Timer Management in Sandia XTP

Yonglin Jiang

A Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 8, 2001

0-612-59328-2

Abstract

Timer Management in Sandia XTP

Yonglin Jiang, Master

Concordia University, 2001

In most network communication protocols, the timer is used as an efficient way to handle various expected and unexpected events. The timer management plays a very important role in the protocol implementation. Also, there are many ways to process the protocol timers. The purpose of this report is to find a protocol timer process mechanism which will improve the implementation performance of the Xpress Transport Protocol (XTP) - Sandia XTP, especially with XTP rate control. Based on previous work (Louis Harvey: In search of a rate control policy for XTP: unicast & multicast) and the analysis of the Sandia XTP implementation, the new timer management is proposed and implemented with Sandia XTP. Some experiments that focus on XTP rate control showed that the new timer management in Sandia XTP improved the XTP rate control and performance.

Acknowledgements

First, I would like to express my sincere gratitude to Dr. J. W. Atwood, my major report supervisor, for his guidance and valuable insight throughout this report. His wisdom and infinite patience have given me great support to finish this report.

Next, I would like to thank Dr. Gregory Butler for his consultations, advice and suggestions which were always of great help.

I am very grateful to have a chance to work in the HSPL (High Speed Protocol Laboratory) of the Computer Science Department. Here, I met several smart people: Louis Harvey, Tian Fang, Tong Ma. I would thank them for their help. Without their help, many things in this report would have taken a very long time. And also I will remember the good times we had together.

Finally, I would like to thank my family: my wife, my proud and cute sons Michel and Joshua, for their unending support and encouragement.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context of the Study

The Xpress Transport Protocol (XTP, website:http://www.ca.sandia.gov/xtp) is a next generation transport protocol with its high speed, reliability, multicast and QoS that has attracted much attention for research and development. At the Sandia National Laboratories, an XTP implementation, written in C++, has been developed. Version 1.5.1 forms the basis for this report. In this version, almost all of the functionality of XTP has been implemented. Concordia University Computer Science HSPL (High Speed Protocol Laboratory) has been working with XTP for many years under the direction of Dr. J. W. Atwood. The research done by Louis Harvey in Concordia HSPL shows that the Sandia XTP 1.5.1 has a problem with XTP rate control. To solve this problem, a new timer management class is introduced, which focuses on the management of the timers used in XTP, and gives a proficient solution. In this document, I describe the detailed XTP modifications and test results based on the new timer management class.

1.2 Report Organization

This report presents the following issues: background of XTP, Sandia XTP implementation, the shortcoming of Sandia XTP rate control, the time management solution, the revision of implementation, and the testing.

In the following, Chapter 2 gives an introduction to the Xpress Transport Protocol (XTP) in general and the necessary related XTP information to understand this report. Chapter 3 presents how the Sandia XTP 1.5.1 is implemented, and the overview of the MTL (Meta Transport Library) and the XTP derivation from MTL. Based on Sandia XTP implementation 1.5.1, there is some discussion about Sandia XTP shortcoming - protocol timer management. To better understand the XTP timer management, the XTP protocol timer usages are presented in more detail. Next, Chapter 4 covers the new timer management prototype and implementation with MTL and XTP based on the discussion in Chapter 3. Also, the Sandia XTP 1.5.1 modifications with new timer management are presented. Chapter 5 will give the experimental results with new timer management Sandia XTP implementation. Finally, Chapter 6 will provide a summary of this report and further work in this direction. An ordered bibliographical index and references, and the listing of related programs close this report.

2 The Xpress Transport Protocol (XTP)

2.1 XTP Overview

Xpress Transport Protocol (XTP) specification and conceptual design was first proposed in 1985. It aims to improve the transport protocol reliability, speed, multicast capabilities and quality of service (QoS). After over ten years of development, XTP today is more specifically designed to embrace high-speed networks and multimedia applications. With efficient error recovery mechanisms, XTP provides significantly higher performance over today's networking infrastructure as a transparent replacement for existing networking protocols such as TCP. Multicast / Unicast feature also allows the user to have more choice and save the communication bandwidth to reduce network congestion. In addition, XTP provides features and services unavailable through other protocols, such as rate control.

Compliant with the OSI-7 layer model, XTP addresses and contributes in the following fields as transport layer protocol:

- Orthogonal protocol functions for separating paradigm from policy
- Separation of rate and flow control
- Explicit first-class support for reliable multicast
- Data delivery service independence

The other features of XTP include: implicit fast connection setup for virtual circuit paradigm, key-based addressing lookup, message priority and scheduling, support for encapsulation and convergence protocols, selective retransmission and acknowledgement, fixed-size 64-bit aligned frame design, 64-bit sequence and connection identifiers, parameterized traffic and quality of service negotiation.

Although the XTP is a complicated and high performance protocol, the communication model is very simple. The following presents the XTP Unicast communication model:

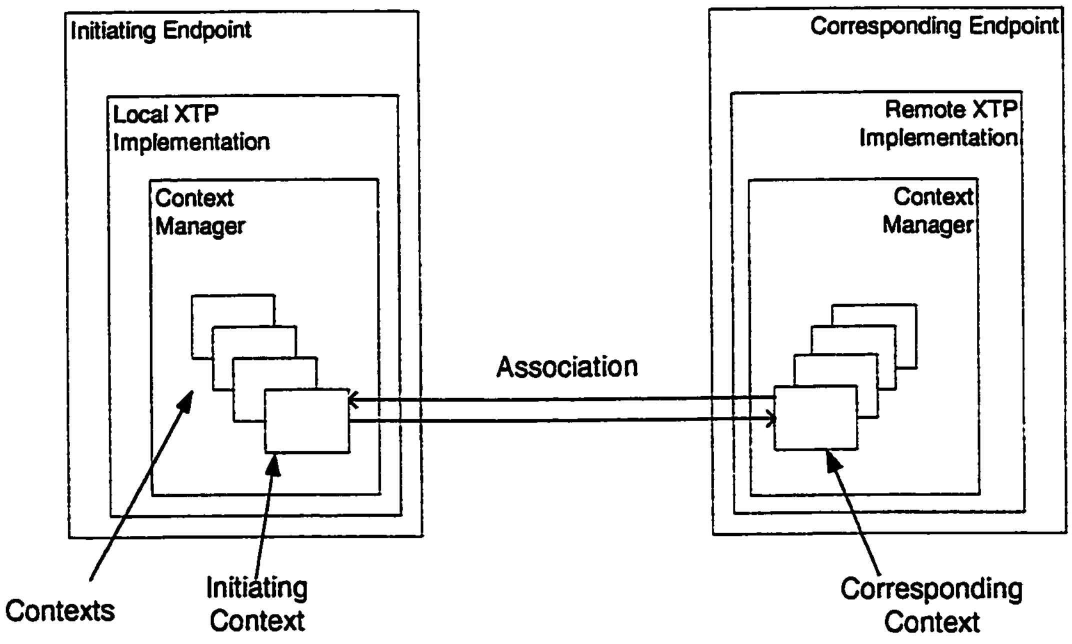

Figure 1: XTP communication model [XTP Revision 4.0b. July 1998]

context:	The collection of information comprising the XTP state at an end-system. It manages both an outgoing data stream and an incoming data stream.
association:	The aggregate of active contexts and the data stream between them. When two contexts start data exchange, the association is set up. When the last packet exchange is finished, the association is destroyed.

In XTP communication model, each XTP endpoint has a local XTP implementation, which is responsible for data exchange between XTP endpoints. Inside the XTP implementation, the contexts are active with unique keys to perform XTP protocol. All contexts are under the management of the context manager.

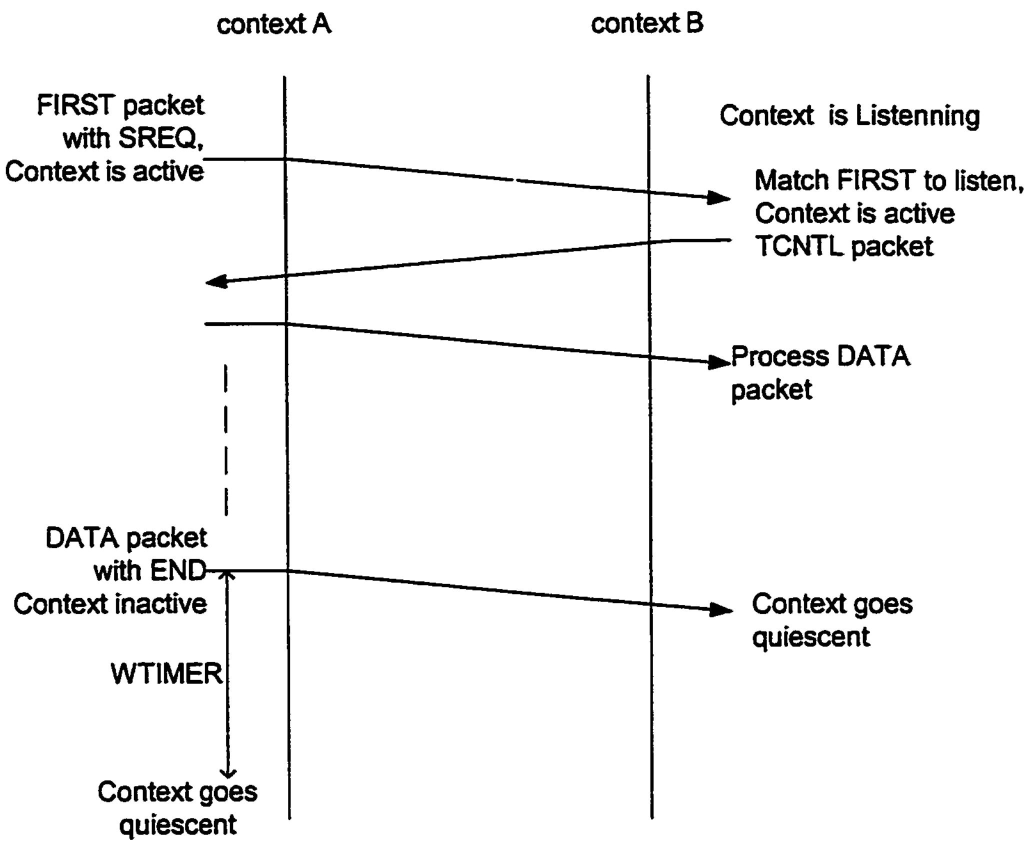

Figure 2: Typical context behaviors of data transfer with no error

The context is always in one of the following states: Quiescent, Listening, Active, and Inactive. Based on XTP user request, the contexts change their state. For the Unicast case, the XTP receiver application always starts first, and the receiver goes to the Listening state from the Quiescent state and waits for the sender packet. The sender issues output command (FIRST packet), then goes to the Active state. After the FIRST packet is accepted by the receiver, the receiver enters the Active state. In that time, the association is established, the sender and the receiver can exchange the well-defined packets freely. When the receiver receives a packet with END flag, the receiver goes back to the Quiescent state. For the sender, after sending the END flag, it enters the Inactive state, finally goes to the Quiescent state. The association is terminated.

2.2 Multcast/Unicast

Like other transport protocols, the XTP provides Unicast support by establishing the connection between two endpoints. XTP Unicast provides a high degree of functionality through orthogonal protocol mechanisms. These mechanisms are in the form of fields and bit flags used during packet exchanges over the lifetime of an association. Association management procedures define how fields and bitflags are used during the lifecycle of the association.

Multicast is a major distinctive feature of XTP. It provides duplicate free data exchange from a single XTP context to a set of XTP receiver contexts. XTP Multicast provides a powerful mechanism for group communication that supports a data service for one-to-many communication. Many-to-many extensions to XTP have been proposed by Ramasinvan [Ramasivan 2000], and are being implemented by Ma [Ma 2001]. XTP Multicast uses the same control algorithms and mechanisms in flow control and rate control as are used for Unicast. The only difference is in the association management. The Multicast association is closely related to the management of a group of receivers. The following are the syntax rules for Multicast:

- Multicast packet

  All packets have the MULTI bit set. The packet from transmitter uses the multicast (group) address. The packet from the receiver uses the transmitter unicast address.

- Multicast receiver join

  Multicast receiver has two ways to join the multicast receivers group: the first is when a listening context receives a multicast FIRST packet. The second is when a receiver sends a JCNTL packet to the transmitter, which responds with a JCNTL packet.

- Multicast address

  XTP is designed to run in different types of network, so it does not define its own network address. It uses the addresses provided by the underlying data delivery service, such as IP address, IPX Address, in its "Address Segment". In order to use XTP multicast, the underlying data delivery service must provide XTP with a multicast or broadcast service. XTP does not define how to assign a multicast group address for the service. So when XTP multicast is requested, the underlying multicast address must be provided.

2.3 Flow Control

XTP flow control uses a regular end-to-end windowing flow control mechanism. Like TCP/IP, XTP flow control aims to prevent swamping a slow receiver with too much data too quickly by transferring control to the receiver who then issues credits to the sender. XTP flow control uses a 64-bit sequence number and a 64-bit sliding window

- XTP reserves two field *alloc* and *rseq* to negotiate the flow control window value.
- Another option for flow control is RES bit and NOFLOW bit. XTP provides user to have choice to setup:

By setting RES bit, the sender instructs receiver to advertise only the actual buffer allocated by the user for the context. This is called reservation mode. In this mode, the receiver is forced to adopt a conservative policy making sure that no packets will be lost due to lack of buffer space.

The NOFLOW bit indicates to the receiver that the transmitter does not wish to adhere to flow control constraints, so flow control in the forward direction will be disabled.

2.4 Rate Control

Unlike flow control, rate control focuses on the relationship of producer/consumer between the XTP end points and considers processor speed and congestion.

The XTP performs rate control by several factors: Rate, Credit and Burst, and by a refresh timer called RTIMER. The relationship among them is the following:

- Rate: the Rate is user expected data output rate.
- Burst: the Burst is another user input parameter, it is the maximum number of data bytes that can be sent at once.
- RTIMER: RTIMER is equal to the value of the Burst divided by the Rate. It is a period timer.
- Credit: Each time when context wants to send data out, it will check the value of Credit. The Credit will decrease to zero while sending data. If no more credit is available, the data packet will put on an out packet FIFO and no data is sent out.

Basically, the context will be assigned new Credit value (the value is equal to Burst) at each RTIMER time period.

2.5 Error Control

In XTP, each packet carries the checksum in the check field. Here the checksum is a 16-bit one's complement sum over all octet pairs. XTP provides a choice to perform only packet header checksum with NOCHECK bit setting, and to perform full packet checksum without NOCHECK bit setting.

If a check error is found, the packet is simply dropped. A receiver will detect a lost packet by checking the incoming packet stream for a gap in the sequence space. In this case, an ECNTL packet is sent back to sender to require the retransmission.

For the sender, if it fails to receive the request acknowledge packet, it starts a Synchronization Handshake. The two timers WTIMER and CTIMEOUT (Refer to section 3.3 for more details) are used to help the sender to detect if the association can be recovered to normal. If the handshake fails, the association is terminated. The communication between sender and receiver is aborted.

3 Sandia XTP Implementation

3.1 Sandia XTP OMT Structure

As one XTP implementation, Sandia XTP is an object oriented implementation of XTP 4.0. The core of base classes used in Sandia XTP comes from the Meta-Transport Library software (also developed at Sandia). Sandia XTP is actually a set of classes derived from MTL base classes. The Meta-Transport Library is a collection of reusable C++ base classes from which specific transport layer protocols can be derived. The object of MTL is to distill the transport protocol commonalties into a set of bases classes.

3.1.1 MTL (Meta Transport Library)

The MTL (Meta Transport Library) is a protocol base class library which eases the building of transport protocols. It covers the following transport protocol common components: some fundamental units for information exchange, data structures to hold each endpoint's state, a processing mechanism to parse incoming packets, a mechanism to construct outgoing packets, and a mechanism to handle service requests.

The Sandia recent MTL version 1.5.1 architecture is shown in figure 3.

