IN SEARCH OF A RATE CONTROL POLICY FOR XTP: UNICAST & MULTICAST

LOUIS HARVEY

A REPORT
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE AT
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

MARCH 1999

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-43554-7

Canada

Abstract

In search of a rate control policy for XTP: unicast & multicast

Louis Harvey

From the sender's viewpoint, the combined underlying data delivery medium and the receiver(s) can be viewed as a global system endowed with absorption capacities. But how could a sender dynamically adjust its data transfer rate to best match the apparent residual absorption capacities of this system? To resolve this question, use could be made of the concept of saturation curves whereby a system responds linearly (or almost) to increasing load and then yield.

The purpose of this report is to investigate the rate control mechanisms and the rate control performance capabilities of one implementation of the Xpress Transport Protocol (XTP) - SandiaXTP. The structure of SandiaXTP as a whole, and also more particularly its rate control mechanisms, are analysed with the help of Object Modeling Notation (O.M.T.). Some changes to the rate control algorithm are proposed, followed by the presentation and interpretation of 26 unicast and multicast rate control experiments done in a LAN environment. The study shows that rate control with a user level implementation such as SandiaXTP needs careful consideration of many factors if the quality of the rate control effectively exercised by the software should meet the expectations.

Acknowledgements

Though they do not wear the official title of **magician**, many persons involved in the Computer Science field are **de facto** magicians. In my view, Dr. J.W. Atwood is certainly one of those modern day magicians, practising his magic in the domain of Computer Science. I was fortunate that he accepted to supervise my work for my Major Report. Countless number of times, when in deep confusion, I would S.O.S. him for a meeting. After the meeting, I would return enthusiastically to my work with the feeling "Surely, Dr Atwood is a magician - how could he turn around my confusion so quickly into insight?". Admittedly, his thinking pervades this whole report, and I owe him my deepest gratitude.

During the course of this study, I was also fortunate to attend to one working session of the XTP Forum at Concordia University, Montréal, December 1997. There, I also observed other *magicians* performing their art of converting problems into plain solutions. Thanks to Dr. Atwood for inviting me to this conference.

Certainly, even if far from sight, Dr. Tim Strayer has been present every day of my study through the provision of the SandiaXTP implementation. And when potential employers ask me to describe my *practical* experience about the client-server model, about object modeling, about C++ programming, about network programming, I can quote my work with SandiaXTP. Thank you very much Dr. Strayer.

Thanks to Dr. Terry Fancott, the examiner for this report. I appreciate his comments.

Thanks also to fellow students Marie A. Wallace, Torsten Auerbach, Deric Sullivan, and Pierre Falcot who have blazed the trail before me. I greatly benefited from their work.

Finally, a special thanks to Lucille, my life companion, for her unfailing support.

Contents

Appendices

List of Figures

List of Tables

1 Summary of the report

1.1 Context of the study

Throughout the period of time devoted to the investigatory work and to the writing of this report, the two most important themes have been: (1) earliest in time, the *multicasting* aspect; (2) then later, increasingly, the *rate control* aspect. For various reasons, such as the non-feasibility of conducting multicast rate control experiments using the Internet/Mbone environment outside the Computer Science Department at Concordia University and also the desirability of a better framework for interpreting former as well as current rate control experiments results, there was a gradual shift of emphasis from the multicast aspect to the rate control aspect.

1.2 Report organization

The logical progression of the subject material contained in this report follows the familiar project life cycle sequence of phases. The report begins with a global planning statement presented in Chapter 2 - **Introductory notes**, and winds up with Chapter 9 - **Concluding remarks**, which is akin to the last phase of a project (i.e., the project review phase).

As the root of the report lies with the Xpress Transport Protocol Revision 4.0 (XTP4.0), Chapter 3 centers around the protocol itself. First, XTP is shown to have been conceived with the OSI-Basic Reference Model (OSI-BRM) in mind. Being a transport level protocol, XTP provides the type of functionalities assigned by the OSI-BRM to this layer, which includes flow control, error control, rate control, and the like. In accordance with the OSI model, XTP presumes that the services of an underlying network layer are available down below. Similarly, XTP provides services to a layer above. Throughout this report, this layer above XTP bears many names, such as "the user", or "the client", or also "the application program". For the later rate control experiments, the role of the "user" (i.e., the above layer) is played by the "mmetric" application program.

Once XTP has been cast within the OSI-BRM, the report gives a brief historical perspective of the evolution of the XTP project, including a glimpse at the functioning of the XTP Forum, the social body that turned XTP from a dream to a reality. More particularly, Section 3.2 shows that XTP can be perceived as a superset of the TCP and UDP protocols, with additional features provided such as multicasting and quality of service, of which the rate control aspect is widely covered in this report.

Then, the report provides a global overview of the protocol itself (Section 3.3). As XTP is a large and sophisticated protocol, it was necessary to favor one angle of attack in accordance with the problem space treated in this report. Such an angle consists of highlighting the parameterized dimension of XTP whereby the protocol provides orthogonal mechanisms that can be selected through the setting of arguments and switches from above. Table 3 presents a summary of these parameterized features.

Gradually, the shift of the XTP global presentation is toward the SandiaXTP implementation. In this direction, Section 3.4 covers the working environment of XTP, showing XTP as peer of other protocols on different LANs. Section 3.5 presents a high level view of the unicast as well as the multicast models as conceived by XTP, to cover more precisely XTP specific concepts and terminology such as *context*. The global presentation of XTP terminates with a discussion of protocol implementation strategies.

The focus of the next two Chapters (Chapters 4 & 5) is to articulate a model view of the SandiaXTP implementation of XTP, which embeds the protocol into a user level daemon process, at the service of "user level" client processes (the layer above XTP). In terms of the project life cycle model, we have now reached the **Design** stage, even though we use the terminology "SandiaXTP **implementation**". To grow this model, and as the *Object paradigm* was used to develop SandiaXTP, we make use of Object Modeling Techniques (O.M.T.), such as Class Diagrams, Object Diagrams, and Event Traces. Being a work of reverse engineering, the model is derived from studying the source code with two scenarios in mind (starting the daemon, and starting one client), then preparing Event Traces, deriving Object Diagrams from the Event Traces, and finally deriving the Class Diagrams from the Object Diagrams.

This work of analysis of the SandiaXTP implementation is performed for two purposes: (1) first to understand the global structure of the SandiaXTP implementation of XTP, which has a set of base classes embedded into a Meta Transport Library (MTL), and a set of protocol specific derived classes (sandiaXTP); (2) then to understand precisely the rate control mechanisms used by SandiaXTP to implement rate control, which eventually culminates into the formulation of the rate control algorithm (as presented in Figure 24), and a finite state machine (Figure 25) to account for the dynamic behavior of the daemon.

Studying the whereabouts of the SandiaXTP rate control algorithm have led to observing some shortcomings, and then to attempts at correcting or improving them. Chapter 6 presents and discusses the changes introduced to the SandiaXTP rate control algorithm, which cover esentially three aspects: (1) introduction of a linked list of timers (for all timers) to help understandability of the rate control algorithm; (2) turning a 50 ms hardcoded minimum timeout value conveyed to the select() system call into a tunable SELECT_FLOOR threshold to reduce the duration of wait periods; (3) introducing a MAXANTICIPATION margin to compensate for the fact that the select() system call may return slightly earlier than specified (causing another long waiting period), and also for *soon to expire* timers (also causing the daemon to yield to select() for a much longer wait period).

Finally, the last three chapters (Chapters 7, 8, and 9) are concerned with the design, presentation and interpretation of the rate control experiments done within the context of a LAN environment. The goals of these experiments are to gain some insight into the rate control limitations of a user level implementation, and also to challenge SandiaXTP with a wide range of command line options so as to learn to circumvent its idiosyncrasies for eventually conducting meaningful multicast rate control experiments within the Internet/Mbone environment.

Chapter 7 defines the data organization for conducting and recording the experiments. Chapter 8 presents the results of the experiments and provides tentative interpretations, as well as discusses the success of the changes introduced to the SandiaXTP rate control mechanisms. Finally, Chapter 9 is a global discussion of the report and includes some recommendations for: (1) revising the specification of the protocol to incorporate success criteria and minima to be met by any implementation for partially attainable features such as rate control; (2) revising SandiaXTP to incorporate some of the changes proposed in this report; (3) conducting further rate control experiments.

The report also includes many detailed appendices, whose information content is in an "unfinished" state, but which could be useful to some other student of XTP.

1.3 How to use this report

Table 1: Thematic guide for using this report

Specific interest of the user	**Parts to consult**
Obtain a global **overview** of the results of the report.	Section 8.4, (p.196)
Read about the various features offered by XTP	Section 3.3, (p.18)
Learn about protocol **implementation strategies**	Section 3.6, (p.48)
Discover how **rate control** is conceived by XTP and how it is implemented by SandiaXTP	Section 3.3.4, (p.22) Chapter 5, (p.94)
Explore the changes made to the SandiaXTP rate control **algorithm**, and see the consequences	Chapter 6, (p.120) Section 8.4, (p.196)
Observe how the **Object paradigm** has been used to implement XTP	Chapter 4, (p.51)
Focus on the **multicast** aspect	Section 3.3.1, (p.20) Section 3.4.3, (p.33) Section 3.5.2, (p.42) Section 8.3.6, (p.192)
Do more rate control experiments with SandiaXTP, such as on the Internet (look up optimized experiments)	Section 8.4, (p.196)
Check the **details** of how the experiments were done (client programs, Perl scripts)	Appendix C, (p.252)

To ease the task of an eventual reader who might be interested to consult or use this report with some specific interest in mind, Table 1 provides a thematic guide juxtaposing anticipated interests and the corresponding most pertinent section numbers.

2 Introductory notes

"Quand on ne sait pas ce que l'on cherche,
on ne voit pas ce que l'on trouve."

Dr Claude Bernard (1813-1878)
Introduction à l'étude de la médecine expérimentale.

English translation:
If you do not know what you are looking for,
you will not see what you find.

Mainly written during the summer of 1997, after a long gestation period devoted to integrating as many multicast concepts as possible, and before effectively starting the live multicast experiments, this part of thc report is meant to give the author (and the reader) a sense of direction and purpose (as per the saying of the great French physiologist, Dr. Claudc Bernard). Only subsection 2.4 (**Update on expectations**) is meant to be written later, when all experiments are done and the report is almost completed.

2.1 Saturation

Saturation is a pervasive phenomenon that has been studied in many areas of activity. In the domain of structural mechanics, a material will withstand increasing load (with accompanying deformation) up to a point where its capacity is exceeded, and failure may follow if the load increase continues. The *stress-strain* curve models this behavior. In the domain of economics, it is reported that additional units of labor input will produce increasing units of output, up to a point where a plateau is reached and more labor input just produces less output. This is known as the *Law of variable proportions.* Thrashing is another well known saturation phenomenon in the domain of Computer Science. With virtual memory, the degree of multiprogramming can be increased, up to a point where the system does more paging than useful work.

With these phenomena, the recurring pattern consists of: (a) a **rising** phase along which the system in question responds positively to an increasing demand; (b) a

plateau where the previous trend changes; (c) a **downfall** where the system fails. Examples of curves depicting this behavior are given in Figure 1. Given the nature of things, a rational approach (one that has been used in many engineering disciplines, such as Structural Engineering) is to experiment with a given system, draw its behavioral curve under loading conditions, and then adopt some sensible policy based on these observations.

2.2 Saturation and the Internet

The Internet is a large sytem, one that is subject to a greatly varying load over time. When its **plateau** is reached, its natural reaction is to silently drop packets. When reliable communication is underway, the saturation phenomenon becomes evident through an increase in activity of the error mechanisms; more packets get lost and need to be retransmitted. For instance, the phenomenon of saturation was well observed in 1986 [VANJ], when a sequence of congestion collapses led to the introduction of a series of new algorithms, collectively known as *slow-start*. These algorithms are now part of the overall TCL/IP protocol suite strategy to effect congestion avoidance and control on the Internet.

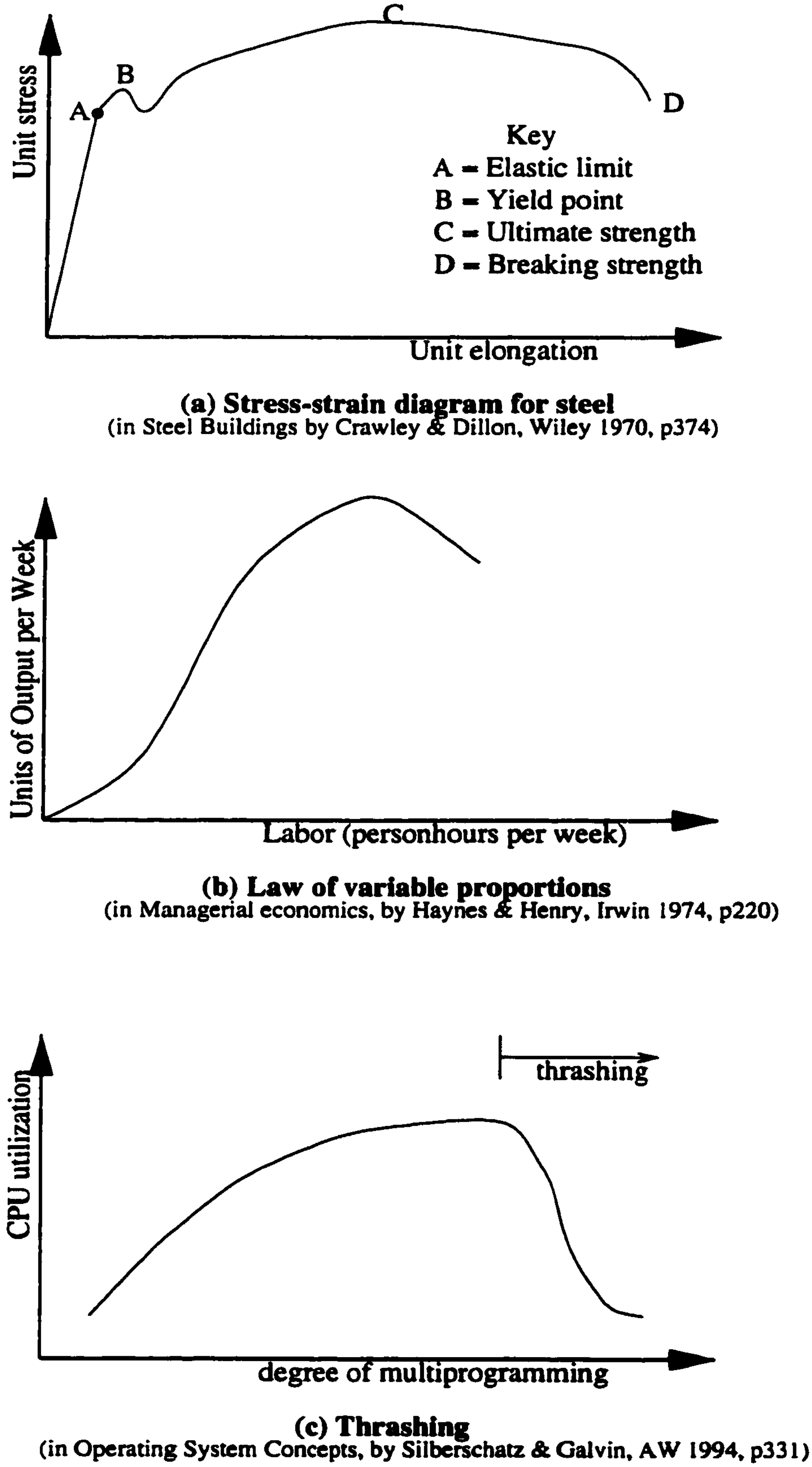
2.3 XTP at the HSP Lab

The Xpress Transport Protocol (XTP) has inbuilt mechanisms for congestion control, but no pre-defined congestion control policy in order to behave as a **nice** user when running above IP in the Internet environment. Under the direction of Dr J.W. Atwood, at the HSP[1] Lab, a series of experimental studies was launched in 1994 to investigate the rate control mechanisms of XTP in a Wide Area Network (WAN) environment, using one implementation of XTP - SandiaXTP.

2.4 Expectations of the present study

The present study is the fourth (for the other ones, refer to ([AUE], [SUL] , [FALCOT]) of the series of studies launched at the HSP Lab to investigate the rate control mechanisms of XTP. The three previous studies explored the rate control mechanisms of XTP for the **unicast** mode of communication. The mandate of the present study is

[1] See Appendix **Keywords summary** - HSP Lab



(a) Stress-strain diagram for steel
(in Steel Buildings by Crawley & Dillon, Wiley 1970, p374)

(b) Law of variable proportions
(in Managerial economics, by Haynes & Henry, Irwin 1974, p220)

(c) Thrashing
(in Operating System Concepts, by Silberschatz & Galvin, AW 1994, p331)

Figure 1: Typical saturation Curves

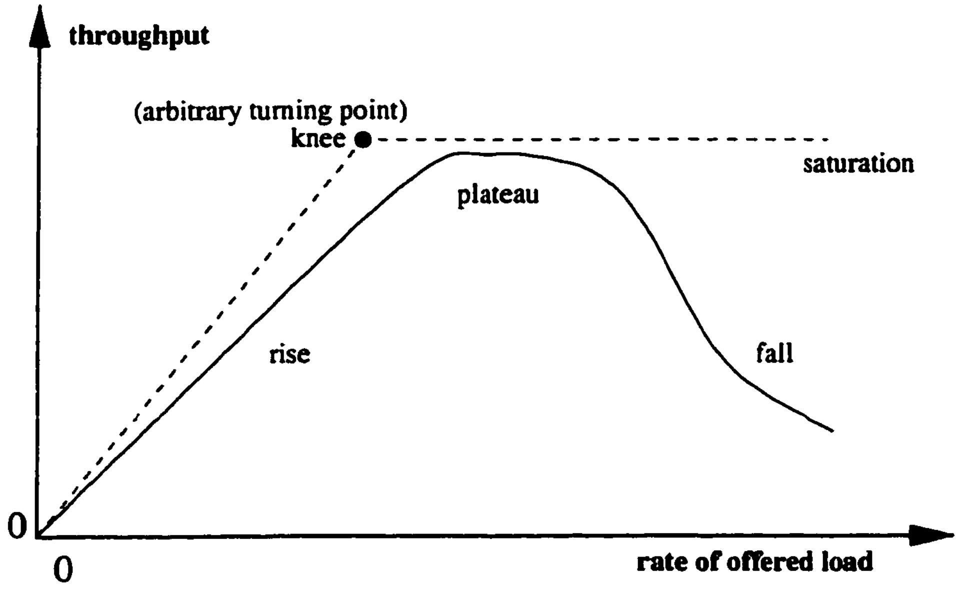


Figure 2: Hypothetical Internet Saturation Curve

to explore the **multicast** case through reliable file transfer in a WAN environment.

In July 1997, at the time of really preparing for the live multicast experiments, the expectations and assumption space of the study are as follows:

1. Saturation curve(s), relating the offered load (the sender side - in bytes/msec) with the throughput (the receiver(s) side - also in bytes/msec), could be found to model the reactive behavior of the Internet. Even if such a curve is valid only at a specific moment in time and for a very small subset of links, this is all that matters for an ongoing communication association. The expectation that the reactive behavior of the Internet could be modeled with the help of saturation curves is based on the intuition that the Internet behaves just as any other system when subjected to increasing load. In a sense, the important dimension would be the bare "existence" of the curve rather than its particulars. As will be explained later, knowing its existence would allow us to eventually devise a valid rate control policy for XTP. A very rough approximation of the type of saturation curve expected for the Internet is shown in Figure 2.

2. Given the existence of "Internet saturation curve(s)", this fact could be used to dynamically adjust the rate of data transfer when doing important communication "jobs" with XTP. With reference to Figure 2, the strategy would be

to gradually increment the offered load up to the point where the drop rate increases significantly. Even if not knowing all the particulars of the saturation curve at this moment in time, and for the links being used, it would mean that the **plateau** phase of the curve has been reached, and that the offered load has to decrease.

3. For the multicast mode of communication, the "presumed Internet saturation curve" will be a resultant of the many Internet links in use for a particular multicast association. For instance, if a multicast sender is located at Concordia University, and the task consists of reliably transfering a file to three receivers (one located in Karlsrhure, Germany; one in Evry, France; and one in North Carolina, USA), then the combined throughput plotted on the curve will be the lowest throughput of the three receivers.

To summarize expectations, this study is based on the hope that the concept of saturation curves can be used to model the reactive behavior of the Internet, more particularly as applied to the multicast mode of communication over the Internet, and that we will be able to produce a set of meaningful saturation curves. The provision of the data and the plotting of these curves mark the limit of our currently (summer 1997) defined mandate.

2.5 Future work

Conducting load experiments with steel and plotting its *stress-strain* curve is only one part of the whole investigatory endeavor. Policy is subsequently needed to manage these "facts of life" rationally. For steel, the policy part comes in the form of Government controlled **Codes of Practice** stipulating the permissible stresses. Given the loads that a structure is planned to withstand, and allowable stresses, one can deduce the amount of steel needed.

To make a "wise" use of the XTP rate control mechanisms, and supposing that the "curve searching phase" has been successfully discharged, analysis of all the studies made and synthesis work will be needed. This work will be part of a subsequent effort. Let us briefly outline the high level algorithm that could be used to exercise dynamic rate control with XTP (if, in fact, the curve has the shape that we expect):

```
at the sender:
  get receiver(s) status with SREQ
  plot point p0
  repeat forever:
    wait DELTA units of time
    get receiver(s) status with sreq
    plot point p1
    if slope of p1-p0 is negative then
      decrease rate of offered load by delta
    otherwise increase rate of offered load by delta
    p0 = p1
```

2.6 Update on expectations

In October 1998, the global direction of the report remains valid, but there were dramatic adjustments made to the modus operandi. One main reason for such an adjustment has been the gradual recognition, after many unsuccessful attempts, that conducting multicast rate control experiments using the Internet/Mbone environment would not be possible, at least from Concordia University. Even within Concordia itself, when conducting most of the rate control experiments in September 1998, multicast was only partially working at the IP/network layer. For instance, it was not possible to exchange multicast packets between machines that belong to the HSP Lab subnetwork (forest or pine) and other machines that belong to other concordia subnets. Another main reason for change has been the gradual discovery that SandiaXTP does not always exercise the quality of rate control that would be expected.

As the original mandate consisted of conducting multicast rate control experiments within the Internet/Mbone environment using SandiaXTP, and given that problems existed with both the Internet and the SandiaXTP premises, there was a gradual shift to: (1) more investigatory work into the rate control mechanisms used by SandiaXTP; (2) proposed changes to the SandiaXTP rate control algorithm; (3) and finally a decision made in May 1998 to conduct the rate control experiments within the Concordia University subnetworks only.

What started as an outward journey was eventually redirected into an inward one, which does not imply that no progress was made. Quite the contrary, the main components of the study are still present (rate control, multicast, saturation curves) and it is felt that the ground is well prepared for further rate control studies using the Internet/Mbone environment when that environment becomes more accessible from Concordia.

3 The Xpress Transport Protocol (XTP)

The investigations and experiments reported here being secondary to the existence of XTP, it is proper to first provide a cursory purview of the protocol itself. We begin with a review of the OSI 7-layer Basic Reference Model (OSI 7-layer BRM).

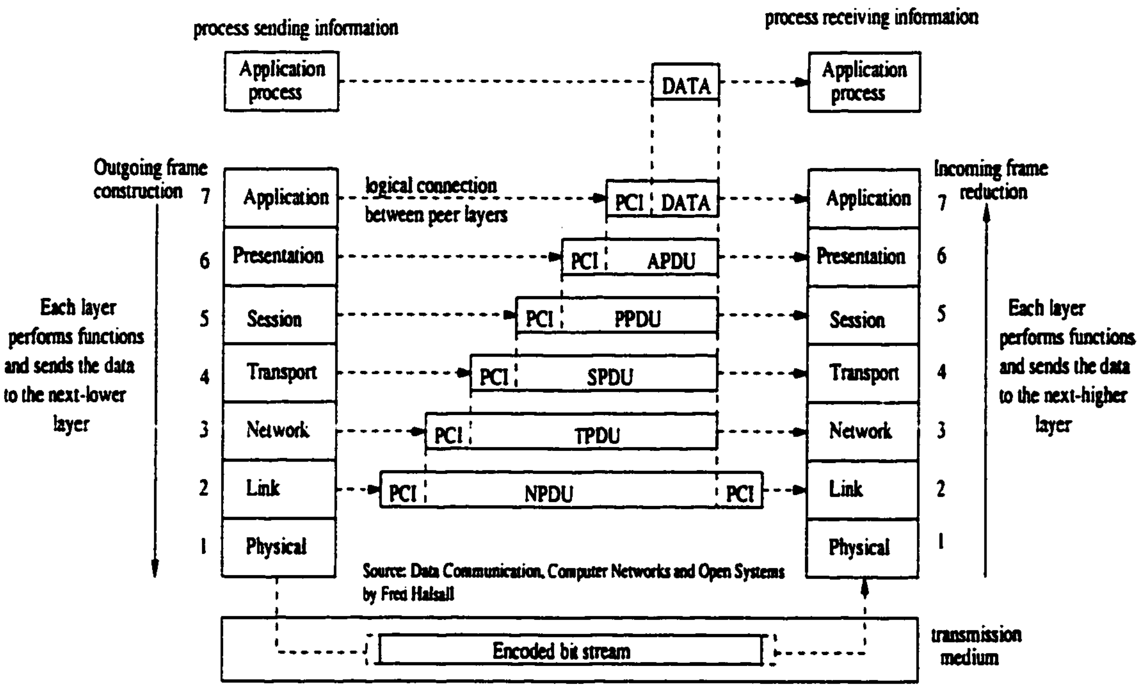
3.1 The OSI Basic Reference Model

One widely used approach to tame complexity is the so-called *divide-and-conquer* strategy. As applied to data communications, this strategy takes the shape of "layering". The technique consists of building a stack of layers, each one assigned responsibility to deal with one part of the total problem, offering services to the layer above it, and after some value-added functionalities of its own, uses the services of the level below it to fulfil its promises.

The OSI 7-layer BRM shown in Figure 3 is a widely known layering conceptual model applied to the field of data communications. The OSI 7-layer BRM provides not only a framework for the development of multi-vendors protocol and network software that are designed to work together, but it also introduces a notation and common terminology used and understood throughout the data communications field. We make extensive use of this terminology throughout the remainder of this report.

Figure 3 also illustrates the activity of sending some data from one application process to another application process. Most of the time, the two communicating processes would be running on different hosts/machines, but it need not necessarily be the case. The communication would still be possible between two processes running on the same machine, though at a higher overhead than using some form of interprocess communication, such as pipes, shared memory or message queues. As per the model, same level layers establish a *logical connection*; for instance, with respect to Figure 3, the left Transport layer is communicating with its peer right Transport layer.

Despite its wisdom, OSI 7-layer BRM offers only a framework, the layer being the locus of residence (so to speak) of some protocol that undertakes to deliver some or all of the responsibilities assigned to that layer. Essentially, a protocol defines the rules



Source: Data Communication, Computer Networks and Open Systems by Fred Halsall

Figure 3: OSI 7-layer BRM and Data encapsulation

of the communication and the data formats to be used in order to fulfil its promises. More than one protocol may belong to a specific layer. In order to establish a successful communication, not only are some same level protocols needed, but these must be identical protocols at both the sending and receiving ends.

With regards to Figure 3, let us briefly discuss the construction of outgoing packets from the left to the right stack. Each layer treats the data passed to it as an opaque object, not to be scrutinized. This is called *encapsulation*. For instance, to use an analogy, a handwritten message to be forwarded would correspond to the DATA part shown at the top of Figure 3. The first element of overhead would consist of inserting this message into an envelope, with some addressing on it and possibly handling instructions (shown as PCI - Protocol Control Information on Figure 3). Next, the letter (with the message encapsulated into it) could be handed to some mail carrier, who might decide to incorporate the letter into its own standard one, possibly fill up records for control, or make a copy for eventual retransmission in case of loss.

Here, we point out the fact that encapsulation does not prevent opening the envelope

Table 2: The functionality of the OSI Reference Model

OSI Layer	Layer Functions
7 - Application	Application-specific services
6 - Presentation	Data compatibility between heterogenous systems
5 - Session	Dialogue maintenance
4 – Transport	Reliable end-to-end data transfer
3 - Network	Routing between network segments
2 - Data Link LLC	Multiplexing users through access points
2 - Data Link MAC	Basic framing and delivery service
1 - Physical	Digital-to-signal transmission for transmission
Source - XTP: The Xpress Transfer Protocol, by Strayer, Dempsey and Weaver	

to make a copy, rather it means that the downstream layer does not depend on the data part passed to it by the upstream layer in order to do its value added work. In the analogy, the same carrier seems to ignore some layer (for instance, no cryptography is used for privacy of the message - which would correspond to the presentation layer responsibility) and to perform the work of more than one layer (for instance, the control information could belong to the Session layer and the copying to the Transport layer). Similarly, in practice, data communication software does not necessarily respect layer boundaries, as long as the work gets done. This process would continue for all layers until the message is actually "put on the wire". When the message reaches the other end, a reverse process occurs, with each peer layer stripping off its header to discharge its assigned duties. Finally, the net DATA is delivered at the top of the right stack to the receiving process.

A high level description of the responsibilities assigned to each layer within the OSI 7-layer BRM is shown in Table 2. Layer **4 Transport** is highlighted to signal the fact that we are mainly concerned with the transport level in this report. Table 2 assigns "Reliable end-to-end data transfer" functionality to the Transport layer. However, the reliability aspect is not assumed by all transport level protocols, as UDP is a transport level protocol, is end-to-end but does not support a reliable service. Other transport levels protocols could also provide the reliability aspect, such as TCP.

3.2 Historical perspective – The XTP Project

The purpose of Figure 4 is threefold: (1) first to trace the timewise evolution of XTP with regards to its life cycle since its inception in the late 1980's (the left part of the diagram, from top to bottom); (2) second to relate XTP functionalities to other pre-existing peer protocols such as TCP and UDP (the upper-right quarter of the diagram); (3) finally to relate the present study to one particular implementation of the XTP protocol, namely SandiaXTP (lower-right quarter of the diagram). We now comment in some detail on the left portion of the diagram, which is meant to summarize the main stages of the evolution of the XTP project.

The box labeled "**perception of the needs for XTP**" marks the origins of the protocol. Since the design of the TCP/IP protocol family in the late 1970's (about 10 years earlier than XTP), there had occured sufficient changes in terms of needs, improved quality of hardware and evolving new technologies (such as ATM & FDDI) to justify the development of an improved transport level data communications protocol. Such a protocol would subsume the functionalities provided by existing ones, such as TCP and UDP, but also provide sufficient convincing new features (such as multicast and quality of service) that would warrant its market acceptability. Rather than linger here on the motives that led to the development of XTP, we refer the reader to a text written by some of the designers of XTP [SDW] that covers this aspect in much more detail.

The box labeled "**organization & management**" is meant to characterize the methods of development and the social structure used by the developers of XTP. The production of XTP is the result of the efforts of a group of international researchers, in an era when the Internet was fully functional, with the support of electronic mail to facilitate exchanges, and also with other protocols (such as TCP) opened to scrutiny. To channel this group effort, the *XTP Forum* was eventually created late 1992 (or early 1993) out of Protocol Engines Inc. The mandate of the XTP Forum was to meet at regular intervals of time to discuss design issues, to agree on further work to be done and related deadlines, and also to publish interim versions of the XTP specification in

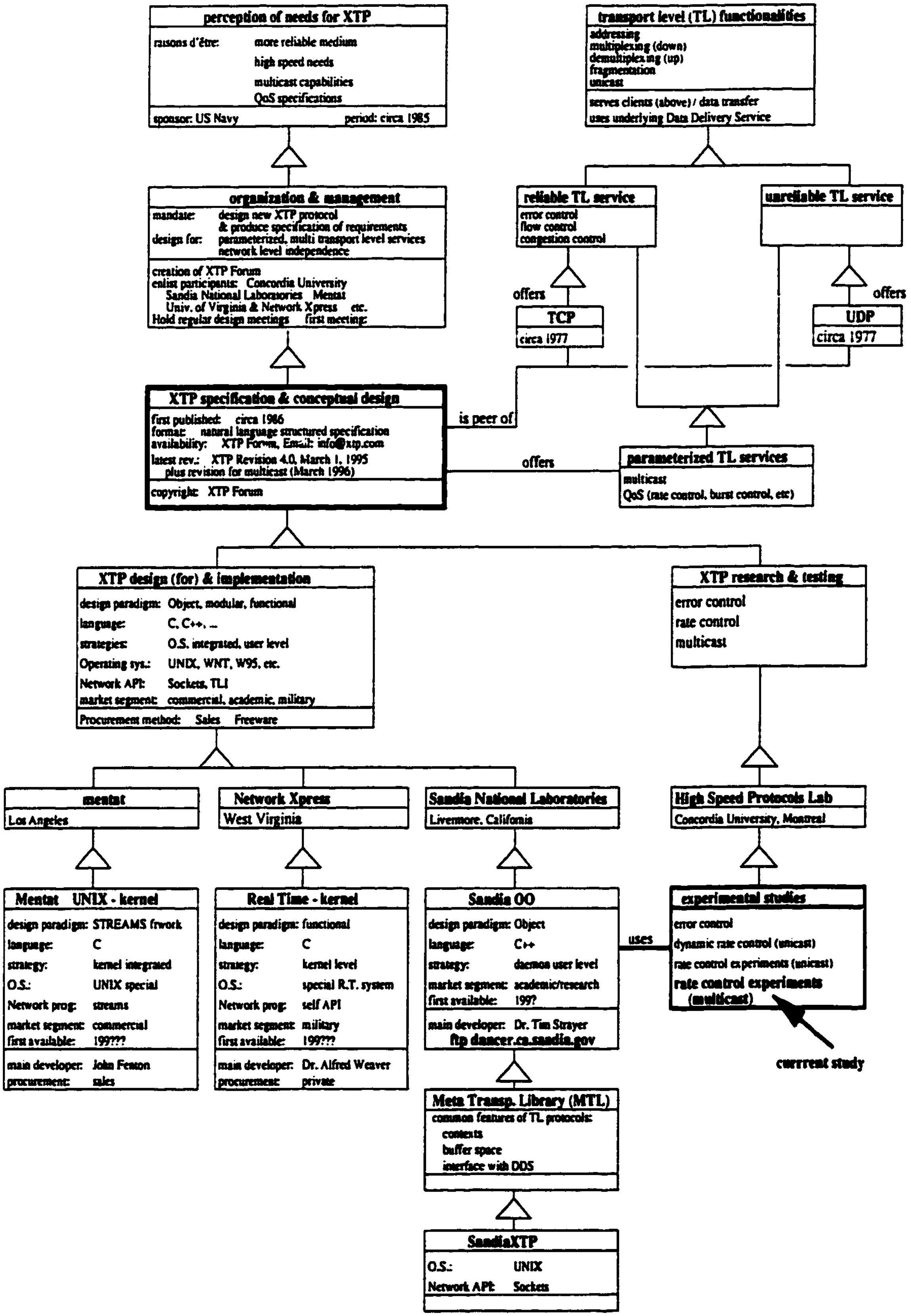

perception of needs for XTP
raisons d'être: more reliable medium
high speed needs
multicast capabilities
QoS specifications
sponsor: US Navy
period: circa 1985
transport level (TL) functionalities
addressing
multiplexing (down)
demultiplexing (up)
fragmentation
unicast
serves clients (above) / data transfer
uses underlying Data Delivery Service
organization & management
mandate: design new XTP protocol
& produce specification of requirements
design for: parameterized, multi transport level services
network level independence
creation of XTP Forum
enlist participants: Concordia University
Sandia National Laboratories Mentat
Univ. of Virginia & Network Xpress etc.
Hold regular design meetings first meeting:
reliable TL service
error control
flow control
congestion control
unreliable TL service
offers
TCP
circa 1977
offers
UDP
circa 1977
XTP specification & conceptual design
first published: circa 1986
format: natural language structured specification
availability: XTP Forum, Email: info@xtp.com
latest rev.: XTP Revision 4.0, March 1, 1995
plus revision for multicast (March 1996)
copyright: XTP Forum
is peer of
offers
parameterized TL services
multicast
QoS (rate control, burst control, etc)
XTP design (for) & implementation
design paradigm: Object, modular, functional
language: C, C++, ...
strategies: O.S. integrated, user level
Operating sys.: UNIX, WNT, W95, etc.
Network API: Sockets, TLI
market segment: commercial, academic, military
Procurement method: Sales Freeware
XTP research & testing
error control
rate control
multicast
mentat
Los Angeles
Network Xpress
West Virginia
Sandia National Laboratories
Livermore, California
High Speed Protocols Lab
Concordia University, Montreal
Mentat UNIX - kernel
design paradigm: STREAMS frwork
language: C
strategy: kernel integrated
O.S.: UNIX special
Network prog: streams
market segment: commercial
first available: 199???
main developer: John Fenton
procurement: sales
Real Time - kernel
design paradigm: functional
language: C
strategy: kernel level
O.S.: special R.T. system
Network prog: self API
market segment: military
first available: 199???
main developer: Dr. Alfred Weaver
procurement: private
Sandia OO
design paradigm: Object
language: C++
strategy: daemon user level
market segment: academic/research
first available: 199?
main developer: Dr. Tim Strayer
ftp dancer.ca.sandia.gov
uses
experimental studies
error control
dynamic rate control (unicast)
rate control experiments (unicast)
rate control experiments
(multicast)
currrent study
Meta Transp. Library (MTL)
common features of TL protocols:
contexts
buffer space
interface with DDS
SandiaXTP
O.S.: UNIX
Network API: Sockets

Figure 4: XTP Project in brief

natural language (English). As advertised at the beginning of the XTP specification, the XTP Forum included a core of about 15 international researchers, with Dr. Tim Strayer of Sandia National Laboratories acting as Editor-in-chief for the specification.

