a second. With a properly programmed multimedia sending application, XUDP would perform quite a bit better.

Measurement	**Result**
Total Data Acknowledged	3009.960 KBytes
Total Transmission Time	39.299 seconds
Network Bandwidth Utilization	76.591 KBytes/second
Average Round Trip Time	46.1 milliseconds
Average Window Size	9.8 packets
Total Parcels Received	198 parcels
Total Transmission Time	36.922 seconds
Total Parcels Skipped	383 parcels
Network Bandwidth Utilization (14.4 KBytes/parcel)	77.222 KBytes/second
Average Parcel Reception Frequency	5.363 parcels/second

Table 8.9: XUDP Test 00:50, March 17, 1997

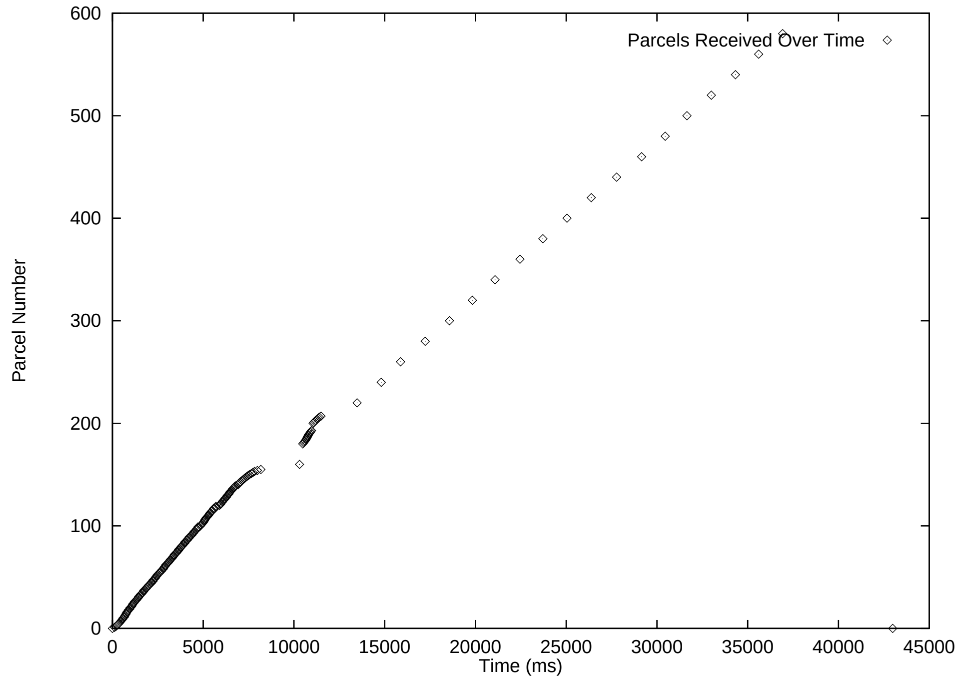

Figure 8.23: Parcels Received (XUDP Test 00:50 3/17/97)

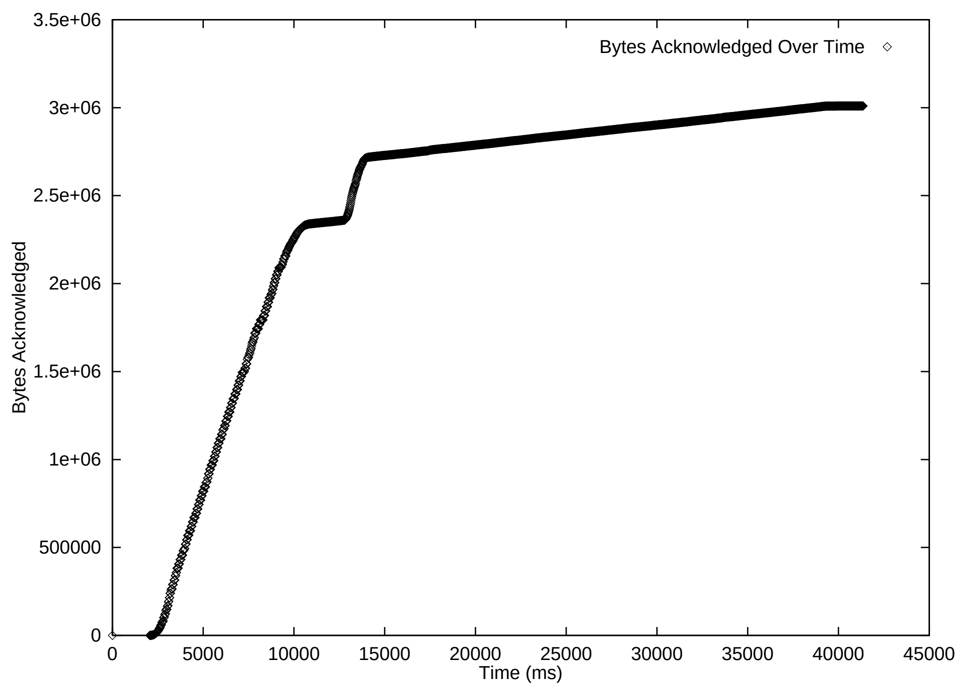

Figure 8.24: Bytes Acknowledged (XUDP Test 00:50 3/17/97)