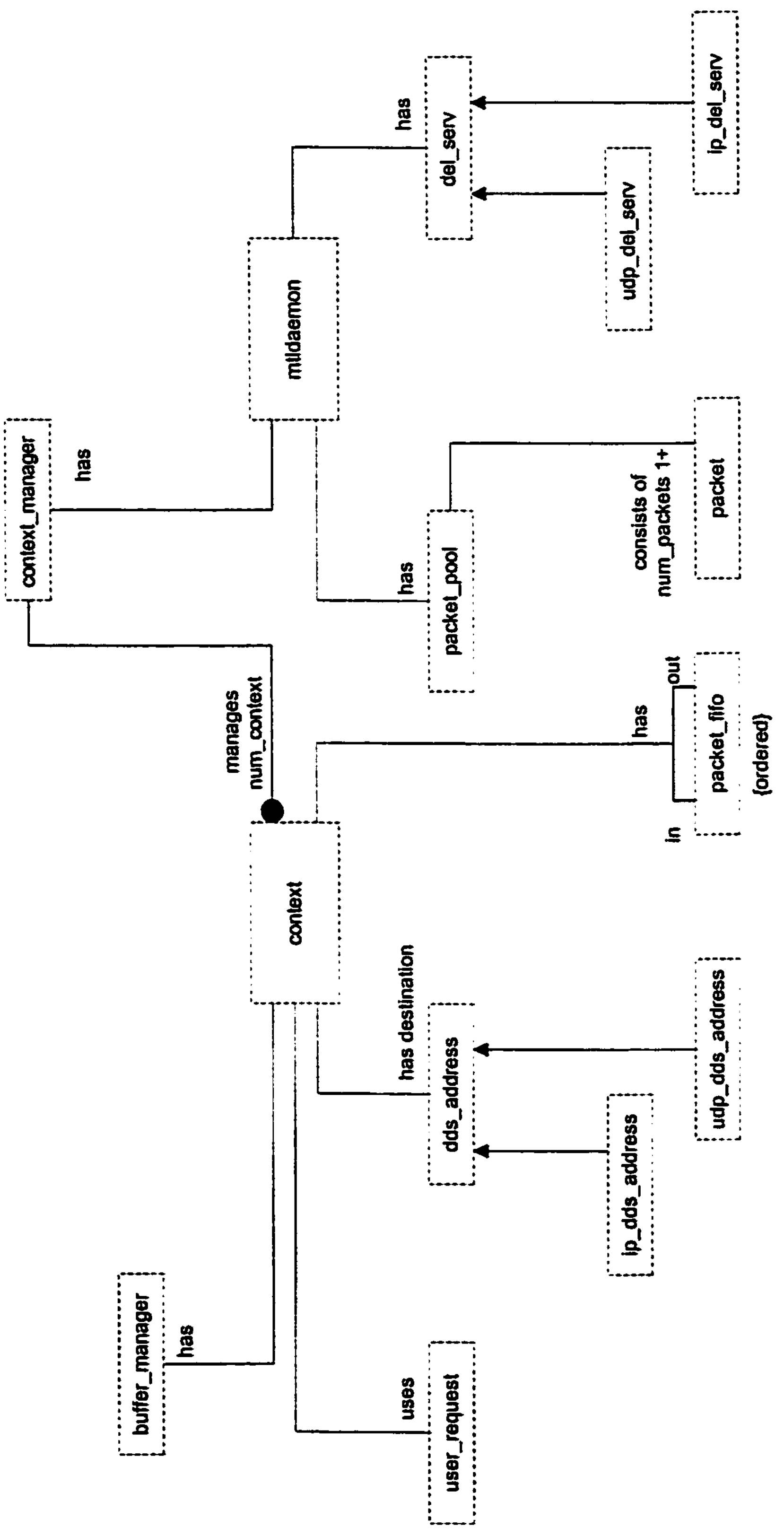

Figure 3: MTL class diagram

- Packets and Packet Manipulators

packet class

Packets are the vehicle for data and information exchange between endpoints. Because the derived protocol defines the structure of its packets, the packet class provided in the MTL does not impose a structure on the packet, but rather provides a packet shell. The packet shell can be manipulated by both MTL manipulator classes (packet_pool and packet_fifo) and the derived protocol.

The packet class has three major functions: as a repository of data, send data via data delivery service, receive data via data delivery service.

packet_pool class

Packet objects are managed by a packet pool object. The packet_pool is a general repository for packet objects. The packet_pool is responsible for allocating all of the packets in the system, and deallocating them when the daemon terminates.

packet_fifo class

The packet_fifo maintains a packet FIFO. Each context has two FIFO, one for receiving and another for sending. When context receives a user request to send packet out without credit, the packet will be put to the sending packet_fifo. Before sending a new user packet, the sending packet_fifo will be checked first. When a context receives packets and for some reason does not process it, this packet will be held by the receiving packet_fifo.

- Contexts and the Context Manger

context class

> A context is the collection of all state information for an endpoint of an association. Most context information is protocol-specific, so there is only some state information that is common to all transport protocols in MTL.

context_manager class

> The context_manager is the container class for all of the contexts in a protocol. The main function is to match the user requests and incoming packet to the appropriate context. In context_managner, two context links are maintained: free context link and active context link.

- Data Delivery

The data delivery service is provided by class del_srv, which is an abstract class specifying the interface to data delivery. Each particular data delivery service is derived from this class. So far, the MTL provides IP and UDP service. The IP data delivery service object is ip_del_srv, and the UDP data delivery service object is udp_del_srv. The delivery service to be used will be specified when XTP daemon starts.

- User Interface

The MTL user interface is provided by the mtlif class, which allows a user protocol application to interface with the protocol daemon. The user application can send request and data packet to the daemon, it also can receive data from the daemon via this class. To reduce the cost of data exchange between the application and daemon, the class buffer_manager is introduced. The buffer_manager will handle the data exchanges between the application and the daemon. In MTL, the buffer_manager uses the UNIX IPC mechanism shared memory to perform the data exchange.

- Daemon

The daemon integrates all classes and provides the protocol service to the user. In MTL, there is one base class for the daemon: mtldaemon. The main function of this class includes initialization, producing the main loop waiting for user requests and incoming packets.

The MTL mtldaemon main_loop architecture is shown in figure 4. In the daemon main_loop, the Unix system function select() is called as the key step. It lets the daemon sleep or wakes up the daemon when an event occurs.

A simple description about select() is the following:

```
int select(int nfds,
           fd_set *readset, fd_set *writeset,
           fd_set *exceptset,
           struct timeval *timeout);
```

where

nfds

the number of FDs (file descriptors) to examine; this must be greater than the largest FD (file descriptor) in any of the fdsets, *not* the actual number of FDs (file descriptors) specified

readset

the set of FDs (file descriptors) to examine for readability

writeset

the set of FDs (file descriptors) to examine for writability

exceptfds

the set of FDs (file descriptors) to examine for exceptional status (note: errors are *not* exceptional statuses)

timeout

NULL for infinite timeout, or points to a timeval specifying the maximum wait time (if tv_sec and tv_usec both equal zero, then the status of the FDs is polled, but the call never blocks)

The call returns the number of `ready' FDs found, and the three fdsets are modified in-place, with only the ready FDs left in the sets.

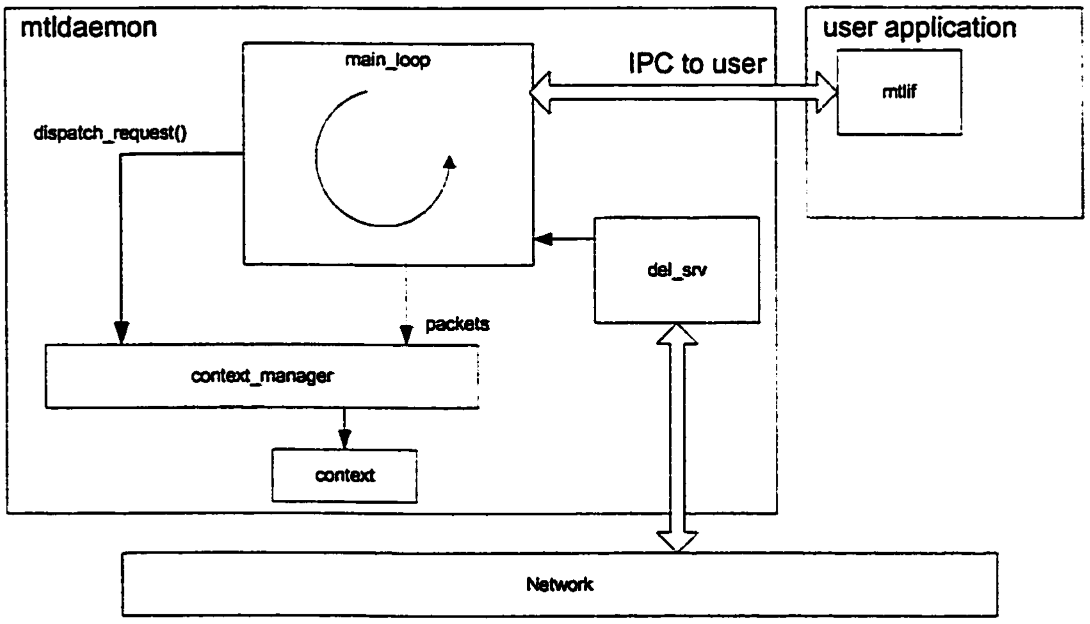

Figure 4: MTL daemon main loop architecture

3.1.2 Sandia XTP

The Sandia XTP implementation of XTP 4.0 is a protocol derived from the base MTL classes. The protocol procedures are implemented in the class XTPcontext, which is derived from the MTL context base class. The packet structures actually form a hierarchy of classes; XTPpacket is derived from the MTL base class packet, and FIRSTpacket, CTNLpacket, etc., are derived from XTPpacket. The XTP class diagram is given in figure 5, on next page.

- Sandia XTP Packets

Packets are structured bins of information that are sent by a delivery service that treats the contents of the packet as uninterpreted payload. A protocol defines these packet structures, and extracts the information from the packets only with knowledge of the structure. As a consequence, the raw payload that is received is cast into some structure so the data can be meaningful. In MTL, the packet base class is simply a byte array with the size of the maximum protocol data unit. There are two base methods, one for getting the address of the start of the packet, and one for sending the packet. There is no receive method, since receiving is not actually done to a packet in the same way that sending is. A protocol-specific packet derived class would specify such methods as value insertion and extraction, and possibly a checksum function.

The derived class XTPpacket adds several XTP-specific methods. Since all XTP packets share a common header, XTPpacket has methods for header placement and extraction, as well as the placement and extraction of several specific header fields. Also, because the size of the header is fixed, there is a method that returns a pointer to the middle part of the packet.

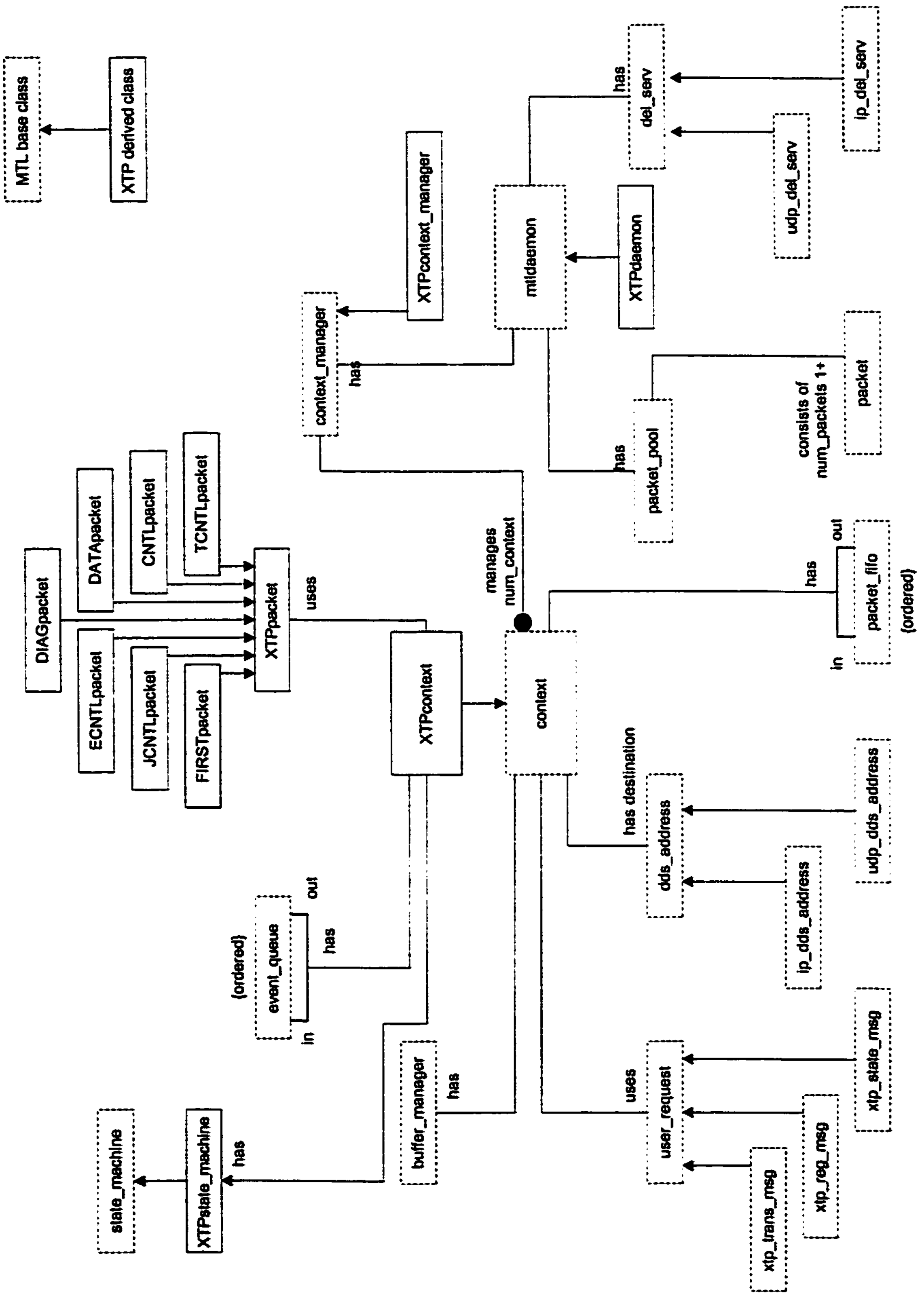

Figure 5: XTP class diagram

The specific XTP packet types are derived from the XTPpacket class. In particular, the FIRSTpacket class has methods for address placement and extraction, the DATApacket class has methods for data insertion, and the control packet classes, CNTLpacket, ECNTLpacket, and TCNTLpacket, have methods for specific control segment placement and extraction.

- Sandia XTP Context and Context Management

The context class holds all endpoint state information. Methods include state transitions, packet parsing, and user request satisfaction. The context also has two buffer managers—one for the send buffer and one for the receive buffer. The context manager class contains and manipulates all of the contexts, and is responsible for associating user requests and newly arrived packets with the proper context. When a packet arrives, the context manager examines the packet only enough to figure out which context should get the packet. The packet is placed on the receive packet fifo (an object owned by the target context) so that the context, when it is allowed to satisfy any outstanding work, can retrieve the packet and process it.

Within the XTPcontext derived class are methods for parsing each type of incoming packet, and responding to the directives contained within that packet. When an XTP context needs to send a packet, an XTPcontext method, send(), constructs either a FIRST or DATA packet and calls the packet's send() method. If a control packet is required during protocol processing, the method send_cntl() constructs the appropriate packet and has it sent. When packets arrive, the method process_packet() does some common packet processing, then determines which type of packet is being parsed. From here type-specific packet parsing methods are called to finish the processing.

3.2 Sandia XTP Shortcomings

Research into XTP has been conducted in the High Speed Protocols Laboratory (HSPL) of the Department of Computer Science for many years, under the director of Dr. J. W. Atwood. In recent years, considerable experimentation has been done on the performance of Sandia XTP.

Taking advantage of object oriented techniques, Sandia XTP provides an excellent vehicle for experimentation with a modern protocol. However, certain aspects of the design of Sandia XTP 1.5.1 exhibit some shortcomings, especially with rate control mechanisms, which are due primarily to the way in which Sandia XTP manages its timers.

In this section, we provide more details on these problems.

3.2.1 Louis Harvey's Major Report

Louis Harvey had worked in HSPL. His study focused on the Sandia XTP rate control behaviors. He found the Sandia XTP throughput rate didn't respect the rate desired by the user. He did many experiments and considerable implementation to demonstrate the following issues:

(1) in Sandia XTP 1.5.1 implementation, the UNIX system call select() plays an important role. It is used to wake up the XTP server when a network packet is received or a user request is received. Its timeout feature is also used to implement various time related issues, such as RTIMER for XTP rate control. The Sandia XTP checks all timers and performs related work when the select() timeout event happens. To guard against the

failure of select() return, the designer of Sandia XTP chose 50ms as the minimum select() timeout parameter value. Unfortunately, this 50ms lower limit becomes a bottleneck to effective rate control if no user requests or packets come. For XTP rate control, if RTIMER should be smaller than 50ms, the Sandia XTP has no guarantee to set a credit at each exact RTIMER period time. So the 50 ms which is set as select() function minimum timeout time, can cause Sandia XTP to have poor quality of rate control.

(2) by exploring the mechanism of Sandia XTP in timer management, found it failed to handle the very short timer.

He also proposed several solutions to improve the situation:

(1) Uses SELECT_FLOOR to replace 50ms, change the SELECT_FLOOR threshold value to improve the rate control quality.

(2) Sets up MAXANTICIPATION margin value for incoming expired timer event to reduce the 50ms effect to rate control.

(3) Uses linked list of timers, which stores timers sorted from earliest to latest to fire, to manage all the timers used by contexts.

He implemented the first two solutions. The development of the third solution is the subject of this Major Report.

3.2.2 Rate Control and Timer Process

For XTP 4.0b specification, the only rate control policy is the following description:

> "Upon each expiration of RTIMER, the internal variable credit is updated with the value burst. That is, credit is updated approximately rate/burst times per second."