The box labeled "**XTP specification & conceptual design**" stands for the key item of the whole XTP Project. Seen from above, it is the output of the work of the XTP designers and the official document published by the XTP Forum (©by XTP Forum). Seen from below, it is the document of reference for all legitimate implementations that can be done of XTP. The revision that we refer to in this report is the **XTP Revision 4.0 - March 1, 1995**, plus an addendum written in March 1996 after the June XTP Forum meeting in Dallas. This addendum contains mainly additional features for the management of multicast groups.

The box labeled "**XTP design (for) & implementation**" and its derivatives labeled "**Mentat**", "**Network Xpress**", and "**Sandia National Laboratories**" (these being names of US organizations) cover the implementation aspects of XTP. Each one of these three widely known implementations of XTP is targeted at a specific market segment and also entails some different technical decisions. For such a general protocol as XTP, it is not incongruous to use the label *implementation design*, as there are numerous implementation strategies that can be used, depending on the targeted market segment.

The reference implementation used for this report is the one done by Dr. Tim Strayer at Sandia National Laboratories (which we shall thereafter refer to as *SandiaXTP*). The market segment targeted by SandiaXTP is the academic/research sector. The source code is freely available and experimenting with the protocol is encouraged. SandiaXTP is characterized as being a user level implementation of the protocol, meant for a UNIX environment, with a daemon (server) process running as a user level process. The object paradigm has been used with the C++ language. The network programming is done with BSD sockets. Many more details about the SandiaXTP implementation are presented later in Chapter 4.

The **Mentat** implementation, developed at a Los Angeles based company of the same

Table 3: XTP parameterized features - summary

features	defines	sub-choices	XTP options	remarks
multicast/ unicast	relationship	Xor	MULTI=0 | 1	default MULTI=0 (unicast)
flow control	reliability	off	NOFLOW=1	
		reservation mode	RES=1	default RES=0
		regular	default	alloc=some value
error control	reliability	off	NOERR=1	
		no check mode	NOCHECK=1	
		go-back-N		implementation issue
		(selective reXmit)		spans(a_i, b_i),nspans...
		agressive	FASTNAK=1	
rate control	traffic spec.			rate=some value
				burst=some value
prioritization	priorities (R.T.)	0..65535	SORT=1	sort=some value
addressing	protocol stack	Internet domain		Ex:
		...		aformat=0x01
		...		adomain=17(UDP)
		Xerox (XNS)		
		IP v6		

name by John Fentat and others, is targeted at the commercial market. It is a kernel level implementation, meant for a UNIX environment, and written in C (for compatibility with the remaining of the kernel source code, and also for speed). The network programming is done with System V-TLI (Transport Layer Interface).

The *Network Xpress* implementation is a special/custom one developed by Dr. Alfred Weaver for one of the sponsors of the XTP Project, namely the US Navy. Being a private implementation, little is known (to the author, at least) about its details.

Globally, we observe that a good twelve years have elapsed since XTP was nurtured in the mind of its sponsors and designers; today, it is still being tested and only at the beginning of its life-span on the protocol market. Such a long time span clearly reveals that the design, implementation, testing, and market penetration of real life data communication protocols is a complex and long term endeavor; one that sharply contrasts with the short time span of the hardware.

3.3 XTP - A bird's eye view of a parameterized protocol

XTP is a parameterized transport level protocol. The design of XTP was guided by two main objectives: (1) to provide to the user (here, the term *user* really means the *network application programmer* - acting on behalf of the user) a large palette of mechanisms to choose from, and let the needs of the applications dictate the policies; (2) through careful observance of the *principle of orthogonality*, make those mechanisms as independent and separately selectable as possible.

As an application example of the principle of orthogonality, consider the problem of packet drop, which could occur mainly at intermediary routers along the path, or at a slow receiving end. One approach, ignoring the principle of orthogonality, would be to presume a unique problem and use the flow control mechanism to resolve it. True enough, if one reduces the window size, this will have the effect of reducing the data flow, as the sender is stalled more often awaiting credits from the receiver. The drawback of this indiscriminating approach is that it may entail an underuse of the resources. Suppose for instance that the bottleneck is not the receiver, but some intermediary router. Suppose also that this router would be quite capable of handling the same number of packets, but provided that their rate of arrival is slightly reduced. If one would split the problem in two sub-problems with focused solutions, the net result might be much lower total duration for the data transfer, as the waiting time for receiver acknowledgements is reduced. Consequently, an alternate approach based on the principle of orthogonality would be to postulate two problems: (1) consider the possibility of swamping a slow receiver with too much data too quickly and use the flow control mechanism to resolve it; (2) consider also a path related congestion problem and use the rate control strategy to resolve it. With the orthogonal approach, not only do we stand better chances of resolving the congestion problem more efficiently, but we also give more freedom of choice, as one mechanism can be selected independently of another one, which means that only one mechanism could be used, or the two together, or even none of them.

The purpose Table 3 is to present a global summary highlighting the parametric nature of the design of XTP. The presentation and the critical comments that follow are based on the author's (limited) knowledge of XTP, plus: (1) an article by Atwood & al [AMZ]; (2) the book by Strayer, Dempsey & Weaver [SDW]; (3) the XTP specification itself [XTP40].

3.3.1 Multicast/unicast

Within the traditional unicast model, the term *connection* implies the *connected* state of two communicating endpoints. To also accomodate multicast, XTP introduces the more general term *association*, which could refer to either unicast or multicast.

Multicast is a major distinctive feature of XTP meant to save network bandwidth. For instance, suppose that a message has to be sent to one thousand receivers from Montreal to various hosts/sites spread out on the territory of France. Furthermore, suppose that the network topology is such that the messages travel though the same sequence of routers whose endpoint routers are called *Montréal/Monty* and *Paris/Étoile*. With unicast, the same message would be sent 1000 times from Montréal, and hence would travel 1000 times the path between routers *Montréal/Monty* and *Paris/Étoile*. With multicast, the message would be sent only once from *Montréal/Monty*, and thus would travel the path *Montréal/Monty* to *Paris/Étoile* only once; the latter router being responsible to disseminate the message to the 1000 sites within France, possibly with further bandwidth savings due to multicast and depending on the details of the local topology.

For the user, the choice between multicast or unicast is mutually exclusive (shown as Xor on Table 3), and done once per association. The MULTI bit in the header options field defines which of multicast or unicast is used. If not set (MULTI=0 - the default), the mode is unicast; otherwise it is multicast.

3.3.2 Flow control

Flow control is a typical end-to-end transport level service. The aim of flow control is 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 offers a threefold range of choice for flow control, whereby services equivalent to well known protocols, such as UDP (no flow control exercised; NOFLOW=1) or TCP (flow control exercised; the default for XTP), could be provided.

XTP also offers an interesting intermediary choice called *reservation mode.* By setting the RES bit in a packet header, the sender instructs the receiver to advertise only the actual buffer space allocated by the user for the context. This forces the receiver to adopt a conservative policy making sure that no packets can be lost due to lack of buffer space at the receiving end. When the RES bit is set, the user is in control. When the RES bit is off (the default for XTP), the implementation is in control.

3.3.3 Error control

Error control is essential for reliability, and cares for problems such as corrupted data, lost data, reordering at reception points and duplicates. As shown on Table 3, XTP offers many possibilities of choice, ranging from "none at all" to an "aggressive" approach (FASTNAK set to 1 by sender) whereby the receiver is requested to signal missing packets as soon as they are detected.

One distinctive feature of XTP for error control is the *nocheck mode* (NOCHECK set to 1 by the sender), whereby the checksum is being computed for the header of the packet only, and not for the data part (payload). Handling the checksum on the header is needed at all times in order to be able to steer incoming packets to proper receiving contexts. As per Atwood & al [AMZ], for some delivery audio, "It is probably not worthwhile to bother to checksum the data, as delivering an invalid set of samples will probably not be any worse than filling the time period with silence".

With regards to retransmission, XTP has the data structures and the mechanisms for both go-back-N and selective retransmission. The feedback from the receiver is triggered by the sender when a packet is sent with the SREQ bit set. The receiver then uses a CNTL packet to forward *alloc* (the limit up to which the sender can send), *rseq* (the sequence number of the highest contiguous packet received so far), *nspan* (number of discontinuous received segments) and *spans*(a_i, b_i) (the limits of these segments) whereby the sender can deduct the missing gaps at the receiver and retransmit accordingly. If go-back-N is used, the sender can then retransmit packets starting with sequence number *rseq* and up to *eseq*-1 (eseq is one greater than the ending sequence number for sent but unacknowledged data). The receiver can always force go-back-N explicitly by issuing a CNTL packet with one span: *spans1=(hseq,hseq)* (hseq being the receiver "high water mark").

Though XTP provides the mechanisms for either go-back-N or selective retransmission, a specific implementation is not forced to support both algorithms. The go-back-N algorithm is appropriate when both the storage capacities of the circuits and the round-trip-time are low. Such would be the case for an Ethernet local area network with a 500 m cable.

At the other end of the spectrum, the selective repeat algorithm is appropriate when both the storage capacities of the circuits and the round trip time are high. Such would be the case for a 7000 Km long data path with a data rate of 10 Mbps. A lost packet could be followed by many other packets "still on the wire" that will be received correctly.

One design criterion for XTP is *interoperability* between implementations that would support only one of the two algorithms, as the ECNTL packet format has fields for both. A go-back-N sender would ignore some information provided by a selective retransmission receiver; a selective retransmission sender being automatically also capable of go-back-N would cooperate seamlessly with a go-back-N receiver. The choice is the responsibility of the implementer; what has been done at the level of the specification is to ensure that whichever choice is made, the implementation will interoperate with an implementation that made the opposite choice.

3.3.4 Rate control

Rate control aims at resolving the problem of congestion along the path, such as at routers, where the data of numerous senders (possibly other routers) combine to overun a receiver-router.

Rate in XTP is specified in **bytes per second**, which differs from the units generally used by the theoretical data communications field where the data rate is expressed in bits per second, such as 56.6 Kbps for modem or 10Mbps for 10 BASE-T Ethernet. In practice, bytes are sent one at a time, but in a sequence up to a certain number of bytes called a *packet*. The sending/receiving granularity is therefore the packet. Some time elapses between the start and the end when sending a packet. Whilst a packet is being sent "rate control" does not govern, as the job is done at hardware speed (for typical software/hardware configurations at least, such as through an operating system and Ethernet network technology); the limiting factor being host processing time or network technology (such as 10 Mbps for 10 BASE-T Ethernet). From this perspective, rate control implies that there could be some **idle time** during which no packet is being sent and yet the job is not over; otherwise, back-to-back packets are being sent at hardware rate and no rate control is exercised.

The quality of the rate control that can be exercised is also subject to the limitations of the underlying software (mainly operating system) and the hardware. For instance, one can send no faster than either host or network technology will allow.

To gain better insights into the workings of XTP rate control mechanisms, let us use speculative reasoning and work out some typical scenarios. Suppose we reason for a period of time of one second. The idle time inherent to rate control could have various distribution patterns, depending on the size of the set-of-packets being sent. For XTP, *burst* defines the size the these sets-of-packets; although burst is expressed in bytes, not in packets.

For instance, an extreme scenario (call it scenario 1) would be for size(set-of-packets) = 1. Only one packet is sent at a time, and total idle time within a second is the sum of equal duration partial idle times. This would yield an *even* distribution of idle times.

Another scenario (call it scenario 2) would be for size(set-of-packets)=n (where n equals all packets to be sent within the second, given the specified rate). For this case, all the packets are sent back-to-back at hardware rate with all idle time concentrated from the moment the last packet is gone until the beginning of the next second. This would yield a *concentrated* distribution of idle time.

Other intermediary scenarios are possible varying the size(set-of-packets) from 1 to n. Let us now further illustrate the scenarios with made up data and using XTP variables names:

made up data:

- rate=25000 bytes/s
- packet size = 1400 bytes
- can send at half Ethernet 10 BASE-T, i.e., 10Mbps/2 or 1250000/2 bytes/s, or 625000 bytes/s, or 445 packets/s
- time to send one byte: 0.0000016 s
- time to send one packet: 0.00224 s
- time to send 25000 bytes: 0.04 s
- presuming plenty of data to send.

Scenario 1 (idle time evenly distributed)

- burst = 1400 bytes
- $RTIMER = \frac{burst}{rate} = \frac{1400}{25000} = 0.056s$, which is much larger than the time required to send one packet (i.e., 0.00224s).

events	**credit**	**time**
arm RTIMER (0.056s)	1400	0
start sending packet		0
end sending packet	0	0.00224s
...		
wait 0.05376s until RTIMER times up...		0.056s
arm RTIMER (0.056s)	1400	0.056s
start sending packet		0.056s
end sending packet	0	0.05824s
...		
wait 0.05376s until RTIMER times up...		0.112s
arm RTIMER (0.056s)	1400	0.112s
etc.		

Scenario 2 (idle time concentrated at the end)

- burst = 25000 bytes
- $RTIMER = \frac{burst}{rate} = \frac{25000}{25000} = 1.0s$, which is still much larger than the time required to send all the bytes/packets (i.e., 0.04s)

events	**credit**	**time**
arm RTIMER (1.0s)	25000	0s
start sending packet 1		0s
end sending packet 1	23600	0.00224s
start sending packet 2		0.00224s
end sending packet 2	22200	0.00448s
...		
start sending packet 18		0.03776s
end sending packet 18	0	0.04s
...		
wait 0.06s until RTIMER times up...		1.00
arm RTIMER (1s)	25000	1.00
...		
etc.		

Scenario 3 (idle time with size(set-of-packets)=3, burst=4200 bytes))

- burst = 4200 bytes
- $RTIMER = \frac{burst}{rate} = \frac{4200}{25000} = 0.168s.$

events	credit	time
arm RTIMER (0.168s)	4200	0s
start sending packet 1		0s
end sending packet 1	2800	0.00224s
start sending packet 2		0.00224s
end sending packet 2	1400	0.00448s
start sending packet 3		0.00448s
end sending packet 3	0	0.00672s
...		
wait 0.16128s until RTIMER times up...		0.168
arm RTIMER (0.168s)	4200	0.168
start sending packet 4		0.168
...		
etc.		

If burst equals zero, then RTIMER equals zero, with the result that the context can send without constrainst (RTIMER forces idle time between bursts; if RTIMER equals zero, then waiting time is null). On the other hand, if rate equals 0, implying a division by zero, a check for this condition is made and transmission is halted immediately until rate has a nonzero value, or burst has a zero value.

The lesson to be learnt from these scenarios is the fragility of the interrelationships between the trio burst, rate and RTIMER with regards to practicability on real computer systems. As shown on Figure 5, if we increase the rate, keeping the burst value constant (line 2), then RTIMER decreases linearly. A similar phenomenon occurs if we decrease the burst value, keeping the rate value constant (line 1). Surely, there must be a lower limit where RTIMER values become meaningless, depending on the precision of the clock and the details of the implementation. Suffice to mention that RTIMER values are generally small fractions of a second, otherwise XTP would not stand to its general promise of being a "fast" protocol.

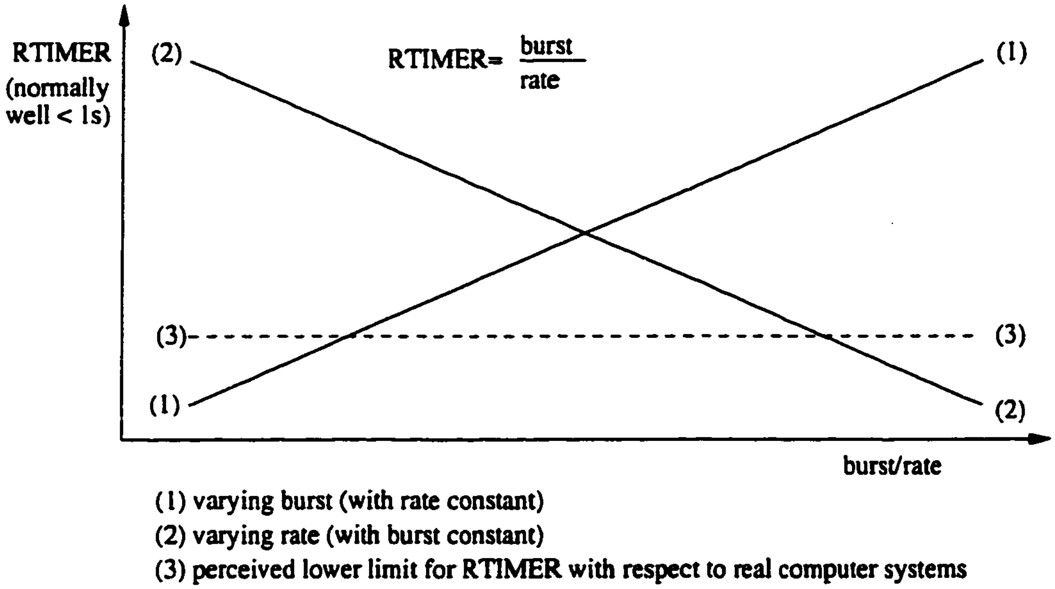


Figure 5: Relationship RTIMER, burst & rate

3.3.5 Prioritization

The SORT option bit, and the corresponding *sort* field, is XTP's response to hierarchy in application needs. For instance, the packets of real time (R.T.) clients can be processed before other competing XTP clients; out-of-band data could also be served at a lower priority level than real time but before regular data. In fact, the *sort* field being a 16-bit number, it is theoretically possible to discriminate amongst 65536 levels of priority. If the sort mechanism is armed, incoming packets are processed in priority order; otherwise, they are handled on a FIFO basis.

Though the number of priority levels appear very abundant (at 65536 different values), their wise management on an open system (i.e., a non dedicated system) does not appear evident from a practical side. Without a coordinating instance, pretentious applications could always claim highest priority, thereby defeating the purpose of the scheme. On the other hand, the mechanisms are available for use on a dedicated system (such as the command and control system on a ship).

3.3.6 Addressing

XTP's approach to addressing is based on the recognition that: (1) there are already many protocol stacks in use, each with their own addressing scheme; (2) many of those schemes are likely to survive and thrive on the market for the forseeable future. XTP's approach to addressing is therefore a pragmatic one that can be summarized with the popular saying "If you can't fight them, join them". Consequently, XTP supports many alternatives, such as Internet Protocol Address (IP), ISO Connectionless Network Layer Protocol Address, Xerox Network System Address, etc.; depending on the particulars of the protocol stack that XTP operates in (essentially defined by the Network Level protocol underneath XTP - more about protocol stacks, layering of protocols in the next subsection).

The address format field (*aformat*) is used to specify the address format, such as Internet Protocol Address. Within an address format, the address domain field (*adomain*) is also needed to demultiplex (to disambiguate) packets destined to different protocols. For instance, within the Internet Protocol Addressing scheme, the *port* number is used as a demultiplexer to target a unique endpoint. However, port number 155 is not enough, as this very same port number could also be used by an application making use of the UDP protocol, or even another application making use of the TCP protocol. Hence the need of the *adomain* field, whose value could be 17 for UDP, 6 for TCP or even 36 for XTP (yes, XTP could be part of a protocol stack with IP underneath along with TCP and UDP as peers).

For the user, XTP does not have any option bit to specify the addressing scheme. Such silence implies that that the provision of the necessary information needed for the values of the *aformat* and *adomain* fields is an implementation issue to be included in the design of that implementation's Application Programming Interface. The necessary information could be supplied implicitly, i.e., derived from the actual format of the address given by the user, or a tagging system similar to the Address Format field could be used.

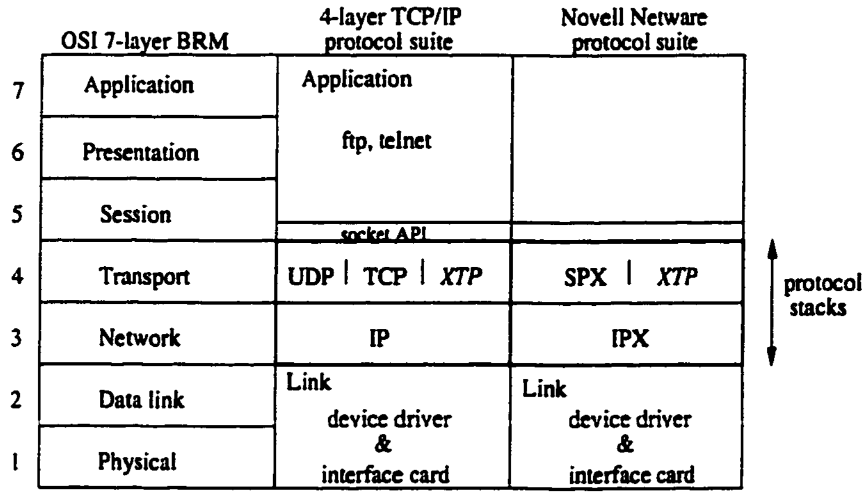


Figure 6: Protocol stacks

3.4 XTP - The working environment

So far, we have only hinted at the working environments of XTP. In this section, the standpoint is to look at XTP from the outside; to expose its intended role and position with regards to other protocols, as if XTP had to find its position within the *society* of data communications protocols.

3.4.1 Protocol stacks

The OSI 7-layer BRM is only a conceptual framework. In practice, either some well rooted protocols were developed before the introduction of the OSI model, or were developed after the model, but not necessarily respecting a one-to-one mapping between the 7 identified layers and the implementation modules. Protocols were often developed as a group, for a particular type of hardware and networking technology, and meant to interconnect together. Examples of such families of protocols are the TCP/IP, DECnet, Appletalk and Novell's SPX/IPX families of protocols. In this context, our use of the term **protocol stacks** refers more specifically at the duo Transport/Network layers, which are accessible to the Network Application Programmer through an Application Programming Interface (API), such as the BSD socket interface.

Figure 6 is meant to illustrate the difference between the OSI 7-layer BRM and actual families of protocols, such as the TCP/IP protocol suite. The two models somewhat differ for the low and high layers: where the OSI model identifies the Data Link and Physical layers, the TCP/IP model identifies only the Link layer; where the OSI model identifies the Session, Presentation and Application layers, the TCP/IP model identifies only an Application layer. Accounting for the differences between the two models not being the purpose of this report, we simply observe that the two models are quite in harmony for the Transport and Network layers, which are the important ones for the present report.

The main protocols that belong to the TCP/IP protocol suite are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) at the transport level. TCP supports reliability whereas UDP does not. IP (Internet Protocol), at the Network level, is mainly responsible for routing.

With reference to Figure 6, XTP is shown at the Transport layer, as peer of TCP and UDP, to highlight the fact that XTP could also be used within the TCP/IP protocol suite, whether as a replacement for either TCP or UDP, or for additional functionalities provided by XTP such as reliable multicast. In this case, XTP would use IP as its data delivery service (DDS). We have seen before that XTP supports the Internet Address format. In fact, for the remaining of this report, the TCP/IP 4-layer protocol stack is the model being used, with XTP at the Transport level evidently. In Figure 6, we have also shown that XTP is meant to work with other protocol stacks such as with Novell Netware, as an alternative to SPX, or for functionalities not provided by SPX.

3.4.2 Communication reachability - unicast

Although XTP was designed, inter-alia, to take advantage of emerging fast technologies, such as Fiber Optics and ATM, it does not necessarily interface directly with them. These technologies are under the control of their own native device drivers, layers below the transport layer at which XTP operates. The purpose of Figure 7 is to illustrate the possibilities of exchange of packets given a hypothetical network

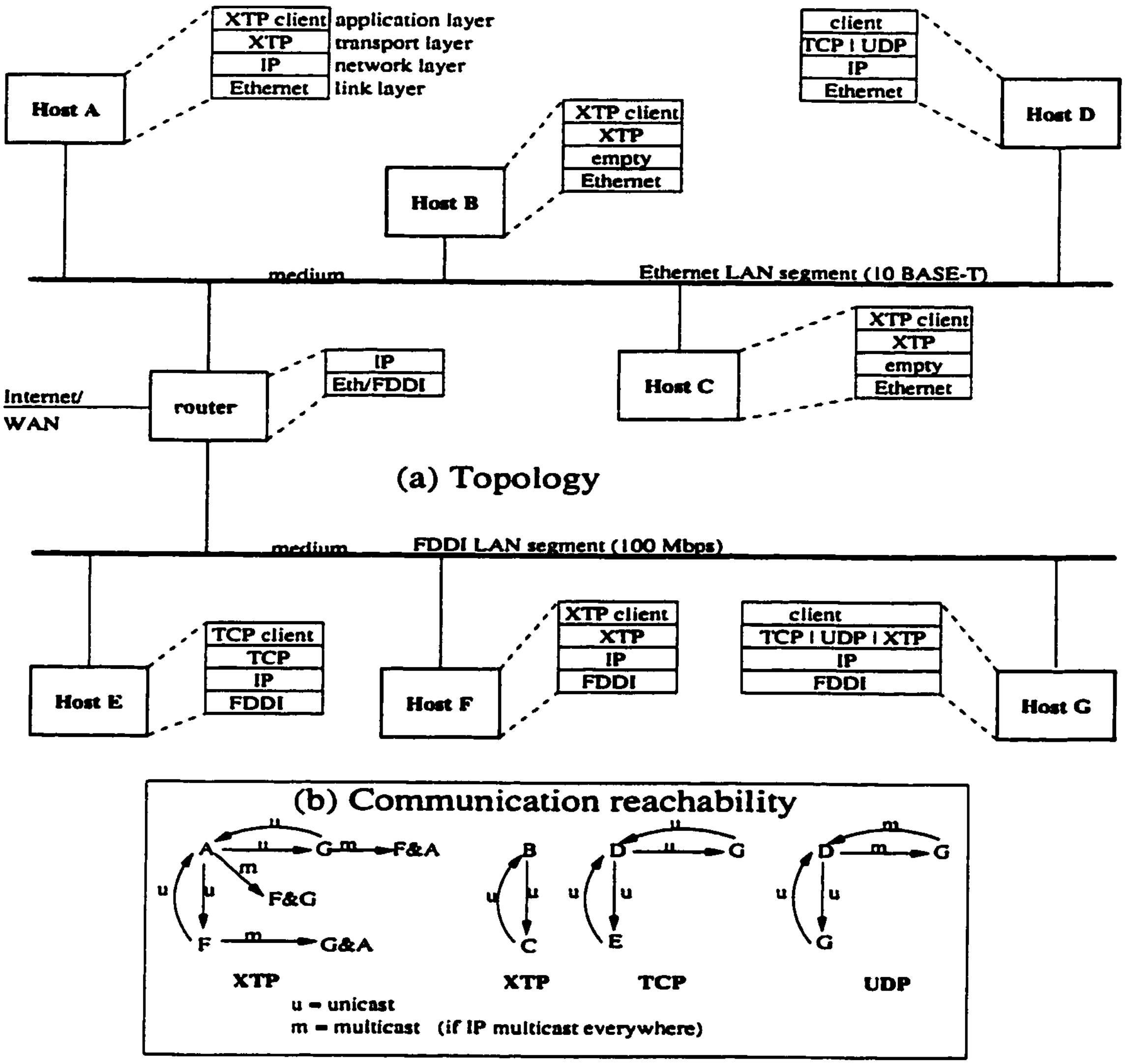


Figure 7: Communication reachability

topology built of two LAN segments interconnected through a router, which also acts as a gateway to Internet, a Wide Area Network.

The technology of one LAN segment is a typical coaxial 10 BASE-T (10 Mbps) Ethernet bus cable. The technology of the other LAN segment consists of an Optical FDDI (100 Mbps) bus cable. Both being bus technologies, all data is broadcast and every host can listen to circulated packets. For the remaining of this subsection, we consider only the exchange of unicast packets, i.e., packets sent from one source host to exactly one targeted destination host. In the next subsection, we consider the multicast case.

Ethernet LAN: Within the Ethernet segment, XTP clients running at Host-B and Host-C can exchange XTP packets because they both use the XTP protocol. The absence of a protocol at the Network level is not a deterrent to their connectivity. Indeed, no routing is needed on this bus broadcast medium. Host-A cannot exchange transport level packets with either Host-B or Host-C. The reason being that XTP at Host-B would receive an IP packet (IP packet format is encapsulated within an Ethernet packet at Host-A, with the Ethernet header stripped off at Host-B, but still delivering an IP packet to XTP), which would be dropped by XTP because of incompatible packet format. In the other direction, IP at Host-A would receive an XTP packet, which would also be dropped by IP because of incompatible packet format; XTP would never be passed the packet. Host-D cannot exchange transport level packets with any host on the Ethernet segment, as no other host has the required TCP or UDP protocols available.

Router: The task of the router is to allow the exchange of packets between the two LANs, even if not of the same network technology. This router would have the hardware interface cards (NIC) of both technologies (Ethernet/FDDI), and would translate link level packets from the Ethernet format to the FDDI format, or vice versa. There is no need for the router to run XTP, as XTP packets are encapsulated within IP packets, which themselves are encapsulated within link level packets. This router also has the role of acting as a gateway to the Internet, a wide area network. The router has the required IP protocol to do its routing work.

FDDI LAN: Within the FDDI segment, Host-E and Host-G can exchange transport level packets, but only through the TCP protocol, as TCP is available at both hosts. Host-F and Host-G can exchange XTP packets, as XTP is also available (as peer of UDP and TCP) at Host-G.

Ethernet-to-FDDI: Now we consider exchanging unicast packets between the two LAN segments. Host-A and Host-F, or Host-A and Host-G, can exchange XTP packets. Host-B and Host-C cannot exchange XTP packets with any host on the FDDI segment, even if XTP is available at Host-F and Host-G. The absence of IP at Host-B and Host-C has the consequence that they are unknown to the router who cannot receive packets from them or pass packets to them. Host-D and Host-E can exchange transport level packets using TCP. Host-D and Host-G can exchange unicast transport level packets using either TCP or UDP. Figure 7 (a) **Communication reachability** summarizes the possibility of packet exchange between the hosts on the two LANs.

3.4.3 Communication reachability - multicast

First of all, let us consider the TCP/IP protocol family with regards to multicast. The Network layer with its routing functionalities also has a very important role to play in order to deliver a multicast service to the client A unicast packet can be sent to a receiver even if this receiver did not ask for it. In contrast, an over-LAN-segment multicast packet is forwarded to a receiver by a router only if the receiver had previously advertised its interests in receiving to the routers, who can then update their routing tables accordingly. This initial receive-request is broadcast to all routers supporting multicast and takes some time to become into effect. If any multicasting is to be done with the Internet stack, a recent revision of IP (such as the one included with Solaris2.5) that supports multicast must be available at the Network level.