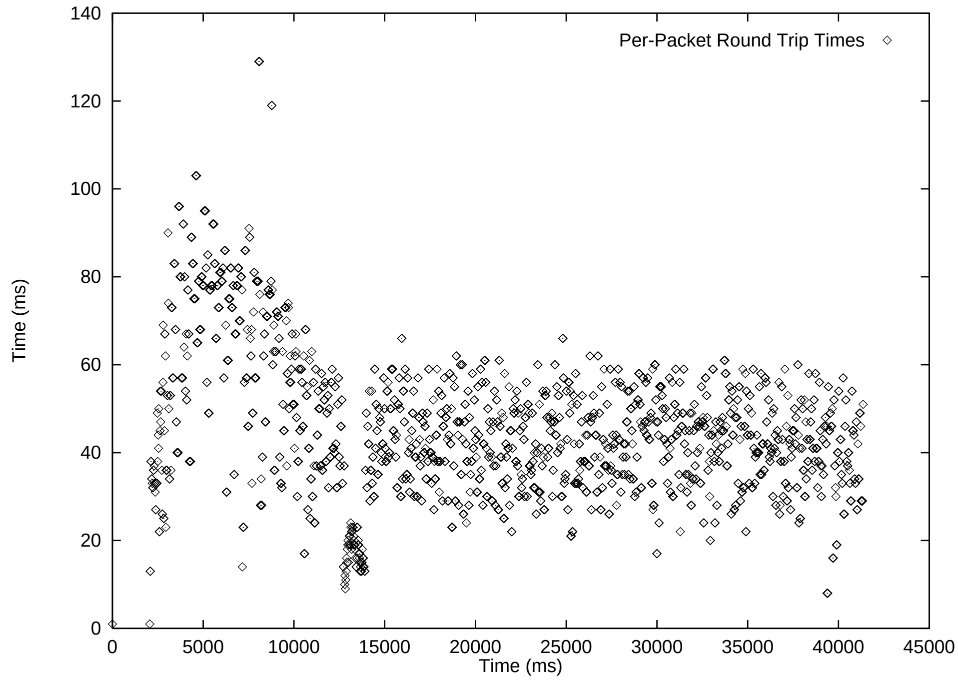

Figure 8.25: Round Trip Times (XUDP Test 00:50 3/17/97)

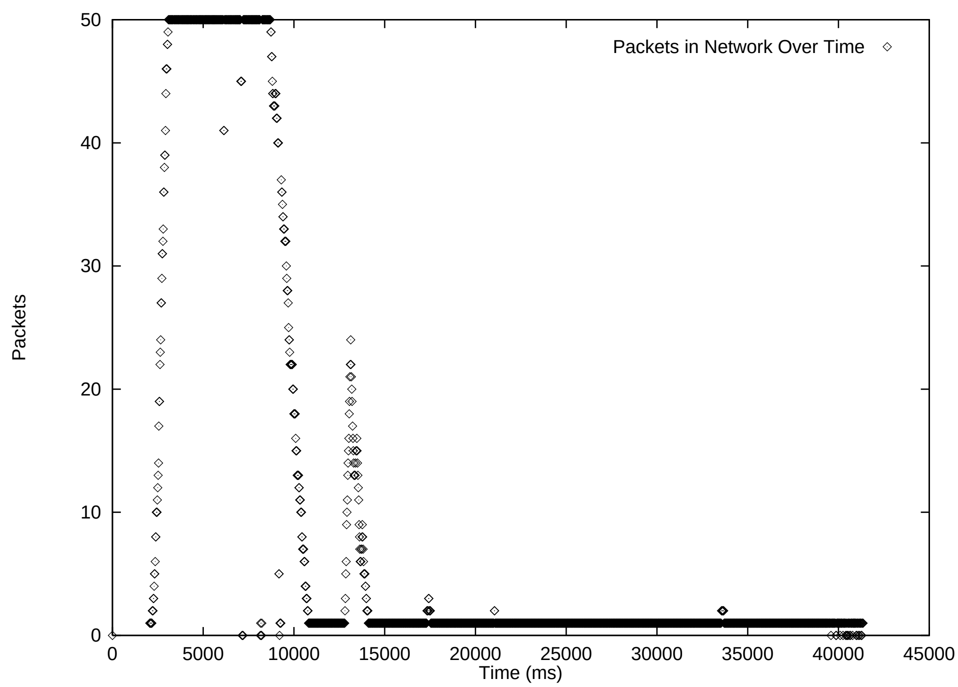

Figure 8.26: Packets In Network (XUDP Test 00:50 3/17/97)

8.7.2 TCP

TCP's transmission of the 30 second video stream completed in 43 seconds, with an average bandwidth utilization of 195KBytes/second (table 8.10), approximately 30KBytes/second lower than that of the unloaded network test.