Based on this description, the basic ideal rate could be worked in this way: at each time of RTIMER expiration, assign the "credit" with "burst" value and immediately start the RTIMER again. The following figure shows the rate control ideal case.

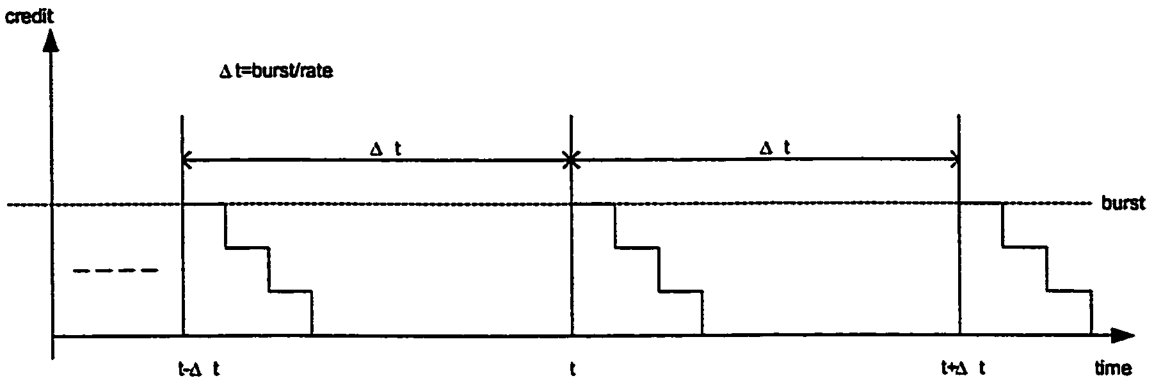

Figure 6: Ideal Rate control diagram

In the Sandia XTP rate control, there is a small difference with the ideal case: there is likely to be a gap between the RTIMER expiration and the RTIMER restart. This gap depends on the CPU execution speed and the XTP client application. Normally, the gap is very short in milliseconds. As we know, the RTIMER value is decreased with increased load rate. If the derived RTIMER value is close to the gap size, at this moment, the gap could affect the throughput rate seriously and cause very poor rate control quality.

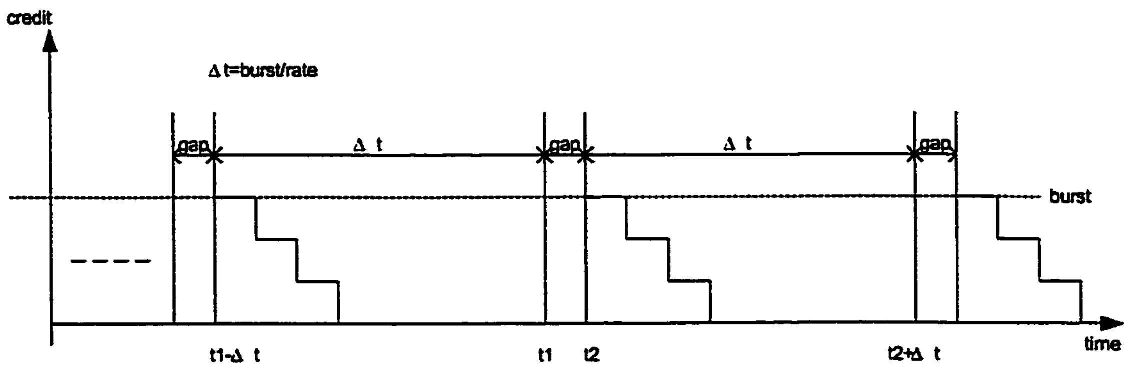

Figure 7: Sandia XTP Rate control

3.2.3 Performance Efficiency

The method satisfy() plays a very important role in Sandia XTP. The method is called each time the timer expires, a user request is received or a new packet is received from the network. Each time the method is called, it will cycle through all contexts: bubble up the shortest time for next select() call timeout value, satisfy any context outstanding work. The method satisfy() is very costly, especially for finding the shortest time, since it will check each context (object) individually. So finding a way to handle the XTP events directly instead of scanning all contexts and all the events for the context is essential to improve the XTP performance efficiency.

3.3 XTP Time Issues

In XTP protocol, there are several timers that are used or maintained during the lifetime of an association. They are:

WTIMER: it is used to bound the amounts of time a context will wait on a response to a status request (a set SREQ bit in any sent packet).

CTIMER: it is a long duration timer used to generate keep-alive packets.

CTIMEOUT: this timer bounds the amount of time an endpoint will try to reestablish the association before giving up.

RTIMER: it is used for rate control, and manages the length of time between bursts of data.

3.3.1 WTIMER

The WTIMER is used to detect the loss of a packet.

Whenever the packet is sent with the SREQ (status request) bit set, the WTIMER will be started with a smoothed round-trip time estimate, and also the *saved_sync* value is increased by transmitter in this packet. When XTP receiver receives the packet with SREQ set, a control packet will be sent back immediately.

If a control packet arrives at the transmitter before the WTIMER expires, the *saved_sync* value will be compared. If the context *saved_sync* is equal with the value in the received packet, the WTIMER will be stopped. If the WTIMER is expired, that means something is wrong, and the context will start the synchronizing handshake to fix the problem.

In Sandia XTP, only one WTIMER is maintained. If another packet with SREQ set is sent before WTIMER is stopped, the WTIMER value will be restarted with round trip time.

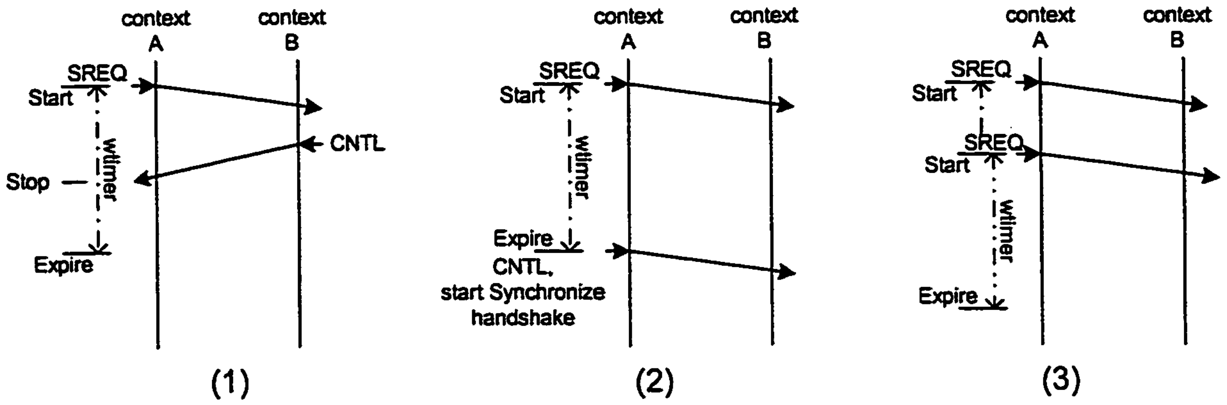

Figure 8: Usages of WTIMER

3.3.2 CTIMER

The CTIMER is used to detect if the XTP association endpoint is still alive.

The CTIMER is a long duration timer. The XTP client should be able to set the length of the CTIMER interval.

When a context becomes active, the CTIMER is started. Each time the packet is received by a context, a packet count is increased by one. When CTIMER is expired, the context examines the packet count. If the count is greater than zero, that means everything is normal and the CTIMER is restarted, also the count is flushed to zero. Otherwise, the CTIMER is reloaded, and the context enters into a Synchronizing Handshake. There is only one CTIMER for each context.

3.3.3 CTIMEOUT

The CTIMEOUT timer limits the amount of time a synchronizing handshake can continue before the context aborts the association. The CTIMEOUT timer is assigned its initial value when a synchronizing handshake is started. If a control packet is received and the association is recovered before the CTIMEOUT timer is expired, the CTIMEOUT timer is stopped. If the CTIMEOUT timer is expired, the association will be aborted by the context.

The CTIMEOUT is also used when a context enters into zombie state after sending a packet with the END bit set. The CTIMEOUT timer can be disabled by setting the CTIMEOUT interval to zero. If the interval sets to zero, the initial value of the *retry_count* for synchronizing handshake must not be zero.

3.3.4 RTIMER

The RTIMER is used to control the sending rate by setting the *burst* variable value to context *credit* variable.

The RTIMER interval is calculated from *burst/rate*. Both *burst* and *rate* can be set by the XTP client.

In Sandia XTP, each time the RTMER is expired, the context will get new credit that is equal to the *burst*. When a XTP client requests to send data packet and there is no credit and RTIMER is stopped, the RTIMER will be started.

4 Sandia XTP Modification with Timer Class

4.1 The Abstraction of Modification

As discussed in section 3.2, the Sandia XTP implementation has several potential problems: (1) the existence of a 50 ms lower limit for timeouts in the select() function call, (2) Using method satisfy() to scan all contexts to get the variable "shortest" (the shortest time for select() function timeout parameter) and to handle the outstanding receiving or sending tasks for the active context list. These two problems cause the Sandia XTP to have poor rate control.

To improve the rate control and protocol efficiency, the following modification is made:

(1) Remove the hard coded 50 ms lower limit for select() from the implementation.

The original Sandia XTP used a 50 ms lower limit on select() timeout to protect the XTP daemon against failure to return from select() system function call. We decided to remove this protection, because we didn't find this problem in Solaris UNIX operating system. We believe there should not be such a limitation for modern UNIX operating systems when we call the select() system function.

(2) Change the mechanism of getting shortest time and handling the outstanding task list.

In the Sandia XTP implementation, when several XTP contexts are active, the shortest time to the next timeout becomes erratic. In addition, the scanning of all the active contexts wastes CPU time, which may cause a delay in serving XTP client requests. The new proposal is to build a timer link that manages all XTP time issues. Only when the timer is armed, it will be inserted to the link. The link is sorted by expiry time, so that the link header always is the next-to-expire timer. The context outstanding task is checked only when the context is being processed.

(3) Adjust the "credit" value every time that the RTIMER expires, rather than simply setting "credit" to "burst".

In the Sandia XTP, the "credit" is simply assigned to the "burst" value when the RTIMER expires. The "credit" is consumed at each time a data packet is sent, until it reaches zero. As we see, the RTIMER timeout returning depends on the UNIX select() function call, especially when the RTIMER is very short, because the RTIMER timeout returning time is not exactly RTIMER. In this case, the XTP Rate control will not work well with setting "credit" to "burst". To improve this situation, the real RTIMER timeout returning time is checked comparing with the desired RTIMER, and the "credit" value will be set depending on the real returning time.

4.2 Timer and Timer Link Class

To be compliant with Sandia XTP OMT architecture, the timer and timer link classes are added as extensions of the MTL. These two classes will manage all time issues described in the protocol specifications. Basically, the timer class will be included into the context class and it will handle the private and special purpose time data for the context. The timer link will manage all active timers that belong to each context in a link, and provide insert, remove, and find methods. The time link object will be placed in the daemon.

4.2.1 Timer Class

4.2.1.1 Class Description

The timer class is used by the context to manage various timeout issues. The timer class is mainly used to store the timeout time and type which tells the context what action the context shall take when a timeout occurs. The context may have several timer objects for different purposes. The timer is identified by variable *type*. The timer class also has a pointer to the context and two links to next and previous instances.

To avoid the conflict of names in MTL, the timer class is named cctimer.

```
class cctimer {
friend class cctimer_link;
private:
  word32 type;            // a type that presents what the timer is used for
  context* owner;         // a pointer to context which owns this timer

protected:
  cctimer* next;              // Linked list next pointer
  cctimer* prev;              // Linked list previous pointer
  word32 due_time;            // the timer due time

public:
// Constructor
cctimer(context* c=(context*) NULL);
// Destructor
~ cctimer();

// time functions
```

```
  void set_type(word32 mtype);
  word32 get_type();
  void set_cctimer(word32 tv);
  void set_cctimer_by_interval(word32 t);
  word32 get_cctimer_val();
  context *get_owner();
  word32 get_shortest();
  word32 timestamp() ;
  int is_expired();
  int is_in_link();
};
```

4.2.1.2 Function Description

void set_type(word32 mtype)

Set the timer type. The type probably decides the timeout action.

word32 get_type()

Return the timer type.

void set_cctimer(word32 tv)

Set the timer due time.

void set_cctimer_by_interval(word32 t)

Set the timer by input interval time.

word32 get_cctimer_val()

Return the timer due time.

context *get_owner()

Return the context that owns the timer.

word32 get_shortest()

Return the timer expire interval time. If the timer already expired, return 0; otherwise return the value of due time subtract the current time.

word32 timestamp()

Return the current computer time in word32 format.

int is_expired()

Check if the timer is expired. If expired, return 1; otherwise return 0.

int is_in_link()

Check if the timer is in the timer link. If the timer is in the timer link, return 1; otherwise return 0.

4.2.2 Timer Link

4.2.2.1 Class Description

The Timer link manages all context active timers. When the timer is armed, it will be inserted into the timer link. If the timer is expired, is stopped or timer owner is inactive, the timer shall be moved from the timer link.

To avoid the conflict of names in MTL, the timer link class is named cctimer_link.

```
class cctimer_link {
private:
        cctimer* link_head;
        cctimer* link_tail;
public:
  // Constructor
        cctimer_link(){link_head=(cctimer*)NULL;link_tail=(cctimer*)NULL;}
  // Destructor
  ~ cctimer_link(){link_head=(cctimer*)NULL;link_tail=(cctimer*)NULL;};

  // Timer linked-list manipulators
  void insert_cctimer(cctimer *);
  void remove_cctimer(cctimer *);
  void remove_cctimer(context *);
  cctimer* find_cctimer(context *, word32);
  void resort_cctimer(cctimer *);
  cctimer* return_first_cctimer(){return link_head;}
};
```

4.2.2.2 Function Description

void insert_cctimer(cctimer *)

Insert a timer into the timer link.

void remove_cctimer(cctimer *)

Remove the timer from the timer link.

void remove_cctimer(context *)

Remove the timer that belongs to the input context.

cctimer* find_cctimer(context *, word32)

Check if the specific timer is in the timer link. If yes, return that timer; otherwise return a Null timer.

void resort_cctimer(cctimer *)

Sort the timer link based on the timer due time.

cctimer* return_first_cctimer()

Return the timer in link head.

4.3 MTL Modification

The cctimer class and cctimer_link class are added as extensions of the MTL. The rest of the classes in the MTL must be modified to accommodate cctimer and cctimer_link classes.

4.3.1 Class context

Two virtual functions handl_timeout() and routine() are added to context class for future use.

virtual void handl_timeout()

This function is reserved for derived class to handle all timeout events.

virtual void routine()

This function is reserved for context to process all unprocessed tasks when the context is running.

4.3.2 Class context_manager

The virtual function word32 satisfy() is deleted.

4.3.3 Class mtldaemon

In mtldaemon class declaration, a timer link object is added. It declared as following:

```
Mtldaemon {
...
        static cctimer_link* timer_link
....
```

}

void main_loop()

Function descriptions:

The main_loop() is the main loop of the daemon. It first parses the arguments, then initializes a few things (such as turning this process into a daemon), then it loops waiting for a packet to arrive (IO signal), the timer to expire (alarm signal), or the user to issue a request to the daemon (msgrcv returns a valid value). This continues forever.

Modification:

The function void main_loop() is rewritten. We do not need satisfy() to do outstanding tasks and obtain the shortest timeout; rather we get the shortest time from the timer link directly and call the context routine() to perform the outstanding tasks. The new main_loop() is shown in the appendix.