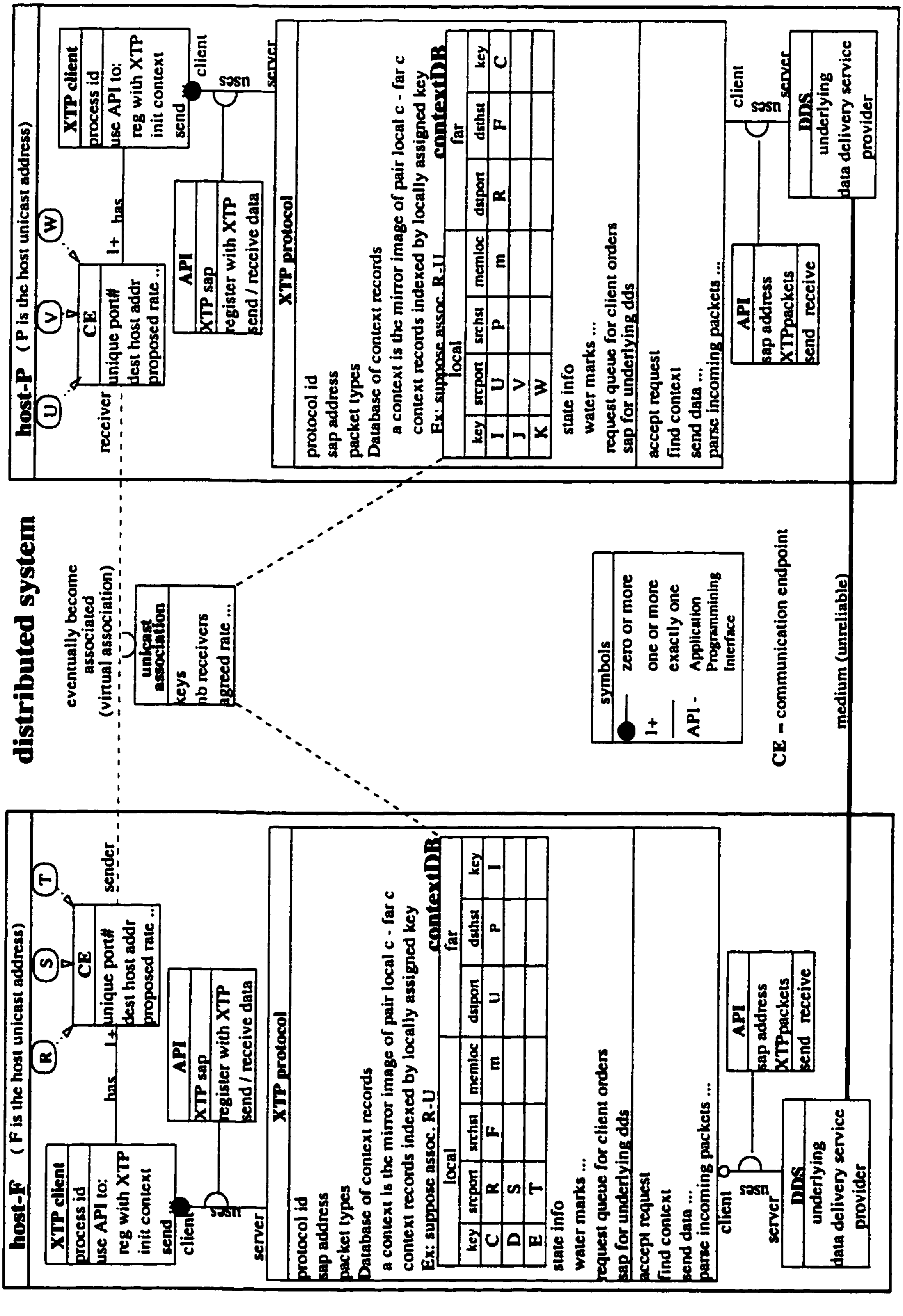
Even if IP multicast is available, any host where only TCP (an unlikely case in practice because UDP is always available with TCP) is available can't send or receive multicast packets because TCP is strictly a unicast protocol. Therefore, Host-E on the FDDI segment cannot exchange muticast packets at the transport level because only TCP is available. On the other hand, UDP is a minimal transport level protocol that offers a datagram service. Through some (socket) options, it offers an interface to IP whereby the services of IP multicast can be marshalled. As UDP is available at Host-D and Host-G, clients running at those two host can use the UDP protocol to exchange transport level multicast packets, provided of course that the receiver (only one possible in this case) had advertised its interest for receiving to the router before the sender starts sending. A scenario not shown on our topology, i.e., bypassing all transport level protocols and opening a direct connection with IP from the application level (IP raw) would also allow clients to exchange multicast packets.

Multicasting with XTP: To illustrate multicasting with XTP, let us suppose that Host-A sends XTP multicast packets on a multicast address such that all other hosts shown in Figure 7 that can receive it will indeed receive it. We presume that potential receivers of XTP packets have previously advertised their interest in receiving to the router, from the Internet/WAN as well as from the FDDI segment, and that a recent version of IP supporting multicast is available at the router. First of all, we note that even if Host-B and Host-C are on the same broadcast LAN segment and XTP is available, they are not able to receive any multicast packet from Host-A. The reason preventing them to receive succcesfully XTP multicast packets is the same as the one given for unicast; namely packets dropped by XTP because of their wrong format (IP format). If IP was available though, they both could receive XTP multicast packets.

On the FDDI segment, Host-F and Host-G are legal receivers of the XTP multicast packets sent from Host-A. In contrast to unicast though, there could be many receiving clients running at Host-F or Host-G. Other multicast receivers situated on the Internet could also receive the XTP multicast packets sent from Host-A, provided that they also have IP and XTP available in their protocol stack. Figure 7 (b) **Communication reachability** also summarizes the situation for the exchange of multicast packets at the transport level with XTP.

3.5 XTP - A system architecture view

The goal of this section is to articulate a system architecture view of XTP based on the interaction of three main components: (1) clients of XTP (Application layer); (2) the XTP specification acting as a server to XTP clients (Transport layer); (3) a data delivery service (DDS) that supports XTP (Network layer). Though the outlook is still quite implementation independent, there is certainly a drift towards turning the high level specification of the protocol into something useful that can eventually be implemented. In the process, we introduce more terminology and idiosyncrasies related to XTP. Using the same overall structure, as displayed in Figure 8 and Figure 9, we first outline the unicast model, and then the multicast model.



contextDB (host-P)

local				far		
key	srcport	srchst	memloc	dstport	dsthst	key
I	U	P	m	R	F	C
J	V					
K	W					

contextDB (host-F)

local				far		
key	srcport	srchst	memloc	dstport	dsthst	key
C	R	F	m	U	P	I
D	S					
E	T					

Figure 8: XTP - the unicast model

3.5.1 The unicast model

Figure 8 shows two large vertical rectangles, labeled respectively **Host-F** and **Host-P**, which represent two interconnected remote computers. The information within a large rectangle refers to concepts related to a particular host, concerning events happening within the hardware boundaries of that machine. The diagram also shows two horizontal lines going from the inside of one machine to the inside of the other machine. The top horizontal line represents a virtual connection only and accounts for a unicast data exchange from Host-F to Host-P. To simplify the diagram, and the presentation, Host-F is attributed the role of the sender and Host-P the role of the receiver. However, in reality, the exchange can be duplex with Host-P returning acknowledgements and control information to Host-F for instance. Below the top horizontal line, the box labeled **unicast association** accounts for the attributes of the relationship between the communicating endpoints. Such relationship being virtual only, a mirror image of it is recreated in the **contextDB** operated by the local implementation instance of XTP, as explained later in more detail.

The thick bottom line represents the physical connection and accounts for the fact that both machines need to be connected through hardware components such as network interface cards (NIC), electrical wires, switches, routers and such. This hardware medium is naturally lossy (unreliable), as packets can be lost due to magnetic storms, overflow at router queues, etc. We emphasize the fact that our aim is to model a distributed system, the physical distance between hosts could be as small as one thousand millimeters, or as large as ten thousand kilometers or more.

We now shift the emphasis to the inside of rectangles labeled **Host-F** and **Host-P**. The information content of both rectangles is quite similar, differing only in the field contents of table **contextDB**. Each rectangle contains the three-component architecture view of XTP referred to previously: (1) the top box labeled **XTP client** stands for the XTP client component; (2) the larger middle rectangle labeled **XTP protocol** stands for the XTP protocol component; (3) and the lower box labeled **DDS** stands for the underlying data delivery service. Two Application Programming Interfaces (**API**) boxes are also shown between the three main components.

Within a given host, there must be only one instance of the XTP protocol in activity, acting as server to local as well as to remote clients. This exclusiveness is necessary to allow the demultiplexing of packets by lower level protocols, such as IP. However, there could be zero or more local clients active at any time (indicated by the • notation), provided of course that an instance of the XTP protocol is on duty. For XTP, communication endpoints (CE) are mirrored with *context*. The bare existence of one XTP client implies at least one context, but there could be many contexts per XTP client (1+ notation in the diagram). An example would be a Netscape type of client that needs many unicast associations with remote servers. A *context* consists of a set of records representing one instance of the use of XTP at an endpoint, including its state. Within a host, each XTP context needs to be uniquely identifiable.

contextDB: With reference to Figure 8, we show examples of XTP clients with three communication endpoint instances, respectively labeled as communication endpoint R, S, T at Host-F, and U, V, W at Host-P (these are transport level port#). Each time one client registers a CE, XTP effectively creates a new *context*, or modifies existing data structures to keep track of it. Table labeled **contextDB** is used as a pseudo minimal database to illustrate important aspects related to XTP record keeping and identification of contexts. The *key*-local, *srcport*, *srchst*, *memloc*, *dstport* and *dsthst* fields pertain to information available locally and can be filled as soon as the local context is created. The *key*-far field pertains to information learned from the remote context with whom the local context may eventually form an association. We review the semantics of these fields in the next paragraphs.

The *key*-local is a unique context identifier assigned locally by XTP. If a received packet (reasoning from the receiver's viewpoint) contains the local key (i.e., the key assigned at the receiving end), then the *abbreviated context lookup* procedure can be used to find the relevant receiving context more quickly than otherwise. However, a received packet does not always include the local key, but it would otherwise always contain the remote key (i.e., far key in receiver's contextDB, the local key of the sender). In this case, use is made of the pair (source address, remote key) to find the proper receiving context. For this case, the XTP specification uses the terminology *full context lookup*.

The *srcport* field is used to store a local "port" number, which identifies a *Transport Service Access Point* (TSAP - in the ISO world). For the unicast model, and given a Transport level protocol such as XTP within a given host, a port uniquely identifies a context with regards to the transactions between the client and XTP. A port is visible to both client and XTP, whereas the key is only visible to XTP. For the unicast model, one port number maps to exactly one key.

For a kernel implementation of XTP, the *srcport* field value is equal to the port number used by the client application when binding a BSD socket (for instance) with the Operating System for receiving purposes. XTP being part of the kernel, the O.S. is therefore aware of the port number.

For a user level implementation of XTP, the *srcport* field value is a *logical* port number known to both the application and XTP, but not known to the Operating System. For such case, another pre-agreed O.S. aware DDS identifier (with UDP or IP raw for instance) would be supplied by the user (or hardcoded) when starting the XTP daemon.

The *srcport*-local being unique, why also invent another unique identifier mechanism, i.e., the *key*? The reason lies with the problem of *aliasing*, which occurs for instance when a receiving endpoint prematurely terminates with possible reallocation of the same identifier to a new receiver. Packets could then reach the wrong receiving context. One solution to the problem of aliasing consists of waiting some delay before reallocating an identifier. The drawback of this method on a highly active system might be a "hold" on a significant part of the identifier numbering space (not counting the overhead of the timers...). XTP's response with the *key* mechanism is to subdivide it in two sub-fields: (1) an index part used for locating the client's context; (2) an instance part used to validate active index values. The *key* is valid only when the two local instance values match with the ones included with a received packet. This decoupled mechanism reduces the chances of packets being delivered to the wrong context.

The *srchst*-local field is used to contain the unicast address of the local host, which uniquely identifies this machine within a network. For convenience here, we assign unicast address "F" to machine labeled Host-F, and "P" to machine labeled Host-P.

The *memloc* field stands for *memory location.* The interaction of XTP with its local clients implies a locus of exchange for the movement of data; i.e., buffer space needed to store incoming data if the client acts as the receiver, or to store outgoing data to be passed to XTP if the client acts as the sender. Such a memory address, which is useful only locally, is nonetheless an important item of information to be maintained by XTP for each context. The two following examples illustrate the potential use of the *memloc* field: (1) when the same XTP client has multiple contexts, such as allowed with the multicast mode, the *memloc* field is a way for XTP to discriminate between contexts for the data movements; (2) for a user level implementation (seen later), and if shared memory is used between XTP and a client, then the *memloc* field automatically targets a specific context for data movements.

The *dstport* field is the reciprocal of the *srcport* field described previously, but for the other (destination) host. In general, a priori knowledge of an O.S. aware port number is needed when sending packets.

For a kernel level implementation, the *dstport* field value is equal to the O.S. port number used to build a destination address structure (containing destination address, destination port number) after opening a BSD socket (for instance). As it is used at the other end by XTP to demultiplex (upwards) to the proper receiving application context, the destination port changes with each different unicast association.

For a user level implementation, the *dstport* field would contain the logical port number known to both the application and XTP, but not known to the Operating System. For such case, another pre-agreed O.S. aware DDS identifier (with UDP or IP raw for instance) would be supplied by the user (or hardcoded) when starting the XTP daemon. The destination logical port number changes with each different unicast association and is used by XTP to demultiplex in the direction of its clients; whereas the O.S. aware DDS identifier remains constant throughout the lifetime of the XTP daemon, and is used upwards by the DDS to deliver packets to XTP.

The *dsthst*-far field corresponds to the unicast address of the other party. For an initially sending context, a priori knowledge of it is needed and would usually be supplied by the user (as well as the *dstport*). For the initially receiving context, XTP soon learns the unicast address of the other party, such as upon reception of a FIRST packet.

The *key*-far field contains the key used by the other party, but as learned by local XTP through receiving a packet from the other party, such as a FIRST or a TCNTL packet. It corresponds to the definition of XTP's *return key*.

Unicast scenario: We now illustrate the functionalities of our make up table contextDB with a scenario, presuming: (1) communication endpoint R at Host-F, acting as the sender, is forming a unicast association with communication endpoint U at Host-P, who is the receiver; (2) a reliable file transfer; (3) the Internet Protocol Address Format; (4) an instance of XTP is on duty at both hosts; (5) a kernel implementation; (6) we abstract out many other important aspects, such as traffic specifications, etc.

At Host-P (unicast): The first move consists of starting up the XTP client at Host-P that registers with XTP for the purpose of receiving. XTP immediately creates a record in its table contextDB for this new context (known as context U), assigns a local key (*key*-local=I) to it and fills up source host address (*srchst*=P) and the memory location (*memlock*= m) fields. The *dstport*, *dsthst* and *key*-far fields are left empty for the time being.

At Host-F (unicast): The next events occur at Host-F. One XTP client is started and registers with XTP for the purpose of sending data. The port number (U) and the address of the destination host (i.e., P for Host-P) are included by the client with the registration request. XTP immediately creates a record in its contextDB for this context, assigns a local key to it (*key*-local=C), fills up the source host address (*srchst*=F), memory location (*memloc*=m), and the address of the remote host (*dsthst*=P). At this point, only the *key*-far field is left empty. XTP, on behalf of context C, then sends a FISRT packet including in it its locally available information (*key*-local=C, *srchst*=F, *srcport*=R), *dsthst*=P, *dstport*=U). The SREQ bit is also set to elicit feedback from the receiver.

At Host-P (unicast): Upon reception of this FIRST packet at Host-P, XTP has to use the "FIRST packet matching" algorithm to find an appropriate listening context. As this first packet is not a duplicate, the matching is done using the information contained in the Address Segment of the FIRST packet and some acceptance criteria (such as traffic specifications - ignored with the present scenario) that were set when context U was registered with XTP. As a result of a successful comparison, context U is identified as the receiving context. XTP at Host-P can now fill up the remaining fields of its table contextDB for context U, i.e., : (1) the key at which the corresponding context is known at the remote Host-F (*key*-far=C) ; (2) the unicast address of the remote host (*dsthst*=F); (3) the port number (*dstport*=R). XTP's contextDB record for context U at Host-P is now complete.

XTP at Host-P can now react and send back a TCNTL packet to Host-F and proceed further with the key exchange procedure. Within the TCNTL packet, XTP also includes its local key for context U (xkey=I) and set the RTN bit to indicate that the key field in the FIRST packet contains a *return key*.

At Host-F (unicast): Upon reception of the TCNTL packet, XTP at Host-F can use the abbreviated context lookup procedure to locate context C and fill up the only remaining field of its contextDB for this record, i.e., the key at which the corresponding context is known at Host-P (*key*-far=I). ContextDB record for context C at Host-F is now complete, and so is the key exchange procedure. Contexts R and U have now formed a unicast association.

Thereafter, both contexts can set the RTN bit in their outgoing packets, signifying that the content of the key field is a return key (i.e., the key was generated by the receiver of the packet). The abbreviated context lookup procedure can then be used at both ends by XTP as an optimum method for mapping an incoming packet to the appropriate context.

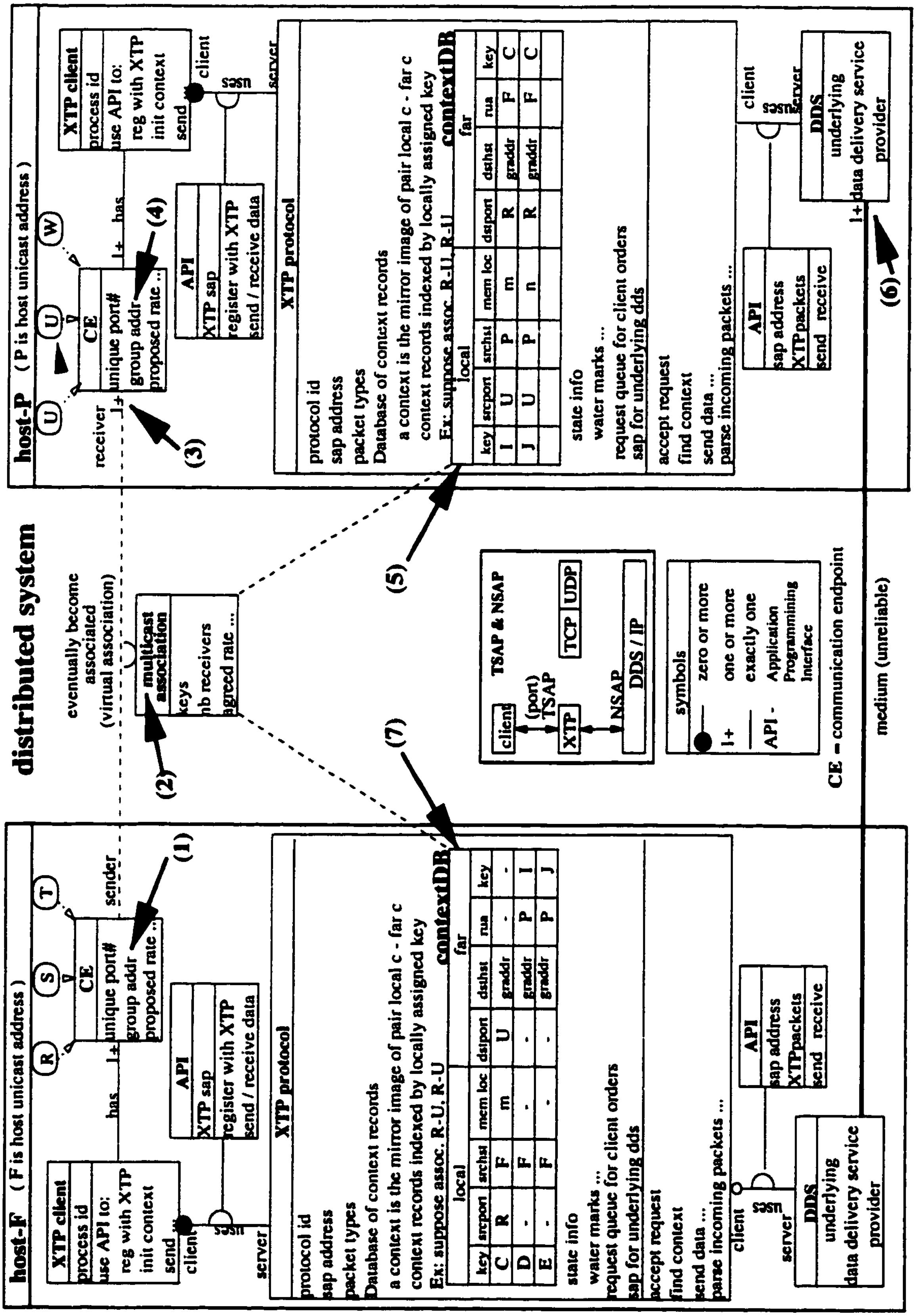
3.5.2 The multicast model

Figure 9 shows the three-component system architecture view of XTP for the multicast model. To describe the multicast model, we use "differential thinking", i.e., we concentrate mainly on the differences specific to the multicast model with respect to the unicast model (our base line). For convenience, the areas of difference are indicated with thick arrows on Figure 9, and also labeled with numbers (Ex: (1)) for cross-referencing purposes between the text and the diagram.

We also develop only the *one-to-many* multicast model (as opposed to the more general *many-to-many* multicast model). The plurality of receivers is illustrated on Figure 9 in many ways: (1) arrow 3 pointing to the **context** box illustrates the case of *one or more* (1+ notation) receiving communication endpoints (CE) that belong to the same XTP client (this same pattern could multiply for the same XTP instance, as there could be many XTP clients); (2) arrow 6 pointing to the **DDS** box illustrates the case of *one or more* (1+ notation) receiving hosts. The latter case is probably more suggestive of real life multicast groups, with a mapping *one context-to-one host.*

Within the XTP multicast model, the multicast mode is generally used for data exchange in the direction from the sending context to the receiving contexts. However, the unicast mode is used when the receiving contexts return their feedback to the sending context. This use of the unicast mode for returning feedback to the sending context is at the root of the "implosion" problem, which occurs when several receiving contexts return their TCNTL packet at approximately the same time to the sending context. The implosion problem is still an active subject of research within the multicast research community. As the XTP specification does not preclude use of the unicast mode by the sending context, for instance to send the END bit to a very slow receiver (ref. [XTP40, p81]), we also take into account this possibility in the following paragraphs.

One important difference of the multicast mode (WRT unicast) consists of the use of the same group address by all members of a multicast group; all receiving contexts listen "on the group address" and the sending context uses this group address when sending data. It is the job of multicast capable routers (network layer) to forward



host-F contextDB:

local					far		
key	srcport	srchst	mem loc	dstport	dsthst	rua	key
C	R	F	m	U	graddr	-	-
D	-	F	-	-	graddr	P	I
E	-	F	-	-	graddr	P	J

host-P contextDB:

local					far		
key	srcport	srchst	mem loc	dstport	dsthst	rua	key
I	U	P	m	R	graddr	F	C
J	U	P	n	R	graddr	F	C

Figure 9: XTP - the multicast model

the packets, as per the semantics of the multicast model. As the sending context issues only one packet, a group addressing technique is needed in order to reach many receiving contexts. Such change is flagged on Figure 9 by arrow 1 and arrow 4. The repercussions for table contextDB are traced with the help of the following **multicast scenario.**

Another important difference of the multicast mode (WRT unicast) consists of the relaxation of the exclusivity rule for the attribution of port numbers by the Operating System. For the unicast mode, the exclusive use of a port number by only one context is dictated **precisely** by the semantics of unicast. However, the semantics of multicast is different and is such that many contexts can receive the same data, though in their own memory address space. One could provide the semantics of multicast by using only the multicast group address to target receiving contexts and not care about the port number. Another tactic would be to use the port number, in addition to the multicast group address, but to allow multiple receiving contexts to listen (to bind) on the same port number within the same host. Many reasons militate in favor of the latter method (i.e., use both port number and group address when sending and receiving): (1) the ubiquity of the unicast mode and the consequent widespread need to specify both destination address and port number when sending data on real computer systems; (2) its better conformance with the semantics of the multicast mode.

Note: Some O.S. may enforce the port number exclusivity within a host even for multicast, thereby preventing several receiving contexts listening on the same port number within the same host. This behavior should however be considered a peculiarity of this O.S., rather in deviance with the semantics of the multicast mode.

contextDB: Figure 9 shows one XTP client at Host-P that has two receiving communication endpoints labeled with the same port mumber U. Table contextDB has one more field (*rua*-far). We now comment on the fields that have a slightly different semantics for the multicast mode.

The *dsthst* filed contains the multicast groups address, as normally supplied by the user.

The *rua* field title stands for return unicast address. This new field is needed with the multicast mode for two reasons: (1) at the receiving sides, to keep track of the sending context unicast address for the purpose of returning feedback (such as acknowledgments) to the sending context; (2) at the sending side, to learn and store the unicast address of each receiving context, thus know who the members of the group are for reliability, and also to potentially send some unicast packest to a particular receiver.

Multicast scenario: Admittedly, our model used to represent XTP multicast falls short of the specification itself and the revision to XTP4.0 (date March 1996) following the June Forum meeting in Dallas. However, given the outlook of the present report, it is judged sufficiently elaborate to provide some insights into the workings of XTP multicast.

For the nexts paragraphs, we refer to table **contextDB** (arrow 5 and 7) shown on Figure 9 and its various fields. Our assumptions for this multicast scenario are the following: (1) communication endpoint R at Host-F, acting as sender, is forming a multicast association with communication endpoints U at Host-P, the receivers; (2) a reliable file transfer; (3) the Internet Protocol Address Format; (4) an instance of XTP is on duty at both ends; (5) a kernel implementation; (6) we abstract out many aspects, such as traffic specifications, etc.

At Host-P (multicast): Same as for unicast, the first move occurs at Host-P and consists of starting the XTP client, who then registers with local XTP two contexts for the purpose of receiving. In contrast to unicast, a multicast client must pass the multicast group address as part of of its registration request (the steps are described in Section 3.4.3). XTP immediately creates two records in its contextDB and assigns keys to them (*key*-local=I to one and *key*-local=J to to the other). The other local fields are also filled up (i.e., *srcport*=U and *srchst*=P for both, *memloc*=m for one and *memloc*=n for the other, and *dstaddr*=graddr for both). The other far fields (i.e., *dstport*, *rua*, and *key*-far) fields are left void, for ignorance of the identity of the sending context at this point.

At Host-F (multicast): The next events occurs at Host-F. One XTP client is started that registers with local XTP one context for the purpose of sending multicast data (MULTI=1). The destination port number (U) and the multicast group address (graddr) are supplied by the client with the registration request. XTP immediately creates one record in its contextDB for this context, known locally as R (*srcport*=R), assigns a local key to it (*key*-local=C), and also fills up the other local fields (i.e., *srchst*=F, *memloc*=m, *dsthst*=graddr and *dstport*=U). The other far fields (i.e., *rua* and *key*-far) for this context will never be filled, as they are not needed for multicast sending (cannot use a return key when sending to many endpoints). XTP on behalf of context C at Host-F, then sends a FIRST packet, including only its locally available information (i.e., using XTP FIRST packet field names key=C, srchst=(**unicast address**)F, srcport=R, dsthst=**multicast**graddr). The SREQ bit is also set to elicit feedback from the receiving contexts, and thus learn who the receiving contexts are.

At Host-P (multicast): Even if there are two listening contexts (a particularity of this scenario only), the semantics of the multicast mode is such that only one FIRST packet hits Host-P (the last router "puts" only one packet on the sub-net, therefore only one packet is delivered to XTP). This FIRST packet is matched against all listening contexts. For this scenario, the only relevant match criteria is the multicast group address (field local *dsthst* in table contextDB). Consequently, contexts I and J are identified as the receiving contexts, having the same multicast group address as the one contained in the FIRST packet. Handling this FIRST packet, XTP also learns the **unicast** address (from the srchst field) and the key of the sending context. XTP can then update its table contextDB accordingly (i.e., *dstport*=R, *rua*=F and *key*-far=C) for the two records. The pair (far unicast address, far key) of sending context will be used thereafter to find listening contexts on incoming packets at Host-P from Host-F. Though the sending context does not yet know about the receiving contexts, the XTP specification considers that a multicast association has been formed at this point. The contextDB is complete at Host-P.

XTP, on behalf of contexts I and J at Host-P, can now return two unicast TCNTL packets to reveal the identities of the receiving contexts to the sending context. As there is only one sending context, XTP at Host-P can use the return key (key=C with RTN bit set to 1) to facilitate context lookup at the other end (abbreviated context lookup). XTP also includes the local keys (i.e., I for one and J for the other)

At Host-F (multicast): The reception of the two TCNTL packets at Host-F allow XTP to update its table contextDB, creating **two new** records to keep track of the two receiving contexts. Each record is assigned a unique local key (*key*-local=D for one and *key*-local=E for the other); the unicast address of the two receiving contexts (*rua*=P for both records) and the return key (*key*-far=I, *key*-far=J) are also stored. Table contextDB is now complete at Host-F, who has a complete knowledge of the multicast group. Therefore, XTP at Host-F has all the necessary information to ensure multicast reliability (knowing the identity of all receivers, it can ensure that feedback is complete) and to possibly engage a unicast exchange with some particular contexts, if and when the need arises.

The necessary unicast exchanges between the sending context and the receiving contexts, while using XTP multicast mode, are done through the same NSAP (Network Service Access Point) opened by XTP with its underlying data delivery service (DDS), rather than the one used to exchange multicast packets. As illustrated on Figure 9 (boxout labeled TSAP & NSAP), a *port* is a Transport Service Access Point used for the exchange of the data part of multicast packets between XTP and its clients. For the unicast packets, the client remains unaware of their exchange, though they transit through the same NSAP; they remain at the XTP (Transport) level and never bubble up to the user (Application) level.

For example, presuming some feedback is returned by XTP from Host-P on behalf of context with *key*-local=I, the packet field contents could be as follows: (1) MULTI=1; (2) dsthst=F; (3) srchst=P; (4) dstport=R; (5) srcport=U; (6) key=C; (7) RTN=1; (8) xkey=I; (9) adomain=0x01; (10) aformat=36(XTP). At the reception end (at Host-F), the packet is delivered to XTP by the DDS and XTP has to handle the packet. With regards to our scenario and table contextDB, an *Abbreviated Context Lookup* with criteria *key*-local=C easily yields the proper context, and the *dstaddr* field reveals the graddr used. A search in contextDB with criteria (*dsthst*=graddr AND *key*-far=I) yields proper local context D.

3.6 Protocol implementation strategies

This section marks a definite transition from our outlook at XTP as an implementation independent specification towards the Sandia implementation presented in the next chapter and used for the rate control experiments. Our assumptions become stronger towards considering XTP in conjunction with an Operating System (O.S.), namely UNIX with its characteristic timesharing and multitasking dimensions, and a particular network environment, namely the Internet with IP at the Network level. Given those assumptions, we develop two implementation strategies: (1) the implementation of XTP as a user level daemon process, as done by Sandia; (2) or as integrated within the kernel code of the O.S. Both strategies are summarized on Figure 10.

3.6.1 User level implementation of XTP

XTP is implemented as a user level daemon process, acting as a server to its clients, but with same status as its clients with regards to the O.S. kernel. The XTP server is a process created, managed and scheduled by the kernel, the same as its XTP clients, and context switches are needed to pass control from server to client and vice versa. Any user can start the XTP daemon, but care must be taken so that only one daemon is running at a given host.

As shown on the left part of Figure 10 (User Level Implementation), there is a need for an Application Programming Interface (API) to define the format of the messages (data structures) and effect the transactions between the daemon and its clients. Such an API must be designed to convey to the daemon the client requests, such as for the setting of option bits (Ex: `MULTI`=1) and other parameters, such as amount of data, quality of service (Ex: rate=25), etc.

Furthermore, with regards to the exchange of the data, the daemon and its clients are in a *producer/consumer* relationship. Consequently, there is also a need for the design of the interprocess communication mechanisms (IPC) to convey the messages between

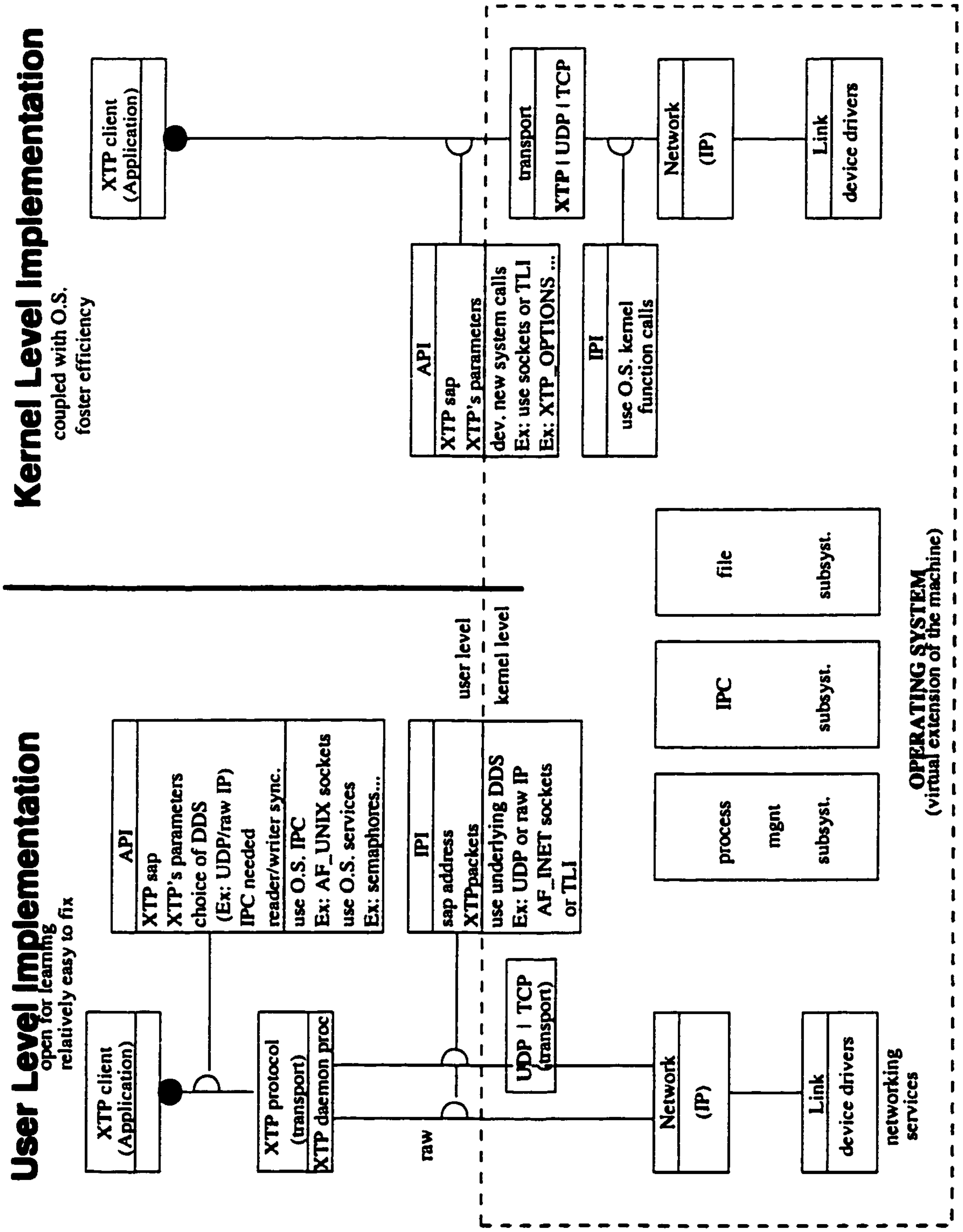


Figure 10: Implementation strategies

the clients and the server and to ensure coordination in their producer/consumer interactions.