Measurement	**Result**
Total Parcels Received	581 parcels
Total Transmission Time	42.986 seconds
Total Parcels Skipped	0 parcels
Network Bandwidth Utilization (14.4 KBytes/parcel)	194.631 KBytes/second
Average Parcel Reception Frequency	13.516 parcels/second

Table 8.10: TCP Test 00:48, March 17, 1997

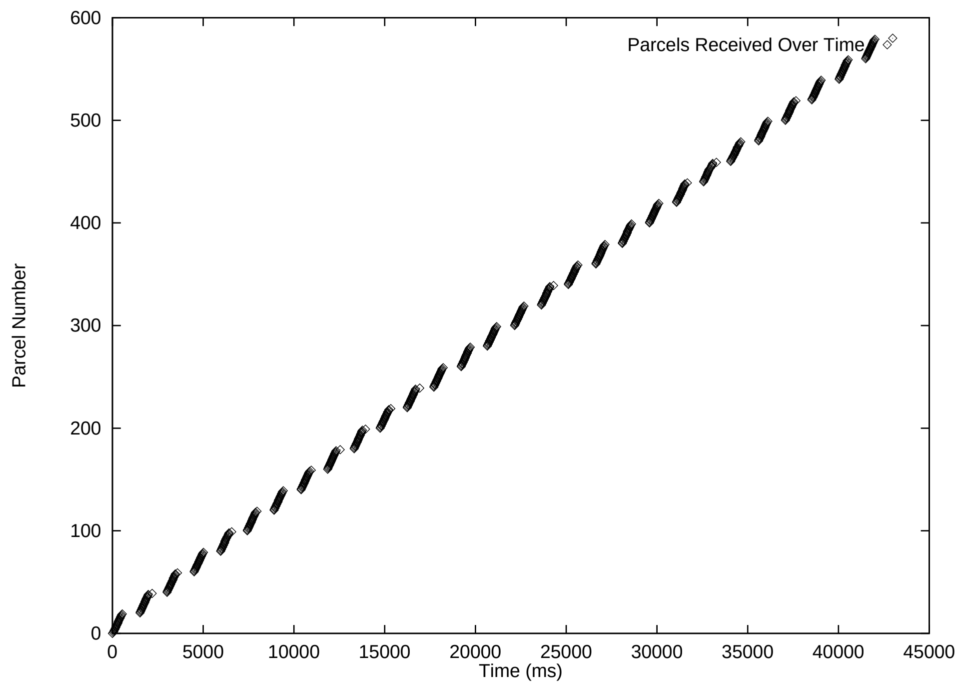

Figure 8.27: Parcels Received (TCP Test 00:48 3/17/97)

8.7.3 Comparison

The comparison graph in figure 8.28 shows that XUDP transmitted a good deal less data than TCP, and did a better job than TCP at preserving the linear progression of the data in time, but still not as good a job as would be desirable.

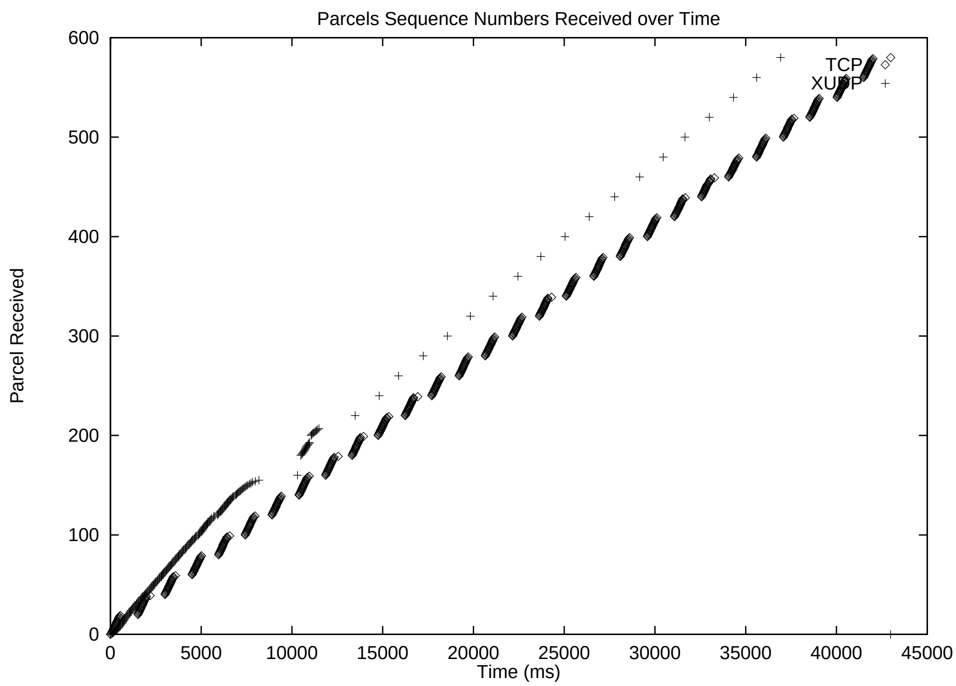

Figure 8.28: Comparison of TCP vs. XUDP Parcels Received (Tests 00:48 and 00:50 3/17/97)

8.8 Simulated 20fps Uncompressed Video Data Under Severe Network Loading, Relieved Near End

This test is similar to the previous test, except the network congestion is relieved manually about three-fourths of the way through the transfer.

8.8.1 XUDP

XUDP received approximately half of the parcels generated by the sending side (table 8.11, with the majority lost during the period of 13 seconds to 28 seconds. The transfer was able to recover nicely once the network congestion was relieved, evidenced by the increased slope and point density in the upper right of the graph in Figure 8.29.