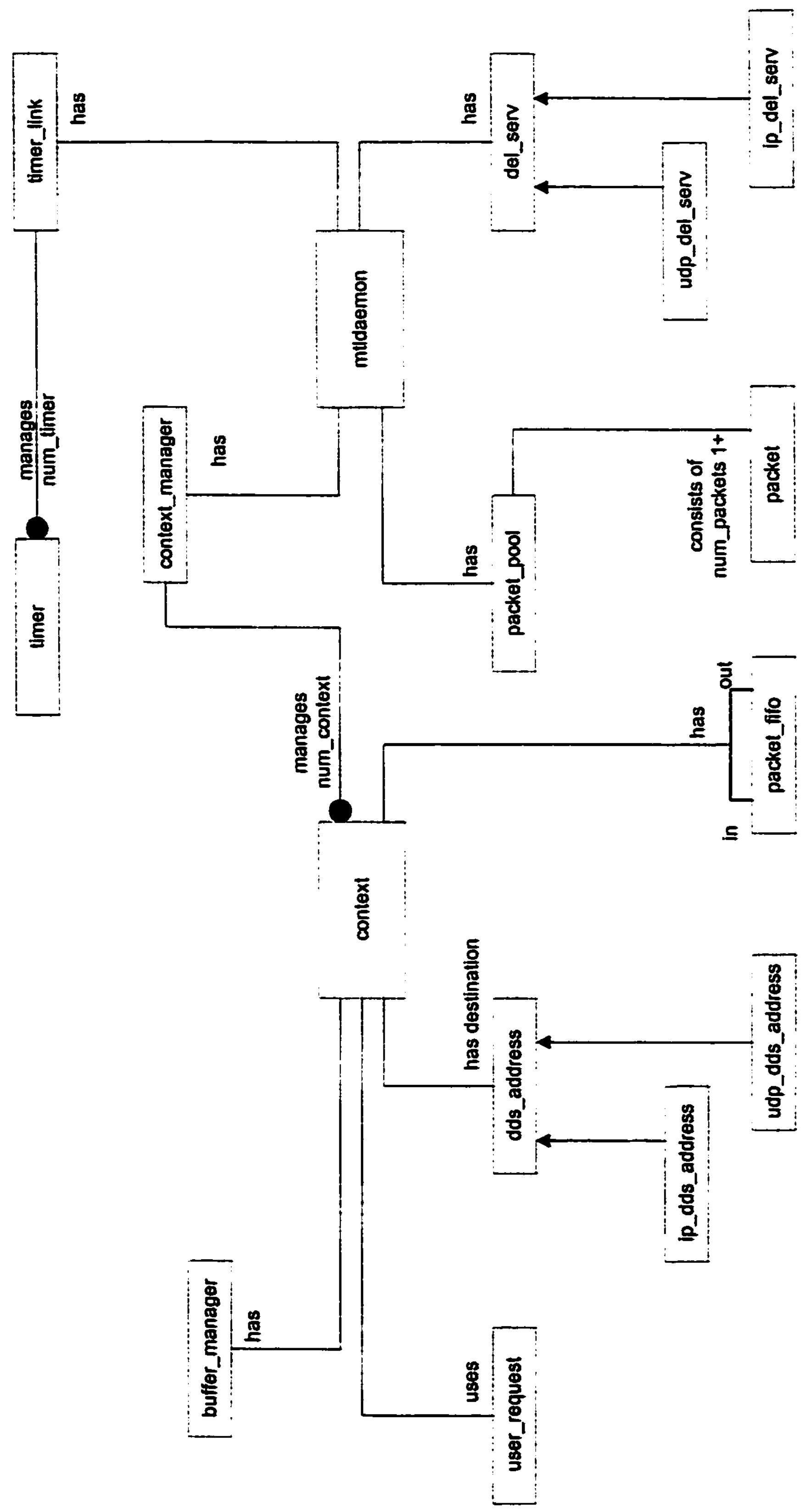

Figure 9: Revised MTL diagram

4.4 XTP Modification

4.4.1 Class XTPcontext

Sandia XTP XTPcontext class develops several timer instances: one CTIMER, one CTIMEOUT, one WTIMER, and one RTIMER. With the cctimer and timer_link classes, the XTPcontext timer instances must be modified.

4.4.1.1 Class Declaration Modification

Redeclare c_timer, c_timeout, w_timer, r_timer with the cctimer class:

```
 XTPcontext:context{
...
//   CTIMER

  //yjiang modify:
  //start
  //word32 c_timer;
  cctimer* c_timer;
  //end
  word32 c_timer_interval;
  word32 pkts_rcvd_in_c_interval;

  //   CTIMEOUT

  //yjiang modify:
  //start
  //word32 c_timeout;
  cctimer* c_timeout;
  //end
  word32 c_timeout_interval;
  //word32 c_timeout_armed;//del by yjiang

  //    WTIMER

  //yjiang modify:
  //start
  //word32 w_timer;
  cctimer* w_timer;
  //end
  //word32 w_timer_armed;//del by yjiang
  word32 w_timer_limit;
  word32 retry_count;
  word32 num_retries;
  word32 backoff_K;
  int sync_handshake_open;

  //    RTIMER

  //yjiang modify:
  //start
  //word32 r_timer;
  cctimer* r_timer;

  //end
  //word32 r_timer_armed;//del by yjiang
  word32 rate;          // Outstream rate
  word32 burst;          // Outstream burst
  int credit;          // Outstream credit
...
}
```

4.4.1.2 Function Modifications

void XTPcontext()

XTPcontext() is the constructor of the XTPcontext class. Here all the cctimer used in the XTPcontext class need to be initializated. The following code is added.

```
....
  //yjiang modify:
  //start
  //   CTIMER
  c_timer= new cctimer(this);
  c_timer->set_type(T_CTIMER);

  //   CTIMEOUT
  c_timeout= new cctimer(this);
  c_timeout->set_type(T_CTIMEOUT);

  //   WTIMER
  w_timer= new cctimer(this);
  w_timer->set_type(T_WTIMER);

  //   RTIMER
  r_timer= new cctimer(this);
  r_timer->set_type(T_RTIMER);
```

void handl_timeout()

handl_timeout() is the handler for XTPcontext timeout events. Based on different expired timer, a corresponding process is performed. For XTP, only following time events will be processed: CTIMER expired, CTIMEOUT expired, RTIMER expired, and WTIMER expired. For the program, see appendix.

void routine()

routine() is the routine process of XTPcontext, which is used to process the XTPcontext outstanding tasks, such as sending the packet that is waiting in out packet FIFO, processing incoming packet, or retransmitting the packet when this context is currently active. For the program, see appendix.

void start_zombie()

start_zombie() is used to arm the CTIMEOUT timer. If the timer interval is zero, the zombie state is bypassed. Clear the sync_handshake if necessary at this point, there is no need to continue it, also clear the RTIMER timer. Normally, it is called after the context sending a packet with END bit set. For the detail modification, see appendix.

void start_wtimer(word32 factor = 1)

This function is used to arm the WTIMER timer. If "factor" is present, it will multiply the duration by a factor. This is mainly for the last timeout, when the END bit has been sent and the context is waiting to go quiescent. The timer duration will be the guessed value of (SRTT+Z*RTTV). Each time when a packet is sent with SREQ flag, this function is called to start the WTIMER. If this function is called again before the WTIMER stops, the WTIMER timer will be assigned a new duration. For the detailed modifications, see appendix.

void stop_wtimer()

Stops and disarms the w_timer and removes the WTIMWER from timer_link. At this time we can release the FIRST packet (if kept) since we know now that it will not need retransmitting. For the detailed modifications, see appendix.

void start_sync_handshake()

When something is wrong, the protocol context goes to synchronization handshake to recover the communication. At the beginning of synchronization, the CTIMEOUT timer shall be armed to protect the context from waiting forever for a response, but only if the CTIMEOUT interval is not 0. If the interval is 0, the CTIMEOUT timer has been disabled. For the detailed modifications, see appendix.

void stop_sync_handshake()

When the synchronization handshake is stopped, the CTIMEOUT timer and WTIMER must be stopped. These two timers shall be removed from timer_link list. For the detailed modifications, see appendix.

void start_rtimer()

This function shall start RTIMER timer: the timer is added to timer link. For the detailed modifications, see appendix.

void stop_rtimer()

Remove the RTIMER timer from timer link. For the detailed modifications, see appendix.

void handle_c_timeout()

It is to handle the expiration of the CTIMEOUT timer. When the END bit is sent, the expiration of CTIMEOUT timer will cause the context to go quiescent. The CTIMEROUT timer also is removed from timer link. For the detailed modifications, see appendix.

void handle_wtimer()

It is handle the expiration of the WTIMER timer. It shall remove the WTIMER timer from the timer link if the timer is in the timer link. For the detailed modifications, see appendix.

void handle_ctimer()

It is handle the expiration of the CTIMER timer. It shall remove the CTIMER timer from timer link if the timer is in timer link. For the detailed modifications, see appendix.

int check_timers()

In Sandia XTP, check_timers() will check all context timers if the timer is expired. The related process must be done by calling the timer handler. With Revised XTP, we do not need to check all context timers because we know which timer is expired and the timer handle is called directly. The check_timers() is deleted.

4.4.2 Class XTPcontext_manager

4.4.2.1 Function Modification

void handle_new_packet(packet* pkt)

The function performs processing of new incoming packets: figure out where the packet shall go. In the normal case, the packet is put on the receiver FIFO and marked for the proper context. If the key in packet is not a return key, the full context lookup is performed. The modification for this function is adding the context routine() call at the packet is put on that context receiver packet FIFO. For details of the modifications, see the appendix.

word32 satisfy()

In Sandia XTP, satisfy() performs the following functions: (1) scan all contexts to find what is the shortest next ; (2) scan all context to try to satisfy the outstanding work. In the Revised XTP, the satisfy() function (1) is replaced by timer time_link class. For the function (2), is replaced by context routine(). So satisfy() is no longer needed in the Revised XTP.

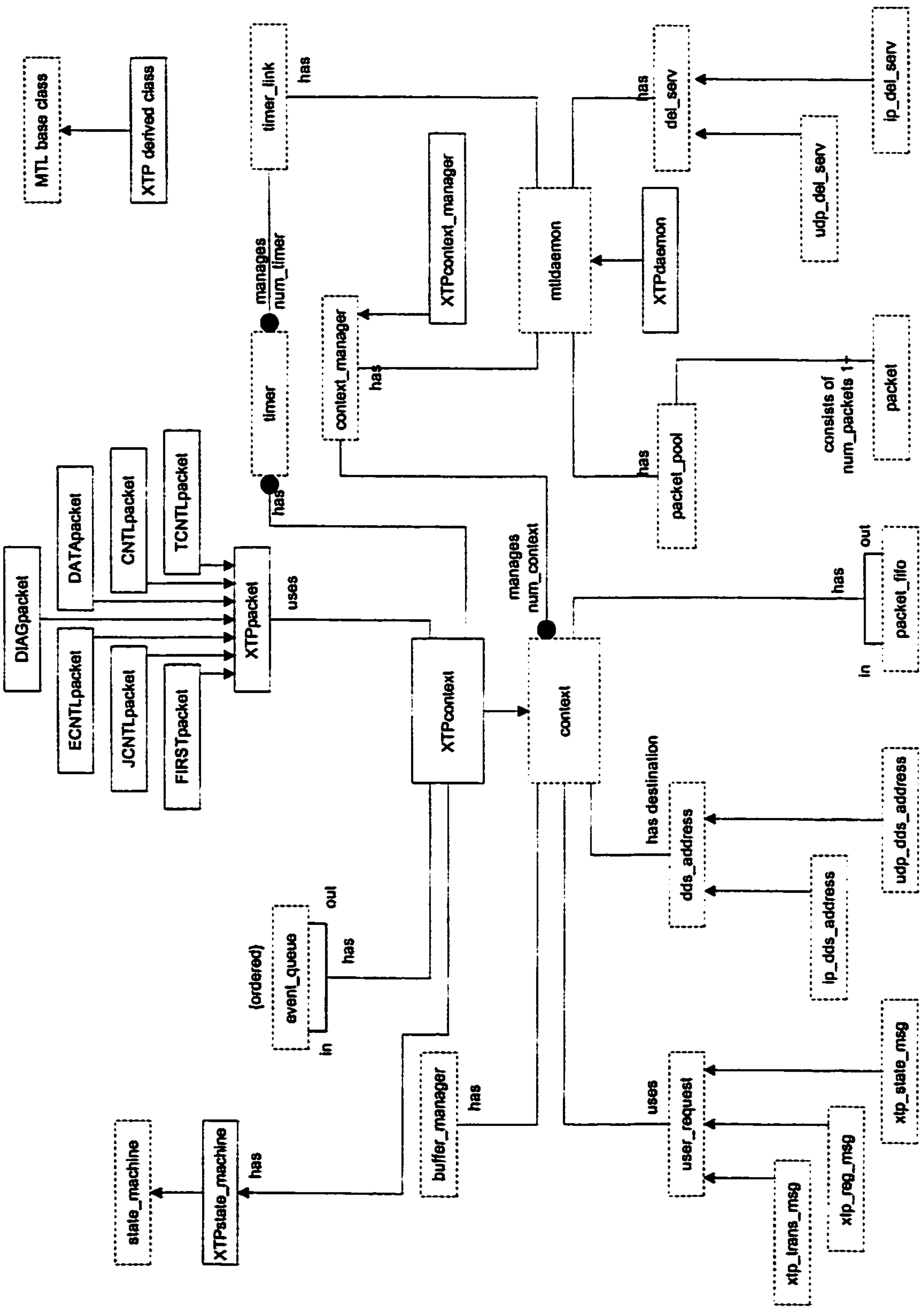

Figure 10: Revised XTP diagram

5 Experiments

As we see, the timer management affects the XTP rate control quality directly. So, the experiments will focus on how the new timer management works with the XTP rate control. The goals of the experiments here are: (1) to prove that the revised XTP has the same functionality as the Sandia XTP 1.5.1; (2) to compare the results of experiments between the Revised XTP and Sandia XTP 1.5.1 with Unicast and Multicast; (3) to find the key parameters that affect the XTP implementation performance.

5.1 Experiment Environment

All experiments are completed using the Concordia University local network. The following figure shows the logical network connectivity, including only the sub networks and the machines used for the experiments. The dotted decimal notation address of the end machines are not of much significance with multicast, as a group address (Class D internet address) is needed. To use maximum network and machine speed, our experiment will be limited on *orchid* and *sunset*. So for the XTP multicast address, it will be 239.159.100.40.

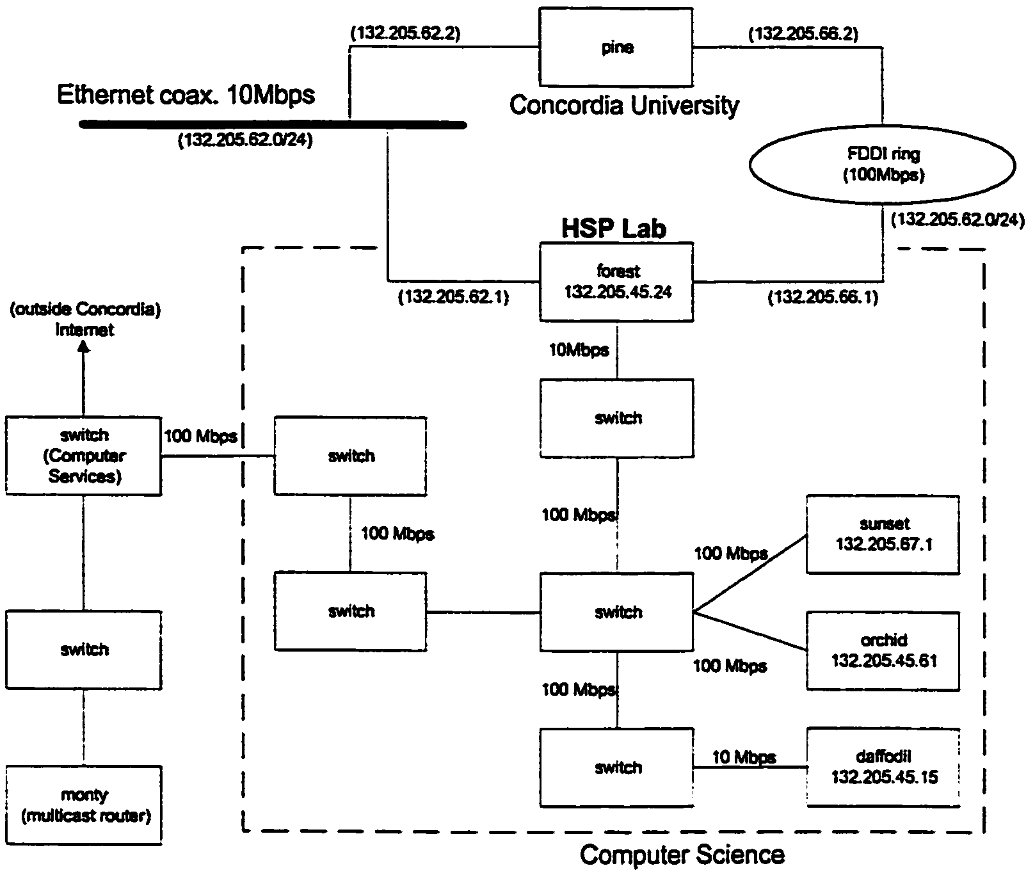

Figure 11: Logical network connectivity in Concordia University

5.2 Experimental Program

The program used to test is the same as used for Louis Harvey's report. The major idea is sending the specified amount of data from one sender to another receiver (Unicast) or to possibly multiple receivers (Multicast) reliably. The time from the moment the program transfers the first buffer full of data to the moment the sender has received the last acknowledgment from the last receiver will be measured. The send rate is specified at the beginning and remains fixed for the whole data transfer.

This program is derived from Sandia XTP example metric and can be used to measure the XTP throughput rate, and the time to transfer amount of data. This program is named mmetric. Typical command line arguments used for the experiments are as follow:

Unicast sender

```
mmetric –t sunset –S –g –f –p1472 –b1440 –C1440 –a1048576 –o250 –W10240 –c10
```

Unicast receiver

```
mmetric –r –S –g –f –p1472 –b1440 –J1440 -o250 –W10240 -j10
```

Multicast sender

```
mmetric –T239.159.100.40 –S –g –f –p1472 –b1440 –C1440 –a1048576 –o250 –W10240 –c10
```

Multicast receiver

```
mmetric –R239.159.100.40 -b1440 –J1440 -o250 –W10240 -j10
```

Where:

-t, -r are Unicast transmitter and receiver. The sunset is the Unicast destination computer name.