At a lower level, the O.S. is effectively acting as the data delivery service (DDS) for the XTP daemon. For multiplexing and demultiplexing purposes, all data transactions towards or from the network must transit through the daemon. Hence, there is a need to use a Network Programming Interface (call it Implementation Programming Interface - IPI) embedded within the code of the daemon to deal with the O.S.. As shown on the left part of Figure 10, the XTP daemon has two alternatives: (1) whether to interface directly to IP, in which case root permission is needed; (2) or interface indirectly to IP through UDP, as only regular user permissions are needed. UDP being a *minimal* protocol, it can be used as a DDS to XTP.

3.6.2 Kernel level implementation of XTP

This is the first and last time we discuss kernel level implementation of XTP, as the Sandia implementation presented in the next chapter and used for the rate control experiments is a user level implementation. With such an implementation, the XTP client (rather the Network Application Programmer using XTP) interfaces directly with the services of the O.S., the same as when using kernel integrated protocols such as TCP or UDP. In fact, the API of a kernel level implementation closely resembles the IPI of a user level implementation. The Network Application Programmer would use such well known API such as BSD sockets or TLI (Transport Layer Interface). For the system programmer, a kernel implementation implies possibly the design and implementation of some new system calls, and at the very least new options to allow the Network Application Programmer to care for the issue of XTP specific options and other numerous parameters.

The code of XTP would be integrated within the kernel code, but as shown on the right part of Figure 10, there would still be a need for an IPI, which would consists of internal kernel function calls to make use of the services of other kernel modules, such as IP.

4 The structure of the SandiaXTP implementation

"The best programmers,
like the best craftsmen,
understand the tools
they use."

Dr Jack Davidson
Professor of Computer Science,
University of Virginia

Taking advantage of the Object paradigm, which features *inheritance* and *component reuse*, the approach adopted at Sandia National Laboratories for implementing XTP has two levels: (1) within the user level architectural framework, isolate a core of Transport level functionalities that would be common to all Transport level protocols; (2) provide XTP's functionalities as a specific extension of this basic core, with view of being able to eventuallly derive other Transport level protocols as well from the same core. This basic core of Transport level functionalities identified at Sandia National Labotatories consists of the following:

1. maintain *contexts*, i.e., structured memory space used to mirror the state of associations (database aspect);

2. provide the services of a *context manager* to demultiplex incoming user requests or incoming packets to the proper context (search and match).

3. supply *packet* shells, i.e., structured memory space used for sending and receiving packets (list management);

4. interact with some underlying data delivery service (DDS) to send and receive the packets referred to in (2) via the network (network programming);

5. provide access to the functionalities of the protocol, i.e., an Application Programming Interface (API);

The result of this work of abstraction is embedded into the *Meta-Transport Library* (MTL), i.e., a set of C++ basic classes designed to provide an infrastructure for deriving Transport layer protocols.

Consequently, Sandia distributes its software in two separate packages respectively identified as follows: (1) **Meta-Transport Library** - A protocol Base Class Library; (2) **SandiaXTP** - An Object-Oriented Implementation of XTP4.0 Derived from the Meta-Transport Library. The MTL package only is needed to compile Client programs, whereas the two packages are needed to compile a daemon program.

Making use of Object Modeling Techniques (OMT), the goal of this Chapter is to articulate a global perspective as well as detailed internal views of the SandiaXTP software as formed of a conglomerate of several MTL base classes coupled with XTP derived classes. First, the daemon is studied, and then the Client.

Admittedly, SandiaXTP has much more inbuilt generality than that exposed in this chapter. The focus has boundary names such as: UDP as the data delivery service; Internet as the network carrier; rate control as the the preferred XTP feature; and the gain of insights for the interpretation of the experimental results and for possibly changing the source code. Furthermore, there is no attempt to paraphrase Sandia's documentation, which is abundant, very well written and was heavily used to develop this OMT based model. Such documentation consists of: (1) comments embedded within the source code; (2) the *Meta-Transport User's Guide*; (3) the *Meta-Transport Library Reference Manual*; (4) the *SandiaXTP User's Guide*; (5) and the *SandiaXTP Reference Manual*.

4.1 About OMT, the diagrams and the notation used

Being a task of "reverse-engineeering", executed by a "newcomer" to both OMT and XTP, one should not develop too stringent expectations regarding the accuracy of the diagrams and the accompanying text produced, when compared to the "real" XTP implementation. The analogy with a geographical chart holds; not fully accurate with regards to the circumvolutions of the terrain, but good enough to guide one's whereabouts.

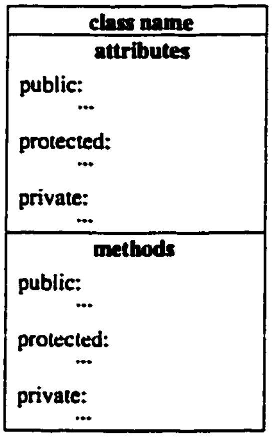
The model being developed is expressed primarily with diagrams having many formats and purposes: (1) Object Model - Class diagram; (2) Object Model - Object diagram; (3) Dynamic Model - Event Trace; (4) Draft Class dictionary. The latter was not meant to be fully developed, but rather used as a convenient memento to support interim progress with the preparation of the diagrams. Being at a level of detail that would disrupt the flow of the text, it is included in its up-to-date draft status in Appendix D.

Though we do not pretend to formalism in building our model, we have tried to adhere as much as possible to the current usage for notation and type of diagrams used, such as presented in "**Object-Oriented Modeling and Design**" by Rumbaugh & al. The diagrams are presented in their "life cycle order", i.e., as they would evolve when OMT is used before writing the code. However, reverse-engineering means that a reverse path/order was followed for preparing the diagrams: first, study the source code; then start preparing a draft class dictionary; then proceed with event traces and Object diagrams; and finally abstract the class diagrams. Of course, the sequence was not that smooth, with frequent re-visits of previous stages.

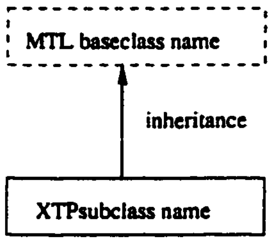
4.1.1 About the Class diagrams used in this report

A **Class diagram** depicts classes, their structure, and the static relationships between them. Normally, a rectangle used to represent a class with OMT has three parts: (1) a top part with **class name**; (2) a middle part listing attributes/variables that belong to the class; (3) a bottom part listing methods/functions. For want of space, many of the Class diagrams used in this report, such as the ones shown on Figure 12 or Figure 16, show only the class names. However, much more detailed information (though still largely incomplete) is presented in Appendix D.

MTL classes, i.e., the ones that are part of the MTL package, are represented with dashed contour rectangles. XTP derived classes are represented with regular contour rectangles, but are also easily recognized for having the prefix *XTP* or *xtp* prepended to their names. Illustration:

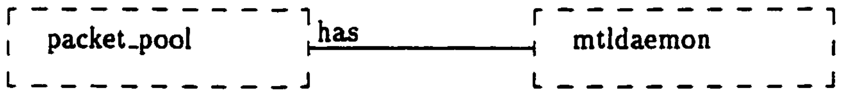


Usual OMT Class diagram



Class diagram used in the text

The relationships between classes are expressed in many ways: (1) **inheritance** is shown with the arrow pointing from the derived class towards the base class; (2) 1+ means **one or more**, the dot • means **zero or more**, ⊢ means **exactly one**; (3) like an arrow with more mass concentrated at the tip of the pointing side, the relative location on the diagram of the link attribute indicates the "direction" of the semantics (Ex: "mtldaemon has packet_pool" or "packet_pool has mtldaemon"? - the fact that **has** is located closer to class packet_pool than to class mtldaemon on the diagram indicates the proper semantics, i.e., "mtldaemon has packet_pool"). Illustration:



4.1.2 About the Object diagrams used in this report

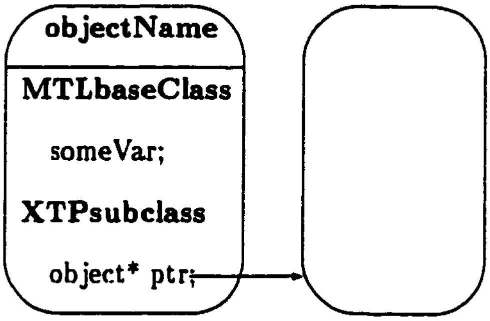
An Object diagram depicts a particular object structure at run time. As they show instances of the classes, the term *instance diagram* is equivalently used for such diagrams. Instance diagrams are mainly used to show examples to help clarify a complex Class diagram, as it conveys much more information regarding the data types and the data structures used, the nature of the asssociation links between the objects shown on the Class diagram, and the number of objects being created.

The symbol used to represent an instance of a class, or of a hierarchy of classes (such as base class←—subclass), is a two part rectangle with rounded edges. The top part, which is relatively small, shows the object identifier, which is borrowed as much as possible from the source code. Sometimes, the class name is used as the object identifier (EX: **context_manager**). Otherwise, the pointer associated with the C++ **new** function call, when the object gets created (dynamically claiming memory space), is used (Ex: from context_manager, **c_s_bm** = new buffer_manager()).

The lower part of the rounded rectangle lists many entries, with class names indicated with bold captions, followed by some illustrative class attributes. Within the lower part, the object hierarchy appears in the bottom-up fashion, i.e., the subclass name appears below the base class name. For example, as class XTPcontext is derived from class context (context←—XTPcontext), the XTPcontext class name and its attributes are listed below the context class name and its attributes.

Whereas arrows are used to convey inheritance relationships on Class diagrams, they are used on Object diagrams to represent a pointer owned by one object and pointing to another object.

Illustration:



4.1.3 About the Event trace diagrams used in this report

An Event trace diagram shows the flow of requests between objects. Figure 14 and Figure 15 are two examples of the particular brand of Event trace diagrams used in this report. The vertical dimension from top to bottom represents the flow of time. Vertical lines represent objects, i.e., compiled instances of their corresponding classes.

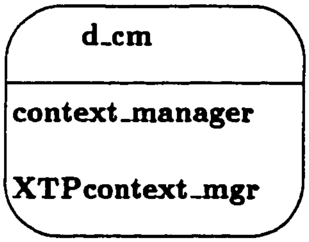
The line for an object becomes solid once the object has been constructed. Horizontal lines represent calls made from one object to another, with tail of the arrow indicating the source object making the call and the head indicating the object that executes the call.

To avoid confusing an underscore (_) with a horizontal line, all the underscores of labels based on the source code are replaced in the Event trace diagrams with a hyphen (Ex: **d_cm** becomes **d-cm**).

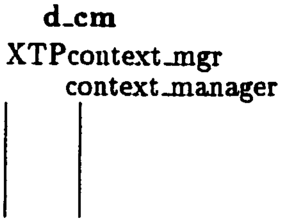
Most (if not all) objects shown on Object diagrams are also represented on their corresponding Event trace diagram, with the same label identifiers appearing highlighted at the top of the vertical lines on the Event trace diagram. To help for clarity, pseudo-objects are sometimes used, such as **user-main()** (see Figure 14) representing the actions of the user, and **O.S.** representing the Operating System.

In as much as possible, the notation used on the Event trace diagrams is the exact replication of some variable, or line of code contained in the source code. However, this rule is not followed strictly (Ex: XTPcontext_manager might be abbreviated to XTPcontext_mgr for want of space, as shown on Figure 14).

The Object identification scheme used at the top of the vertical lines is coordinated with the one used on the Object diagrams. For example, **d_cm** (ref. Figure 14) refers to the context manager object instance. The fact that this object instance is a compound made up of a MTL base class and a XTP derived class is represented with two vertical lines situated at a close distance. Of the two, the left line represents the derived class component, with corresponding label indented relatively to the left and at the top (i.e., XTPcontext_mgr for this example); the right vertical line represents the base class component, with its corresponding label indented a bit to the right (i.e., context_manager is indented to the right of XTPcontext_manager). Illustration:

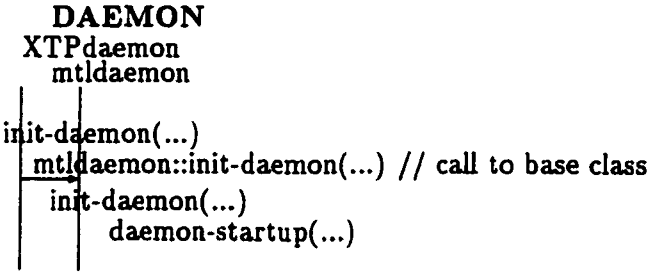


Object diagram



Event trace

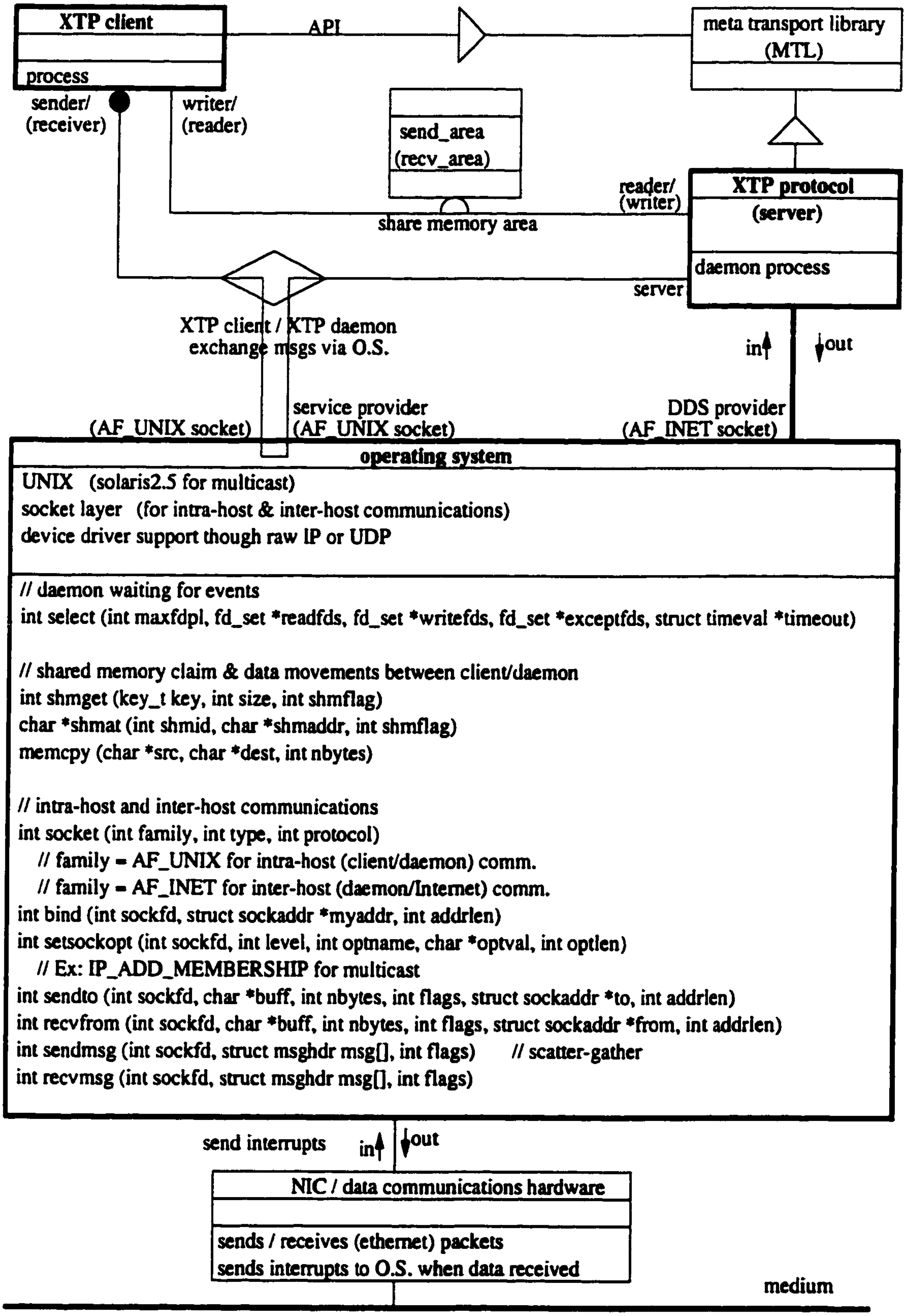
The labels shown on top or between the horizontal arrows are function calls or statements extracted from the source code. Some "code independent comments" (starting with //) are also presented. In most cases, identation matters. A line of code that belongs to a MTL base class, such as the init_daemon() method implemented by mtldaemon, would be indented to start over the vertical line representing the mtldaemon class component; with following calls properly indented. For example, start_daemon() is indented to the right of init_daemon(), as the call is made from init_daemon. Illustration:



4.2 SandiaXTP global architecture and dynamics

The formulation of the model starts with a global presentation of the SandiaXTP implementation, to be refined and elaborated in later subsections.

Figure 11 is meant to illustrate the global architecture and dynamics of SandiaXTP at a given host. To facilitate recognition with the three-component view of XTP introduced previously (see Figure 8 & Figure 9), these same three components appearing again in Figure 11 have been highlighted (bold contour) and are described as follows: (1) the **XTP client** takes the form of user level processes and is represented by the upper-left class rectangle labeled as such; (2) the **XTP protocol** takes the form of a user level daemon process and is represented by one upper-right class rectangle


XTP client
process
API
meta transport library (MTL)
sender/ (receiver)
writer/ (reader)
send_area
(recv_area)
reader/ (writer)
XTP protocol
(server)
daemon process
share memory area
server
XTP client / XTP daemon exchange msgs via O.S.
in
out
(AF_UNIX socket)
service provider (AF_UNIX socket)
DDS provider (AF_INET socket)
operating system
UNIX (solaris2.5 for multicast)
socket layer (for intra-host & inter-host communications)
device driver support though raw IP or UDP
send interrupts
in
out
NIC / data communications hardware
sends / receives (ethernet) packets
sends interrupts to O.S. when data received
medium

```
// daemon waiting for events
int select (int maxfdpl, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)

// shared memory claim & data movements between client/daemon
int shmget (key_t key, int size, int shmflag)
char *shmat (int shmid, char *shmaddr, int shmflag)
memcpy (char *src, char *dest, int nbytes)

// intra-host and inter-host communications
int socket (int family, int type, int protocol)
   // family = AF_UNIX for intra-host (client/daemon) comm.
   // family = AF_INET for inter-host (daemon/Internet) comm.
int bind (int sockfd, struct sockaddr *myaddr, int addrlen)
int setsockopt (int sockfd, int level, int optname, char *optval, int optlen)
   // Ex: IP_ADD_MEMBERSHIP for multicast
int sendto (int sockfd, char *buff, int nbytes, int flags, struct sockaddr *to, int addrlen)
int recvfrom (int sockfd, char *buff, int nbytes, int flags, struct sockaddr *from, int addrlen)
int sendmsg (int sockfd, struct msghdr msg[], int flags)      // scatter-gather
int recvmsg (int sockfd, struct msghdr msg[], int flags)
```

Figure 11: SandiaXTP global architecture

labeled as such; (3) the role of the **DDS** is undertaken by the operating system and is represented by the large class rectangle labeled as **operating system.**

The MTL is shown at the top-right of the diagram. We show that both client and XTP inherit (with the usual arrow pointing to the base class) from the MTL. The client inherits the API by including the MTL in its source code; XTP inherits many base classes as will be explained with many more details later. The lower part of Figure 11 shows the host networking hardware (NIC) and the medium. This layer (Link layer) is under the control of the Operating System with its device drivers and need not concern us any more.

The role of the operating system is paramount. For the daemon, it is the DDS. It also facilitates all communications between the daemon and all client processes. Such interprocess communication needs take two forms: (1) regarding the movement of net data to be sent or received; (2) regarding user requests passed to the daemon about the net data to be sent or received. To illustrate the dynamics and the interactions between the three main components, two scenarios are used; one for sending data, one for receiving data.

Scenario for sending data: First, the daemon is presumed to be on duty, but idle (i.e., not running) at the present time. Having made use of the select() system call, it is probably awaiting on some I/O queue for the earliest occurence of one of the three following events: (1) reception of some user request; (2) arrival of some packet(s) from the network; (3) or a timeout if neither of the preceeding two events had occurred. At this point, the operating system is in full control, as it is responsible for the handling of these three events. User requests are conveyed to the daemon via internal sockets (AF_UNIX), an IPC mechanism operated by the kernel. Incoming packets from the network are handled by modules integrated within the kernel, such as the network device driver necessary the handle the interrupt when a packet comes in, and IP for demultiplexing which protocol should the packet be delivered to. The clock for the timeout is also under the control of the Operating System.

To start the ball rolling, the client process would deposit some data into the shared memory send area (send_area). At this point, the daemon is still unaware of the action and intents of the client. To express its needs to the daemon, the client fills up a data structure, including its expectation to send the data, the offset from the beginning of the send area, and the data byte count. The client would then convey its request to the daemon using an internal socket (AF_UNIX) IPC mechanism, and possibly block waiting for the daemon's response. It now presumed that the client process becomes idle (i.e., is placed waiting on some I/O queue for instance).

The issue of such user request to send data being one of the three events described previously, the operating system would now schedule the daemon process for running. The daemon would then take action in accordance with the client's expectations contained in the user request data structure, apply the rules of the protocol, and make up a complete XTP DATA type packet with header. It would then call again on the services of the operating system (with system calls such as sento() or sendmsg()) for further handling, such as by IP to include routing information and the network device driver to "put" the packet on the wire. Our high level send scenario is now complete.

Scenario for receiving data: It is presumed that a client has blocked after having issued a user request to the daemon and is waiting the reception of data from the network. Packets can arrive any time (asynchronously) from the network. The occurrence of such an event would make the daemon return from the select() system call (the work of the operating system again) with a copy of the packet being made from kernel space to XTP's own memory space (packet shell). Using the header information, it is the job of the context manager (one part of the daemon) to find the appropriate awaiting context (probably using the *Abbreviated Context Lookup* procedure).

The XTP daemon would then engage in a three-phased action: (1) use again the services of the operating system to copy the data part of the packet from its own memory space to the shared memory receive area (recv_area); (2) fill up a message structure, including offset from the beginning of the shared memory receive area and the data byte count; (3) send this message to the client via an internal socket (AF_UNIX) IPC mechanism, thereby using again the services of the operation system.

The client can now be scheduled (again the operating system - in its scheduling capacities) to run and act upon the data. The high level receive scenario is now also complete.

4.3 SandiaXTP Daemon with O.M.T.

4.3.1 Daemon startup - Class diagram

Figure 12 illustrates the class diagram for the SandiaXTP daemon. The small diagram labeled **keymap**, and situated at the upper-left corner of Figure 12, reproduces at a reduced scale the three-component user-level implementation architecture used in SandiaXTP. This medallion, also to appear on some subsequent diagrams, is meant to highlight the component(s) (here the XTP protocol embodied in the daemon) that the remainder of the diagram focuses on.

Some comments follow regarding thc semantics conveyed by the Class diagram for the daemon. At the highest level of abstraction the classes can be lumped in two clusters (call it the *daemon cluster* and the *context cluster*), whose focal points are highlighted on Figure 12. Expressed in a nutshell, the daemon cluster plays the role of a nervous reactive system, offering common services to all contexts, whether before a particular context has been found, or in rcaction to calls made from a particular context. In contrast, the viewpoint of the context cluster is rather egocentric. When a particular context has been identified, control is passed to the context cluster with per context items and actions.

The focal point of the daemon cluster is the pair mtldaemon base class and its corresponding XTPdaemon subclass. The satellite classes in the daemon cluster are: (1) the pair context_manager/XTPcontext_ manager; (2) the MTL packet_pool class and its related packet class; (3) the MTL del_srv base class with its ip_del_srv and udp_del_srv subclasses.

The focal point of the context cluster is the pair MTL context base class with its Sandia XTPcontext subclass. The satellite classes members of the context cluster include (anticlockwise on Figure 12): (a) the XTPpacket base class and its six subclasses, each one corresponding to some XTP packet type such as the FIRST-

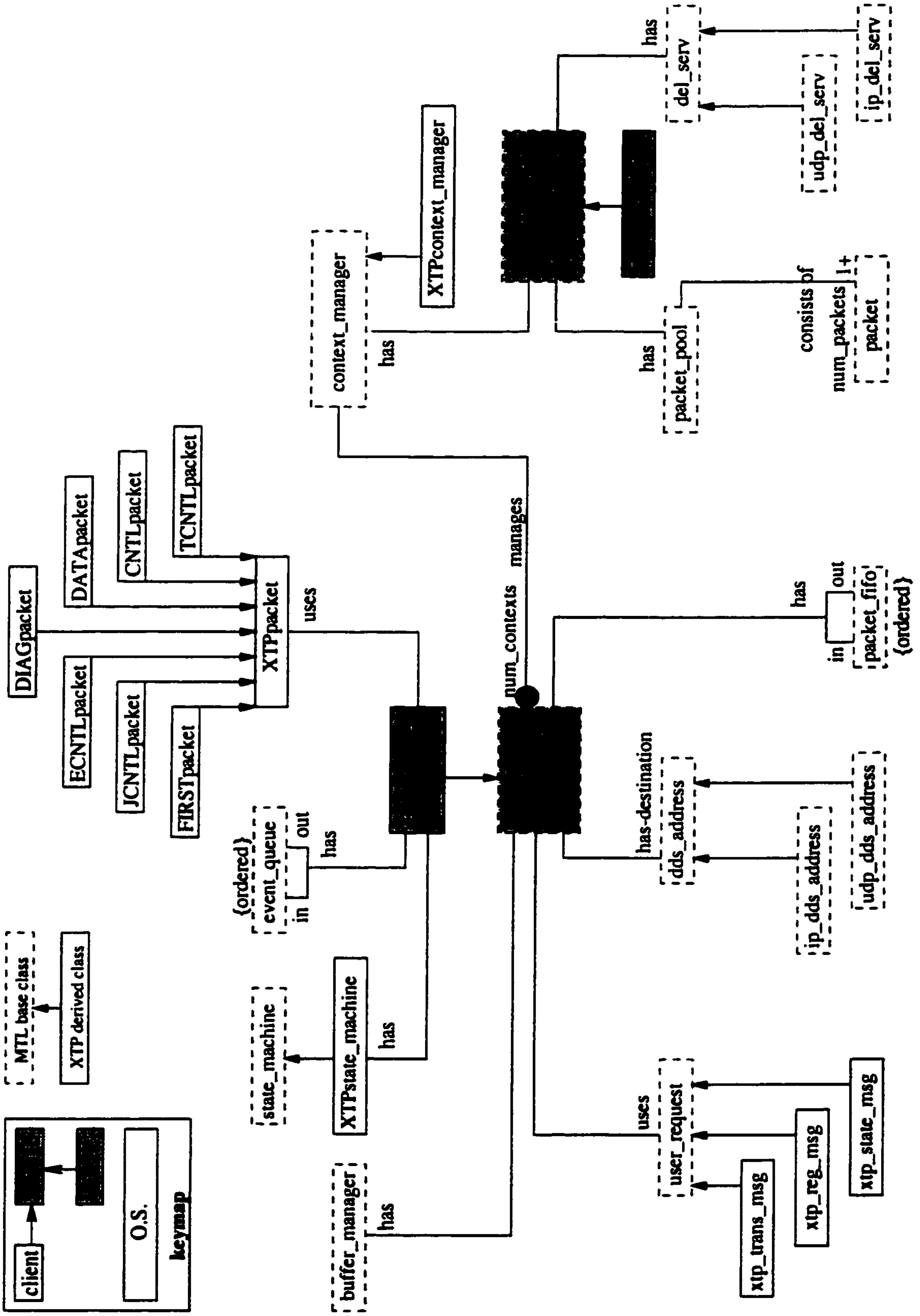


Figure 12: SandiaXTP daemon startup - Class diagram

packet type/subclass, the DATApacket type/subclass, etc.; (b) the MTL event_queue class; (c) the MTL state_machine base class and its XTPstate_machine derived class; (d) the MTL buffer_manager class; (e) the MTL user_request base class and its Sandia xtp_trans_msg, xtp_req_msg and xtp_state_msg derived classes; (f) the MTL dds_address base class and its MTL ip_dds_address and udp_dds_address subclasses; (g) the MTL packet_fifo class. Further comments are presented around this "two clusters" theme.

THE DAEMON CLUSTER

Classes **mtldaemon & XTPdaemon:** The behavior of the whole daemon process is *event driven.* With regards to the subsequent processing, the mtldaemon base class acts as the work engine of the whole daemon. Directly, or indirectly though its allied classes, the mtldaemon class cares for the following aspects:

1. Directly, it provides a method for the parsing of command line arguments (virtual int parse_args()), as specified by the user. Being a virtual method, only a default implementation is provided and the derived protocol is encouraged to provide its own implementation.

2. Directly, it has method init_daemon(), which converts the user process into a Unix daemon (call to private method start_daemon()), then installs the signal handlers (call to method install_handlers()), and also initializes other services covered by following items.

3. Directly, it has the main_loop() method for awaiting endlessly for some event to occur. Such an event could consist of an incoming packet from the network, a user request from some client, or a timeout if none of the previous two events had occurred.

4. Directly, it operates one FIFO request queue for receiving orders from clients, and eventually responding. For instance, method setup_request_queue() is used to set up the request queue; while methods recv_request() and send_reply() hide the underlying IPC mechanisms needed for the exchanges.

5. Indirectly, through its allied del_srv base class and derivatives ip_del_srv and

udp_del_srv subclasses, it deals with the DDS/O.S. for the purpose of receiving packets from the network, or for sending packets through the network.

6. Indirectly, through its allied packet_pool and packet classes, it operates a pool of packet shells for the temporary storage of protocol packets.

7. Indirectly, through its allied context_manager class, it is responsible for the demultiplexing of incoming packets to the proper receiving context, or of incoming user requests to the proper context.

The **XTPdaemon** derived class contributes few functionalities beyond the ones provided by the mtldaemon base class. One main contribution of the XTPdaemon subclass consists of reimplementing the MTL virtual parse_args() method. XTP knows best what command line arguments are needed from the user to start the daemon component. One such argument consists of the choice of the underlying data delivery service, whether raw IP or UDP (this is strictly not XTP specific, but needs to be included with the reimplementation of method parse_args()). If UDP is selected, then some mtldaemon protected variable is set accordingly (i.e., d_in_dds_t = UDP_type). Other important daemon startup command line arguments include the selection of the number of contexts and of the number of packet shells (mtl protected variables respectively called d_num_contexts and d_num_packets).

Classes **context_manager & XTPcontext_manager**: The context manager acts as matching entity between incoming packets or user requests on the one hand, and the appropriate context on the other hand. To eventually find a match, the context manager must conduct searches, which implies search criteria, proper organization and free access to the data. Class context_manager is therefore responsible for assigning *keys* to contexts (method init_each_context()), which are then later used as search criteria to retrieve a context given a key value (method find_context() - effectively implementing the *abbreviated context lookup* procedure). To access the data, class context_manager is declared a friend by class context. Exploiting the fact that class context has pointer variables for linking contexts together, class context_manager operates linked lists of active and quiescent contexts (EX: context* cm_active_list protected variable, and method add_active_context()). Virtual function handle_new_packet(), meant to be superseded by the protocol specific implementation, is designed to match an incoming packet from the network to a specific context and place the packet on that context's incoming FIFO queue.

Class **XTPcontext_manager** adds little to base class context_manager. One major contribution of class XTPcontext_manager consists of reimplementing the virtual method handle_new_packet().

Classes **del_srv, ip_del_srv & udp_del_srv:** The role of class del_srv, and its ip_del_srv and udp_del_srv subclasses, is to deal on behalf of the mtldaemon class with the underlying DDS for the purpose of sending packets through the network, or to receive packets from the network. Base class del_srv is an abstract base class that plays the role of an interface for the choice of a particular DDS. It provides many virtual methods, such as send() and receive() to be implemented by each of its subclasses. Class ip_del_srv is based on raw IP, whereas class udp_del_srv is based on UDP.

Classes **packet & packet_pool:** Much more detailed descriptions of class packet_pool and class packet are provided in Appendix D. To explain the role of both classes, a limited outlook is adopted. On their way in, net transport level packets are delivered to XTP without even knowing their type, the identity of the receiving context, or if any receiving context at all will be found. The daemon object must therefore have a supply of packet recipients, first for its own use while searching for a receiving context, then that can be placed on the incoming queue of a particular context if one is found, and eventually to be recycled for later reuse. These activities imply packet holders with suitable data structures for list management. The solution developed at Sandia National Laboratories is to make use of packet shells (class packet) to serve as packet holders and of a packet pool entity (class packet_pool) needed to create the required number of packet shells and to manage them with a list.

A packet shell is a data structure with many more fields than the ones strictly needed to contain the actual transport level packet. Amongst its many fields, a packet shell includes: (1) a pointer (packet* p_pnext) for linking packets shells together; (2) a flag to indicate whether it is being used as a monolith container or for scatter-gether

I/O; (3) an array of contiguous bytes in case it is used as a monolith; (4) a list of many address/length pairs in case it is used for scatter-gather I/O. In addition to providing the data structure, class packet has methods such as is_mono() to test if a packet shell is being used as a monolith or not, and method pkt_start() to return the beginning of the data part if monolith.

Class packet_pool has methods to claim a packet shell from the pool list (get()) and to replace it for eventual reuse while it is not needed (put_back()).

THE CONTEXT CLUSTER

Classes **context & XTPcontext:** The role of the context is to act as a repository containing the dossier for the end point of an association. As much of the state information is protocol specific, one can expect that the derived class will add much to the base class. Knowing all about these particulars, the context is also a launching base by making calls to methods that belong to other classes for finally sending or receiving the data. Directly, or indirectly through its allied classes, base class context takes care of the following aspects:

1. Directly, it stores identity information such as the key, the port number (TSAP) and the user's process id.

2. Directly, it has vitual methods for sending or receiving the data, such as send() and receive(), which are meant to be redefined by the derived protocol.

3. Indirectly, though its allied buffer_manager base class, it manages the shared memory send and receive areas.

4. Indirectly, through its allied state_machine base class, it provides an interface (pure abstract class) for the implementation of protocol specific state machine.