Measurement	**Result**
Total Data Acknowledged	4033.607 KBytes
Total Transmission Time	33.206 seconds
Network Bandwidth Utilization	121.472 KBytes/second
Average Round Trip Time	50.1 milliseconds
Average Window Size	16.3 packets
Total Parcels Received	265 parcels
Total Transmission Time	32.219 seconds
Total Parcels Skipped	316 parcels
Network Bandwidth Utilization (14.4 KBytes/parcel)	118.439 KBytes/second
Average Parcel Reception Frequency	8.225 parcels/second

Table 8.11: XUDP Test 18:54, March 17, 1997

Figure 8.30 shows the plateau in bytes received during the period of time the network was congested.

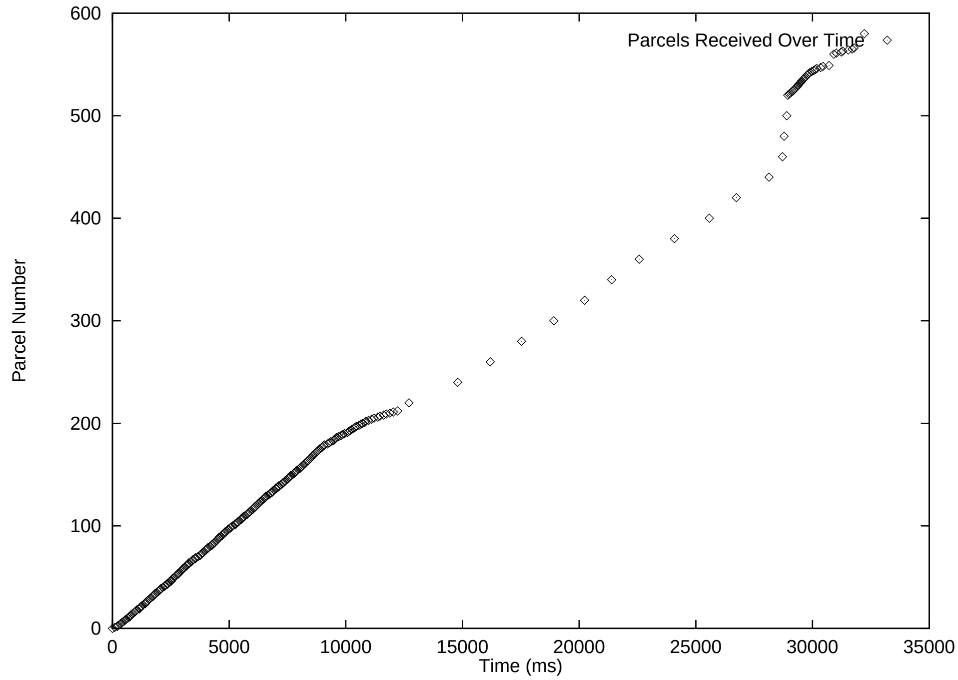

Figure 8.29: Parcels Received (XUDP Test 18:54 3/17/97)

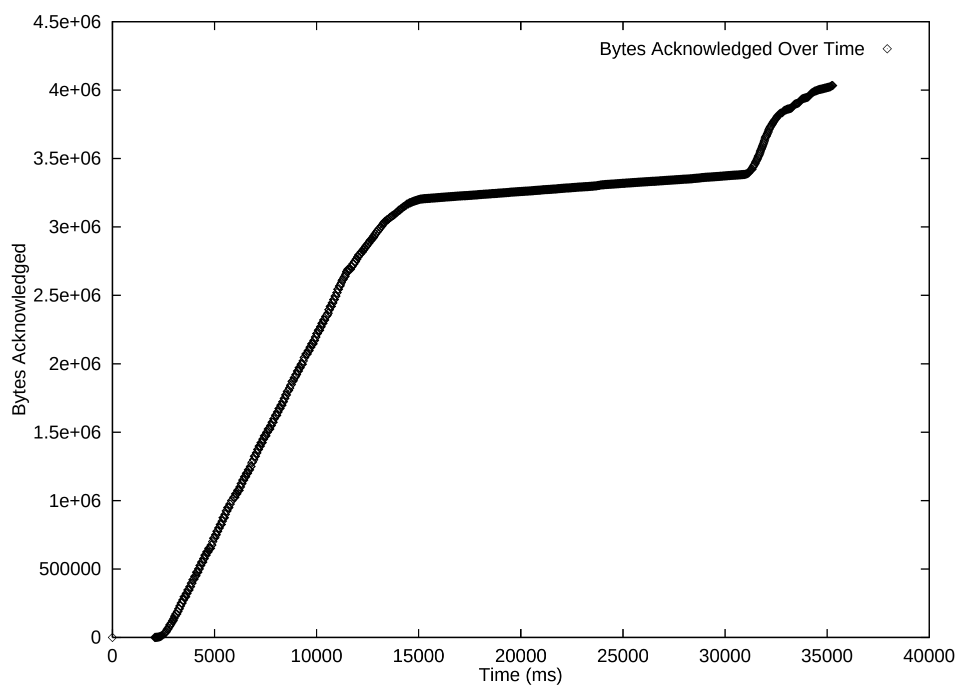

Figure 8.30: Bytes Acknowledged (XUDP Test 18:54 3/17/97)