-T, -R are Multicast transmitter and receiver. The 239.159.100.40 is Multicast group address.

-S is used to specify selective retransmission mechanism.

-g is used to block the sender on acknowledgements.

–f is used to set SREQ in the FIRST packet.

–p is used to set PDU size. The PDU size is set to 1472 in the experiments.

–b is used to define the user level buffer size.

–C is used to suggest output burst value. With "harmonization Burst /Rate", this value will be changed respect to rate.

–a is set the amount of data to be sent.

–o is used to define the initial round trip time.

–W is used to define the send window size.

–c is used to set the load rate. The unit is bytes per ms (*Bpms*). The load value is progressively increased during Unicast and multicast experiments to cover all test rates.

5.3 Experiments Limitation with XTP Rate Control

The following tables show some physical characteristics used for the experiments and the relationship between RTIMER and Rate (this investigation was done by Louis Harvey in his Master major report.). These will be helpful to explain the XTP Rate Control behavior in our experiments.

Table 1: Machine characteristics

Name	**Mean delay1 (ms)**	**Derived Capacity (Bpms)**	**Features**	**O.S.**
sunset	0.5	2880	Sun 2X UltraSPARC-II 296MHz, 2 cpus, 896MB mem	2
orchid	1.0	1440	Sun 2X UltraSPARC 168MHz, 2 cpus, 640MB mem	2
sunset	2.5	576	Sun SPARCstation-10 50MHz, 448MB mem	2
daffodil	3.6	400	Sun SPARCstation-10 36MHz, 256MB mem	2
forest	7.0	206	Sun 4_75 SPARCstation 2 40MHz, 32MB mem	2
pine	7.0	206	Sun 4_50 SPARCstation IPX 40MHz, 32MB mem	2
1 Mean delay between sending consecutive back-to-back 1440 byte packets 2 Solaris 2.5				

Table 2: Evaluation of RTIMER when burst = 1440 bytes

RTIMER(ms):	144	96	72	57.6	48	41.1	28.8	19.2	14.4	11.5	9.6
Rate(Bpms):	10	15	20	25	30	35	50	75	100	125	150

RTIMER(ms):	8.2	7.2	5.7	4.8	4.1	3.6	2.8	2.4	2.0	1.8	1.6
Rate(Bpms):	175	200	250	300	350	400	500	600	700	800	900

RTIMER(ms):	1.4	1.3	1.2	1.1							
Rate(Bpms):	1000	1100	1200	1250							

As we see, the machine Mean delay decides the machine Derived Capacity. This will be the bottleneck of data transfer. When we do XTP Rate Control test with high rate, we must consider the machine factors.

Another factor we also consider is time resolution. In the whole XTP implementation, the time resolution is one millisecond, that means we can not guarantee the results with two timers that have difference under 1 millisecond.

5.4 Functionality Test

To make sure the new revised Sandia XTP has the same functionality as the original Sandia XTP version, the mixed environment is created: one XTP server is started with revised Sandia XTP, another XTP server is started with Sandia XTP original version. The data transferring between those two XTP servers is conducted to verify the revised Sandia XTP does not change the XTP protocol. The following tests are done:

1.Unicast, Sender: revised Sandia XTP; receiver: Sandia XTP

Table 3: Unicast test results (1)

	XTP Sender	XTP Receiver
Host Name	forest	Orchid
XTP Serve	revised Sandia XTP	Sandia XTP
Data Transfer Mode	unicast	Unicast
Test Program and Command	mmetric -t orchid -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10	mmetric -r -S -g -f -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10
Test Results	Sun Jan 14 16:56:54 2001 mmetric starting (13542) on Host:forest mode UNICAST, outburst=1440, off_load=10 Bpms, WinSiz=102400 bytes, mode UNICAST, off_load=10 Bpms, WinSiz=102400 bytes, Transmitting_to:orchid 1048576 bytes using buffers of size 1440 bytes Timing: 109286 ms Throughput: 9.595 Bpms - Bytes per ms xy: 10 9.595 Throughput: 0.077 Mbits/sec Number of calls: 729 Latency: 149.912 ms/call Sent 1048576 bytes Sun Jan 14 16:58:44 2001	Sun Jan 14 16:50:42 2001 mmetric starting (23785) on Host:orchid mode UNICAST, inburst=1440, inrate=10 Bpms, WinSiz=102400 bytes, mode UNICAST, input_rate=10 Bpms, WinSiz=102400 bytes, Receiving with buffers of size 1440 bytes rTiming: 482098 ms rThroughput: 2.175 Bpms - Bytes per ms xy: 10 2.175 rThroughput: 0.017 Mbits/sec Number of calls: 730 Latency: 660.408 ms/call Received 1048576 bytes Sun Jan 14 16:58:44 2001

2.Unicast, Sender: Sandia XTP; receiver: revised Sandia XTP

Table 4: Unicast test results (2)

	XTP Sender	XTP Receiver
Host Name	Orchid	forest
XTP Serve	Sandia XTP	Revised Sandia XTP
Data Transfer Mode	Unicast	unicast
Test Program and Command	mmetric -t froest -S -g –f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10	mmetric -r -S -g -f -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10
Test Results	Sun Jan 14 16:47:13 2001 mmetric starting (23748) on Host:orchid mode UNICAST, outburst=1440, off_load=10 Bpms, WinSiz=102400 bytes, mode UNICAST, off_load=10 Bpms, WinSiz=102400 bytes, Transmitting_to:forest 1048576 bytes using buffers of size 1440 bytes Timing: 110232 ms Throughput: 9.512 Bpms - Bytes per ms xy: 10 9.512 Throughput: 0.076 Mbits/sec Number of calls: 729 Latency: 151.210 ms/call Sent 1048576 bytes Sun Jan 14 16:49:05 2001	Sun Jan 14 16:46:56 2001 mmetric starting (13523) on Host:forest mode UNICAST, inburst=1440, inrate=10 Bpms, WinSiz=102400 bytes, mode UNICAST, input_rate=10 Bpms, WinSiz=102400 bytes, Receiving with buffers of size 1440 bytes rTiming: 128714 ms rThroughput: 8.147 Bpms - Bytes per ms xy: 10 8.147 rThroughput: 0.065 Mbits/sec Number of calls: 729 Latency: 176.562 ms/call Received 1048576 bytes Sun Jan 14 16:49:05 2001

3.Multicast, Sender: revised Sandia XTP; receiver: Sandia XTP

Table 5: Multcast test results (1)

	XTP Sender	XTP Receiver
Host Name	Forest	orchid
XTP Serve	revised Sandia XTP	Sandia XTP
Data Transfer Mode	Multcast	multcast
Test Program and Command	mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 –o 250 -W 102400 -c 10	mmetric -R 239.159.100.40 -S -g -f -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10
Test Results	Sun Jan 14 16:40:28 2001 mmetric starting (13512) on Host:forest mode MULTICAST, outburst=1440, off_load=10 Bpms, WinSiz=102400 bytes, mode MULTICAST, off_load=10 Bpms, WinSiz=102400 bytes, Transmitting_to:239.159.100.40 1048576 bytes using buffers of size 1440 bytes Timing: 109371 ms Throughput: 9.587 Bpms – Bytes per ms xy: 10 9.587 Throughput: 0.077 Mbits/sec Number of calls: 729 Latency: 150.029 ms/call Sent 1048576 bytes Sun Jan 14 16:42:19 2001	Sun Jan 14 16:40:25 2001 mmetric starting (23692) on Host:orchid mode MULTICAST, inburst=1440, inrate=10 Bpms, WinSiz=102400 bytes, mode MULTICAST, input_rate=10 Bpms, WinSiz=102400 bytes, Receiving with buffers of size 1440 bytes rTiming: 113805 ms rThroughput: 9.214 Bpms - Bytes per ms xy: 10 9.214 rThroughput: 0.074 Mbits/sec Number of calls: 730 Latency: 155.897 ms/call Received 1048576 bytes Sun Jan 14 16:42:19 2001

4.Multicast, Sender: Sandia XTP; receiver: revised Sandia XTP

Table 6: Multcast test results (2)

	XTP Sender	XTP Receiver
Host Name	Orchid	forest
XTP Serve	Sandia XTP	revised Sandia XTP
Data Transfer Mode	Unicast	unicast
Test Program and Command	mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 –o 250 -W 102400 -c 10	mmetric -R 239.159.100.40 -S -g -f -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10
Test Results	Sun Jan 14 16:32:57 2001 mmetric starting (23550) on Host:orchid mode MULTICAST, outburst=1440, off_load=10 Bpms, WinSiz=102400 bytes, mode MULTICAST, off_load=10 Bpms, WinSiz=102400 bytes, Transmitting_to:239.159.100.40 1048576 bytes using buffers of size 1440 bytes Timing: 110278 ms Throughput: 9.508 Bpms – Bytes per ms xy: 10 9.508 Throughput: 0.076 Mbits/sec Number of calls: 729 Latency: 151.273 ms/call Sent 1048576 bytes Sun Jan 14 16:34:48 2001	Sun Jan 14 16:32:51 2001 mmetric starting (13507) on Host:forest mode MULTICAST, inburst=1440, inrate=10 Bpms, WinSiz=102400 bytes, mode MULTICAST, input_rate=10 Bpms, WinSiz=102400 bytes, Receiving with buffers of size 1440 bytes rTiming: 117291 ms rThroughput: 8.940 Bpms - Bytes per ms xy: 10 8.940 rThroughput: 0.072 Mbits/sec Number of calls: 729 Latency: 160.893 ms/call Received 1048576 bytes Sun Jan 14 16:34:48 2001

5.5 Unicast without Harmonization Burst/Rate

When calculating the value to be loaded into RTIMER, the ratio Burst/Rate (burst size / data rate) is used to calculate the number of milliseconds until the next timer event should occur. Louis Harvey observed in his Major Report [Harvey 1999] that if the "burst" size is increased as the "rate" is increased, then the calculated value for RTIMER can be kept larger than 50ms, which avoids the previously-noted problems with the (hardcoded) 50ms lower limit for timeouts. He used the term "harmonization of Burst/Rate" to imply increasing the "burst" size as the "rate" increases, to ensure that the RTIMER value is never less than 50ms. We do experiments in both "harmonization" and without "harmonization" cases, and check how the RTIMER value affect the XTP rate control quality.

The experiments to compare Sandia XTP and Revised XTP are divided into four parts: Unicast without harmonization Burst/Rate; Unicast with harmonization Burst/Rate; Multicast without harmonization Burst/Rate; and Multicast harmonization Burst/Rate.

For Unicast without harmonization Burst/Rate experiment, the "burst" is set to 1440 bytes, which equals to the XTP packet size used in our experiments and let us to achieve most efficiency to transfer XTP packets. The load rate is progressively assigned the value from 10 Bpms to 1255 Bpms. The throughput rate and timing will be collected at sender under the Original Sandia XTP server and Revised XTP server. The experiment and results are described in table 7 and figures 12 and 13:

Sender: orchid   Receiver: sunset

Table 7: Unicast without harmonization Burst/Rate data summary

Load Rate (Bpms)	Original_Sandia_ Throughput (Bpms)	Original_Sandia_ Timing (ms)	Revised_Sandia_ Throughput (Bpms)	Revised_Sandia_ Timing (ms)
10	9	113352	10	104860
15	13	78867	15	69901
20	16	64249	20	52402
25	21	50805	25	41544
30	25	42059	30	34915
35	27	39479	35	29873
50	27	38347	51	20419
75	27	38177	76	13884
100	27	38514	102	10302
125	28	37964	128	8194
150	8	38064	159	6591
200	28	37840	203	5141
250	27	38441	265	3964
300	29	36400	323	3241
350	29	36437	353	2971
400	29	36400	426	2461
500	29	36369	539	1942
600	29	36358	616	1703
700	29	35903	687	1527
800	40	26261	829	1264
900	30	35071	912	1150
1000	33	32116	987	1062
1100	58	17961	986	1063
1200	58	18001	1069	980
1250	32	33052	1063	986

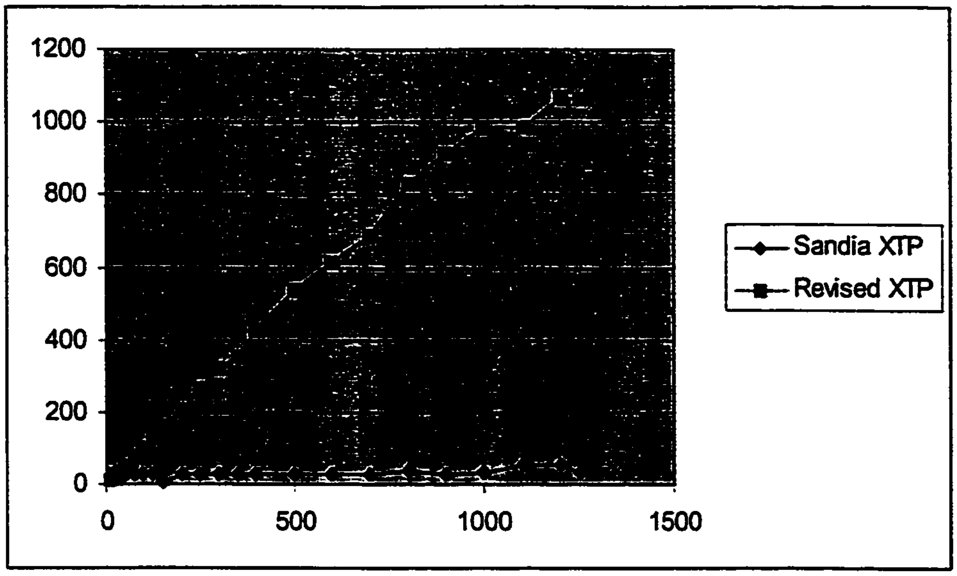

Figure 12: Unicast without harmonization Burst/Rate saturation curve

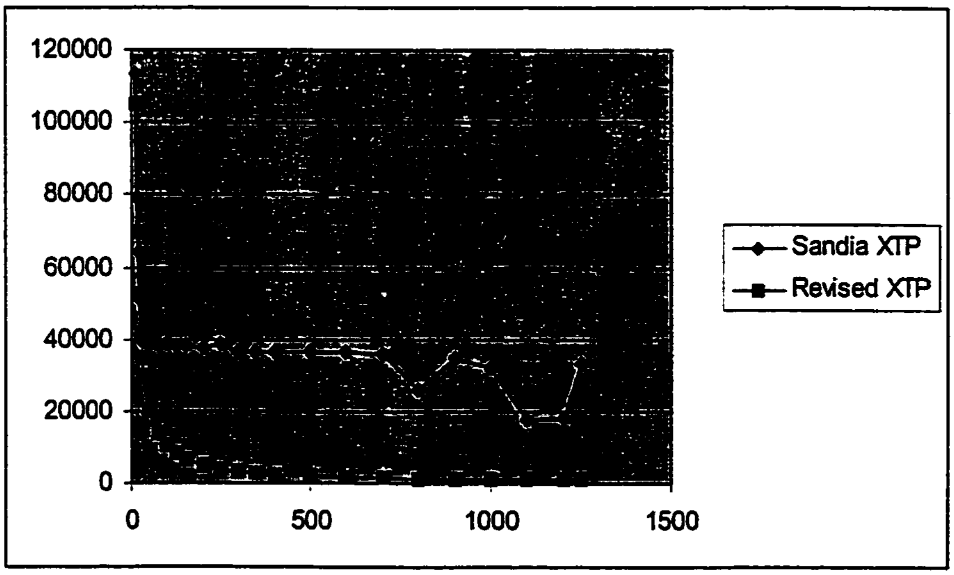

Figure 13: Unicast without harmonization Burst/Rate timing curve

Before comparing the revised XTP with Sandia XTP, we look at the Sandia XTP saturation curve first. As we see, for the Sandia XTP, the throughput almost stop to increase after the load rate is higher than 30Bpms. That is because at that particular rate (30Bpms), the value of RTIME equals to 48ms, which is close to the threshold value of

50ms used by the Sandia XTP implementation as the minimum timeout parameter to the select() UNIX system call. This phenomenon is called "SELECT FLOOR" effect in Harvey's Major Report. The "SELECT FLOOR" effect dominates only within a subset of range of sending rates. Another Bombardment phenomenon is introduced the throughput rate may be jump to higher when the load rate goes higher. As we see, when load rate reaches 1100Bpms, the Sandia XTP throughput rate jumps from around 30Bpms to 58Bpm. The "Bombardment" effect is caused by the incessant up coming of event, such as incoming packet or incoming request issued by the user, with the consequence that the system select() call returns before the specified timeout value. Compared the saturation curve and time consume curve between Sandia XTP version and Revision. The revision of XTP achieves great improvement on rate control quality. There is no "SELECT FLOOR" and The "Bombardment" effect with revised XTP. For revised XTP, the throughput rate does not follow the load rate very well when the load rate is set to very high (over 800 Bpms) , this because in that load range, the RTIMER is changed in 1 millisecond, that against the XTP implementation time resolution (1 millisecond).