5. Indirectly, through its allied event_queue base class, it provides methods for the queuing of events that drive the protocol specific state machine. The events themselves are encoded by the derived protocol.

6. Indirectly, through its allied user_request base class, it provides a basic format

and methods for interacting locally with client processes. Class user_request serves as a base class for protocol specific user requests.

7. Indirectly, through its allied dds_address class, ip_dds_address and udp_dds_address subclasses, it cares for the formulation and storage of the addressing needed by the various DDS.

8. Indirectly, through its allied packet_fifo base class, class context cares for possibly enqueuing incoming or outgoing packets waiting to be further processed whether by the context or by the DDS.

Subclass **XTPcontext** is very important and adds much to the functionality provided by class context. For example, class XTPcontext must cater for the several parametric dimensions of XTP, such as rate, credit, burst and r_timer variables associated the rate control mechanism. For incoming packets, the context manager is responsible to demultiplex a packet to a proper receiving context. However, to do so, it makes calls to methods provided by class XTPcontext, such as method get_fclu() to conduct a full context lookup (the abbreviated context lookup procedure is handled by class context - method find_context(key)). Furthermore, XTPcontext's method process_packet() is used to process an incoming packet, which in turns makes other local calls to methods such as process_FIRST_packet(), depending on the packet type. When a context needs to send data, method send() redefined by XTP creates a packet out of the information given in the shared memory send area of the appropriate context's shared memory, as specified by the user request. The packet might be sent immediately, or enqueued on this context's outgoing packet_fifo queue for later processing.

Class **XTPpacket & subclasses:** Base class XTPpacket abstracts and encapsulates some unchanging aspects common to all XTPpacket types, which are based on the 32-bit fixed length header used for all XTP packet types. Consequently, class XTPpacket has methods for header placement and extraction (put_header(), get_header()), for placement and extraction of many specific header fields (insert_checksum(), get_key()), and to return a pointer to the middle part of a packet (middle_ptr()).

A set of subclasses is also derived from class XTPpacket, one corresponding to each of the seven packet types defined by XTP, with appropriate methods. For example, subclass FIRSTpacket has methods for address placement and extraction (put_address(), get_address()); subclass CNTLpacket has methods for placement and extraction of control information (put_cntl(), get_cntl()); etc.

The relationship between MTL base class packet and the XTPpacket base class needs clarification. The documentation provided by Sandia and the source code appears to support two conflicting viewpoints, which are now reviewed.

Thesis 1: class XTP packet **is not** a subclass of MTL class packet

> "The packet class is not designed to be a base class, as other MTL classes are. The packet class is just a repository for data to be sent or received via the data delivery service; packet classes from the derived protocols should contain a member variable of type *packet**, and their member functions should impose some order onto the raw data contained within." [Meta-Transport Library, User's Guide, June 12, 1997, p16]
>
> "The packet class is not designed to be the base class for protocol specific packet structure. Rather, protocol-specific packet classes should be defined so that they contain a member variable of the type packet*, where the packet shell is held..." [Meta-Transport Library, Reference Manual, MTL version 1.5, December 1996, packet(3)]
>
> Supporting the same thesis, the code declaration for class XTPpacket has the following form:

```
class XTPpacket {
private:
protected:
  packet* x_pkt;
  ...
}
```

```
[ File XTPpacket.h - SandiaXTP-1.5 ]
```

Thesis 2: class XTP packet is a subclass of MTL packet class

> "...the XTPpacket is derived from the MTL base class packet, and FIRSTpacket, CNTLpacket, etc., are derived from XTPpacket" [SandiaXTP User's Guide, June 12, 1997, p.8]

> "Class XTPpacket is the protocol data unit for XTP. It is derived from the MTL base class packet ..." [SandiaXTP Reference Manual, SandiaXTP version 1.4, February 1996, XTPpacket(3)]

As shown on the daemon startup Class diagram (Figure 12), the modeling approach favored consists of giving priority to the source code and shows no inheritance association links between MTL class packet and class XTPpacket. It is presumed that class XTPpacket and its derivatives are used to cast a structure on the packet shell's field used for containing the packet data, which is mandatory for the parsing and interpretation of the packet.

Class **event_queue:** A protocol specific taxonomy of events is needed to drive the XTP state machine, which is designed for two data streams (one incoming and one outgoing). The role of the MTL class event_queue is to provide an infrastructure consisting of: (1) a 32-bit value available for the protocol specific encoding of events; (2) methods for enqueuing and dequeueing events, or for peeking without removing (respectively put(), pull() and peek()). The XTP implementation can instantiate two such event_queue objects. The encoding of XTP specific events is given in file XTPtypes.h (EX: EOM, WCLOSE, END, etc.).

Classes **state_machine & XTPstate_machine:** The MTL state_machine is a pure abstract class, i.e., an interface from which protocol-specific state machines are derived. The role of the XTPstate_machine subclass is to hold the state for the context and the association as defined in the XTP Context State Machine ([XTP40, p52]) and the XTP Association State Machine ([XTP40, p53]). It has many related methods, such as: (1) go_listening() to place a context in a listening state; (2) is_Quiescent() to test if a context is in Quiescent state; (3) is_OutStream_AssocClosed() to test if OutStream is in AssocClosed state or not; (4) trans_on_rcvd(event) to perform a state transition after a receive operation in *event.*

Class **buffer_manager:** Recall that shared memory is used for moving the data between a client and the daemon's context. This shared memory area comes in pairs of segments; one being used for sending and another used for receiving. A different buffer manager manages each. Class buffer_manager has a method to create one shared memory area, which is used only twice per context, i.e., once to create the send area and once to create the receive area. The creation task is triggered from class mtldaemon following a registration request received from the client. Through some IPC mechanism, the daemon can return the share memory id (shmid) to the client that created its own two reciprocal buffer managers (the mtlif class is seen later when tracing a client startup) and attach to the same shared memory areas. Class buffer_manager has several methods for managing data pointers in the shared memory area and used to move the data such as: (1) read() to copy *len* bytes of data from the shared memory segment to a user buffer; (2) write() to copy *len* bytes of data from a user buffer into the attached shared memory segment.

Class **user_request & subclasses:** Within the Sandia user level implementation architecture, there is a need for interprocess communications between the daemon and client processes. The Operating System supplies the IPC mechanisms, but cannot define the formats or the semantics of the messages. Class user_request is a base class for protocol specific user requests. SandiaXTP derives three subclasses as follows: (1) subclass **xtp_req_msg** which is used for registering with the daemon. Subsequent actions include the creation of two shared memory segments (if necessary), attachment by the daemon to these segments, and the handling of many other options, such as rate control, etc.; (2) subclass **xtp_trans_msg** used for the many regular transactions between the daemon and client such as SEND, RECEIVE, etc.; (3) subclass **xtp_state_msg** for change of traffic specifications.

Class **dds_address and subclasses:** The use of a DDS entails the preparation of corresponding address structures containing, for instance, one's own port number and network address as well as the port number and network address of the correspondent receiving context. Class **dds_address** is a pure virtual class that serves as an interface for the implementation of its virtual functions by derived classes, such as **ip_dds_address** (when raw IP is used as the DDS) and **udp_dds_address** (when UDP is used as DDS). The constructor instantiates a DDS specific address structure, fills it with blanks, automatically sets the address family (here to AF_INET) and the default daemon port number (to DAEMON_PORT, i.e., 2795). Method put_hostid() is used to put the host identifier (i.e., the 32-bit netid/hostid) into the address structure; method get_hostid() returns a pointer to the host identifier.

Class **packet_fifo:** Two per-context packet FIFO queues are needed for the temporary storage of: (1) incoming packets from the DDS, after the context manager has found a receiving context, but before that context can further handle the packets; (2) outgoing packets towards the DDS, after the context has built the packet, but before it can be passed to the DDS. Class **packet_fifo** manages such a FIFO queue of packets awaiting processing. The contructor returns two pointers of type packet (head=tail=(packet*)NULL). Method put() enqueues a packet at the rear of the FIFO structure; method get() dequeues a packet from the head.

4.3.2 Daemon startup - Object diagram

Figure 13 shows the instance diagram corresponding to the daemon startup Class diagram shown on Figure 12. As can be inferred from the **keymap** appearing on Figure 13, the instance diagram fuses a hierarchy of classes into a single object instance. The term "hierarchy" is defined here to mean a base class and its derived classes. Illustratively, while the **keymap** on Figure 12 shows two class rectangles MTL←—SXTP, the **keymap** on Figure 13 shows only one rounded edge rectangle labeled **xtpd**. This pattern is repeated throughout the whole instance diagram, as explained further in the following paragraphs.

Here follow some specific observations explaining how the daemon startup Object diagram contributes to a better understanding of the SandiaXTP software.

DAEMON: On the class diagram, the relationship between class mtldaemon and class context_manager is expressed with a line linking the rectangles and a label (**has**).

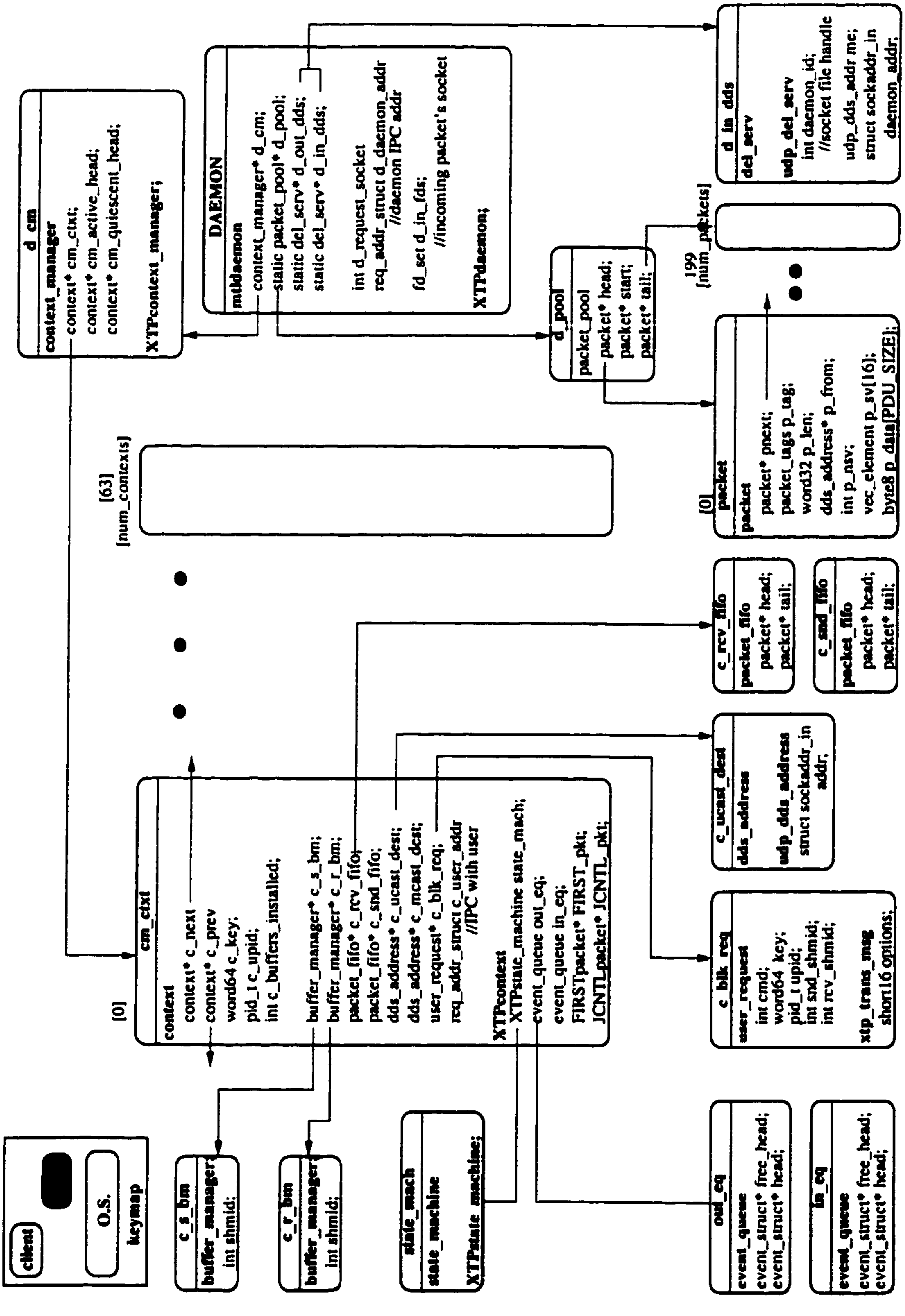


Figure 13: SandiaXTP Daemon startup - Object diagram

The semantics of this notation was defined as "daemon has a context_manager". The Object diagram reveals the exact programmatic nature of this link, in the form of a pointer (context_manager* d_cm) to object context_manager owned by DAEMON. Here, the arrow represents "a pointer to". The same explanation applies to packet_pool and the DDS.

Delivery service (d_in_dds): The daemon is designed to have two distinct data delivery services, one for incoming packets and one for outgoing data packets. The Object diagram shows that the two are actually merged (a socket offering duplex exchanges) and one single object instance gets created.

packet & packet_pool: The organization of the packet_pool, with pointers to packet objects for list management, is made more explicit with the Object diagram. An array of packet shells objects get created immediately at daemon startup, with default size of 200 packet shells. The structure of a packet shell is also made more evident, which could be used as a monolith or for scatter-gather I/O, depending on the p_tag flag.

context_manager: The setup of the context_manager, with pointers to first context object (context[0]), and pointers of context type for managing the active and quiescent lists of contexts is made more explicit with the Object diagram.

context & XTPcontext: An array of context object instances gets created immediately (by the context_manager) at the daemon startup, having default size of 64 contexts all ready to be assigned. The links with most satellite object instances, such as buffer_manager, are made more explicit with pointers. In contrast, the state_mach, out_eq and in_eq are *embedded within* a context object, rather than being created as separated objects, with context having pointers to them as it is done for the other satellite objects.

buffer_manager (c_s_bm & c_r_bm): The Object diagram clearly shows that two buffer_manager object instances get created per context, one for managing the send and one for managing the receive shared memory segments.

4.3.3 Daemon startup - Event Trace

Figure 14 and Figure 15 show an Event trace for the initialization of the SandiaXTP daemon, and consists of the sequence of method calls made between objects in response to starting the daemon by the user (xtpd -d udp). This Event trace is split in two parts, appearing on different report size sheets for reason of space. Markers are given to link the two parts (Ex: (1) on part 1 - Figure 14 continues to (1) on part 2 - Figure 15). The interested reader might decide to make copies and paste the two parts together before continuing.

Some observations are now presented regarding the semantics conveyed by the daemon startup Event trace diagram. When starting the daemon program, the user can set a number of switches through command line arguments. For instance, the user can specify which DDS to use (UDP or raw IP), the number of contexts or of packet shells to instantiate. For the Event trace, UDP is presumed to be the DDS selected, letting the default values apply for all other options. The daemon program would then be started with command line **xtpd -d udp** as shown at the top-left portion of Figure 14.

Internally, a driver program including the C++ main() function (file xtp.C) is used to compile the daemon process, with all major later calls being made from main() (i.e., DAEMON→parse_args(), DAEMON→init_daemon(...), DAEMON→main_loop()). Right at the onset, a daemon object instance gets created, with global variable **DAEMON** being a pointer to this object.

The next call made from main() is to parse the command line arguments given by the user (DAEMON→parse_args()). The program might terminate at this point if parsing reveals errors. Otherwise, much will happen before the next call (DAEMON→init_daemon(...)) made from main() returns. The role of function daemon_start() is to convert the user process into a Unix daemon process that can survive after the initiating user logs out.

The second object being created consists of the data delivery service (dds_in_srv = new ...). As UDP is being used, as datagram type of socket (SOCK_DGRAM) is opened with the operating system, and added to the set of file descriptors for even-

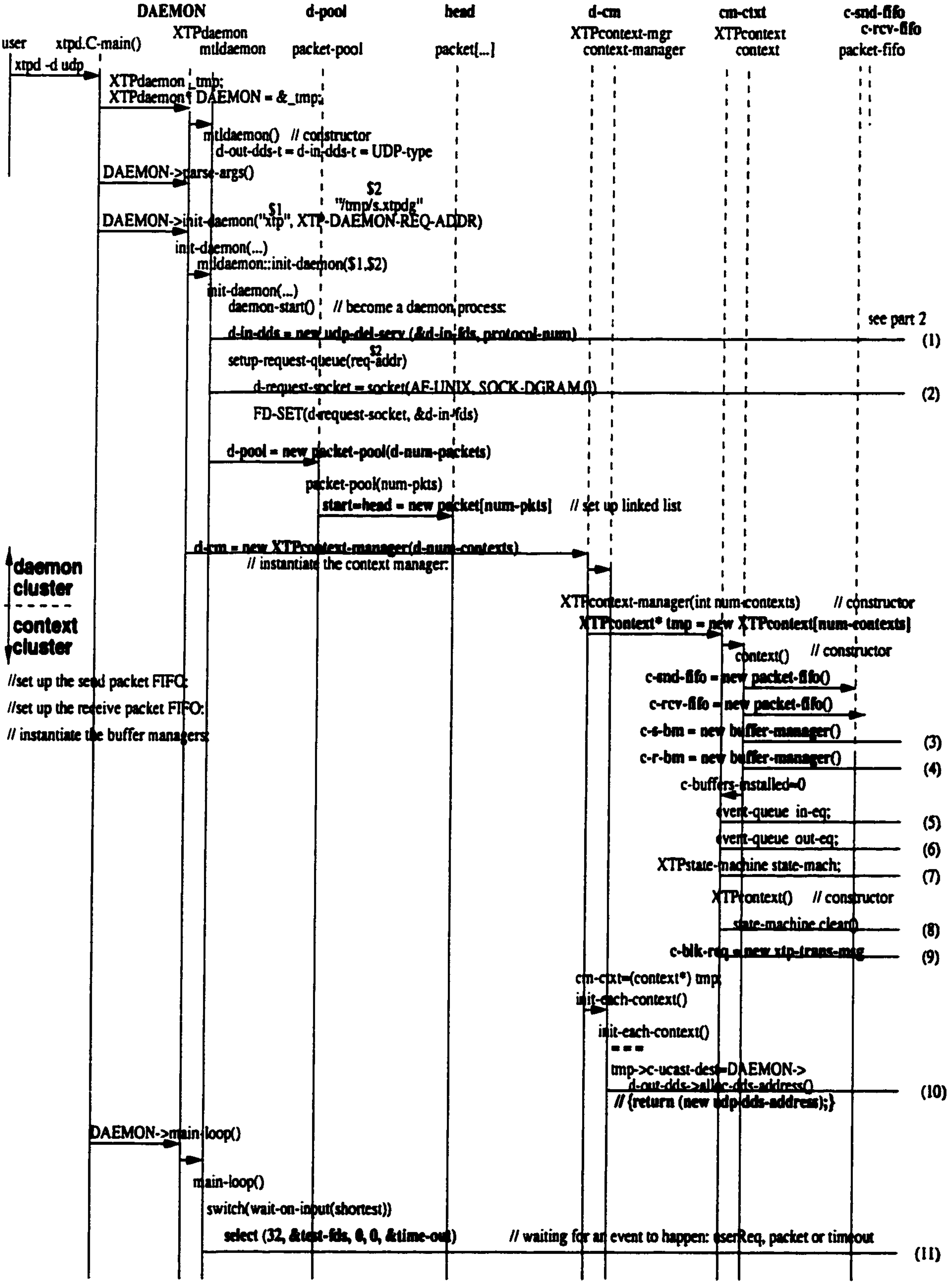

DAEMON
d-pool
head
d-cm
cm-ctxt
c-snd-fifo
c-rcv-fifo
user
xtpd.C-main()
XTPdaemon
mtldaemon
packet-pool
packet[...]
XTPcontext-mgr
context-manager
XTPcontext
context
packet-fifo
xtpd -d udp
XTPdaemon _tmp;
XTPdaemon* DAEMON = &_tmp;
mtldaemon() // constructor
d-out-dds-t = d-in-dds-t = UDP-type
DAEMON->parse-args()
$2
"/tmp/s.xtpdg"
$1
DAEMON->init-daemon("xtp", XTP-DAEMON-REQ-ADDR)
init-daemon(...)
mtldaemon::init-daemon($1,$2)
init-daemon(...)
daemon-start() // become a daemon process:
see part 2
d-in-dds = new udp-dd-serv (&d-in-fds, protocol-num)
(1)
$2
setup-request-queue(req-addr)
d-request-socket = socket(AF-UNIX, SOCK-DGRAM,0)
(2)
FD-SET(d-request-socket, &d-in-fds)
d-pool = new packet-pool(d-num-packets)
packet-pool(num-pkts)
start=head = new packet[num-pkts]
// set up linked list
d-cm = new XTPcontext-manager(d-num-contexts)
// instantiate the context manager:
daemon cluster
context cluster
XTPcontext-manager(int num-contexts)
// constructor
XTPcontext* tmp = new XTPcontext[num-contexts]
context()
// constructor
//set up the send packet FIFO:
//set up the receive packet FIFO:
// instantiate the buffer managers:
c-snd-fifo = new packet-fifo()
c-rcv-fifo = new packet-fifo()
c-s-bm = new buffer-manager()
(3)
c-r-bm = new buffer-manager()
(4)
c-buffers-installed=0
event-queue in-eq;
(5)
event-queue out-eq;
(6)
XTPstate-machine state-mach;
(7)
XTPcontext() // constructor
state-machine clear()
(8)
c-blk-req = new xtp-trans-msg
(9)
cm-cxt=(context*) tmp;
init-each-context()
init-each-context()
= = =
tmp->c-ucast-dest=DAEMON->
d-out-dds->alloc-dds-address()
(10)
// {return (new udp-dds-address);}
DAEMON->main-loop()
main-loop()
switch(wait-on-input(shortest))
select (32, &test-fds, 0, 0, &time-out)
// waiting for an event to happen: userReq, packet or timeout
(11)

Figure 14: Sandia XTPdaemon startup dynamic model (part 1)

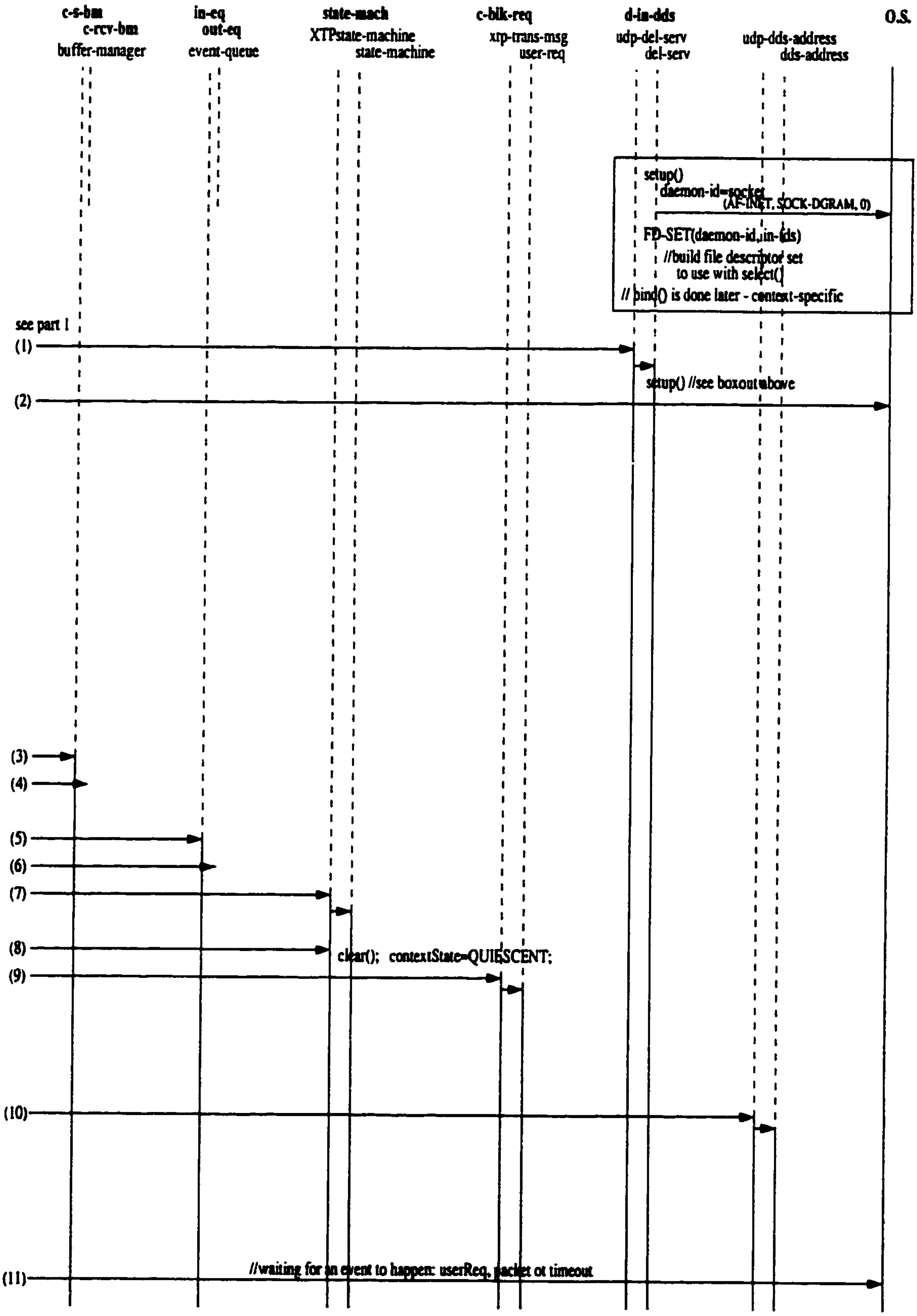


Figure 15: Sandia XTPdaemon startup dynamic model (part 2)

tual use with the select() system call. The long arrow, all the way to the rightmost vertical line of Figure 15 with label **O.S.** illustrates the opening of the socket with the Operating System.

Next, the DAEMON object opens an internal Unix socket (family AF_UNIX), of type datagram, with the Operating system (d_request_socket = socket(AF_UNIX, SOCK_DGRAM,0), which is also added to the FD_SET for eventual use with the select() system call. Through select(), the daemon is effectively requesting the O.S. to be notified whenever something happens on those two sockets. The path name used to fill up the internal address structure is "/tmp/s.xtpd". The communication path with the client is now opened, and clients can specify this pathname when sending requests to the daemon. There is only one such Unix domain socket to streamline all client requests, which will thus be processed by the daemon on a FIFO basis. Note that the UNIX datagram facility does **not** provide flow control.

Next, still executing mtldeamon::init_daemon() code, the packet pool object gets created (d_pool = new packet_pool(...)). In turn, when the packet_pool object initializes itself (constructor), it triggers the creation of an array of two hundred
(start=head = new apcket[num_pkts]) packet objects.

Subsequently, the last satellite object instance of the daemon cluster (show by a marker on the left margin of Figure 14) is created, namely the context_manager object (d_cm = new XTPcontext_manager(...)). The creation of this object entails the creation of all the other objects that belong to the context cluster. More precisely, the creation of the XTPcontext_manager (constructor) entails the subsequent creation of: (1) the two packet_fifo objects (c_snd_fifo, c_rcv_fifo); (2) the two buffer_manager objects (c_s_bm & c_r_bm); (3) the two event_queue objects (in_eq & out_eq); (4) the state machine object (state_mach); (5) a transaction user request message (c_blk_req), which will allow the daemon to receive the first incoming user request from clients. Terminating this object creation frenezy, function init_each_context() is called to initialize each context and control is finally returned to main().

The last call made from main(), i.e., DAEMON→main_loop(), results in the execution of the main loop code that belongs to the mtldaemon base class. The daemon stalls on executing the select() system call, effectively relinguishing control to the operating system. Because of the former measures taken with FD_SET, the daemon will be awakened by the operating system, on the earliest occurence of one the three following events: (1) an incoming packet; (2) an incoming user request; or (3) a clock timeout if none of the preceeding events had occured. The timeout is necessary to ensure the proper long term functionality of the daemon (not taking into consideration preventing being swapped out from memory). Each time it occurs, the daemon does a systematic check on pending work to be done such as: (1) checking XTP timers for eventual retransmissions; (2) sending a new burst of packets subject to rate control; etc.

All the object instances shown on the daemon startup Object diagram have been created and the daemon process is fully functional.

4.4 SandiaXTP Client with O.M.T.

In principle, clients running at each host could simultaneously manage one outgoing and one incoming data stream, with each client assuming alternatively the role of sender and receiver. To simplify the preparation and the presentation of the Client's Object Model (O.M.), a clear cut division of responsibilities is presumed, with one client acting as the sender and the other one acting as the receiver (unicast mode). For the Event trace at least, the emphasis is also from the receiver viewpoint, as starting a receiver is the next logical step to perform once a daemon is up and running.

The details of the model have been inferred from tracing and studying the source code, using program bulk (file bulk.C and common.h) as the starting point. Program bulk, which is one of the several examples of network application client programs included with the SandiaXTP distribution, is meant to illustrate bulk data transfer. It supports unicast as well as multicast.

Recall also that there is a need for an Application Programming Interface (API) between the client and the running instance of XTP for a user level implementation of XTP (see Figure 10). Base class mtlif and the xtpif subclass embody this API for the SandiaXTP implementation. Consequently, the model building path is straightforward: from program bulk, follow the calls made to mtlif/xtpif methods(); which lead to the SandiaXTP source code, and eventually infer a meaningful Object Model (Event trace, Object diagram and Class diagram) for the client.

4.4.1 Client-Receiver startup - Class diagram

The fundamental entity representing the Client consists of some application program, which has to marshal the services of the SandiaXTP daemon for performing some useful task for the user, such as reliable file transfer. However, in order to marshal the services of the daemon successfully, the programming constructs used in the application program would have to follow the rules of a well defined Application Programming Interface (API).

Globally, the Class diagram shown on Figure 16 illustrates the class organization of one client of the SandiaXTP daemon, with the pair mtlif baseclass/xtpif subclass providing the functionalities of the API (call it the *interface*). The lib_manager is a *per-process* entity. Only one instance of the lib_manager gets created (the role of each class is better described later) per application program. The mtlif constructor calls the lib_manager initialize() method, which effectively executes itself only once because of some internal flag that gets set the first time it is called.

Though this is not necessarily frequent, there could exist many instances (1+ notation on the Class diagram) of the interface (mtlif/xtpif) created from the same application program. Recall (from Section **XTP - A system architecture view**) that XTP models each endpoint of a communication as a different *context*, and there could be many contexts per client, with each one possibly represented at the client's level by one interface object instance.

Each new interface instance entails the creation of two buffer managers; one to manage the send shared memory segment, and another one to manage the receive shared memory segment. The daemon also has reciprocal buffer managers for managing the same shared memory segments on its side. For each interface, some user request ob-

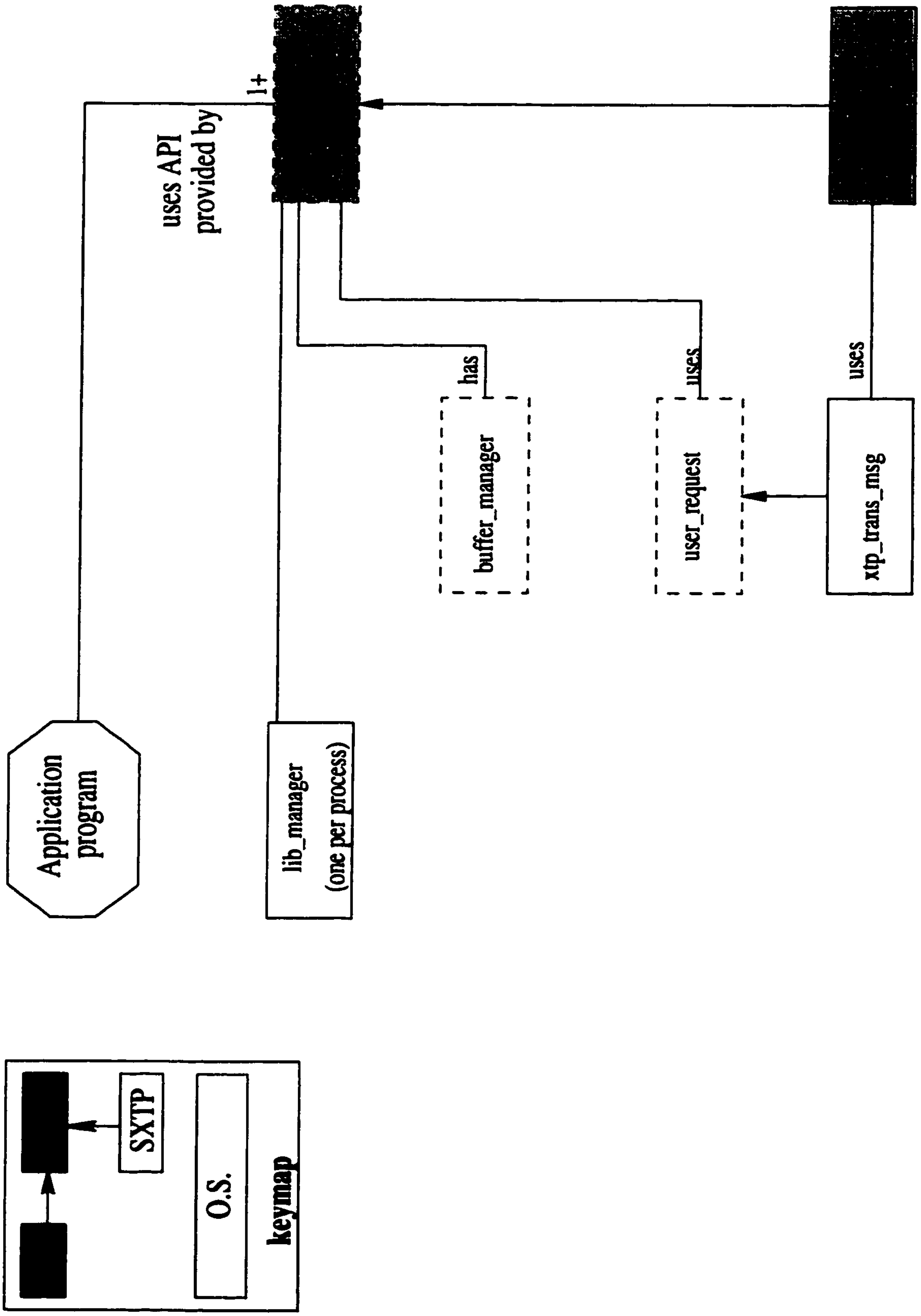


Figure 16: Receiver startup - Class diagram

jects also get created.