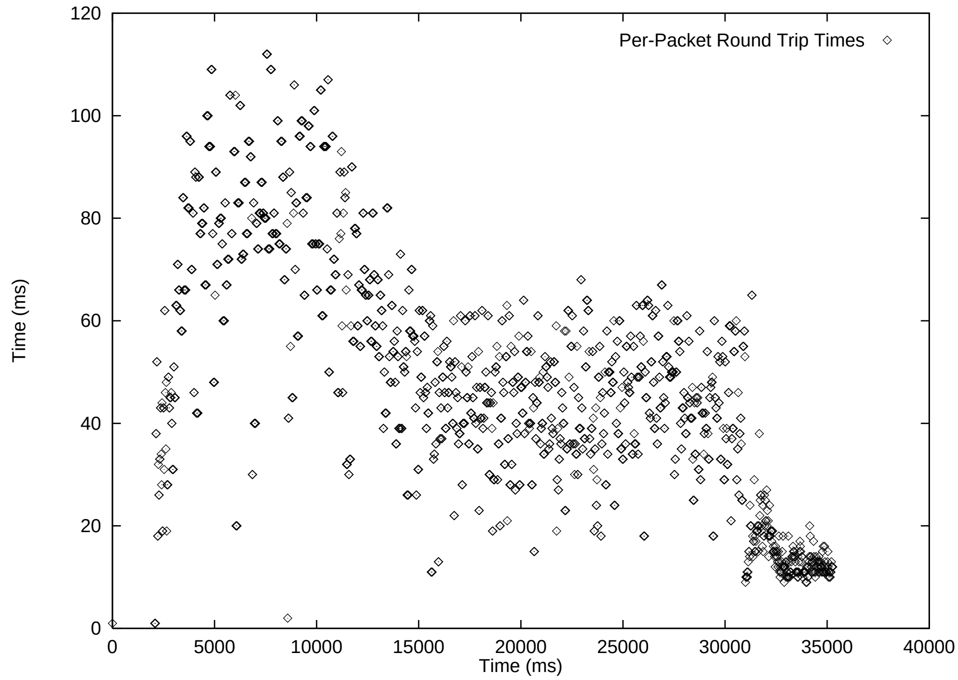

Figure 8.31: Round Trip Times (XUDP Test 18:54 3/17/97)

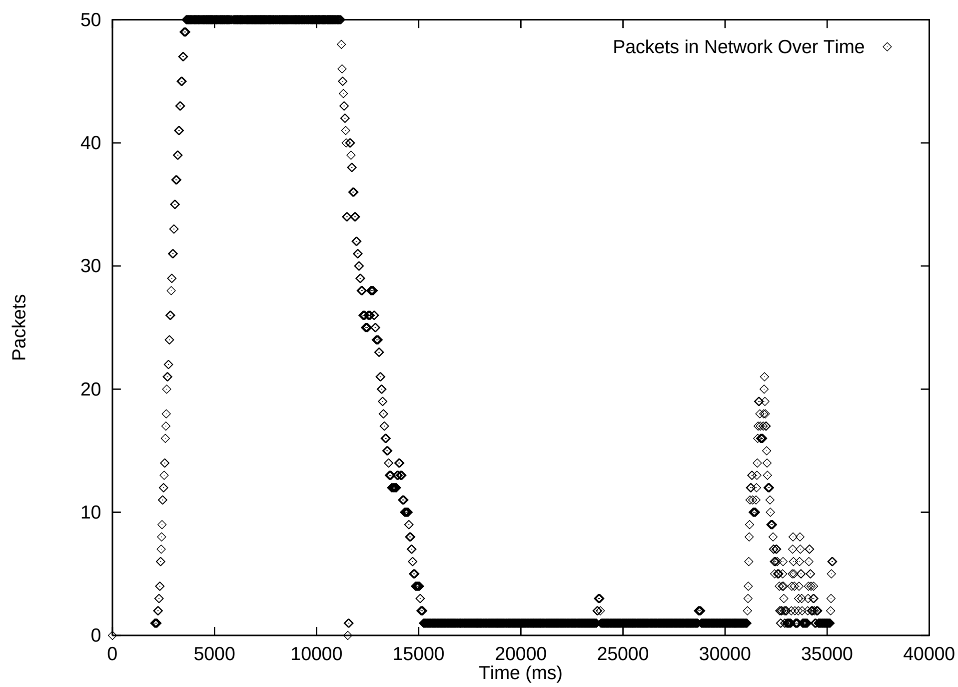

Figure 8.32: Packets In Network (XUDP Test 18:54 3/17/97)

The congestion window (figure 8.32), interestingly did not adjust itself once network congestion had been relieved. The activity from 31 seconds until 35 seconds is due to residual acknowledgments remaining in XUDP's queues (acknowledgments of XUDP command parcels, primarily).

8.8.2 TCP

TCP held close to a 200KBytes/second bandwidth for the duration of the transmission (table 8.12). Note that the slope of the parcel received curve (figure 8.33 rises towards the end of the connection, indicating TCP's response to the increase in available bandwidth.