5.6 Unicast with Harmonization Burst/Rate

For Unicast with harmonization Burst/Rate experiment, the "burst" size will increase as the load rate increases: burst=rate*1000*0.1. That means the RTIMER interval is 100ms at any load rate. The load rate is progressively assigned the value from 10 Bpms to 1255 Bpms. The throughput rate and timing will be collected at sender under the Sandia XTP server and Revised XTP server. The experiment and results described in table 8 and figures 14 and 15:

Sender: orchid Receiver: sunset

Table 8: Unicast with harmonization Burst/Rate data summary

Load Rate (Bpms)	Original_Sandia_ Throughput (Bpms)	Original_Sandia_ Timing (ms)	Revised_Sandia_ Throughput (Bpms)	Revised_Sandia_ Timing (ms)
10	8	125899	10	104887
15	15	70169	15	69875
20	19	54050	20	52410
25	25	41949	25	41949
30	30	35139	30	34832
35	34	30600	35	30018
50	50	21051	50	20903
75	75	13950	76	13864
100	100	10466	101	10416
125	124	8441	126	8311
150	148	7100	152	6908
200	197	5319	204	5145
250	249	4203	255	4106
300	306	3425	307	3412
350	331	3165	360	2913
400	379	2770	415	2526
500	420	2499	521	2013
600	498	2105	600	1748
700	729	1438	734	1429
800	855	1227	858	1222
900	904	1160	939	1117
1000	978	1072	1035	1013
1100	1087	965	1146	915
1200	1268	827	1283	817
1250	1222	858	1290	813

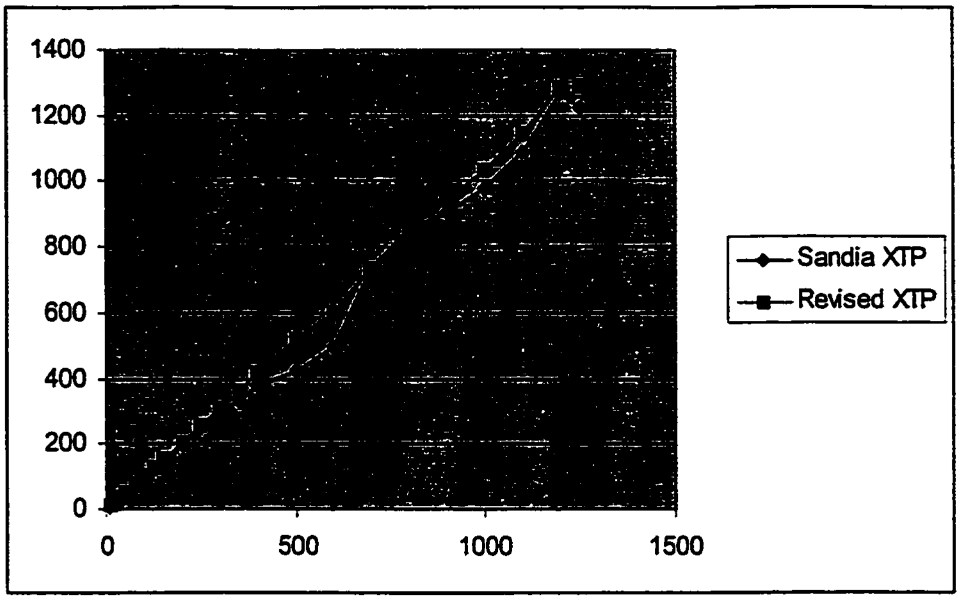

Figure 14: Unicast with harmonization Burst/Rate saturation curve

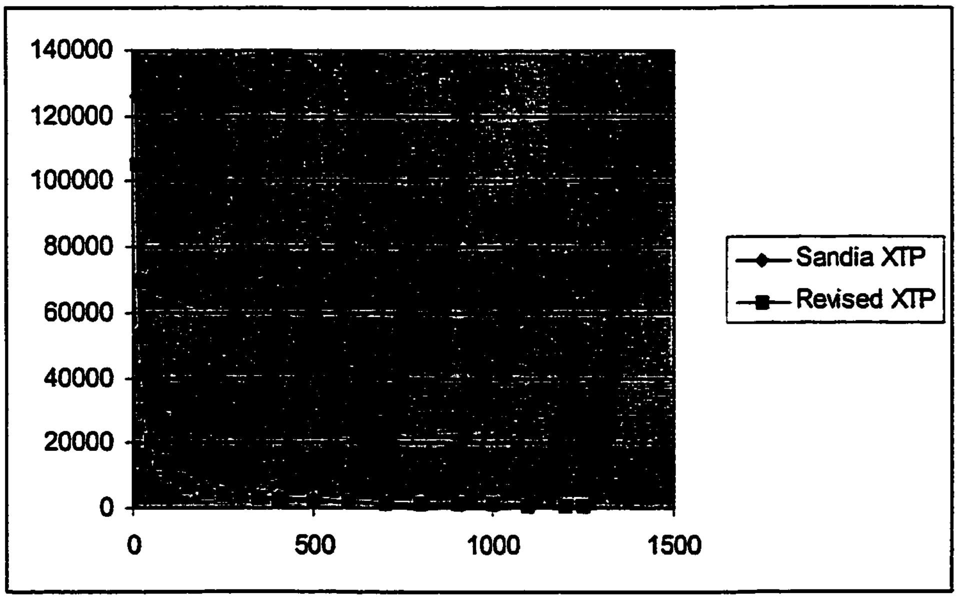

Figure 15: Unicast with harmonization Burst/Rate timing curve

As we see, the saturation curve and time consumed curve between the Sandia XTP and the revised XTP, are very close, the rate control QoS appears very good. This is because the RTMER interval (100ms) is larger than 50 ms, which removes the "SELECT FLOOR" effect. In this case there is not much difference between the revised XTP and the original Sandia XTP.

5.7 Multicast without Harmonization Burst/Rate

For Multicast without harmonization Burst/Rate experiment, the "burst" is set to 1440 bytes, and the load rate is progressively assigned the value from 10 Bpms to 1255 Bpms. The throughput rate and timing will be collected at sender under the Sandia XTP server and Revised XTP server. The experiment and results are described in the table 9 and figures 16 and 17:

Sender: orchid Receiver: sunset

Table 9: Multicast without harmonization Burst/Rate data summary

Load Rate (Bpms)	Original_Sandia_ Throughput (Bpms)	Original_Sandia_ Timing (ms)	Revised_Sandia_ Throughput (Bpms)	Revised_Sandia_ Timing (ms)
10	10	109860	10	104960
15	14	73191	15	70060
20	18	58961	20	52572
25	24	43960	25	41718
30	29	36724	30	35153
35	29	36640	35	30062
50	29	36710	51	20653
75	29	36631	74	14097
100	29	36567	99	10550
125	29	36370	125	8412
150	29	36441	153	6835
200	29	36552	193	5432
250	29	35946	252	4165
300	29	36441	300	3490
350	29	36330	326	3220
400	29	36282	388	2706
500	29	35791	497	2110
600	30	35500	546	1922
700	29	36321	602	1740
800	40	26147	812	1291
900	38	27360	852	1231
1000	61	17181	883	1188
1100	60	17390	797	1316
1200	34	30661	955	1097
1250	34	30840	983	1067

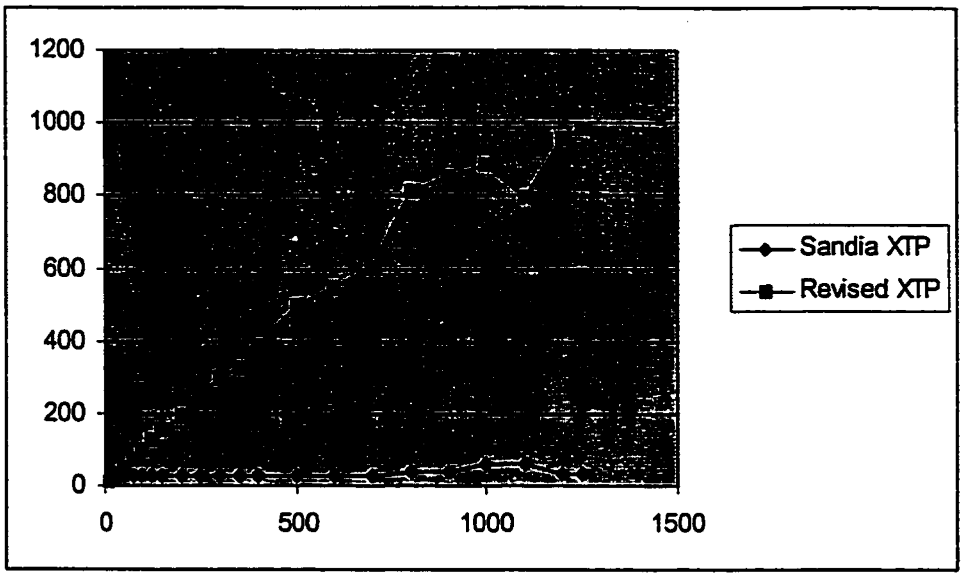

Figure 16: Multicast without harmonization Burst/Rate saturation curve

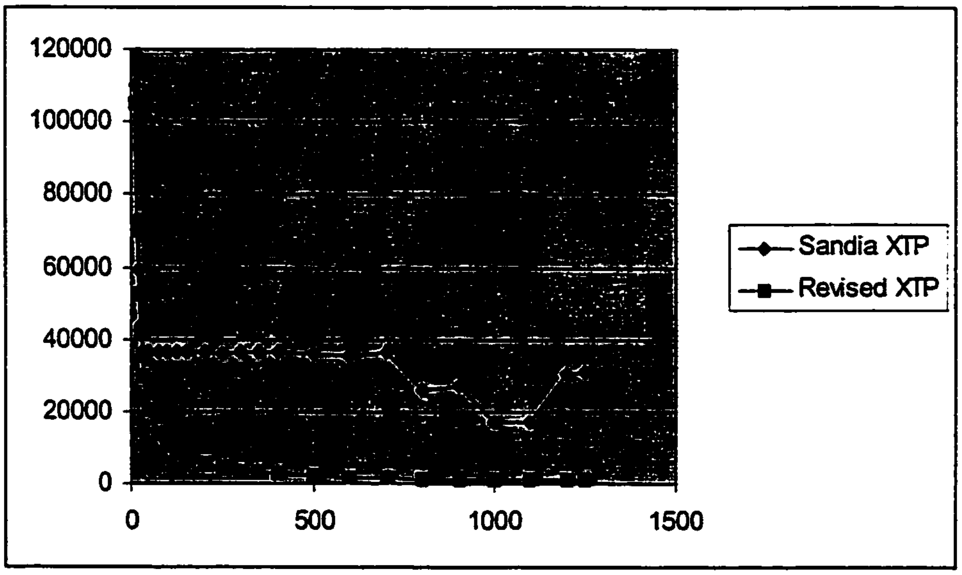

Figure 17: Multicast without harmonization Burst/Rate timing curve

Very similar with Unicast experiments result, the saturation curve and time consumed curve between the Sandia XTP version and the revised XTP shows the achievement of the revised XTP with rate control quality and performance. For the Sandia XTP, the

"SELECT FLOOR" is there like in Unicast experiments and makes the rate control quality very poor.

Again we also see the XTP implementation time resolution affects the throughput rate when the load rate is set to high rate range.

The saturation and time consumed curves show that the Revised XTP achieves great improvement on rate control quality. Just like the Unicast without harmonization Burst/Rate, when the load rate is higher than 150Bpms, the rate control results are very poor.

5.8 Multicast with Harmonization Burst/Rate

For Multicast with harmonization Burst/Rate experiment, the "burst" is equal to rate*1000*0.1, and the load rate is progressively assigned the value from 10 Bpms to 1255 Bpms. The throughput rate and timing will be collected at sender under the Sandia XTP server and Revised XTP server. The experiment and results described in the following table and figure:

Sender: orchid Receiver: sunset

Table 10: Multicast with harmonization Burst/Rate data summary

Load Rate (Bpms)	Original_Sandia_ Throughput (Bpms)	Original_Sandia_ Timing (ms)	Revised_Sandia_ Throughput (Bpms)	Revised_Sandia_ Timing (ms)
10	10	107870	10	105261
15	14	72721	15	70402
20	20	52602	20	52613
25	24	43213	25	42173
30	29	36321	29	36264
35	35	30312	35	30253
50	49	21302	49	21264
75	71	14723	73	14312
100	95	11063	97	10808
125	105	9942	119	8796
150	147	7152	146	7202
200	192	5452	195	5367
250	239	4396	240	4378
300	286	3669	283	3703
350	331	3170	329	3190
400	379	2765	374	2802
500	463	2265	457	2295
600	514	2039	524	2000
700	627	1672	619	1694
800	676	1552	650	1612
900	739	1419	747	1403
1000	807	1300	813	1289
1100	826	1270	875	1198
1200	888	1181	945	1110
1250	937	1119	983	1067

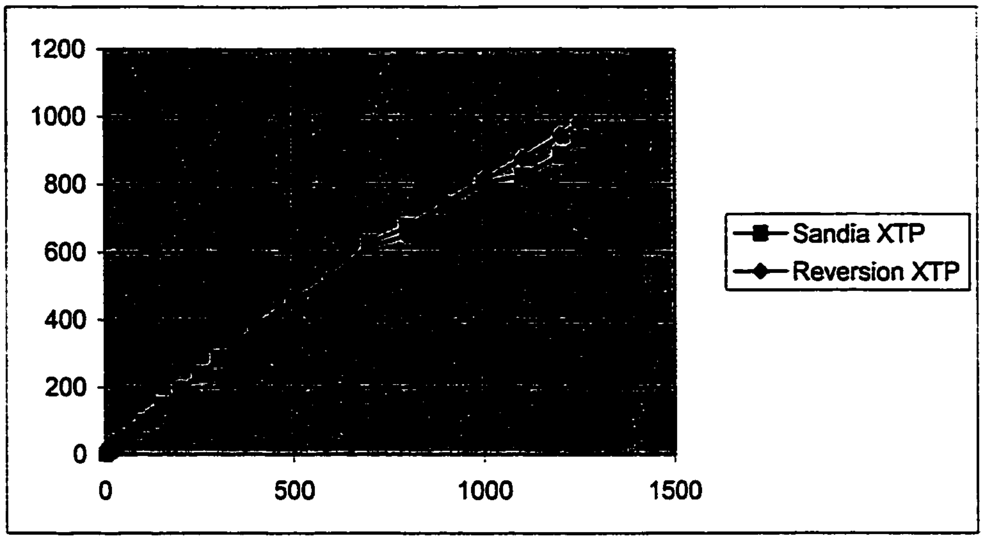

Figure 18: Multicast with harmonization Burst/Rate saturation curve

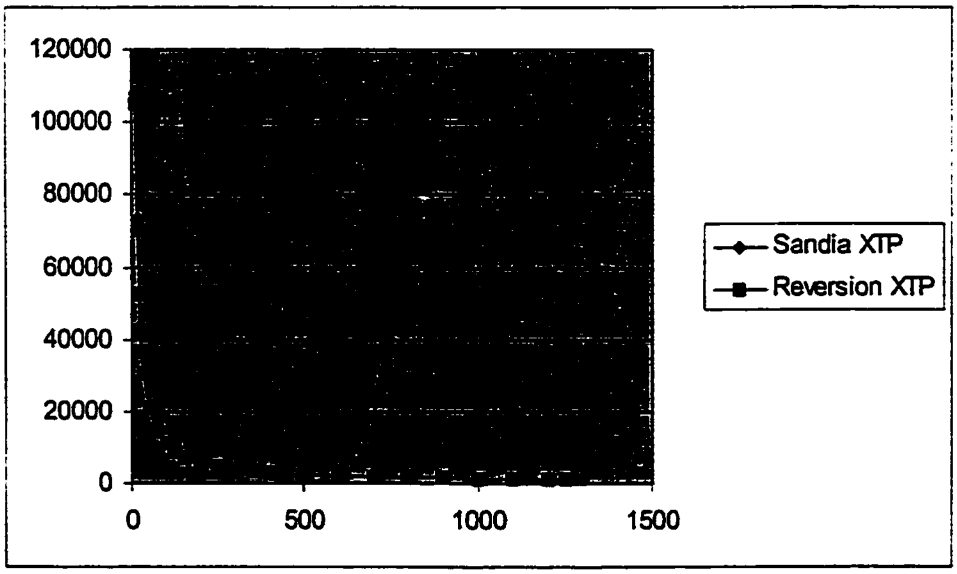

Figure 19: Multicast with harmonization Burst/Rate timing curve

As we see, with harmonization Burst/Rate, the throughput rates of both the Sandia XTP and the revised XTP are very close to load rate. However, the burst size may get very high with high load rate. If XTP can not distribute the data packet evenly in RTIMER interval, it may cause network congestion.