The creation of an interface object by the application program does not necessarily imply a *legitimate* communication endpoint. Some contacts with the daemon are needed for this purpose. After creating an interface object, the application program can: (1) register with the daemon, i.e., communicate with the daemon for the assignment of a context corresponding to the interface object, create the shared memory segments, and get back from the daemon the *key* and the shared memory identification numbers; (2) bind an address with the daemon, thereby allowing the match of a FIRST packet with a listening context; (3) communicate with the daemon to get/set the traffic specifications; (4) again communicate with the daemon for entering the listening states; (5) communicate with the daemon to receive or send data. The application need not step through all the phases for all the interface objects it creates.

Classes **mtlif & xtpif:** Basically, the role of the mtlif baseclass is twofold: (1) to provide the necessary infrastructure to allow clients to contact the daemon and receive replies; (2) indirectly, through its associate buffer_manager class, to manage the shared memory segments on the client's side.

As part of its attributes, class mtlif has the data members that act as global variables for its class (`static`) such as: (a) a pointer to the first mtlif created from the same application program; (b) a pointer to the last interface object created; (c) the interface socket descriptor for sending requests to the daemon; (d) one address structure needed by all interface instances to get replies from the daemon; (e) one address structure needed when sending requests to the daemon. As part of its other attributes (i.e., the ones particular to each interface object), class mtlif also has: pointers to the next and the previous interface object, if any; the process identification number; two embedded buffer managers for managing the two shared memory segments.

Class mtlif has methods for communicating with the daemon, such as issue(), send_to_daemon() and recv_from_daemon(). It also has some virtual methods such as reg() (to register with the daemon) and release(), to be reimplemented the protocol-specific implementation.

Subclass **xtpif** adds few attributes but many XTP specific methods to the mtlif baseclass. Some of the attributes added by XTP include: (1) the amount of data unread waiting in the receive shared memory segment; (2) a configuration structure needed for passing information to the daemon; (3) one embedded XTP transaction user request (in fact a data structure); (4) a structure to hold traffic segment fields.

Some of the methods added by the xtpif subclass include: (a) config(), which effectively copies some XTP configuration parameters given as command line arguments by the user into the configuration structure known by the interface (i.e., item (2) of the preceding paragraph - the original one contained all default values); (b) reg(), to register with the daemon, i.e., get back from the daemon a context identifier (*key*) and the shared memory identifiers; (c) bind(), to bind one address with the daemon for the purpose of FIRST packet context matching; (d) overloaded send() and receive(), with one version for regular I/O and another version for scatter-gather I/O, for sending and receiving data.

Class **buffer_manager:** This MTL class is described in Section **Daemon startup - Class diagram.**

Class **lib_manager:** Class lib_manager is not really part of the class design. Its declaration and definition, normally in separate files (declaration.h and definition.C), are included in the mtlif definition file (file mtlif.C). The declaration is made as follows: `class lib_manager{...} _lm`. Class lib_manager has only one attribute, a flag (int ready), which is set when the lib_manager::initialize(...) method is called the first time by the mtlif() constructor.

Class lib_manager has only two methods, namely initialize() and cleanup(). The purposes of method initialize() are manifold: (1) build two address structures, one to target the daemon when issuing user requests, the other to be reached by the daemon (call it interface address); (2) open an interface socket, and bind with the O.S. the interface address on this socket. This *one* IPC communication channel is valid for all interface objects created from this application program/process. The application program does the demultiplexing automatically by making API calls from a particular interface object; (3) activate the interrupt handlers.

Method initialize() is called by the mtlif() constructor. Therefore, it is called each time one interface object is created, but it is effectively executed in its entirety only once (the first time) because of the ready flag (initialize(...) if(ready) return(1) ...). Method cleanup() is called by the ~lib_manager() destructor to release the interface objects and the interface socket.

Classes **user_request & xtp_trans_msg:** A client formulates requests and the daemon responds. A data structure is needed to encode the contents of any message exchanged between the client and the daemon. Base class user_request provides a common header for all types of user requests, with fields such as key and shared memory identification numbers. Subclasses xtp_reg_msg, xtp_state_msg and xtp_trans_msg provide protocol-specific user requests for respectively containing registration information, traffic specification information for instance, and regular transaction information with regards to the movements of the data to/from the shared memory segments.

Class mtlif provides a handle to base class user_request so that header fields are accessible with protocol-specific user requests. The mtlif class declaration (mtlif.h) embeds only the xtp_trans_msg user request, probably because this particular one is needed throughout the lifetime of the client process. Other short lived user requests, such as xtp_reg_msg and xtp_state_msg are created for specific purposes within the scope of a particular API method call.

4.4.2 Client-Receiver startup - Object diagram

Figure Figure 17 shows the instance diagram corresponding to the Receiver startup Class diagram (Figure 16). Here follows some specific observations explaining how the Client startup Object diagram contributes to a better understanding of the SandiaXTP software.

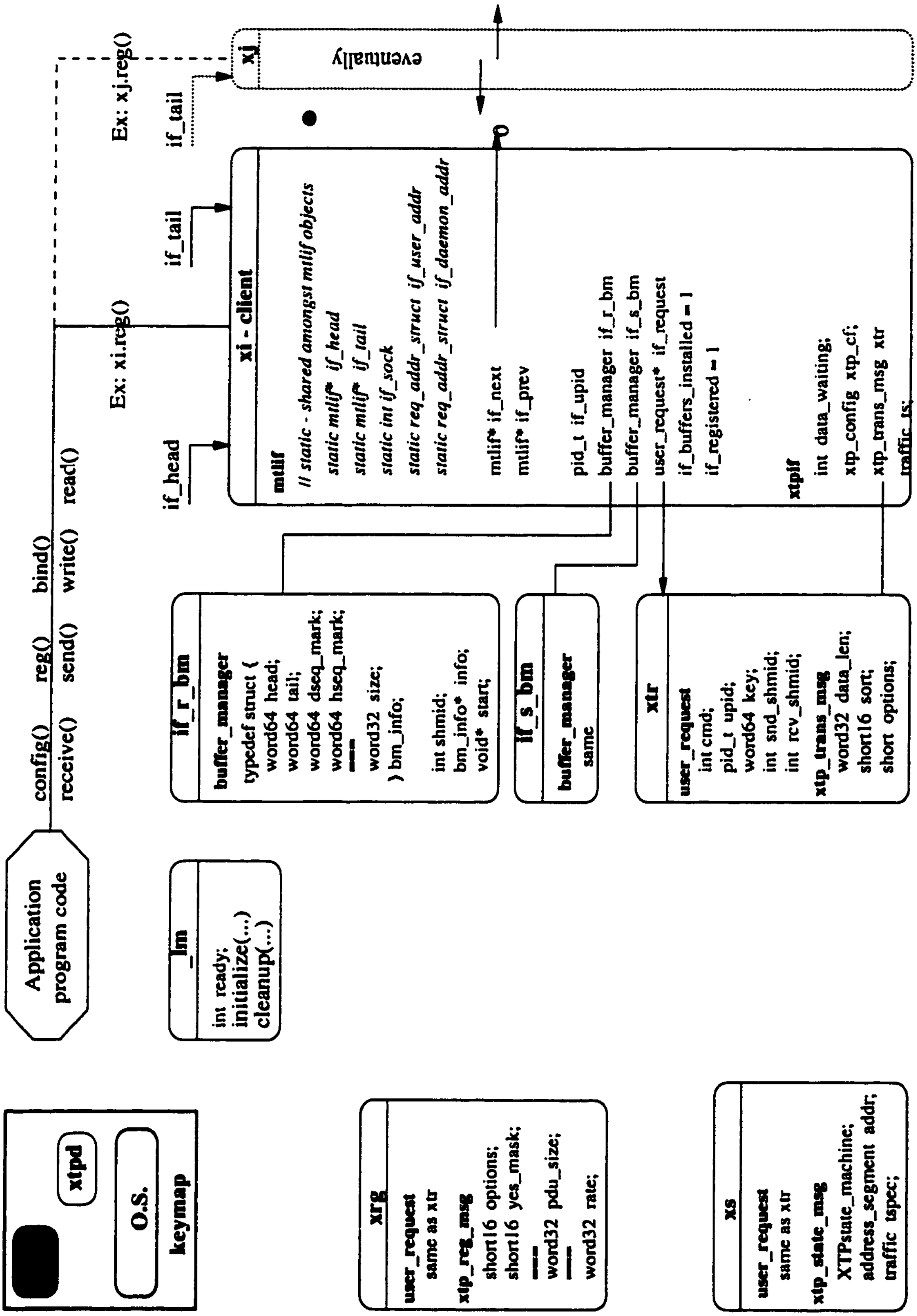


Figure 17: Receiver startup - Object diagram

_lm (lib_manager): The Object diagram shows clearly that there is *only one* lib_manager object instance per application program, no matter how many interface objects get instantiated from this application program. No programming link is shown with other objects, as the lib manager object exists independently of the other objects. The lib_manager object is created as part of the compilation process (see description of Class **lib_manager**); its method initialize() is called by the mtlif() constructor; the destructor (~lib_manager() cleanup(...);) calls its own method cleanup(), which in turn calls the mtldaemon's shutdown() method.

xi - interface: The object diagram illustrates clearly the organization to the interface: (1) with new interface instances inserted at the tail of a linked list; (2) with *per-interface-instance* embedded objects such as the two buffer managers (**if_r_bm, if_s_bm**) object instances and one xtp transaction user request object instance. The mtlif pointer **if_request** provides access to common header fields such as *key*.

xrg & xs: The Receiver startup Object diagram also shows separately one xtp registration user request (**xrg**) and one xtp state user request (**xs**) object instances, which can be used respectively for registering a context with the daemon and obtaining/changing a traffic specification.

4.4.3 Client-Receiver startup - Event Trace

When staring program bulk, the user has to specify (through command line arguments) all the options defining the role of the client and the details of the data communication task. Some of these arguments are used by the client program (bulk) to determine which mtlif/xtpif method() to call, whilst many others are eventually conveyed to the daemon. Examples of such command line arguments include: (1) the role of the client, be it sender or receiver; (2) the mode, unicast or multicast (SandiaXTP examples are built around the convention that a lower case letter defining the role implies unicast whilst an upper case letter implies multicast (Ex: -r would mean a unicast receiver, -R would mean a multicast receiver); (3) the output rate; (4) etc.

As the emphasis of this subsection is neither the API nor a comprehensive setting of the XTP parameters, a minimal set of illustrative command line arguments is

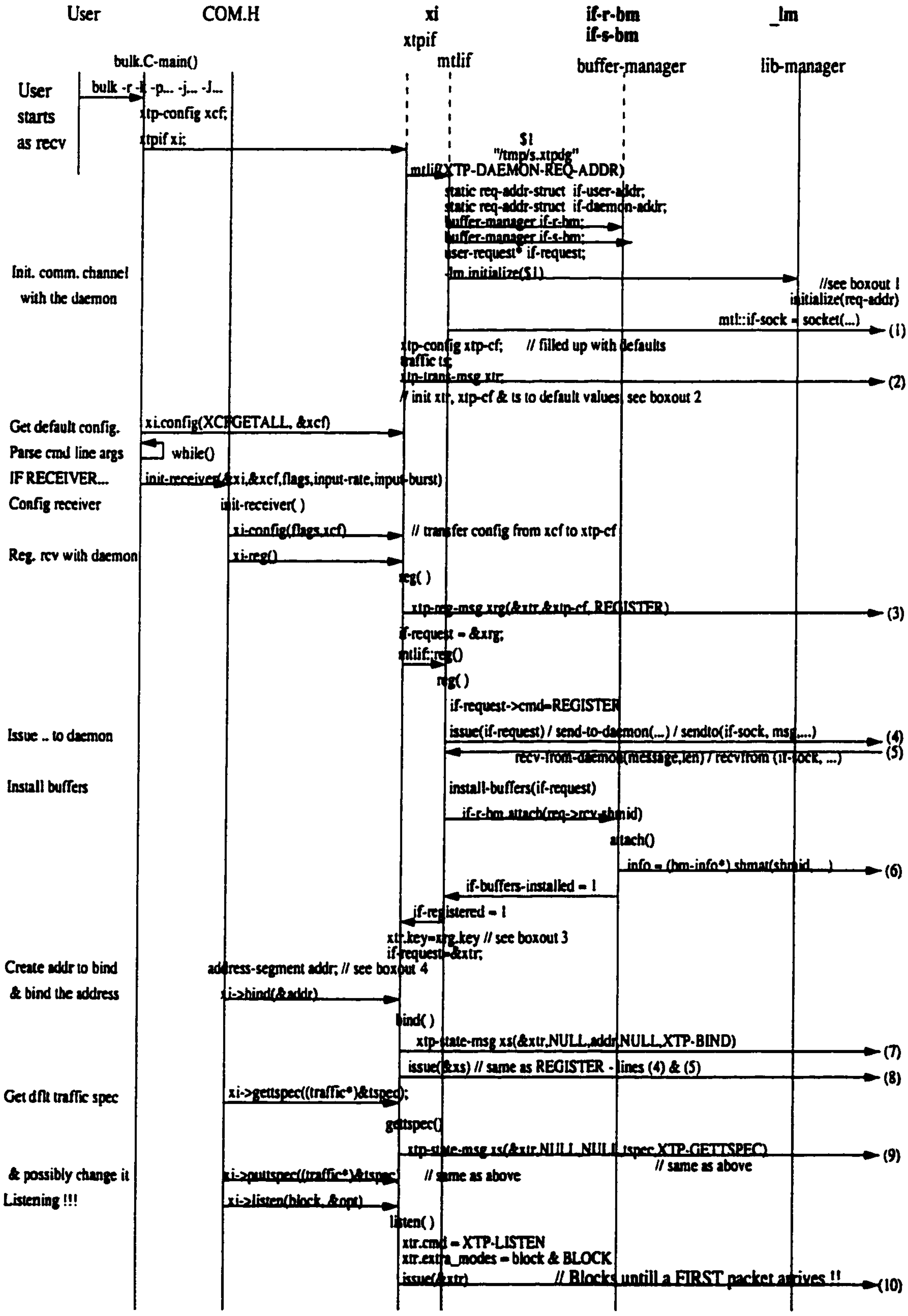
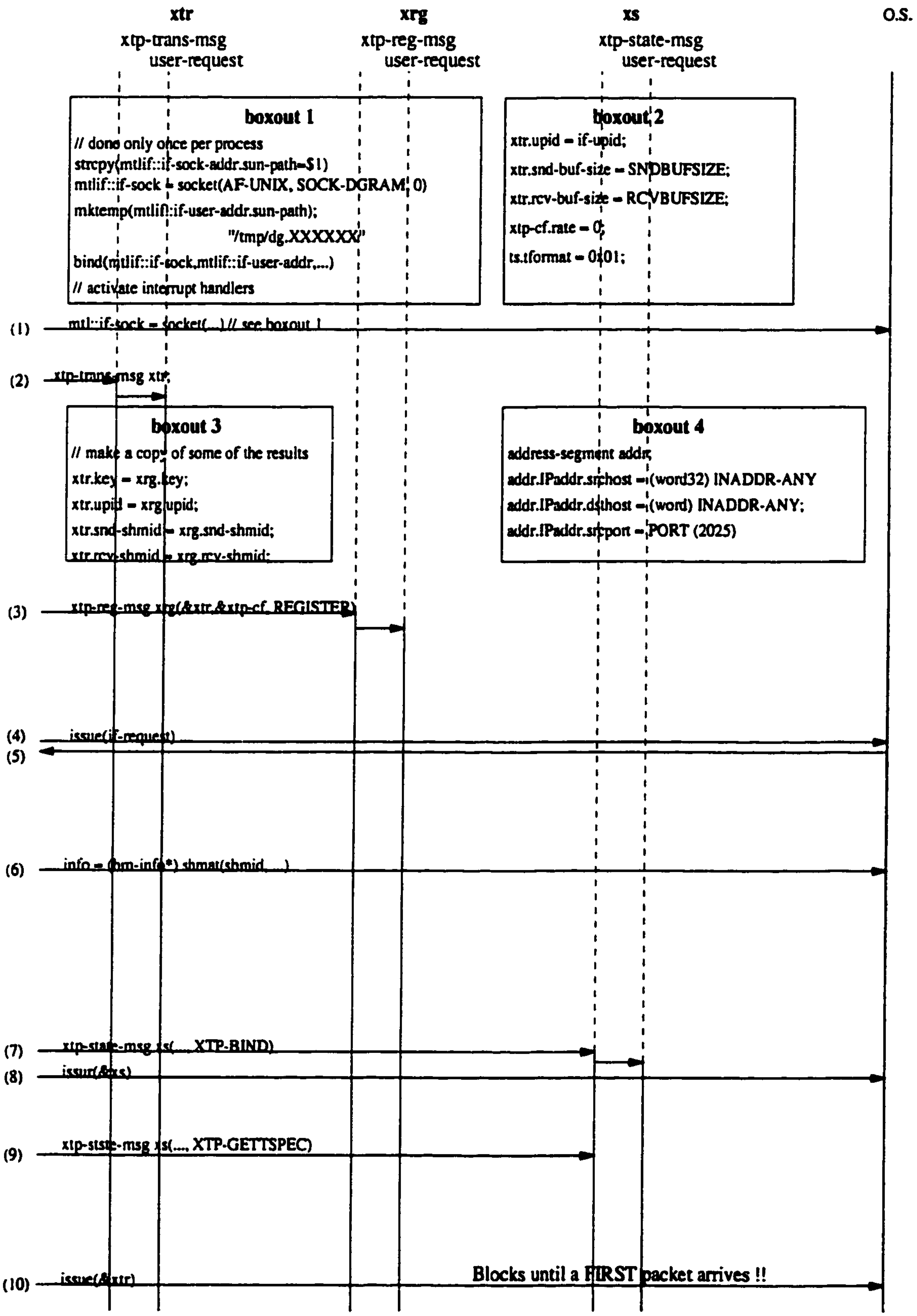

User
COM.H
xi
xtpif
mtlif
if-r-bm
if-s-bm
buffer-manager
_lm
lib-manager
bulk.C-main()
User
starts
as recv
bulk -r -l -p... -j... -J...
xtp-config xcf;
xtpif xi;
$1
"/tmp/s.xtpdg"
mtlif(XTP-DAEMON-REQ-ADDR)
static req-addr-struct if-user-addr;
static req-addr-struct if-daemon-addr;
buffer-manager if-r-bm;
buffer-manager if-s-bm;
user-request* if-request;
Init. comm. channel
with the daemon
lm.initialize($1)
//see boxout 1
initialize(req-addr)
mtl::if-sock = socket(...)
(1)
xtp-config xtp-cf; // filled up with defaults
traffic ts;
xtp-trans-msg xtr;
(2)
// init xtr, xtp-cf & ts to default values, see boxout 2
Get default config.
xi.config(XCFGETALL, &xcf)
Parse cmd line args
while()
IF RECEIVER...
init-receiver(&xi,&xcf,flags,input-rate,input-burst)
Config receiver
init-receiver( )
xi-config(flags,xcf)
// transfer config from xcf to xtp-cf
Reg. rcv with daemon
xi-reg()
reg( )
xtp-reg-msg xrg(&xtr,&xtp-cf, REGISTER)
(3)
if-request = &xrg;
mtlif::reg()
reg( )
if-request->cmd=REGISTER
Issue .. to daemon
issue(if-request) / send-to-daemon(...) / sendto(if-sock, msg,...)
(4)
(5)
recv-from-daemon(message,len) / recvfrom (if-sock, ...)
Install buffers
install-buffers(if-request)
if-r-bm.attach(req->rcv-shmid)
attach()
info = (bm-info*) shmat(shmid, ...)
(6)
if-buffers-installed = 1
if-registered = 1
xtr.key=xrg.key // see boxout 3
if-request=&xtr;
Create addr to bind
& bind the address
address-segment addr; // see boxout 4
xi->bind(&addr)
bind( )
xtp-state-msg xs(&xtr,NULL,addr,NULL,XTP-BIND)
(7)
issue(&xs) // same as REGISTER - lines (4) & (5)
(8)
Get dflt traffic spec
xi->gettspec((traffic*)&tspec);
gettspec()
xtp-state-msg xs(&xtr,NULL,NULL,tspec,XTP-GETTSPEC)
(9)
// same as above
& possibly change it
xi->puttspec((traffic*)&tspec)
// same as above
Listening !!!
xi->listen(block, &opt)
listen( )
xtr.cmd = XTP-LISTEN
xtr.extra_modes = block & BLOCK
issue(&xtr) // Blocks untill a FIRST packet arrives !!
(10)

Figure 18: Client-receiver startup Event trace - part 1


xtr
xtp-trans-msg
user-request
xrg
xtp-reg-msg
user-request
xs
xtp-state-msg
user-request
O.S.
(1) mtl::if-sock = socket(...) // see boxout 1
(2) xtp-trans-msg xtr;
(3) xtp-reg-msg xrg(&xtr,&xtp-cf, REGISTER)
(4) issue(if-request)
(5)
(6) info = (hm-info*) shmat(shmid, ...)
(7) xtp-state-msg xs(..., XTP-BIND)
(8) issur(&xs)
(9) xtp-stste-msg xs(..., XTP-GETTSPEC)
(10) issue(&xtr)
Blocks until a FIRST packet arrives !!

boxout 1

```
// done only once per process
strcpy(mtlif::if-sock-addr.sun-path=$1)
mtlif::if-sock = socket(AF-UNIX, SOCK-DGRAM, 0)
mktemp(mtlif::if-user-addr.sun-path);
                    "/tmp/dg.XXXXXX"
bind(mtlif::if-sock,mtlif::if-user-addr,...)
// activate interrupt handlers
```

boxout 2

```
xtr.upid = if-upid;
xtr.snd-buf-size = SNDBUFSIZE;
xtr.rcv-buf-size = RCVBUFSIZE;
xtp-cf.rate = 0;
ts.tformat = 0x01;
```

boxout 3

```
// make a copy of some of the results
xtr.key = xrg.key;
xtr.upid = xrg.upid;
xtr.snd-shmid = xrg.snd-shmid;
xtr.rcv-shmid = xrg.rcv-shmid;
```

boxout 4

```
address-segment addr;
addr.IPaddr.srchost = (word32) INADDR-ANY
addr.IPaddr.dsthost = (word) INADDR-ANY;
addr.IPaddr.srcport = PORT (2025)
```

Figure 19: Client-receiver startup Event trace - part 2

presumed, which consists of the following: (1) client is to act as a unicast receiver (hypothetical option label **-r**); (2) Client is to block (BLOCK) until the arrival of matching FIRST packet (hypothetical option label: **-k**); (3) user sets the protocol data unit size (PDU) (hypothetical option format: **-p** ...); (4) user sets the incoming data rate (hypothetical option format: **-j** ...); (5) user sets the incoming burst value (hypothetical option format:**-J** ...).

The -r argument is used by the client (bulk) program to determine which interface method to call, such as receive(). The other arguments (-k, -p, -j, -J) are eventually conveyed to the daemon via arguments to interface methods called from main(). Hence, the hypothetical client would be started with the following command:

```
bulk -r -k -p ... -j ... -J ...
```

The following paragraphs make reference to the Receiver startup Event trace shown in Figure 18 and Figure 19. The upper-left arrow with label **bulk -r -k -p ... -j ... -J ...** represents the initial action of the user starting program bulk.

Create the interface (xi) and the buffer manager objects (if_r_bm & if_s_bm): The driver program (main()) uses the **xcf** configuration structure (xtp_config is defined in file XTPtypes.h) to store various options typed by the user having implications for the daemon. The xcf structure is filled up later when parsing the command line arguments. At the beginning of main(), the declaration **xtpif xi** also triggers the creation of the **xi** interface object, which is a compound of MTL baseclass mtlif with the xtpif subclass. The xtpif() subclass constructor immediately calls the mtlif() base class constructor, passing as parameter the local address for sending requests to the daemon. This call has the following form:
xtpif::xtpif():mtlif(XTP_DAEMON_REQ_ADDR). The local address of the daemon is defined in file XTPtypes.h (i.e., `''/tmp/s.xtpdg''`). Recall that this same address was used by the daemon when setting its request queue (see Figure 14, line (2) ff).

The mtlif object uses two address structures: (1) **if_daemon_addr** to store the local address of the daemon; (2) **if_user_address** to store the local address of the client. The function of these two data structures will become evident shortly. The mtlif constructor also creates two buffer managers for this client (**if_r_bm & if_s_bm**). These will manage respectively the receive and the send shared memory segments for the client. The daemon has its own reciprocal buffer managers. Even if the client is to act exclusively as receiver, the daemon still creates two shared memory segments. On the client's side, at least two buffer managers are created by the mtlif object.

Initialize communication channels needed to communicate with the daemon: A call is then made from the mtlif() constructor to the lib_manager for initializing the communication channels needed by the client to communicate with the daemon (_lm.initialize($1) on Figure 18). Such a call is made only once per process when the mtlif() constructor is first called, no matter how many interface objects are later created from the same process.

Let us now focus on the details of the work done by the lib_manager, as shown on Figure 19, boxout 2. First of all, the daemon local path address string (‘‘/tmp/s.xtpdg’’) received as an argument gets copied into the if_daemon_addr structure (refer to beginning of boxout 1: strcpy(mtlif::if_daemon_addr.sun_path = $1)). Not shown on the diagram, the family field would also be filled in (i.e., mtlif::if_daemon_addr.sun_family = AF_UNIX). This completed if_daemon_addr structure is used later when the client needs to send requests to the daemon. It is passed as the *to* argument with the sendto(...,struct sockaddr *to,...) system call, which specifies the protocol-specific address of where the data are to be sent.

Next, it is also necessary to make sure that the client can be reached by the daemon. As shown on Figure 19, within boxout 1, the steps are as follows: (1) open a UNIX domain datagram socket with the Operating System, which is always acting as the middle person for the exchange of messages (mtlif::if_sock=socket(AF_UNIX, SOCK_DGRAM,0)); (2) fill up a local address structure:
(a) mtlif::if_user_addr.sun_family=AF_UNIX),
(b) strcpy(mtlif::if_user_addr.sun_path,‘‘/tmp/dg.XXXXX’’),
(c) mktemp(mtlif::if_user_addr.sun_path). The mktemp() system call generates a unique pathname within this host; (3) bind the local address with the operating system so that the client can be reached by other processes, i.e., by the daemon in

this case (bind(mtlif::if_sock,...)).

At this point, the client has performed all the necessary steps to be reached by the daemon. The links are to be done by the Operating System, which knows the address structure of the client because of the bind() system call. When receiving, the daemon uses the recvfrom(..., struct sockaddr **from*,...) system call, which fills in the protocol-specific address of who sent the data into *from*. The daemon can then use this *from* address structure as an argument when using the sendto() system call to return a reply to the local client.

Start building the data structures for contacting the daemon: Now, back to the mainstream of Figure 18. As the IPC mechanisms are in place, the focus becomes preparing the message contents for registering with the daemon. For this purpose, the xtpif object has three attributes: (1) **xtp_cf**, a configuration structure (xtp_config xtp_cf), which gets filled up with all the default values by the xtpif() constructor; (2) **ts**, a traffic structure (traffic ts), which also gets filled up with default values by the xtpif() constructor; (3) **xtr**, a xtp transaction structure (xtp_trans_msg xtr - arrow (2) on Figure 18). Sample initialization values used by the xtpif() constructor to fill up these three data structures are shown on Figure 19, boxout 2.

Parse the command line arguments: With regard to the handling of command line arguments, the logic of the program steps into a three-phase operation:

1. From main(), using the interface config() method, obtain a copy of the default configuration parameters as known to the interface object
(i.e., xi.config(XCFGETALL, &xcf) - which triggers a copy of xtp_cf into xcf);

2. Parse command line arguments, set some flags and possibly change some configuration parameters copied to the xcf structure controlled by main(). For instance: (a) the **-r** argument would result in the setting of some flag, such as is_recv=TRUE; (b) **-k** would result in changing the extra modes field (i.e., xcf.extra_modes |=BLOCK); (c) **-p** would result in xcf.pdu_size=atoi(optarg), flags|=XCFPDUSIZE; (d) **-j** would result in input_rate=atoi(optarg); (e) **-J** would result in input_burst=atoi(optarg). If some other command line argument were used, such as to set the output rate for a sender, then some field of

the configuration structure would be set accordingly (i.e., xcf.rate=atoi(optarg), flags|=XCFRATE).

3. From main(), return the modified configuration structure (xcf) to the interface object, which will eventually convey it to the daemon. This latter phase is greatly expanded in the Event trace diagram, as explained in the following paragraphs.

Given the **-r** option, call init_receiver(&xi,&xcf,flags) made from main() has many implications. Its arguments have the following purposes: (1) &xi, a memory address, is needed to specify which interface object is concerned; (b) &xcf, conveys the address of the configuration structure as known by main() after the parsing of command line arguments; (3) flags specify which fields of the configuration structure were changed, if any. The goal of the next call, xi_config(), is to update the xtp_cf interface configuration structure from the xcf structure provided by main(). So doing, control is passed to the xi interface for the next several calls.

Build the registration request object: Always keeping in mind the goal of establising the first contact with the daemon, the next significant call is made from the xtpif object, i.e., xtp_req_msg(&xtr,&xtp_cf,REGISTER). This call results in the creation of a xtp registration user request (i.e., a data structure), with all the necessary fields automatically filled up when the object gets instantiated, such as: (a) xrg.cmd=REGISTER; (b) xrg.extra_modes=xtp_cf.extra_modes;
(c) xrg.pdu_size=xtp_cf.pdu_size; (d) xrg.upid=xtr.upid. A pointer is set to this data structure (if_request=&xrg).

Contact the daemon: The stage is now ready for contacting the daemon, which is done from mtlif with the issue(if_request) method. Method issue(..) is defined as follows (file mtlif.h):

```
int issue(user_request* req) {
  return(((send_to_daemon(req, req->len) == EXOK) &&
     (recv_from_daemon(req) == EXOK)) ? EXOK : EXCOMM);
}
```

```
int send_to_daemon(void* message, int len) {
   return((sendto(if_sock, (char*)message, len, 0,
                (struct sockaddr*)&if_daemon_addr,
                if_servlen) != len) ? EXCOMM : EXOK);
}
```

```
int recv_from_daemon(void* message) {
   return((recvfrom(if_sock, (char*)message, MAX_MSG_SIZE, 0,
          (struct sockaddr*)0, &if_tmp) < 0) ? EXCOMM : EXOK);
}
```

Consequently, the client blocks at this point awaiting an answer from the daemon. The message is conveyed to the daemon using the if_socket, but providing the if_daemon_addr structure as the destination address argument. This exchange with the daemon, via the Operating System, is illustrated on Figure 18 and Figure 19 with lines (4) and (5) spanning both parts of the Event trace diagram.

Synopsis of the processing at the daemon: Though tracing the ensuing sequence of events on the daemon side is not the purpose of this subsection, nor is it shown on the Event trace diagram (in fact, it happens between lines (4) and (5)), a brief outline is given here. The daemon would return from the select system call (main_loop(); switch(wait_on_input(shortest); select()) and control is passed to method XTPdaemon::dispatch_request(...), case REGISTER. A call is then made to the context manager to initialize a context (d_cm→init_context(...)). The context_manager assigns a *key*, and stores it in the user request structure so that it gets returned to the client. Similar steps are performed for the shared memory identification numbers.

Complete registration request: Control is back to the client, which got its reply from the daemon in a user request data structure. The base class user_request has fields such as *key*, *snd_shmid* and *rcv_shmid*. As the client's buffer manager objects already exist, calls to buffer_manager's attach() method are made to get the starting address of the two shared memory segments, using the shmid's returned by the daemon. Only the call for the receive segment is shown on the Event trace diagram. The end of the registration procedure includes making copies of essential information for later transactions with the daemon (see Figure 19, boxout 3).

Create and bind an address with the daemon: The next step consists of filling up and binding an address structure so that this bound address can be used as a filter in discriminating against incoming FIRST packet addresses (the sender must specify the same transport level address when sending). Otherwise, how could the daemon be able to identify a proper listening context? SandiaXTP being a user level implementation, a DDS port number would steer a FIRST packet to XTP, but no further. For this purpose, XTP dispenses *logical* port numbers to each context. The creation and the completion of the client's address structure for use with the bind() method is illutrated on Figure 19, boxout 4. Following the same sequence of calls made for the REGISTRATION request, method issue() is used to contact the daemon for binding the address to the context. Note that the daemon makes no bind() system call to the Operating System, keeping the issue at the user level.

Traffic specifications: Now at the bottom of the Event trace diagram, the next step triggered from main() (common.h) consists of obtaining the current traffic specifications from the daemon, and making changes as per the command line arguments typed by the user (-p, -j and -J arguments). Handling this isssue independently from the registration request confers more generality to the mechanism, which could again be used later to change the traffic specifications. The daemon returns the information into an xtp_state_msg structure, which includes as one of its fields a *traffic* structure (trafic tpesc; with fields maxdata, inrate, inburst,...). The client can then set the desired values(from main() - common.h) to: (a) tspec.ts1.maxdata=xcf→pdu_size);
(b) tspec.ts1.inrate=input_rate;
(c) tspec.ts1.inburst=input_burst. Method xi→puttspec(...) is then used to convey the revised traffic specification values to the daemon, with issue(...) and following the same sequence of calls made for the REGISTRATION request.

Listening: From main() (common.h), interface method xi→listen(0,&opt) is the last one called. Because of the **-k** command line argument, block has a non-zero values and the method will block until the arrival of a FIRST matching packet. The description of the Receiver startup Event trace is now complete.

5 Rate Control with SandiaXTP

The goal of this section is to disclose the detailed mechanisms used by SandiaXTP to implement the XTP rate control feature. The methodology used consists of a three phase abstraction process:

1. First, starting from the source code and with the help of Event trace diagrams, stage the interactions of a sender process with the daemon for sending data, eliminating as many details as possible that have no critical impact on rate control.

2. Second, using the Event trace diagrams prepared in step 1, abstract (preferably on a single page) the salient points of the rate control algorithm buried in the SandiaXTP source code.

3. Third, using the algorithm produced in step 2, apply it to a typical send scenario with the intent of gaining insight into its functioning.