Measurement	Result
Total Parcels Received	581 parcels
Total Transmission Time	42.571 seconds
Total Parcels Skipped	0 parcels
Network Bandwidth Utilization (14.4 KBytes/parcel)	196.528 KBytes/second
Average Parcel Reception Frequency	13.648 parcels/second

Table 8.12: TCP Test 18:44, March 17, 1997

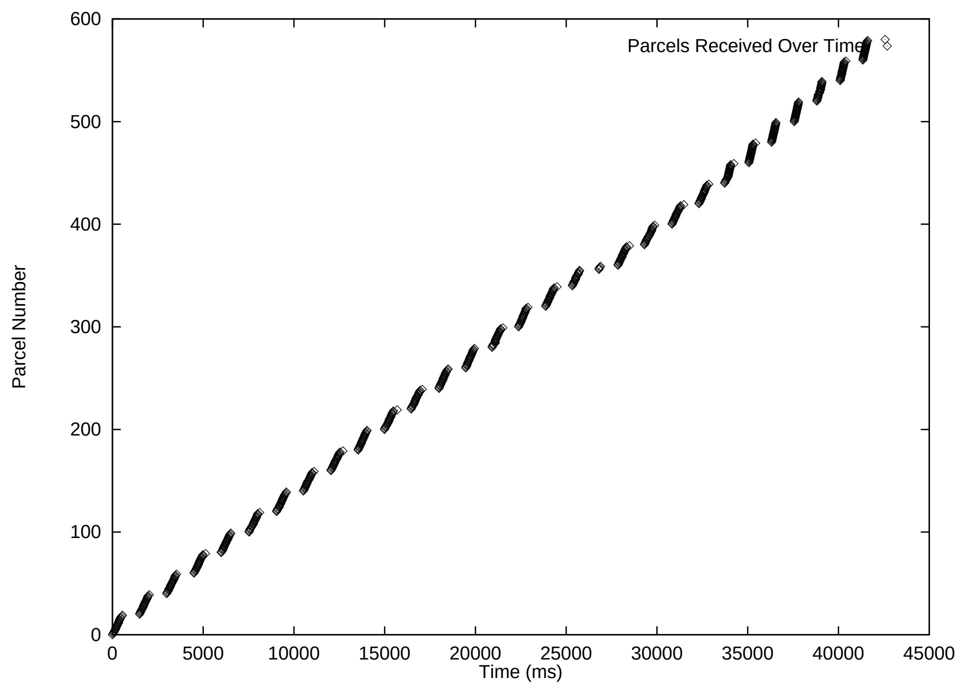

Figure 8.33: Parcels Received (TCP Test 18:44 3/17/97)

8.8.3 Comparison

This example is by far the best in this suite at illustrating the functionality of XUDP's timed-obsolescence algorithm. XUDP's stream is finished in just over 32 seconds, while TCP's stream isn't finished until the 42 second mark - *a ten second difference.* A XUDP application could have presented the XUDP stream unbuffered with satisfactory results, while the TCP's unbuffered stream would, on average, get presented 25% slower!

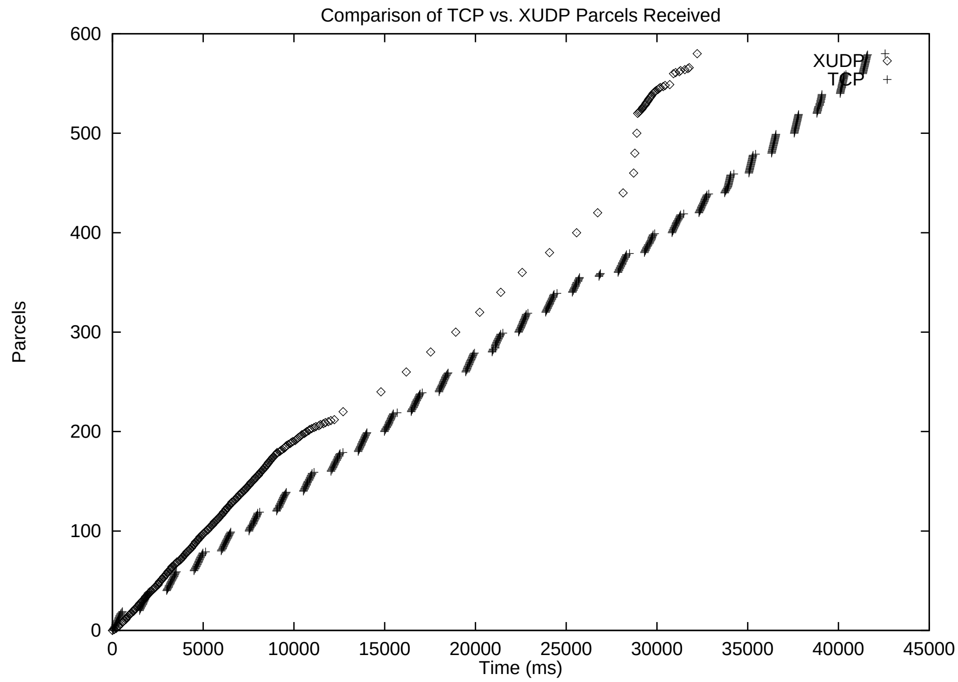

Figure 8.34: Comparison of TCP vs. XUDP Parcels Received (Tests 18:44 and 18:54 3/17/97)

Chapter 9

Conclusions and Directions for Future Work

The main goal of this MQP was to develop an improved method for transporting real-time, multimedia data over the internet. Timed obsolescence provided a means for achieving our goal, but necessitated an entirely new transport layer protocol (XUDP) for a complete implementation. This task was completed, producing an extension to the User Datagram Protocol, with flow control mechanisms similar to that found in the Transmission Control Protocol. As evidenced by the results presented earlier, XUDP successfully preserves the linear time relationships of real-time multimedia data under a variety of network conditions.