6 Conclusion

With the timer management class, the Revised XTP has improved the rate control. Obviously, the timer management class is a way to handle the all timer events accurately and efficiently. This project also gives us a very good example to handle time issues in the development of a network protocol implementation.

Using object oriented techniques to implement the protocol is good in the network implementation domain. But the cost of time and memory may be expensive. As we see in the experiments, sometimes the software execution time may be a bottleneck of the network. How to reduce the cost and also take advantage of object oriented techniques can be future work.

For the Sandia XTP rate control, we can get very good throughput when we set harmonization Burst/Rate. But when the Rate goes too high, the Burst also becomes bigger, which means a lot of packets are sent out in a very short time. This burst may crash or jam the network. So sending the packets out smoothly is very useful.

7 Dictionary

association:	two or more contexts connected by an active data stream.
congestion:	an overload phenomenon observed at gateways and other parts of network where the data rates of numerous senders combine to overrun a receiver.
context:	the set of state variables representing an instance of the use of XTP at an endpoint; one half of an association. A context can be both a sender and receiver.
data stream:	a simplex, sequenced data flow. An association consists of two data streams, one in each direction.
endpoint:	a host participating in an association.
end-to-end:	inclusion of all processing for sending data from one endpoint to another endpoint.
handshake:	a message exchange between two hosts where, once a host sent the initial message, it repeatedly retransmits that message until a response is obtained from the intend receiver host.
PDU:	protocol data unit. It is a data structure with protocol information.
receiver:	the context for which particular data are incoming.
rtt:	round trip time, defined as the time between when a sender transmits a packet and when it receives an acknowledgement for that packet.
sender:	the context for which particular data are outgoing.

8 Reference

1. Mentat Inc., Xpress Transport Protocol Specification, XTP Revision 4.0b. July 1998.
2. Louis Harvey, Concordia University, Computer Science, Master Major Report: In Search of a Rate Control Policy for XTP: Unicast & Multicast. March 1999
3. Sandia National Laboratories. Meta-Transport Library User's Guide, Meta-Transport Library – A protocol Base Class Library, Release 1.5.1. By: Infrastructure and Networking Research Sandia National Laboratories, P.O. Box 969 Mailstop 9011, Livermore, California 94551-0969. June 1997
4. Sandia National Laboratories. Sandia XTP User's Guide – Sandia XTP An Object Orientied Implementation of XTP 4.0 Derived from the Meta - Transport Library, Release 1.5.1. By: Infrastructure and Networking Research Sandia National Laboratories, P.O. Box 969 Mailstop 9011, Livermore, California 94551-0969. June 1997
5. Kamal Ghander, Concordia University, Computer Science, Comp 490 course project: Requirement Documentation for XTP time management Base Class. Jan. 10, 2000
6. W. Richard Stevens. Advanced Programming in the UNIX Environment. April, 1992
7. Uresh Vahalia, UNIX Internals, The New Frontiers. EMC Corporation, Hopkinton, MA. October 1995

9 Appendix

9.1 void mtldaemon::main_loop()

```
{
...
         // yjiang modify
         //satrt
        // Get the shortest from timer link
         first_cctimer=timer_link->return_first_cctimer();
         if      (first_cctimer)
                  shortest= (int) first_cctimer->get_shortest();
                 else
                         shortest = -1;
         //end
    switch(wait_on_input(shortest)) {
      case TMOUT:   // Timeout
        // Try to satisfy any outstanding work
        //yjiang modify
        //old:shortest = d_cm->satisfy(); // remove the satisfy
        if       (first_cctimer){
                                 if (first_cctimer->is_expired()){
                                         c_temp=first_cctimer->get_owner();
                                         if (c_temp){
                                                 if (c_temp->is_quiescent()){
                                                 d_cm->quiescent_routine(c_temp);
                                                 }
                                                 else{
                                                         c_temp-
>handle_timeout(first_cctimer->get_type());
                                                         c_temp->routine();
                                                         if (c_temp->is_quiescent()){
                                                                 d_cm-
>quiescent_routine(c_temp);
                                                                 }
                                                         }
                                                 }
                                         }
                                 }
        break;

      case USERREQ:  // User Request

        // Try to satisfy any outstanding work

        if (!daemon_stop) {
          get_packets_from_dds();
          //shortest = d_cm->satisfy();//yjiang modify: remove the satisfy
        }

        break;

      case PACKET:  //  Incoming Packet

        //  Get any incoming packets
        get_packets_from_dds();
        //  Try to satisfy any outstanding work
        //shortest = d_cm->satisfy();//yjiang modify: remove the satisfy
        break;
    }

  } while (!daemon_stop);
}
```

9.2 XTPcontext::XTPcontext() : context()

```
{

  DEBUG(0x0004, "XTPcontext::XTPcontext(constructor)");

  //  The context() constructor:
  //    -- sets the next and prev context pointers (used for placement
  //       on the active or quiescent list) to NULL
  //    -- constructs the snd_fifo and rcv_fifo
  //    -- allocates address structures for the unicast and multicast
  //       destination addresses

  //  Set the initial SDU size based on the header size

  c_sdu_size = c_pdu_size - HDR_LEN;

  state_mach.clear();

  c_blk_req = new xtp_trans_msg;

  //yjiang modify:
  //start
  //   CTIMER
  c_timer= new cctimer(this);
  c_timer->set_type(T_CTIMER);

  //   CTIMEOUT
  c_timeout= new cctimer(this);
  c_timeout->set_type(T_CTIMEOUT);
  // WTIMER
  w_timer= new cctimer(this);
  w_timer->set_type(T_WTIMER);

  // RTIMER
  r_timer= new cctimer(this);
  r_timer->set_type(T_RTIMER);

  // STIMER
  mcast_info.s_timer= new cctimer(this);
  mcast_info.s_timer->set_type(T_STIMER);
}
```

9.3 void XTPcontext::start_zombie()

```
{
  DEBUG(0x0004, "XTPcontext::start_zombie");
  // If the c_timeout_interval is zero, the zombie state is bypassed.

  if (c_timeout_interval == 0) {
    go_quiescent();
    return;
  }

  // Clear the sync handshake if necessary -- at this point there's
  // no need to contine it. Also clear the rtimer.

  stop_sync_handshake();
  stop_rtimer();

  // Start up the c_timeout timer

        //yjiang modify
        //start
        //old:word32 now = DAEMON->timestamp();
        //old:c_timeout = now + c_timeout_interval;
        c_timeout->set_cctimer_by_interval(c_timeout_interval);
        if (c_timeout->is_in_link())
                DAEMON->timer_link->remove_cctimer(c_timeout);
        DAEMON->timer_link->insert_cctimer(c_timeout);
        //end

  //c_timeout_armed = 1;
}
void XTPcontext::go_quiescent() {
```

```
DEBUG(0x0004, "XTPcontext::go_quiescent");

// Don't go any further if this context is already quiescent

if (is_quiescent()) {
  return;
}

// Free up the spans list

if (reader_spans) {
  free((char*)reader_spans);
  reader_spans = (span*)NULL;
}
if (pending_retrans.spans) {
  free((char*)pending_retrans.spans);
  pending_retrans.spans = (span*)NULL;
}

// Return the FIRST and JCNTL packets if necessary

if (FIRST_pkt) {
  DAEMON->d_pool->put_back(FIRST_pkt->pkt());
  delete(FIRST_pkt);
  FIRST_pkt = (FIRSTpacket*)NULL;
}
if (JCNTL_pkt) {
  DAEMON->d_pool->put_back(JCNTL_pkt->pkt());
  delete(JCNTL_pkt);
  JCNTL_pkt = (JCNTLpacket*)NULL;
}

// Empty the event queues

word64 seqno;
while (in_eq.pull(seqno) != 0);
while (out_eq.pull(seqno) != 0);

stop_sync_handshake();

if (!state_mach.is_Quiescent()) {
  state_mach.clear();
  if (TRACE && trace) {
    log_event();
    log_status();
  }
  }

  // Clean up the stuff initialized in the base class

  context::go_quiescent();

  fclu_valid = 0;

  // If this is a multicast master, make all auxils to quiesent.

  if (is_mcast_xmitter()) {

    XTPcontext* runner = get_mcast_aux_list();

    while (runner) {
      runner->go_quiescent();
      runner = runner->get_mcast_aux_next();
    }
  }
}
```

9.4 void XTPcontext::handle_timeout(word32 ttype)

```
{

  DEBUG(0x0004, "XTPcontext::handle_timeout");
  switch (ttype){

  case T_CTIMEOUT:// A CTIMEOUT expiration is serious -- abort this context.
                handle_c_timeout();
                break;

  case T_RTIMER://RTIMER expired,
                if (TRACE && trace) {
                        log_event();
        log_print("%lx RTIMER expired\n", key());
                }

                stop_rtimer();

          credit = burst;


                break;

        case T_STIMER:// Check the STIMER
          if (is_mcast_xmitter() && mcast_info.s_timer_interval > 0) {
                start_stimer();
                }
                else{
                        DAEMON->timer_link->remove_cctimer(mcast_info.s_timer);
                        }
               break;

        case T_WTIMER:// Check the WTIMER
               handle_wtimer();
               break;

        case T_CTIMER:// Check the CTIMER
               handle_ctimer();
               break;
        }
}
```

9.5 void XTPcontext::routine()

```
{
  DEBUG(0x0004, "XTPcontext::routine");

    // Ceck to see if any work can be done, like processing
    // awaiting packets.

   if (!is_registered() && !(c_rcv_fifo->empty())) {
      drain_snd_fifo();
      process_packet();
    }

    // If this context is not a multicast auxillary, check
    // retransmissions and try to drain the send fifo.

    if (!is_quiescent() &&
       !(is_mcast_auxil() || is_mcast_rcvr())) {
      retransmit();
      drain_snd_fifo();
    }

    if (is_satisfied()) {

      // If there is someone blocked waiting for the processing of
      // this context (specifically, an incoming packet), then
      // unblock that user

      unblock_user();
    }
}
```

9.6 void XTPcontext::handle_c_timeout()

```
{
  DEBUG(0x0004, "XTPcontext::handle_c_timeout");

  if (TRACE && trace) {
    log_event();
    log_print("%lx CTIMEOUT expired\n", key());
  }

  // CTIMEOUT is also set when the END bit is sent -- expiration will
  // cause the context to go quiescent. We just want to hang around
  // long enough for a lost packet to be recognized.

  if (is_zombie()) {
    if (is_blocked()) {
      c_blk_req->result_code = EXOK;
      unblock_user();
    }
    go_quiescent();
    return;
  }

  // Otherwise, this was a real CTIMEOUT expiration...

  if (TRACE && trace) {
    log_print("\tContext %lx going quiescent\n", key());
  }

  if (is_blocked()) {
    c_blk_req->result_code = EXTMOUT;
    unblock_user();
  }
  go_quiescent();
  return;
}
```