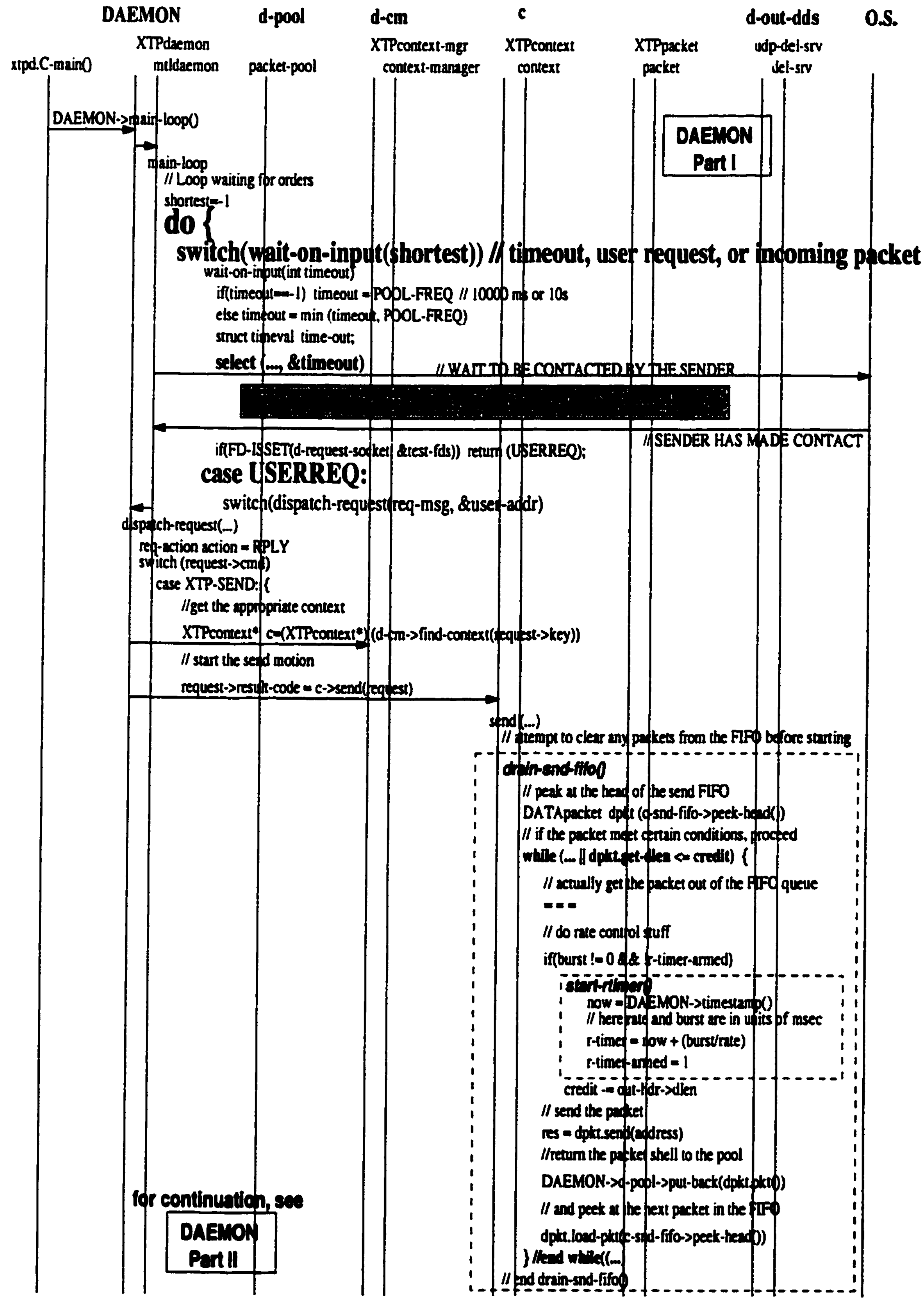
The following subsections present the results of this investigatory work into the rate control mechanisms used by SandiaXTP.

5.1 Rate control Event traces (Sender and Daemon)

The rate control Event trace for the sender fits on a single page and is shown on Figure 23. The one for the daemon is much more intricate and consists of parts I, II, and III, respectively shown on Figures 20, 21, and 22. These two Event trace diagrams are commented on in the following paragraphs, which are ordered as per the interleavings of events that would normally occur when a client process sends data to a remote receiver via the daemon.

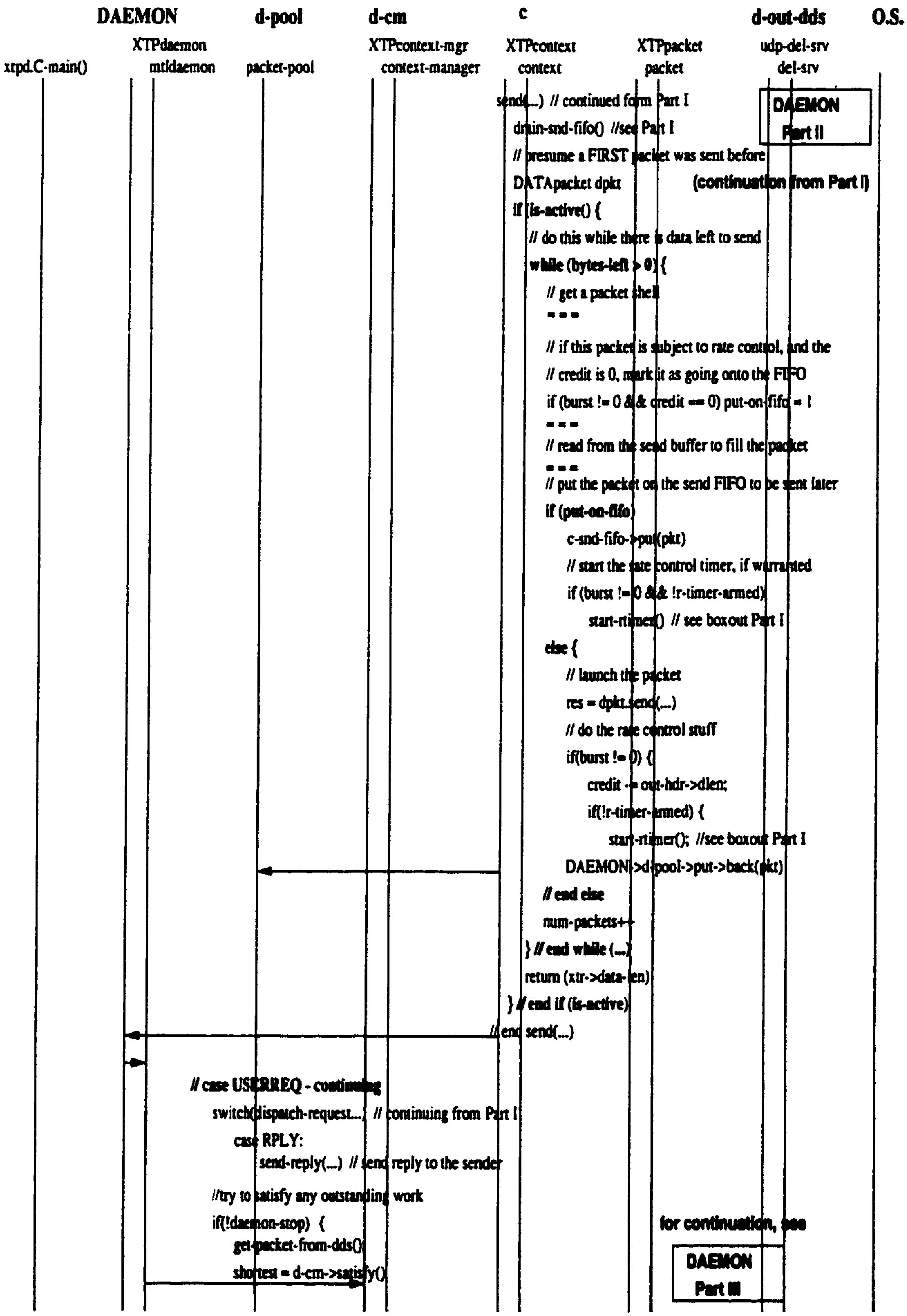
DAEMON - Part I (Figure 20)

The dynamic behavior of the daemon is implemented with a *do {...} while (!daemon_stop)* loop. At this point, it is presumed that the daemon was started at some host by the user and has halted execution on the *select(..., &timeout)* system call waiting for client orders. The previous calls for starting up the daemon are not shown on



```
DAEMON->main-loop()
main-loop
  // Loop waiting for orders
  shortest=-1
  do {
    switch(wait-on-input(shortest)) // timeout, user request, or incoming packet
      wait-on-input(int timeout)
        if(timeout==-1)  timeout = POOL-FREQ // 10000 ms or 10s
        else timeout = min (timeout, POOL-FREQ)
        struct timeval  time-out;
        select (..., &timeout)
        if(FD-ISSET(d-request-socket, &test-fds))  return (USERREQ);
    case USERREQ:
      switch(dispatch-request(req-msg, &user-addr)
dispatch-request(...)
  req-action action = RPLY
  switch (request->cmd)
    case XTP-SEND: {
      //get the appropriate context
      XTPcontext*  c=(XTPcontext*)(d-cm->find-context(request->key))
      // start the send motion
      request->result-code = c->send(request)
send(...)
  // attempt to clear any packets from the FIFO before starting
  drain-snd-fifo()
    // peak at the head of the send FIFO
    DATApacket  dpkt (c-snd-fifo->peek-head())
    // if the packet meet certain conditions, proceed
    while (... || dpkt.get-dlen <= credit)  {
      // actually get the packet out of the FIFO queue
      ===
      // do rate control stuff
      if(burst != 0 && !r-timer-armed)
        start-rtimer()
          now = DAEMON->timestamp()
          // here rate and burst are in units of msec
          r-timer = now + (burst/rate)
          r-timer-armed = 1
        credit -= out-hdr->dlen
      // send the packet
      res = dpkt.send(address)
      //return the packet shell to the pool
      DAEMON->d-pool->put-back(dpkt.pkt())
      // and peek at the next packet in the FIFO
      dpkt.load-pkt(c-snd-fifo->peek-head())
    } //end while((...)
  // end drain-snd-fifo()
```

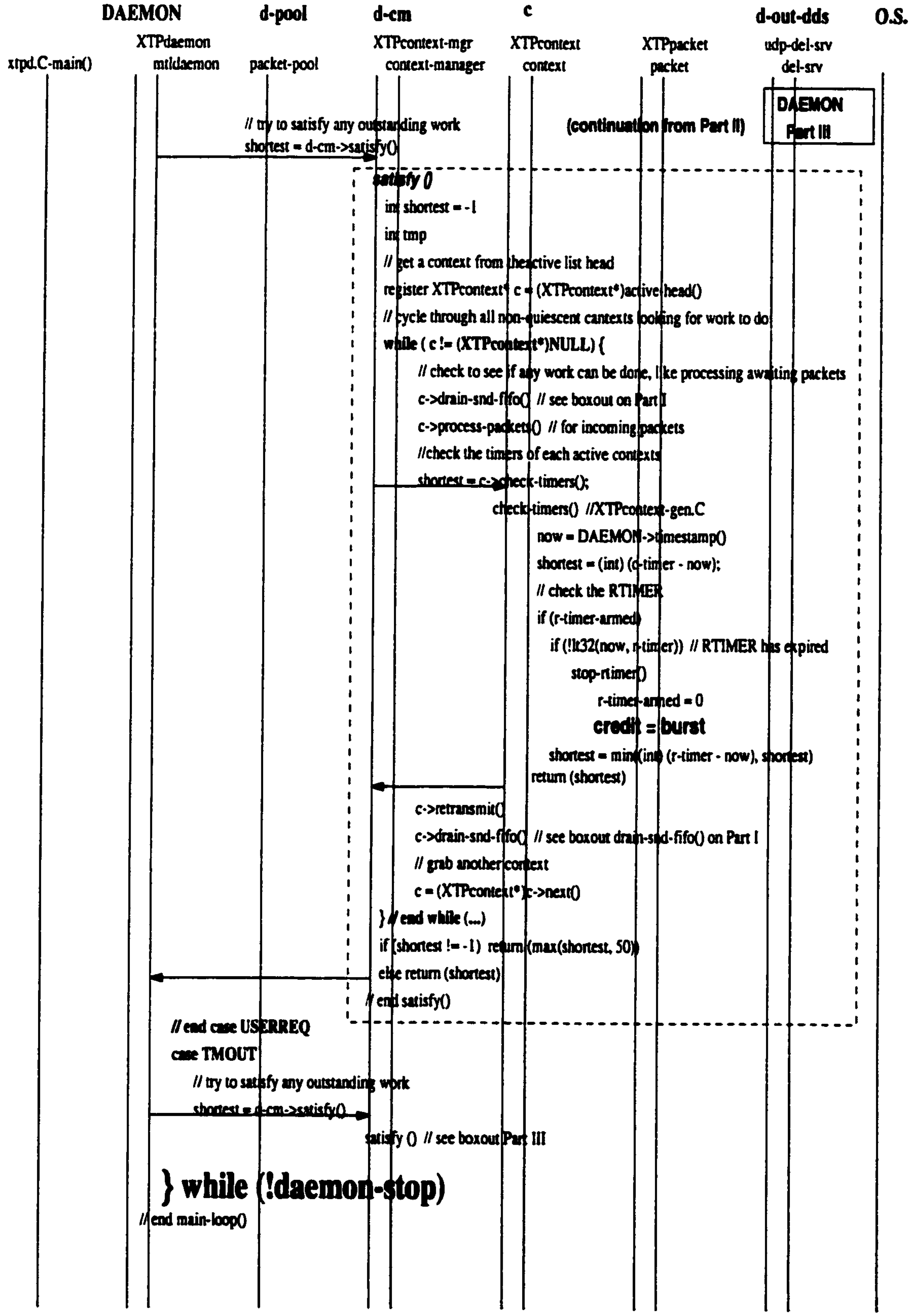
Figure 20: SandiaXTP rate control - daemon Event trace Part I



```
send(...) // continued form Part I
  drain-snd-fifo() //see Part I
  // presume a FIRST packet was sent before
  DATApacket dpkt
  if (is-active() {
    // do this while there is data left to send
    while (bytes-left > 0) {
      // get a packet shell
      ...
      // if this packet is subject to rate control, and the
      // credit is 0, mark it as going onto the FIFO
      if (burst != 0 && credit == 0) put-on-fifo = 1
      ...
      // read from the send buffer to fill the packet
      ...
      // put the packet on the send FIFO to be sent later
      if (put-on-fifo)
        c-snd-fifo->put(pkt)
        // start the rate control timer, if warranted
        if (burst != 0 && !r-timer-armed)
          start-rtimer() // see boxout Part I
      else {
        // launch the packet
        res = dpkt.send(...)
        // do the rate control stuff
        if(burst != 0) {
          credit -= out-hdr->dlen;
          if(!r-timer-armed) {
            start-rtimer(); //see boxout Part I
        DAEMON->d-pool->put->back(pkt)
      // end else
      num-packets++
    } // end while (...)
    return (xtr->data-len)
  } // end if (is-active)
// end send(...)
```

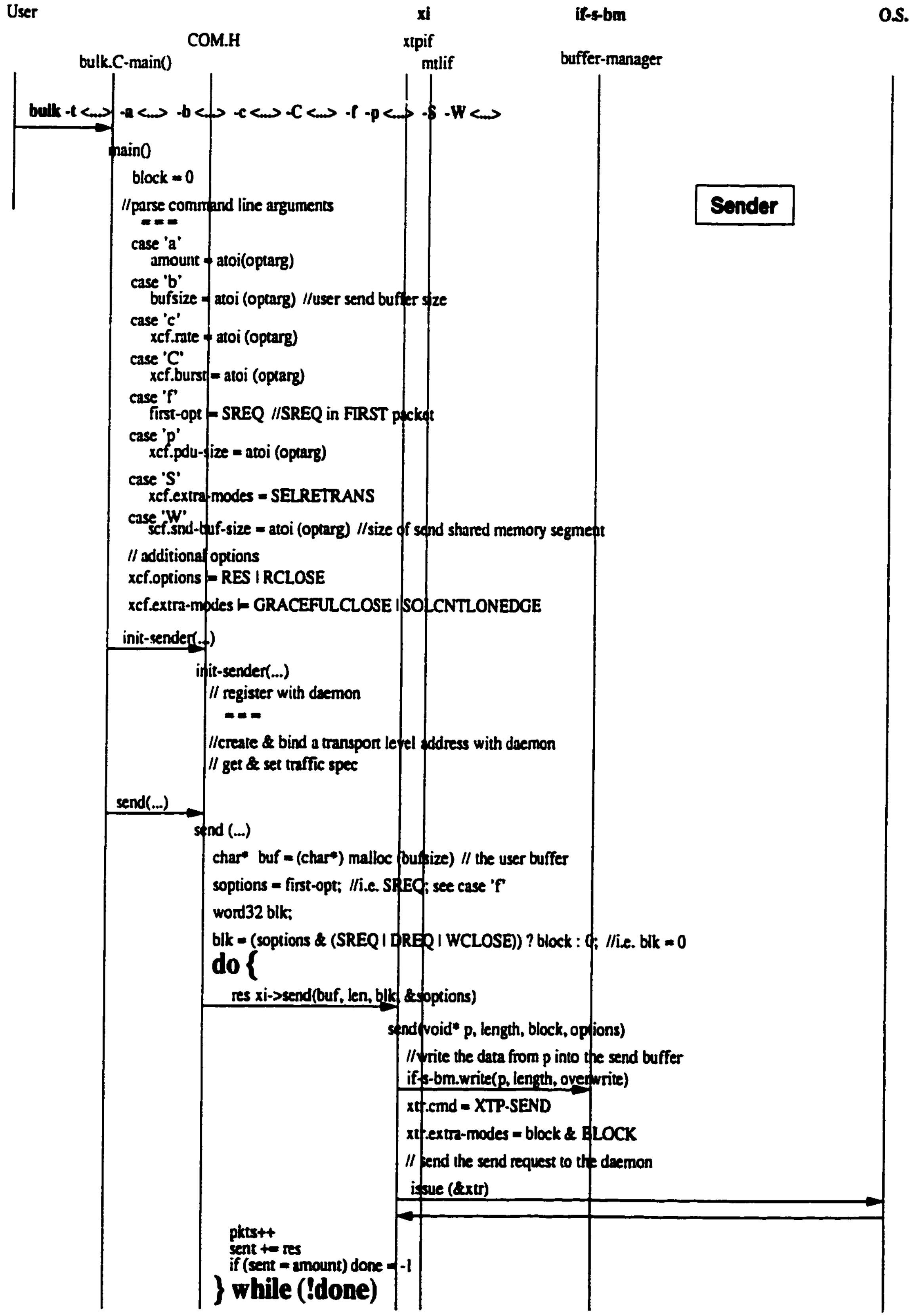
```
// case USERREQ - continuing
  switch(dispatch-request... // continuing from Part I
    case RPLY:
      send-reply(...) // send reply to the sender
  //try to satisfy any outstanding work
  if(!daemon-stop) {
    get-packet-from-dds()
    shortest = d-cm->satisfy()
```

Figure 21: SandiaXTP rate control - daemon Event trace Part II



```
// try to satisfy any outstanding work
shortest = d-cm->satisfy()
satisfy ()
  int shortest = -1
  int tmp
  // get a context from theactive list head
  register XTPcontext* c = (XTPcontext*)active head()
  // cycle through all non-quiescent cantexts looking for work to do
  while ( c != (XTPcontext*)NULL) {
        // check to see if any work can be done, like processing awaiting packets
        c->drain-snd-fifo()  // see boxout on Part I
        c->process-packets()  // for incoming packets
        //check the timers of each active contexts
        shortest = c->check-timers();
                          check-timers()  //XTPcontext-gen.C
                                now = DAEMON->timestamp()
                                shortest = (int) (c-timer - now);
                                // check the RTIMER
                                if (r-timer-armed)
                                  if (!lt32(now, r-timer))  // RTIMER has expired
                                      stop-rtimer()
                                          r-timer-armed = 0
                                       credit = burst
                                  shortest = min((int) (r-timer - now), shortest)
                               return (shortest)
        c->retransmit()
        c->drain-snd-fifo()  // see boxout drain-snd-fifo() on Part I
        // grab another context
        c = (XTPcontext*)c->next()
  } // end while (...)
  if (shortest != -1)  return (max(shortest, 50))
  else return (shortest)
 // end satisfy()
//end case USERREQ
case TMOUT
    // try to satisfy any outstanding work
    shortest = d-cm->satisfy()
                  satisfy ()  // see boxout Part III
} while (!daemon-stop)
// end main-loop()
```

Figure 22: SandiaXTP rate control - daemon Event trace Part III



```
bulk -t <...> -a <...> -b <...> -c <...> -C <...> -f -p <...> -S -W <...>
main()
  block = 0
//parse command line arguments
  ...
  case 'a'
    amount = atoi(optarg)
  case 'b'
    bufsize = atoi (optarg) //user send buffer size
  case 'c'
    xcf.rate = atoi (optarg)
  case 'C'
    xcf.burst = atoi (optarg)
  case 'f'
    first-opt = SREQ //SREQ in FIRST packet
  case 'p'
    xcf.pdu-size = atoi (optarg)
  case 'S'
    xcf.extra-modes = SELRETRANS
  case 'W'
    scf.snd-buf-size = atoi (optarg) //size of send shared memory segment
// additional options
xcf.options = RES | RCLOSE
xcf.extra-modes |= GRACEFULCLOSE | SOLCNTLONEDGE
init-sender(...)
        init-sender(...)
        // register with daemon
          ...
        //create & bind a transport level address with daemon
        // get & set traffic spec
send(...)
        send (...)
        char* buf = (char*) malloc (bufsize) // the user buffer
        soptions = first-opt; //i.e. SREQ; see case 'f'
        word32 blk;
        blk = (soptions & (SREQ | DREQ | WCLOSE)) ? block : 0; //i.e. blk = 0
        do {
          res xi->send(buf, len, blk, &soptions)
                send(void* p, length, block, options)
                //write the data from p into the send buffer
                if-s-bm.write(p, length, overwrite)
                xtr.cmd = XTP-SEND
                xtr.extra-modes = block & BLOCK
                // send the send request to the daemon
                issue (&xtr)
          pkts++
          sent += res
          if (sent = amount) done = -1
        } while (!done)
```

Figure 23: SandiaXTP Rate Control - Sender Event Trace

the diagram.

The *select()* system call allows a user process (i.e., the daemon) to instruct the operating system kernel to monitor the eventual occurence of several events and to wake up the process only when one of these events occur. Effectively, the semantics of *select()* is: "return when one of the specified descriptors is ready for I/O, but don't wait beyond a fixed amount of time, specified by variable `timeout`". The descriptors could be a user request coming through an internal UNIX socket, or an incoming packet coming through an Internet UNIX socket.

The monitoring activity of the daemon is illustrated by the highlighted rectangle shown on Figure 20. The focus of the inquiry now switches to the Event Trace diagram of the sender shown on Figure 23.

The sender (Figure 23)

To embody the sender process, program **bulk** (one of SandiaXTP example programs) is used with typical rate control related command line arguments to suit the present purpose. These options are now reviewed:

- **-a** <...> specifies the amount of data to be sent, in bytes (say of the order of 1 MB)
- **-b** <...> specifies the size of the user buffer for transferring the data from user space to the shared memory send_area
- **-c** <...> specifies the suggested output rate, in bytes per millisecond (Bpms)
- **-C** <...> specifies the suggested output burst, in bytes
- **-f** specifies that the FIRST packet be sent with the SREQ bit set so that the receiver reacts immediately
- **-p** <...> specifies the Protocol Data Unit (PDU) size, in bytes
- **-S** specifies usage of the *selective retransmission* flow control mechanism (the other being go-back-N).

- **-W** <...> specifies the size of the send and receive shared memory areas.

Given these configuration parameters, a call is made to initialize the sender (*init_sender(...)*), which is not traced further, as the steps are very similar to the ones needed to initialize a receiver. Such a scenario has already been traced in detail in Section 4.4.3. Then, the sender is about to engage into the send operation with call to method *send(..)*. Memory is claimed for a user buffer of size `bufsize`, as specified by the user.

This same buffer will be used repeatedly to transfer all the data to the send shared memory area; its size defines the granularity at which the data will be passed to the send shared memory area; and its size combined with `amount` defines the number of send requests that will be issued to the daemon for sending all the data.

The options have been selected such that the user does not block on acknowledgements (i.e., blk=0), which means that the sender will block on each send request issued (each time method *issue()* is called) to the daemon, but the daemon won't delay much before returning its reply to the sender. Otherwise, each time method *issue()* is called (see later the *do {...} while (!done)* loop), the daemon would delay its reply until an acknowledgement is received for the last packet issued as a result of this request.

Then the sender begins its send loop (*do {...} while (!done)*). This loop is cycled repeatedly to send `amount` of data. The quotient of `amount` by `bufsize` defines the number of cycles, plus eventually one more cycle for the remainder. The services of the *xi* interface, and subsequently of the buffer_manager are invoked to copy the data into the shared memory area (*if_s_bm.write(p, length, overwrite)*). The request is then conveyed to the daemon via the the *issue()* method, with command `XTP_SEND`. The focus now switches back to the daemon, never to return to the sender.

DAEMON Cont. - Part I (Figure 20)

As a consequence of the user request issued by the sender, the daemon is awaken by the Operating System, i.e., it returns from the select() system call. Being a request to send data (*case USERREQ: & case XTP_SEND:*), the XTPcontext method *send(...)* is called, and much will happen before it returns.

Method *send(...)* is essentially composed of four parts: (1) on behalf of the sender, a call is made to method *drain_snd_fifo()* to clear any outstanding packets that may have accumulated on the send FIFO queue of this context, if the value of variable `credit` (and other conditions) allow it; (2) the discharge of the work being consequent to the request, resulting whether in sending a packet immediately if possible, or otherwise enqueuing the packet on the send FIFO queue to be sent later; (3) the issue of the reply to the sender (method *send_reply()*) so that the sender can do one more cycle in its *do {...} while (!done)* loop (see Figure 23); (4) an attempt to satisfy any outstanding work, now on behalt of all contexts that the daemon may be dealing with. With respect to rate control, it is the call to method *satisfy()* that is really significant. Method *satisfy()* is shown on Part III of the daemon Event Trace (Figure 22) and its logic is explored later while describing the rate control algorithm.

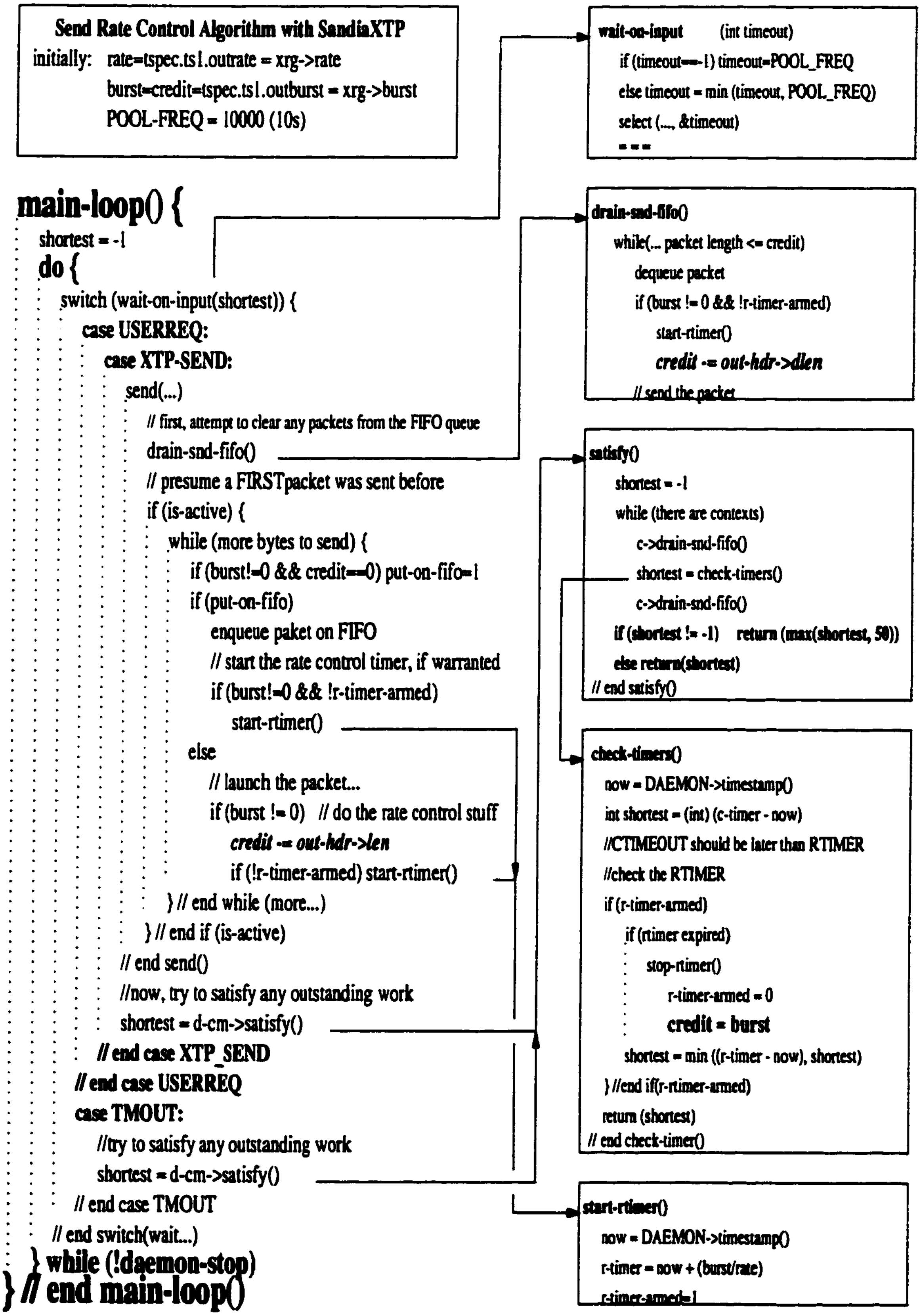
In terms of tracing, *case TMOUT* adds nothing new as it calls only method *satisfy()*, which will be dealt with shortly.