However, work remains to be done regarding the optimization of XUDP and the preparation for eventual distribution as a unique contribution to the growing array of multimedia transport layer protocols. As of this stage, much of the code comprising XUDP is designed for easy modification, readability and expansion and is not optimized for speed. Below are some items that can be cleared up in future project work:

- **Minimize context switches in the main loop.** XUDP is constructed around a series of queues that are searched for data to process by a number of functions accessed sequentially from a main program loop. Quite often, the functions are called only to determine that there is no work to be done, and control is returned to the loop. If the queues are checked for data to be processed on the main loop level, costly function calls can be avoided and stack times will be significantly reduced.

- **Making use of hash tables to interface with queues.** Currently in all cases, XUDP uses a linear search through linked-list nodes to locate information. It may be worthwhile to investigate the possibility of the creation of dynamic hash tables that would significantly reduce the amount of searching necessary.

- **Less costly RTT statistical calculations.** The round trip time statistics are calculated with floating point arithmetic on a per-packet basis. *Congestion Avoidance and Control[19]* outlines a method of using integer arithmetic on scaled powers of two, which is cheap and effective on any platform.

- **Theoretically derived RTO and window sizing algorithms** XUDP is using the retransmission timeout with variance estimation algorithm first developed for 4.3BSD TAHOE TCP. Worse, the congestion anticipation window sizing algorithm currently responds too quickly to changes in round trip times. Some time can be spent determining more suitable solutions to these problems.

- **IPC with shared memory.** Parcels are being sent from the user's application to the local XUDP server through UNIX streams. If shared memory handling functions were added, some protocol stack delay time could be eliminated.

- **Implementation directly on top of IP.** XUDP is dependent on UDP for its transport mechanism. An implementation with raw sockets built on top of IP would place XUDP at the same layer as TCP and UDP, further reducing stack times.

Bibliography

[1] Ahn, J. S., Danzig, P. B., Liu, Z., and Yan, L. Evaluation of TCP Vegas: Emulation and Experiment. Tech. rep., University of Southern California, 1995. http://excalibur.usc.edu/research/vegas/doc/vegas.html.

[2] Bakker, R., and van Oorschot, J. Tuning TCP/IP. Tech. rep., Delft University of Technology, 1995. http://dnpap1.et.tudelft.nl/Documents/bones-article/article.html.

[3] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *Computer Communication Review 24*, 4 (Oct 1994), 24–35. ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.ps.

[4] Chiueh, T., and Venkatramani, C. Supporting Real-Time Traffic on Ethernet. Tech. rep., State University of New York at Stony Brook, 1994. http://www.cs.sunysb.edu/ chitra/rether/papers/sigcomm.ps.

[5] Chiueh, T., and Venkatramani, C. Design, Implementation, and Evaluation of a Software-based Real-Time Ethernet Protocol. Tech. rep., State University of New York at Stony Brook, 1995. http://www.cs.sunysb.edu/ chitra/rether/papers/rtss.ps.

[6] Clark, D. D. RFC 813: Window and Acknowledgment Strategy in TCP. ftp://ds.internic.net/rfc/rfc813.txt, 1982.

[7] Clark, D. D. RFC 817: Modularity and Efficiency in Protocol Implementation. ftp://ds.internic.net/rfc/rfc817.txt, July 1982.

[8] Clark, D. D., Jacobson, V., Romkey, J., and Salwen, H. An analysis of tcp processing overhead. *IEEE Communications 29*, 6 (June 1989), 23–29.

[9] Comer, D. E. *Volume I: Principles, Protocols and Architecture*, third ed., vol. 1 of *Internetworking With TCP/IP*. Prentice Hall, Englewood Cliffs, NJ, 1995. ISBN 0-13-216987-8.

[10] Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., and Lumley, J. Afterburner. *IEEE Network 7*, 4 (July 1993), 36–43.

[11] Deering, S. E. Multicast routing in internetworks and extended lans. In *SIGCOMM '88* (Stanford, CA, Aug 1988).

[12] Deering, S. E. RFC 1112: Host extensions for IP multicasting. ftp://ds.internic.net/rfc/rfc1112.txt, Aug 1989.

[13] Deering, S. E. Multicast Routing in a Datagram Internetwork. Tech. rep., Stanford University, 1991. ftp://gregorio.stanford.edu/vmtp-ip/, Report no. STAN-CS-92-1415.

[14] Doeringer, W., Dykeman, D., Kaiserswerth, M., Meister, B. W., Rudin, H., and Williamson, R. A survey of light-weight transport protocols for high-speed networks. *IEEE Transactions on Communications 38*, 11 (Nov 1990), 2025–2039.

[15] Hazzard, B. K. Evaluation of Optimal Socket Buffer Sizes for the Internet Protocol Suite (draft). Tech. rep., Worcester Polytechnic Institute, 1996. http://vision3.WPI.EDU:8080/TCP/Documents/BuffEval/buffEval.html.

[16] Hunt, C. *TCP/IP Network Administration*, fourth ed. O'Reilly & Associates, Inc., Sebastopol, CA, 1992. ISBN 0-937175-82-X.

[17] Hutchinson, N. C., and Peterson, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan 1991), 64–76. ftp://ftp.cs.arizona.edu/xkernel/Papers/architecture.ps.