9.7 void XTPcontext::handle_wtimer()

```
{
  DEBUG(0x0004, "XTPcontext::handle_wtimer");

  // If this is a multicast master, check that all of the
  // members of the multicast group have responded. If they
// haven't, do the normal sync handshake.

if (is_mcast_xmitter() && mcast_all_responded()) {

  if (sync_handshake_open) {
    stop_sync_handshake();
  }
  else {
    stop_wtimer();
  }
  if (FIRST_pkt) {
    DAEMON->d_pool->put_back(FIRST_pkt->pkt());
    delete(FIRST_pkt);
    FIRST_pkt = (FIRSTpacket*)NULL;
  }

  // Stop the tspec negotiation (even if it's not open).

  tspec_neg_open = 0;

  // Update the retransmission threshold

  word64 m_rseq = mcast_auxils_rseq();
  kseq = m_rseq;

  // Update the alloc.

  alloc = mcast_auxils_alloc();

  // Move the dseq mark in send buffer

  c_s_bm->acknowledged(m_rseq);

  // If a user is blocked...

  if (is_blocked()) {

    // if this is a response to an SREQ or DREQ, then update the
    // send buffer manager and unblock the user. We know
    // that this is a response by matching the sequence numbers;
    // if the assoc is closed, we don't care. The only two flags
    // we need to communicate to the blocked process are the
    // END and RCLOSE flags.

    if (((get_blk_type() == XTP_SEND) ||
        (get_blk_type() == XTP_SEND_CNTL)) &&
        ((m_rseq >= c_blk_seq) || is_assoc_closed())) {
       ((xtp_trans_msg*)c_blk_req)->options |= local_modes;
       unblock_user();
     }
   }
  return;
}

// If this wtimer expiration is not part of a sync hanshake, start
// the sync handshake up.

int res;

if (!sync_handshake_open) {

  if (TRACE && trace) {
    log_event();
    log_print("%lx WTIMER expired\n", key());
  }

  start_sync_handshake();
}

else {

  // The expiration is part of the handshake, check the retries.

  if (num_retries != 0 && retry_count == 0) {
    log_event();
    log_print("%lx Retry count exceeded:\n", key());
    log_print("\tsynchronizing handshake failed, ");
    log_print("giving up on context.\n");
    if (is_blocked()) {
      c_blk_req->result_code = EXTMOUT;
      unblock_user();
    }
    go_quiescent();
    return;
  }

  if (TRACE && trace) {
    log_event();
    log_print("%lx WTIMER expired, retry count = %u\n",
          key(), retry_count);
  }
}
```

```
// If the FIRST packet has not been answered (the WTIMER expires
// without any received packets) then retransmit the FIRST and
// treat this just like a sync handshake.

if (FIRST_pkt && num_pkts_rcvd == 0) {

  saved_time_valid = 0;

  if (TRACE && trace) {
    log_event();
    log_print("%lx Retransmitting: ", key());
    FIRST_pkt->log_pkt(1);
    log_print("\tDestination: %s\n", is_mcast_xmitter()?
                        c_mcast_dest->pr_hostid():
                        c_ucast_dest->pr_hostid());
  }

  res = FIRST_pkt->send(is_mcast_xmitter()?c_mcast_dest:c_ucast_dest);
  if (res < 0) {
    log_event();
    log_print("ERROR: send failed for packet: %d\n", res);
  }

  if (!is_mcast_xmitter() && num_retries != 0) {
      retry_count--;
  }
  start_wtimer(backoff_K);

  // Increase the backoff exponentially, watching for overflow

  if (backoff_K < 0x4FFFFFFF) {
    backoff_K *= 2;
  }
  return;
}

// If we're listening to join a multicast association, send
// the JCNTL again and hope for a response. This is multicast
// receiver polling.

if (is_listening() && JCNTL_pkt) {

  if (TRACE && trace) {
    log_event();
    log_print("%lx Retransmitting:", key());
```

```
      JCNTL_pkt->log_pkt(1);
      log_print("\tDestination: %s\n", c_mcast_dest->pr_hostid());
    }

    saved_time_valid = 0;
    if ((res = JCNTL_pkt->send(c_mcast_dest)) < 0) {
      log_event();
      log_print("ERROR: send failed for packet: %d\n", res);
    }

    start_wtimer(backoff_K);

    // Increase the backoff exponentially, watching for overflow

    if (backoff_K < 0x4FFFFFFF) {
      backoff_K *= 2;
    }
    return;
  }

  // If we've kept a copy of the JCNTL packet, then failure
  // to get a response from this packet's SREQ caused the
  // timeout, so retransmit the JCNTL.

  if (JCNTL_pkt) {

    JCNTL_pkt->get_header()->cmd.options |= SREQ;
    saved_sync++;
    saved_sync_seq = JCNTL_pkt->get_header()->seq;
    saved_time = DAEMON->timestamp();
    saved_time_valid = 1;
    JCNTL_pkt->get_header()->sync = saved_sync;

    if (TRACE && trace) {
      log_event();
      log_print("%lx Retransmitting:", key());
      JCNTL_pkt->log_pkt(1);
      log_print("\tDestination: %s\n", c_ucast_dest->pr_hostid());
    }

    if ((res = JCNTL_pkt->send(c_ucast_dest)) < 0) {
      log_event();
      log_print("ERROR: send failed for packet: %d\n", res);
    }
    if (num_retries != 0) {
      retry_count--;
      }

      start_wtimer(backoff_K);

      // Increase the backoff exponentially, watching for overflow

      if (backoff_K < 0x4FFFFFFF) {
        backoff_K *= 2;
      }
      return;
    }

    // Otherwise just send a control packet. If the traffic
    // specification negotiation is open, send a TCNTL;
    // otherwise, send a (E)CNTL

    if (tspec_neg_open) {
      send_tcntl(SREQ | get_sent_modes());
    }
    else {
      send_cntl(SREQ | get_sent_modes());
    }

    // Start the WTIMER with backoff_K times a smoothed
    // round-trip time estimate. Double the backoff for
    // use next time.

    if (num_retries != 0)
      retry_count--;

    start_wtimer(backoff_K);

    // Increase the backoff exponentially, watching for overflow

    if (backoff_K < 0x4FFFFFFF) {
      backoff_K *= 2;
    }
  }

  void XTPcontext::handle_ctimer() {

    DEBUG(0x0004, "XTPcontext::handle_ctimer");

    if (TRACE && trace) {
      log_event();
      log_print("%lx CTIMER expired\n", key());
```

```
}

// For a CTIMER expiration, we should just see if any activity has
// happened during this interval. If not, we check to see if the user's
// process is still alive (is_user_alive()). This check is useful
// so that we can clean up any dead user's shared resources. We must
// set CTIMER so that it will expire before a process id or key value
// roll over. To this end the protocol directs that this timer should
// have an interval less than 3600 seconds (one hour).
//
// If the process is really dead, we try to clean up the ipc
// structures it may have left behind, and go quiescent.

// Did we do any work?

if (pkts_rcvd_in_c_interval > 0) {

  // Yes: Clear the counters for another interval and return

  pkts_rcvd_in_c_interval = 0;

       //yjiang modify
       //start
       //old:c_timer += c_timer_inter;
       c_timer->set_cctimer(c_timer_interval+c_timer->get_cctimer_val());
       if (c_timer->is_in_link())
              DAEMON->timer_link->remove_cctimer(c_timer);
       DAEMON->timer_link->insert_cctimer(c_timer);
       //end
      if (TRACE && trace) {
  log_event();
  log_print("%lx CTIMER Start\n", key());
}
  return;
}

// If the process is still alive, start a synchronizing handshake if
//   (1) this context is not just simply waiting for an
//       association startup packet, or
//   (2) a sync handshake is not already going.

if (is_user_alive()) {
  if (!is_registered() && !is_listening() &&
              !c_timeout->is_in_link()) {
    //!c_timeout_armed) {//yjiang del
    start_sync_handshake();
```

```
      if (tspec_neg_open)
        send_tcntl(SREQ | get_sent_modes());
      else
        send_cntl(SREQ | get_sent_modes());
    }

          //yjiang modify
          //start
          //old:c_timer += c_timer_inter;
          c_timer->set_cctimer(c_timer_interval+c_timer->get_cctimer_val());
          if (c_timer->is_in_link())
                  DAEMON->timer_link->remove_cctimer(c_timer);
          DAEMON->timer_link->insert_cctimer(c_timer);
          //end
        if (TRACE && trace) {
    log_event();
    log_print("%lx CTIMER Start\n", key());
  }
    return;
  }

  // Otherwise, try to clean up the remains.

  log_event();
  log_print("%lx Process %d died, cleaned up context.\n", key(), upid());

  // Go quiescent

  go_quiescent();
}
int XTPcontext::initialize(user_request* request) {

  DEBUG(0x0004, "XTPcontext::initialize");

  register xtp_reg_msg* xrg = (xtp_reg_msg*)request;

  // First record the buffers sizes

  c_snd_buf_size = xrg->snd_buf_size;
  c_rcv_buf_size = xrg->rcv_buf_size;

  // Then call context::initialize() to
  //    -- set the key to the currently assigned key
  //    -- record the user's pid
  //    -- install the buffers
```

```
if ((request->result_code = context::initialize(request)) != EXOK) {
  return(request->result_code);
}

// Set the context shell states

state_mach.clear();
c_blk_state = NOT_BLOCKED;
sent_FIRST = 0;
should_send_END = 0;

// Clear the options masks

sticky_mask = (short16)0;
local_modes = (short16)0;
sent_modes = (short16)0;
perm_modes = (short16)0;
extra_modes = (word32)0;
yes_mask = (short16)0;
no_mask = (short16)0;

// Fill in the general header; most of this will remain constant over
// all packets sent out from this context

memset((char*)&hdr, (char)0, sizeof(header));
hdr.key = key();

// The stickies are initialized here, and set during packet exchanges.

set_stickies(xrg->options);
perm_modes = xrg->options;
set_sent_modes(xrg->options);

// Set the local modes.

set_local_modes(xrg->options & MULTI);

// Set the extra modes.

set_extra_modes(xrg->extra_modes);
state_mach.set_extra_modes(xrg->extra_modes);

// The yes_mask and no_mask are used for traffic negotiaion. The
// yes_mask indicates what modes must be set, the no_mask indicates
// what modes must not be set. We "or" in the MULTI bit since this
// mode can never be changed during an association; it also helps
// distinguish which FIRST packets are acceptable.

yes_mask = xrg->yes_mask | xrg->options & MULTI;
no_mask = xrg->no_mask | (xrg->options & MULTI)?(short16)0:MULTI;

// Set the starting sequence numbers

hdr.seq = c_s_bm->get_head();
hseq = cntl.rseq = c_r_bm->get_tail();
dseq = c_s_bm->get_dseq();

// Sort information

hdr.sort = (hdr.cmd.options & SORT)?(short16)priority():(short16)0;

dreq_request = 0;  // No requests for CNTL via a DREQ yet.
saved_edge = 0;   // Initial EDGE value is 0
cur_edge_val = (short16)0;  // Initial outgoing EDGE value is 0

// Fill in the general control segment; this is where some of
// the context's state is kept

cntl.echo = 0;
saved_sync = 0;
saved_sync_seq = word64(0);
rcvd_sync = 0;
rcvd_echo = 0;
num_pkts_rcvd = 0;
expected = 0;
FIRST_pkt = (FIRSTpacket*)NULL;
JCNTL_pkt = (JCNTLpacket*)NULL;
pending_retrans.nspan = 0;

cntl.alloc = c_r_bm->get_alloc();
excess_alloc = xrg->excess_alloc;

if (!(hdr.cmd.options & RES))
  cntl.alloc += excess_alloc;
reader_nspan = 0;

maxspans = umax32(xrg->maxspans, 1);
if ((reader_spans = (span*)malloc(maxspans * sizeof(span)))
                              == (span*)NULL) {
  xrg->result_code = EXMEM;
  return(EXMEM);
}
```

```
if ((pending_retrans.spans = (span*)malloc(maxspans * sizeof(span)))
                                    == (span*)NULL) {
  xrg->result_code = EXMEM;
  return(EXMEM);
}

// Adjust the PDU and SDU sizes. Later, PDU and SDU sizes can be
// further adjusted if the endpoints of the association negotiate
// traffic parameters.

c_pdu_size = min(DAEMON->d_out_dds->get_maxpdu(), xrg->pdu_size);
c_sdu_size = c_pdu_size - sizeof(header);

// Set the edge frequency

edge_freq = pkt_count = xrg->edge_freq;

// Initialize the retransmission marker

kseq = word64(0);
kseq_sync = 0;
tseq = word64(0);

// Set the initial alloc. The "10" is just a guess, there needs to
// be a better way to get the initial allocation

alloc = 10 * c_sdu_size;

// Set the tspec to the default. Until told otherwise, we'll
// assume that traffic specifier format 0x01 is used.

tspec.tlength = (short16)(sizeof(traffic_1) + 4);
tspec.service = UNSPEC;
tspec.tformat = (byte8)0x01;

// Set the maxdata, rate and burst values. Maxdata is the maximum
// Information segment size, which is the sdu_size.

maxdata = tspec.ts1.maxdata = c_sdu_size;

if (extra_modes & OUTRATEOFF)
  rate = tspec.ts1.outrate = 0;
else if (xrg->rate)
  rate = tspec.ts1.outrate = xrg->rate;
else
  rate = tspec.ts1.outrate = DAEMON->d_out_dds->get_rate();
```

```
if (extra_modes & OUTBURSTOFF)
  burst = credit = tspec.ts1.outburst = 0;
else if (xrg->burst)
  burst = credit = tspec.ts1.outburst = xrg->burst;
else
  burst = credit = tspec.ts1.outburst = DAEMON->d_out_dds->get_burst();

if (extra_modes & INRATEOFF)
  inrate = tspec.ts1.inrate = 0;
else
  inrate = tspec.ts1.inrate = DAEMON->d_in_dds->get_rate();

if (extra_modes & INBURSTOFF)
  inburst = tspec.ts1.inburst = 0;
else
  inburst = tspec.ts1.inburst = DAEMON->d_in_dds->get_burst();

tspec_changed = 0;
tspec_neg_open = 0;

// Clear the address segment and fclu my_entry; the address_segment
// gets set when the context is bound (see the context_manager), and
// the fclu my_entry gets set when the FIRST packet arrives.

memset((char*)&address, (char)0, sizeof(address_segment));
memset((char*)&my_entry, (char)0, sizeof(my_entry));
fclu_valid = 0;

// Set the time-out values.

c_timer_interval = CTIMER_INTERVAL * 1000; // express in msec

//yjiang modify
//start
//c_timer = DAEMON->timestamp() + c_timer_interval; // start CTIMER
      if (TRACE && trace) {
  log_event();
  log_print("%lx start CTIMER\n", key());
}
c_timer->set_cctimer_by_interval(c_timer_interval);
if (c_timer->is_in_link())
        DAEMON->timer_link->remove_cctimer(c_timer);
DAEMON->timer_link->insert_cctimer(c_timer);
//end
```

```
pkts_rcvd_in_c_interval = 0;
saved_time_valid = 0;
saved_time = 0;

// Set the retry count and C_TIMEOUT timer

num_retries = xrg->retry_count;
c_timeout_interval = xrg->c_timeout_interval * 1000;  // express in msec

// Set the WTIMER limit

w_timer_limit = xrg->w_timer_limit * 1000;  // express in msec

// One of these mechanisms must be enabled

if (num_retries == 0 && c_timeout_interval == 0) {
  xrg->result_code = EXMECH;
  return(EXMECH);
}

// Disarm the WTIMER, RTIMER, and CTIMEOUT; they will be armed during a
// synchronizing handshake

//w_timer_armed = r_timer_armed = c_timeout_armed = 0;//del by yjiang
sync_handshake_open = 0;

//yjiang modify
//start
DAEMON->timer_link->remove_cctimer(w_timer);
DAEMON->timer_link->remove_cctimer(r_timer);
DAEMON->timer_link->remove_cctimer(c_timeout);
//end

// Set the initial roundtrip estimate and the variables for
// calculating the smoothed RTT.

rtt = SRTT = xrg->init_rtt;
RTTV = 0;

// Initialize multicast information

memset((char*)&mcast_info, (char)0, sizeof(mcast_info));
mcast_info.max_act_rcvrs = xrg->mcast_max_act_rcvrs;
mcast_info.min_act_rcvrs = xrg->mcast_min_act_rcvrs;
mcast_info.s_timer_interval = 0;
```

```
  // Become registered

  state_mach.initialize();
  if (TRACE && trace) {
    log_event();
    log_status();
  }

  xrg->result_code = EXOK;

  return(EXOK);
}
```

9.8 void XTPcontext::start_wtimer(word32 factor)

```
{

  DEBUG(0x0004, "XTPcontext::start_wtimer");

  // Can't have a "0" factor

  if (factor == 0) factor = 1;

  // Take an initial guess at the duration

  word32 duration = SRTT + 2*RTTV;

  // Calculate the maximum factor allowed so as not to overflow the
  // word32 timer; the real factor is the min of this max and the
  // offered parameter

  word32 maxfactor = (word32)(0x4FFFFFFF / duration);
  factor = min(maxfactor, factor);

  // Get the real duration; if there is a w_timer_limit, let the
  // duration be limited by it

  if (w_timer_limit != 0) {
    duration = umin32((word32)(duration*factor), w_timer_limit);
  }
  else {
    duration = (word32)(duration*factor);
  }

  if (TRACE && trace) {
    log_event();
    log_print("%lx Starting WTIMER, duration: %u\n", key(), duration);
  }

        //yjiang modify
        //start
        //old:word32 now = DAEMON->timestamp();
        //old:w_timer = now + duration;
        w_timer->set_cctimer_by_interval(duration);
        if (w_timer->is_in_link())
                DAEMON->timer_link->remove_cctimer(w_timer);
        DAEMON->timer_link->insert_cctimer(w_timer);
        //end


  //w_timer_armed = 1;//del by yjiang
}
```

9.9 void XTPcontext::stop_wtimer()

```
{  DEBUG(0x0004, "XTPcontext::stop_wtimer");

  //if (TRACE && trace && w_timer_armed) {//yjiang modify
  if (TRACE && trace && w_timer->is_in_link()) {
    log_event();
    log_print("%lx Stopping WTIMER\n", key());
  }

  //w_timer_armed = 0;//yjiang:shall be delete

  //yjiang modify
  //start
  DAEMON->timer_link->remove_cctimer(w_timer);
  //end

  if (FIRST_pkt) {
    DAEMON->d_pool->put_back(FIRST_pkt->pkt());
    delete(FIRST_pkt);
    FIRST_pkt = (FIRSTpacket*)NULL;
  }
}
```

9.10 void XTPcontext::start_rtimer()

```
{  DEBUG(0x0004, "XTPcontext::start_rtimer");

  if (rate == 0) return;

                if (TRACE && trace) {
                        log_event();
```

```
        log_print("%lx RTIMER Start\n", key());
                }

        //yjiang modify
        //start
  //old:word32 now = DAEMON->timestamp();

  // Here rate and burst are in units of msec.
  //old:r_timer = now + (word32)(burst/rate);
        r_timer->set_cctimer_by_interval((word32)(burst/rate));
        if (r_timer->is_in_link())
                DAEMON->timer_link->remove_cctimer(r_timer);
        DAEMON->timer_link->insert_cctimer(r_timer);
        //end


  //r_timer_armed = 1;//yjiang: shall be remove

}
```

9.11 void XTPcontext::stop_rtimer()

```
{  DEBUG(0x0004, "XTPcontext::stop_rtimer");

  //r_timer_armed = 0;//yjiang: shall be remove

  //yjiang modify
  //start
  DAEMON->timer_link->remove_cctimer(r_timer);
  //end

  credit = burst;
}
```

9.12 void XTPcontext::start_stimer()

```
{  DEBUG(0x0004, "XTPcontext::start_stimer");

  //yjiang modify
  //start
  //old:mcast_info.s_timer = DAEMON->timestamp() + mcast_info.s_timer_interval;
        mcast_info.s_timer->set_cctimer_by_interval(mcast_info.s_timer_interval);
        if (mcast_info.s_timer->is_in_link())
                DAEMON->timer_link->remove_cctimer(mcast_info.s_timer);
        DAEMON->timer_link->insert_cctimer(mcast_info.s_timer);
        //end
}
```

9.13 void XTPcontext::start_sync_handshake()

```
{  DEBUG(0x0004, "XTPcontext::start_sync_handshake");

  if (TRACE && trace) {
    log_event();
    log_print("%lx Starting synchronizing handshake, rtt: %u\n",
           key(), rtt);
  }

  word32 now = DAEMON->timestamp();//yjiang:shall be remove

  //  Load the CTIMEOUT timer with its initial value, but only if
  //  the interval is not 0. If the interval is 0, the CTIMEOUT timer
  //  has been disabled.

  if (c_timeout_interval != 0) {

         //yjiang modify
    //start
    //old:c_timeout = now + c_timeout_interval;
        if (TRACE && trace) {
    log_event();
    log_print("%lx C_TIMEOUT Start\n", key());
  }
         c_timeout->set_cctimer_by_interval(c_timeout_interval);
         if (c_timeout->is_in_link())
                DAEMON->timer_link->remove_cctimer(c_timeout);
         DAEMON->timer_link->insert_cctimer(c_timeout);
         //end

    //c_timeout_armed = 1;
  }

  //  Reset the retry_count to its initial value...

  retry_count = num_retries;

  //  And set an exponential backoff variable K to 1.

  backoff_K = 1;

  sync_handshake_open = 1;
}
```

9.14 void XTPcontext::stop_sync_handshake()

```
{  DEBUG(0x0004, "XTPcontext::stop_sync_handshake");
```

```
  if (sync_handshake_open && !state_mach.is_Quiescent()) {
    if (TRACE && trace) {
      log_event();
      log_print("%lx Stopping synchronizing handshake\n", key());
    }
  }

  //c_timeout_armed = 0;//yjiang:shall be remove
  //w_timer_armed = 0;//yjiang: shall be remove

  //yjiang modify
  //Start
  DAEMON->timer_link->remove_cctimer(c_timeout);
  DAEMON->timer_link->remove_cctimer(w_timer);
  //end

  backoff_K = 1;
  sync_handshake_open = 0;
}
```

9.15 void XTPcontext::start_zombie()

```
{  DEBUG(0x0004, "XTPcontext::start_zombie");

  // If the c_timeout_interval is zero, the zombie state is bypassed.

  if (c_timeout_interval == 0) {
    go_quiescent();
    return;
  }

  // Clear the sync handshake if necessary -- at this point there's
  // no need to contine it. Also clear the rtimer.

  stop_sync_handshake();
  stop_rtimer();

  // Start up the c_timeout timer

        //yjiang modify
        //start
        //old:word32 now = DAEMON->timestamp();
        //old:c_timeout = now + c_timeout_interval;
        c_timeout->set_cctimer_by_interval(c_timeout_interval);
        if (c_timeout->is_in_link())
                DAEMON->timer_link->remove_cctimer(c_timeout);
        DAEMON->timer_link->insert_cctimer(c_timeout);
        //end

  //c_timeout_armed = 1;
}
```

Index