[18] Jacobson, V. The van jacobson classic messages on fast recovery/retransmit. Courtesy of Bob Braden. 1987-1990 email, collected in message ¡199407201713.AA20911 @zephyr.isi.edu¿, end2end-interest archives.

[19] Jacobson, V. Congestion avoidance and control. In *SIGCOMM '88* (Palo Alto, CA, Aug 1988). ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[20] Jacobson, V. Performance. Usenet, comp.protocols.tcp-ip Newsgroup, Message-ID ¡8811231121.AA19744@helios.ee.lbl.gov¿, Nov 1988.

[21] Jacobson, V. Re: maximum ethernet throughput. comp.protocols.tcp-ip Newsgroup, Mar 1988.

[22] Jacobson, V. Berkeley tcp evolution from 4.3-tahoe to 4.3-reno. In *Proceedings of the Eighteenth Internet Engineering Task Force* (University of British Colmbia, Vancouver, B.C., Sep 1990), p. 365.

[23] Jacobson, V. Modified tcp congestion avoidance algorithm. end2end-interest mailing list, Apr 1990.

[24] Jacobson, V. A conferencing architecture for 'light-weight sessions' (viewgraphs). In *MICE seminar series* (Nov 1993). ftp://ftp.ee.lbl.gov/talks/vj-lwsarch.ps.Z.

[25] Jacobson, V. Flexible, efficient resource management for datagram networks (viewgraphs), Oct 1993. ftp://ftp.ee.lbl.gov/talks/vj-cbq.ps.Z.

[26] Jacobson, V. Lightweight sessions - a new architecture for realtime applications and protocols (viewgraphs). In *Networkshop '93* (Melbourne, Australia, Nov 1993). ftp://ftp.ee.lbl.gov/talks/vj-nws93-2.ps.Z.

[27] Jacobson, V. Some design issues for high-speed networks. In *Networkshop '93* (Melbourne, Australia, Nov 1993). ftp://ftp.ee.lbl.gov/talks/vj-nws93-1.ps.Z.

[28] Jacobson, V., Braden, R. T., and Borman, D. A. RFC 1323: TCP Extensions for High Performance. ftp://ds.internic.net/rfc/rfc1323.txt, May 1992.

[29] Jacobson, V., and McCanne, S. The bsd packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX Technical Conference* (San Diego, CA, Jan 1993). ftp://ftp.ee.lbl.gov/papers/bpf-usenix93.ps.Z.

[30] Jain, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling.* Wiley, NY, 1991.

[31] Kale, C., and Socolofsky, T. RFC 1180: A TCP/IP Tutorial. ftp://ds.internic.net/rfc/rfc1180.txt, Jan 1991.

[32] Karn, P., and Partridge, C. Improving round-trip time estimates in reliable transport protocols. *Computer Communication Review 17*, 5 (Aug 1987), 2–7. ftp://sics.se/users/craig/karn-partridge.ps.

[33] Kay, J., and Pasquale, J. The importance of non-data touching processing overheads in tcp/ip. In *Proceedings of ACM SIGCOMM '93* (San Francisco, CA, Sep 1993), pp. 259–268.

[34] Mogul, J. C. IP Network Performance. In *Internet System Handbook*, D. C. Lynch and M. T. Rose, Eds. Addison-Wesley, Reading, MA, 1993, pp. 575–675.

[35] Papadopoulos, C., and Parulkar, G. Experimental evaluation of sunos ipc and tcp/ip protocol implementation. *IEEE/ACM Transactions on Networking 1*, 2 (Apr 1993). http://www.ccrc.wustl.edu/ christos/PostScriptDocs/ToN93-paper.ps.Z.

[36] Parker, T. *Teach Yourself TCP/IP in 14 Days.* Sams, Indianapolis, IN, 1994. ISBN 0-672-30549-6.

[37] Partridge, C. *Gigabit Networking.* Addison-Wesley, Reading, MA, 1994. ISBN 0-201-56333-9.

[38] Partridge, C., and Pink, S. A faster udp. *IEEE/ACM Transactions on Networking 1*, 4 (Aug 1993), 429–440.

[39] Parulkar, G. Protocols for Network Computing. Tech. rep., Washington University, 1994. http://arl1.wustl.edu/ guru/g-troup-short-fix/g-troup-short-fix.html.

[40] Postel, J. B. RFC 791: Internet Protocol. ftp://ds.internic.net/rfc/rfc791.txt, Sept 1981.

[41] Postel, J. B. RFC 793: Transmission Control Protocol. ftp://ds.internic.net/rfc/rfc793.txt, Sept 1981.

[42] Stevens, W. R. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1990. ISBN 0-13-949876-1.

[43] Stevens, W. R., and Wright, G. R. *Volume II: The Implementation*, vol. 2 of *TCP/IP Illustrated.* Addison Wesley, Reading, MA, 1995. ISBN 0-201-63354-X.

[44] Tannenbaum, A. S. *Computer Networks*, third ed. Prentice Hall, Upper Saddle River, NJ 07458, 1996. IBSN 0-13-349945-6.

[45] Weaver, A. C. *Xpress Transport Protocol Version 4.* University of Virginia, Charlottesville, VA 22903.

[46] Wortelboer, J. Snifd, a network measurement tool Survey on the development of Snifd, . Tech. rep., Delft University of Technology, 1995. http://dnpap1.et.tudelft.nl/Documents/snifd_survey/snif.html.