5.2 Rate control algorithm with SandiaXTP

The process of abstraction is now carried one step further in this subsection through the presentation of the rate control algorithm inferred from the Event trace of the daemon, which is shown on Figure 24. As space is at a premium on this diagram, it has dictated the exact form of the language used to express the algorithm, which is a mixture of pseudo-code and code constructs borrowed from the SandiaXTP source code. At times, when the source code is too complex or would look rather incomprehensible, it is replaced by a C++ style comment. For example, *//send the packet* is used in lieu of the more precise source code, which would be:

```
int res = dpkt.send(is_mcast_xmitter()?c_mcast_dest:c_ucast_dest)
```

In any case, the labels used on the Event trace diagram are much more precise and can be used to clarify statements that would appear confusing in the algorithm. The relationships between the rate control Event trace diagram for the daemon presented



```
Send Rate Control Algorithm with SandiaXTP
initially:  rate=tspec.ts1.outrate = xrg->rate
            burst=credit=tspec.ts1.outburst = xrg->burst
            POOL-FREQ = 10000 (10s)
```

```
main-loop() {
  shortest = -1
  do {
    switch (wait-on-input(shortest)) {
      case USERREQ:
        case XTP-SEND:
          send(...)
            // first, attempt to clear any packets from the FIFO queue
            drain-snd-fifo()
            // presume a FIRSTpacket was sent before
            if (is-active) {
              while (more bytes to send) {
                if (burst!=0 && credit==0) put-on-fifo=1
                if (put-on-fifo)
                  enqueue paket on FIFO
                  // start the rate control timer, if warranted
                  if (burst!=0 && !r-timer-armed)
                    start-rtimer()
                else
                  // launch the packet...
                  if (burst != 0)   // do the rate control stuff
                    credit -= out-hdr->len
                    if (!r-timer-armed) start-rtimer()
              } // end while (more...)
            } // end if (is-active)
          // end send()
          //now, try to satisfy any outstanding work
          shortest = d-cm->satisfy()
        // end case XTP_SEND
      // end case USERREQ
      case TMOUT:
        //try to satisfy any outstanding work
        shortest = d-cm->satisfy()
      // end case TMOUT
    // end switch(wait...)
  } while (!daemon-stop)
} // end main-loop()
```

```
wait-on-input       (int timeout)
    if (timeout==-1) timeout=POOL_FREQ
    else timeout = min (timeout, POOL_FREQ)
    select (..., &timeout)
    ===
```

```
drain-snd-fifo()
    while(... packet length <= credit)
        dequeue packet
        if (burst != 0 && !r-timer-armed)
            start-rtimer()
            credit -= out-hdr->dlen
        // send the packet
```

```
satisfy()
    shortest = -1
    while (there are contexts)
        c->drain-snd-fifo()
        shortest = check-timers()
        c->drain-snd-fifo()
    if (shortest != -1)    return (max(shortest, 50))
    else return(shortest)
// end satisfy()
```

```
check-timers()
   now = DAEMON->timestamp()
   int shortest = (int) (c-timer - now)
   //CTIMEOUT should be later than RTIMER
   //check the RTIMER
   if (r-timer-armed)
      if (rtimer expired)
         stop-rtimer()
            r-timer-armed = 0
            credit = burst
      shortest = min ((r-timer - now), shortest)
   } //end if(r-rtimer-armed)
   return (shortest)
// end check-timer()
```

```
start-rtimer()
   now = DAEMON->timestamp()
   r-timer = now + (burst/rate)
   r-timer-armed=1
```

Figure 24: Send rate control algorithm with SandiaXTP

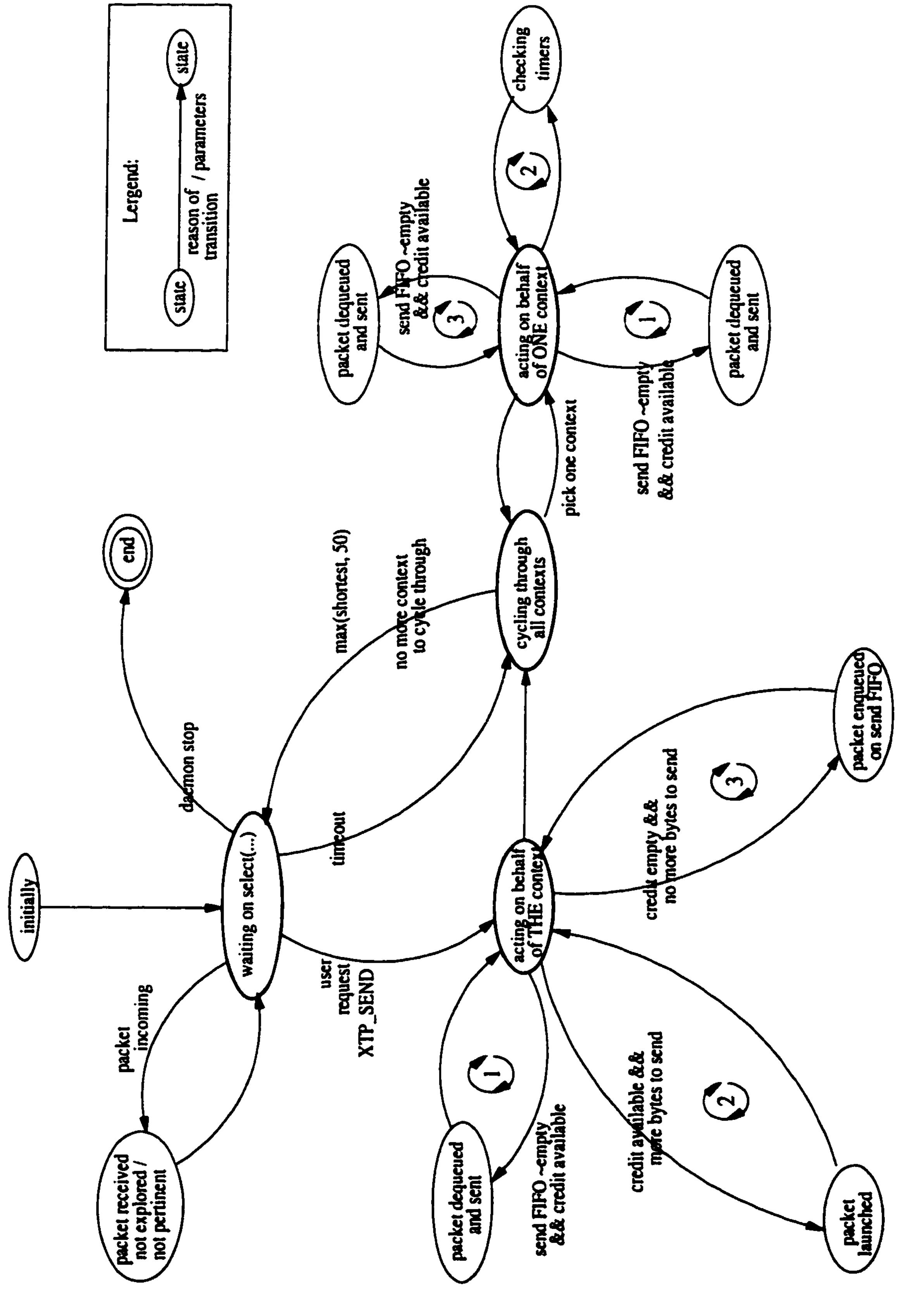
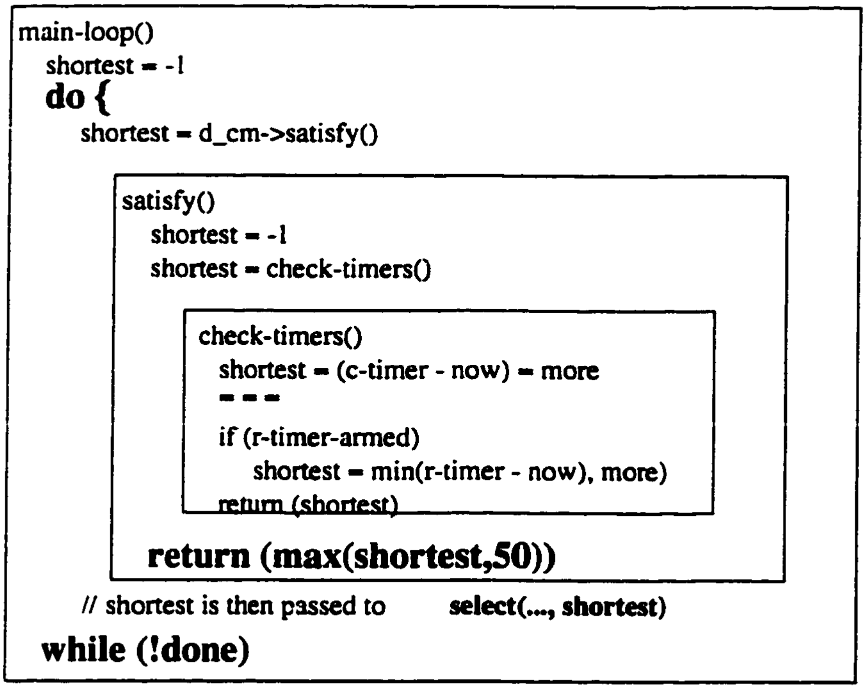

Lergend:
state
reason of transition
/ parameters
state
initially
waiting on select(...)
end
daemon stop
packet incoming
packet received not explored / not pertinent
timeout
max(shortest, 50)
no more context to cycle through
cycling through all contexts
user request XTP_SEND
acting on behalf of THE context
pick one context
acting on behalf of ONE context
checking timers
2
send FIFO ~empty && credit available
3
packet dequeued and sent
1
packet dequeued and sent
send FIFO ~empty && credit available
credit empty && no more bytes to send
3
packet enqueued on send FIFO
1
packet dequeued and sent
send FIFO ~empty && credit available
credit available && more bytes to send
2
packet launched

Figure 25: SandiaXTP rate control algorithm - EFSM



```
main-loop()
  shortest = -1
  do {
    shortest = d_cm->satisfy()

      satisfy()
        shortest = -1
        shortest = check-timers()

          check-timers()
            shortest = (c-timer - now) = more
            = = =
            if (r-timer-armed)
               shortest = min(r-timer - now), more)
            return (shortest)

        return (max(shortest,50))
    // shortest is then passed to      select(..., shortest)
  while (!done)
```

Figure 26: SandiaXTP rate control Algorithm - Updating of `shortest` and `timeout`

in the previous subsection, and the rate control algorithm presented on Figure 24 should be relatively easy to establish by the reader, as many of the keywords that appear on both diagrams have the same font size and also have been highlighted (bold) (Example: keywords **do {** and **} while (!daemon_stop)**).

The right hand part of Figure 24 shows some methods, circumscribed with boxouts, which are generally called more than once from the mainstream algorithm shown on the left hand part of the same figure. To save space and also help following the logic of the algorithm, only the name of these repetitive methods are given in the main algorithm, with arrows pointing to the boxouts that contain the method itself. For convenience, the dynamic aspect of the rate control algorithm shown on Figure 24 is also shown on Figure 25 in the form of an extended finite state machine (EFSM).

Before describing the logic of the algorithm itself, let us consider separately two of its important dimensions and their related state variables: time and credit management.

Time management

The management of time is of the essence in rate control. The two main variables names used by the algorithm to manage the time aspects are `shortest` and `timeout`. Variable `shortest`, in fact, appears at three scopes of visibility. The highest most general scope is within the main_loop() itself. The second level of visibility of variable `shortest` is within the scope of method satisfy(). The third, most restricted, level of visibility is within the scope of method *check_timers()*. In fact, variable `shortest` could have different names within the scope of these three methods. During the course of the execution of the algorithm, the value of `shortest` is bubbled up from the lowest level (i.e., check_timers()) to the most general level which is within the scope of the main_loop(). As seen later, there are some conditions that may restrict the exact value of `shortest` that is bubbled up.

Variable `timeout` is particular to method *wait_on_input()* (see boxout). In fact, once the daemon is rolling, it is the value of `shortest` as it is at the end of one cycle of the **do {...} while (!daemon_stop)** loop that imposes itself to `timeout`, i.e., `timeout` being a parameter of method wait_on_input(`timeout`) takes the value of `shortest`.

With regards to the dynamics of the program, variable `timeout` is particularly important, because it is passed to the operating system via the select system call (select(..., &`timeout`)), and it effectively defines how long the daemon will sleep if no user request is issued in the meanwhile or no packet comes in from the network.

The algorithm is such that `timeout` is never larger than POLL_FREQ (i.e., 10s), such as when the daemon is first started up, or never smaller than 50 ms (see boxout with method satisfy()). Generally, for the present staging, the value of `timeout` is equal to RTIMER if larger than 50ms, or 50ms otherwise. The implications of this minimum of 50ms will be explored extensively later in this report.

Figure 26 is meant to clarify the levels of visibility of variable `shortest` and how it is passed to select.

Credit management

The management of variable credit is also critical to the operation of the rate control algorithm abstracted. Before starting a new burst of packets, the value of credit equals the value of variable burst, which itself is specified by the user. Then credit is decremented each time a packet is sent, until the value of credit is smaller than the size of a packet. This occurence signifies the end of sending a burst of packets.

With regards to the algorithm presented on Figure 24, decrementing credit is done at two locations (both highlighted with bold italic fonts). One location is the third line after the *else* statement, which represents the case when the daemon has been contacted by the sender and is reacting to this request by sending a packet. Another location is at the end of method *drain_snd_fifo()*, which occurs when the daemon checks for work to be done.

In contrast, variable credit is incremented to the value of variable burst only once in the algorithm, i.e., towards the end of method *check_timers()*. This occurence represents the case when the rtimer has expired and the daemon is about to begin the operation of sending the next burst of packets.

Dynamics of the algorithm

The execution of the algorithm is started at the beginning of the main_loop(), as shown on Figure 24. At this point, variable shortest is set to -1. Consequently, variable timeout takes the value of POLL_FREQ (i.e., 10s) and select is called for waking up the daemon at most 10 seconds later (select(..., &timeout=10000)).

case USERREQ:

Soon, probably before the timeout occurs, the daemon is likely to awaken as a result of a send user request issue by the sender. With regards to the algorithm, this event translates to *case USERREQ: & case XTP_SEND:*. Now, the algorithn is being executed by the daemon on behalf of the particular context that sent the user request.

Immediately at the beginning of method *send(...)*, a first attempt is made to drain the send FIFO queue of this context.

This step is reasonable, as the send operation may have been ongoing for a while and some packets may already have accumulated on the send FIFO queue of this context. These packets should have priority over the one(s) underlying the current user request.

If there is still `credit` available and the `r_timer_armed` flag is not set, it effectively means that the daemon is starting to send a new burst of packets out of the send FIFO queue, and measures should be taken now to detect the end of the current burst. Consequently, method *start_timer()* is called from *drain_snd_fifo()*. The effect is of computing a new value for the `r_timer` variable, and arming it.

On the other hand, if the `r_timer_armed` flag is set to 1, then it just means that the daemon is continuing to send a burst of packets, and only decrementing the value of `credit` is needed.

The execution of the algorithm has now reached the line *if (is_active)*. The goal is to handle the data that underlies the **XTP_SEND** request passed by the sender; i.e., make packets and either send them immediately, or place them on the send FIFO queue of this context for sending later. If the value of `credit` is nil (credit == 0 is true), then the packet has to be enqueued on the send FIFO queue; consequently, the flag `put_on_fifo` is set accordingly and the packet is enqueued. If the `r_timer_armed` flag is not set, then method start_timers() is called to prepare for the next burst of packets to be sent.

Otherwise, if the value of `credit` allows it, the packet is sent immediately (follow up after the *else* statement). After decrementing the value of the `credit` variable, method start_timer() is also possibly called. The tracing of the algorithm has now reached the end of method *send (...)*.

The next step of the algorithm, still within *case USERREQ:*, consists of trying to satisfy any outstanding work, with call **shortest** = *d_cm→satisfy()*. The daemon now switches from acting on behalf of the particular context that made the request to cycling through all contexts that it has to serve with method *satisfy()*.

The outlook is now from method *satisfy()* (see boxout), cycling through all contexts. Whilst executing method *satisfy()*, two attemps are made to drain the send FIFO queue of each context. The first one is made prior to calling method *check_timers()*, and another one is made afterwards. As method *check_timers()* may result in refurbishing the value of `credit` to its full `burst` value, presumably the first call to method *drain_snd_fifo()* is needed to make sure that packets that would belong to the current burst on the send FIFO queue are sent (if `credit` allows it of course), prior to engaging in the next burst. Then the statement `shortest` = *check_timers()* gets executed, which may possibly result in upating the value of `shortest` passed to the main_loop() and also the value of `credit`.

Method check_timers()

All the timers for a particular context are checked. For the present rate control study, it is presumed that the value of the connection timer (`CTIMEOUT`) is much larger, and only the `RTIMER` needs consideration. Variable `credit` gets incremented to the value of `burst` if the `r_timer_armed` flag was set and it has expired, which means that it is now time to replenish `credit` in prevision for the next burst of packets. This is the only location in the algorithm where the value of `credit` is incremented.

Within *check_timers()*, the statement that does the updating of `shortest` is as follows:

$$\texttt{shortest} = \min((\texttt{r\_timer})\ \texttt{- now)},\ \texttt{shortest}$$

As `shortest` was set to `CTIMER` at the beginning of method *check_timers()*, it should be much larger that the value of `RTIMER` and could be ignored. Hence, the equation to update `shortest` within method *check_timer()* is as follows:

$$\texttt{shortest} = \texttt{r\_timer - now}$$

This value is returned to method *satisfy()*.

Variable **shortest** is updated if the **r_timer_armed** flag was set to 1, whether it has expired or not. If the **r_timer_armed** flag was set (i.e., the rtimer was armed) but had not expired, then a new value for **shortest** needs to be recomputed. Given the old value of variable **r_timer**, and the value of variable **now** being larger than it was, then the updated value of **shortest** will be smaller than its previous value, which is reasonable. If, on the other hand, the **r_timer_armed** flag was set to 1 and **RTIMER** had expired, then a new value for **shortest** is also recomputed, but using the expired **r_timer** value. Consequently, **shortest** takes a negative value (as **now** is larger in **r_timer-now**). Later on, this negative value for **shortest** implies that 50 ms might be bubbled up to select(), and not a timer related value as expected. Even if **RTIMER** has expired, other timers might be pending, and **shortest** should convey the next one to expire.

Method satisfy() - back

Now, the second call to method *drain_snd_fifo()* is made, which may be necessary because of replenishment of variable **credit** and having not finished up draining the send FIFO when method *drain_snd_fifo()* was called before. Method *start_rtimer()* is likely to be called again, with updating of the **r_timer** value and the setting of the **r_timer_armed** flag. However, this recalculation of the **r_timer** value happens **too late** to be conveyed to the main_loop() and the select() system call via **shortest**, as **shortest** within the scope of method *satisfy()* is already fixed.

The tracing for *case USERREQ:* is now complete.

case TMOUT:

The focus is now on *case TMOUT:*, which would occur when the value of **shortest** passed to select() times out and neither a user request nor any packet was received by the daemon in the meanwhile. As per the algorithm shown on Figure 24, the only call being made is to method *satisfy()* as follows:

shortest = *d_cm→satisfy()*

Having no particular context in view, the daemon has to cycle through all contexts in search of work to do. The logic of the algorithm for method *satisfy()* is not traced again, as this work was done in the preceeding paragraphs while discussing *case USERREQ:*. The presentation of the algorithm is completed. General comments and observations about the algorithm are presented later in Section 5.4.

5.3 Rate control analysis with a scenario

The purpose of this subsection is to explore more precisely the working of the algorithm with the help of a typical scenario, similar to the ones presented in Section 3 for the rate control feature. As expressed previously, the exercise of rate control implies some periods of **idle time** during which no packet is being sent. Otherwise, the send operations are done at the hardware send rate capacity, and no rate control is effectively exercised. The scenarios used take into consideration the manner with which the idle time is distributed within the time needed to send a burst of packets.

Scenario - idle time evenly distributed, rate=25 Bpms

The general assumptions that are valid for this scenario are as follows:

- a sender is reliably sending a large amount of data to a receiver, using the unicast mode;
- the connection establishment phase is over, i.e., a FIRST packet was sent and replied to and the sender is about to contact the daemon for sending data packets;
- the size the send window (i.e., the size of the send shared memory area) is very large so that the operation of the flow control algorithm has minimal impact on the rate control algorithm;
- a logical clock is used to keep track of time; for processing at the sender, only the time needed to "put a packet on the wire" is accounted for; the send operation is presumed to be done at half the Ethernet speed (i.e., about **3 ms** for a packet of PDU size of 1440 bytes); the time needed to enqueue packets, etc., is ignored.

Table 4: Scenario - State Summary (25/1440)

	logical	shortest (ms)			timeout	r_timer(ms)		
	clock			(check	(wait_on	= now +	r_timer	credit
state	= now	(main())	(satisfy())	_timers())	input())	(burst/rate)	_armed	(bytes)
S_0	0	-1			10000		0	1440
S_1	3	57.6	57.6	57.6		60.6	1	0
S_2	say 3	57.6	57.6	57.6				
•								
S_n	60.6		0	0			0	1440
S_{n+1}	60.6	50???				118.2	1	0
•								
S_m								
S_{m+1}								
etc								

rate = 25 Bpms; burst = 1440 bytes
RTIMER = (burst/rate) = (1440/25) = 57.6 ms
S_0 - at the beginning, halted on select()
S_1 - case USERREQ: after launching a first DATApacket
S_2 - case USERREQ: after placing a packet on the send FIFO queue
S_n - case TMOUT: halfway through satisfy(), before 2nd drain_snd_fifo()
S_{n+1} - case TMOUT continued: after drain_snd_fifo() - 1 packet sent
S_m - case TMOUT: halfway through satisfy(), before 2nd drain_snd_fifo()
S_{m+1} - case TMOUT continued: after drain_snd_fifo() - 1 packet sent

Table 4 shows a summary of the data for the simulated execution of the scenario. To produce an even distribution of the idle time, the burst is defined to be the size of one packet (i.e., 1440 bytes). Given a rate of 25 bytes per millisecond, the value of the RTIMER is 57.6 ms.

State S_0 - at the beginning, halted on select()

At this point, the daemon has opened a context for the sender, but has not started the task of sending data yet. It is waiting on select() to be contacted by the sender with the request for sending data (see boxout **wait_on_input**). The first row of Table 4, corresponding to `state` = S_0, shows a summary of the state variables being traced at this point: `shortest` = -1 within the main_loop(); `timeout` = `POOL_FREQ`= 10000; the rtimer is not armed (`r_timer_armed`=0); and `credit` = `burst` = 1440.

State S_1 - case USERREQ: after launching a first DATA packet

With respect to the execution of the algorithm since state S_0, the daemon returned from the select() system call, and branched to *case USERREQ:*. The *send(...)* method was called, which eventually resulted in launching a packet after the *else* statement. The send activity is presumed to have taken 3ms, and the logical clock was updated accordingly on the second row of Table 4. Then variable `credit` got decremented (now `credit` = 0), and method *start_timer()* was invoked with updating of the state variables (i.e., `now`=3; `r_timer=now` + (`burst/rate`) = 60.6; `r_timer_armed`=1).

Method *satisfy()* was called, but as the send FIFO queue was empty, it did not follow through full execution. Method *check_timers()* was called; and as `r_timer_armed`=1, `shortest` = min((60.6 - 3),CTIMEOUT) = 57.6 got returned to satisfy() (`shortest` = *check_timers()*). From *satisfy()*, as `shortest` != -1, statement *return(max(57.6, 50))* got executed and the value `shortest` = 57.6 got returned to the main_loop() and subsequently passed to *select (..., &*`timeout=57.6`*)*. The summary of the state variables is shown on 4, row S_1.

State S_2 - case USERREQ: after inserting a packet on the send FIFO queue

With respect to the execution of the algorithm since state S_1, the sender continued copying data into the send shared memory area and issued another `XTP_SEND` request to the daemon. The path followed by the daemon was similar to S_1, but with slight modifications. As the value of `credit` was nil, the flag `put_on_fifo` was set to 1, and the packet was enqueued on the send FIFO queue of the sender context. As the rtimer was already armed, method *start_timers()* was not called. Again the statement `shortest` = *d_cm→satisfy()* got executed. Because there is no more `credit` available, the first call to method *drain_snd_fifo()* did not follow through. From *satisfy()*, `shortest`=*check_timers* got executed. As `r_timer_armed`=1, statement `shortest`=min((`r_timer` - `now`), `CTIMER`) got executed, but eventually the same value for `shortest` (i.e., 57.6ms) got returned at the scope of the main_loop, and the daemon is back waiting on *select*(..., &`timeout`=57.6).

Row S_2 of Table 4 presents a summary of the state variables with practically no change as compared to row S_1.

Between S_2 and S_n

Given that there is much data to be sent, it is presumed that not much change in terms of the state variables will occur. Only more packets will be enqueued on the send FIFO queue of this context. Change will however occur at some point when it is found out that the rtimer has expired.

State S_n - case TMOUT: halfway through satisfy(), but before the 2nd call to drain_snd_fifo()

At this point, it is presumed that the send FIFO queue has filled up, the logical clock has advanced such that the daemon returned from the select() system call with a rtimer timeout (*case TMOUT:*). From the main_loop, the only method that got called is
`shortest`=*d_cm→satisfy()*. From *satisfy()*, the first call to method *drain_snd_fifo()* did not follow through full execution because the value of `credit` was still nil. However, this time, the statement `shortest`=*check_timers()* has produced different results.

As the rtimer had expired, method *stop_rtimer()* was called, which reset some of the state variables (i.e., `r_timer_armed`=0; `credit` = `burst`). As the value of the `r_timer` variable had not been updated at this point, the statement `shortest` = min((60.6-60.6), CTIMEOUT) returned 0 (possibly in reality, it would return less than 0, but does not change much the current logic) to the scope of *satisfy()*.

State S_{n+1} - case TMOUT: continued from the 2nd call to drain_snd_fifo()

Now that the value of `credit` had improved, the second call made to method *drain_snd_fifo()* from *satisfy()* did follow through to full execution, and one packet was sent from the send FIFO queue. As the `r_timer_armed` flag was not set, method *start_rtimer()* was called with consequent review of some state variables (i.e., `r_timer`

= 60.6 + 57.6 = 118.2; r_timer_armed = 1). As shortest = 0 at the level of *satisfy()*, statement *return(max(0, 50))* resulted in returning shortest = 50 at the level of the main_loop to be conveyed to select via timeout.

It is here that I feel that the algorithm does not behave as expected. The value for shortest that gets returned to select() should be around 57.6 ms, and not 50, more particularly for this case when the value of credit is only one packet. The next burst should time out 57.6 ms later. The rationale underlying this expectation is explained in the following paragraphs.

Regarding the evolution of the algorithm, the set of specific circumstances are that a timer has just fired, the implied task handled and the daemon is about to yield to the O.S. for a maximum timeout duration. All events that could follow can be classified in two categories: (1) predictables ones such as the ones for which the daemon already has a context opened and that are timer paced; (2) unpredictable ones such as user requests for sending or receiving data, user requests for opening a context or packets received. Unpredictable events have no bearing on the timeout valued used to yield to the O.S. When they happen, the daemon is awakened for other reasons than the expiration of the timeout value.

The claim here is that the daemon should always yield to the O.S. for a predictable timer related timeout value (preferable the earliest timer to fire). In contrast, one observes that in the specific circumstances described above, the daemon always yields for a 50 ms timeout value (whatever the value of RTIMER for example, be it greater than or smaller than 50 ms). As long as a timer has not fired yet, shortest has a positive value and it is faithful to its semantics (one indicator of the timing of the next timer paced job to be done). However, once a timer has fired, shortest has a negative value and results in yielding to the O.S. for a timeout value of 50 ms when a timer related value should more appropriate. Waking up the daemon earlier (say 50 ms later rather than 57.6 ms for the illustrated case) implies that the timer will not have fired. Besides reducing understandability, the costs are a spurious wake up of the daemon and possibly sending the daemon back to sleep for a much longer period than the 7.6 ms left (50 ms is the minimum timeout used).

5.4 Concluding remarks about rate control with SandiaXTP

Figure 27 stresses a contrast between *ideal* rate control on the one hand, and the kind of *practical* rate control exercised by SandiaXTP on the other hand.

Ideal rate control [Figure 27 (a)]

RTIMER defines the beat for rate control activities, similar to the *year* for the recurrent rotation of the earth around the sun. For the ideal case, the timing up of RTIMER marks the end of an *idle time* (possibly of duration 0) period and the beginning of a new cycle of rate control related activities, which would include the following management activities, data transmission activities, and inactivity:

1. Replenishing *credit* to a value equal to *burst* so that no more data than what "ought to be" sent is effectively sent;

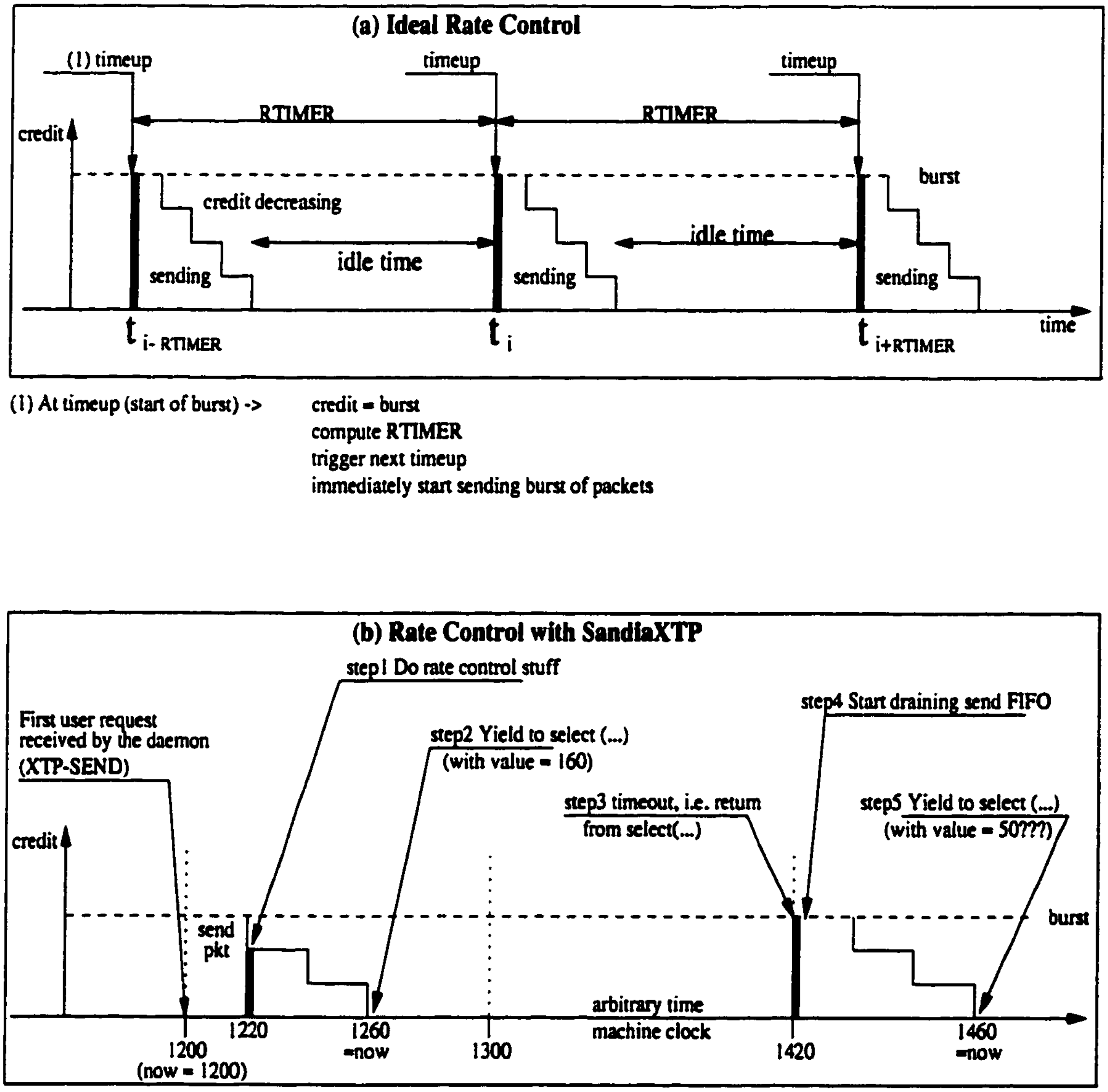
2. Computing a new value for RTIMER and starting some sort of alarm clock of RTIMER duration to trigger the beginning of the next cycle;

   Note: the fact that these management activities are of short duration, but nonetheless take some time, is indicated with **thick** vertical lines on Figure 27.

3. Transmitting the data packets, thereby consuming and decrementing *credit* until it becomes nil. This is shown by a staircase effect on Figure 27.

4. Finally, possibly a period of relative inactivity (packets could still be built and enqueued).

Being an ideal case, such details as the method used to account for the passage of time and the accuracy of the timing measurements are of no concern; we just presume that it is done somehow.


(a) Ideal Rate Control
(1) timeup
timeup
timeup
RTIMER
RTIMER
credit
burst
credit decreasing
idle time
idle time
sending
sending
sending
idle time
$t_{i-RTIMER}$
t_i
$t_{i+RTIMER}$
time
(b) Rate Control with SandiaXTP
step1 Do rate control stuff
step4 Start draining send FIFO
First user request received by the daemon (XTP-SEND)
step2 Yield to select (...) (with value = 160)
step3 timeout, i.e. return from select(...)
step5 Yield to select (...) (with value = 50???)
credit
send pkt
burst
arbitrary time
machine clock
1200 (now = 1200)
1220
1260 =now
1300
1420
1460 =now

(1) At timeup (start of burst) -> credit = burst
compute RTIMER
trigger next timeup
immediately start sending burst of packets

```
step1 //do rate control stuff
   now = timestamp() (=say 1220)
   r-timer = now + (burst/rate)
   (say: pdu = 1440 bytes;
       burst = 4320 bytes or 3 packets;
       rate = 20 Bpms;
       time to send 1 packet = 20 ms)
   // RTIMER = (burst/rate) = 4320/20 = about 200
   r-timer = 1220 + 200 = 1420
```

```
step2 shortest = check-timers()
       check-timers()
           shortest = min((r-timer-now),CTIMEOUT)
           shortest = min(1420-1260), ...) = 160
           return 160
       return (max(160, 50))
    return 160
```

```
step3 shortest = check-timers()
       check-timers()
           armed && expired
               rtimer-armed = 0
               credit = burst
           shortest = min ((r-timer-now, ...)
           shortest = min (0, ...)
           return 0
```

```
step4 r-timer = 1420 + 200 = 1620
    r-timer-armed = 1
```

```
step5 return (max(0,50))
```

Figure 27: Ideal *VS* SandiaXTP rate control

Practical rate control [Figure 27 (b)]

With SandiaXTP, rate control timing considerations are much more complicated than for the previously outlined *ideal* rate control case. Both the client and the daemon are user level processes, and they have no direct notion of the passage of time. For timing considerations, they have to rely on the operating system.

At this point, it is necessary to distinguish between **RTIMER** and **r_timer**. RTIMER is an interval of time; in practice, it is only useful to be added to whatever cumulative time it is now, and thus determine some cumulative time in the future (this due time is in effect r_timer) to mark the beginning of a next cycle. This practical way of doing things has many implications that are worth mentioning:

- Precisely at what moment r_timer is being computed matters. For the ideal case, we presumed that the end of a cycle is immediately followed by the beginning of another one. For the practical case, there might be an interval of time between the end of a cycle and the beginning of the following one. As it is used as a comparative reference to determine if it has timed up, the computing of r_timer effectively marks the beginning of a cycle, and its computation should be delayed until there are indeed data to send. Hence the necessity to uncouple the two events: *end of one cycle* and *beginning of the next one.*

- Once computed, r_timer is not conveyed as such to the operating system. Between the moment r_timer is being computed, and the moment the daemon yields to the operating system, the data packets have to be sent, and this activity takes time. When the daemon is ready to yield to the operating system, a new timestamp is taken and it is the interval between r_timer and now (r_timer-now) that is reported to the operating system.

- When the daemon wakes up again for whatever reason, it will have to check again which timer has timed up, if any, and this implies that the discovery is made after the fact. How much later depends on practical considerations such as how many contexts are active and the level of activity.

Figure 27 (b) illustrates more precisely the mechanisms used by SandiaXTP for rate control, presuming the aforementioned algorithm and send scenario. As the detailed scenario was covered previously, only the most salient points are summarized here:

- When the daemon resumes execution after a TIMEOUT event (step 3), method satisfy() is called;
- As there is no credit available, method drain_snd_fifo() is called for the first time, but does not follow through;
- Method check_timer() is called, which results in replenishing the `credit` equal to `burst`; a value for `shortest` is also returned based on the previous `r_timer` value, which implies a 0 value or most likely a negative value (presumably an updated value for `r_timer` cannot be used at this point, as it would imply starting the next cycle, and there is no evidence yet that there are data to be sent - we are only sure of the timeout).
- Method drain_snd_fifo() is called for a second time, which results in recomputing a new value for `r_timer` (if we have reached this point, there are indeed data to send - see step 4).
- Ultimately (step 5), a value of 50 ms (a non-timer related value) is passed to select().

This tracing of the rate control algorithm is not fulfilling our expectations. It was judged sufficient to continue inquiring into it, which is the object of the next chapter.

More general remarks about rate control SandiaXTP

Finally, the type of rate control exercised by SandiaXTP, given the user level implementation architecture used, occurs between the memory space of the daemon process and the memory space of the operating system. The sending of a packet by the daemon results in a system call such as *sendto()*, and the choice of the precise moment when the packet leaves the machine is relinguished to the operating system and the underlying network technology. The select () system call itself is blocking, but when it returns it does not necessarily mean that the O.S. has effectively sent the packet. When opening a socket, a kernel send buffer is associated with it (SO_SNDBUF, probably of size 32768 bytes). This buffer space allows the Operating System the possibility to do the send operation asynchronously, depending on its own priorities.

6 Changes to SandiaXTP rate control mechanisms

The goal of this chapter is to motivate and to expose some changes made to rate control as exercised by SandiaXTP. Whereas the method used to abstract the rate control algorithm in Section 5, and to trace its behavior with a minimal scenario, was mainly of a speculative nature, the approach used in this Chapter is essentially experimental. First, are presented the effects of additional traces aimed at monitoring the behavior of the algorithm and recreating the minimal scenario presented in Section 5.3. As these changes led to further observations, many other modifications aimed at enhancing the algorithm are presented.

6.1 Tracing the behavior of the rate control algorithm

The hypothetical scenario presented in Section 5.3 that led to apparently unexpected behavior needed to be exposed experimentally in order to test its accuracy and to expose its repercussions. Consequently, many monitoring changes were introduced in the source code at key points as per the view of the algorithm presented in Figure 24. In addition to numerous timestamps, these changes consist of the following:

- immediately before returning from method check_timers() (file XTPcontext_gen.C), display the value of variable `shortest` that is being conveyed to the scope of method satisfy(). This is the lowest level towards returning a time duration value to be used as the `timeout` argument value to system call select().

- immediately before returning from method satisfy() (file XTPcontext_manager.C), display the value of variable `shortest` that is being conveyed to the scope of the main_loop.

- immediately before yielding to the operating system through the select() system call, display the value of variable `timeout` (file mtldaemon.C, method wait_on_input()) conveyed to select(), and a timestamp to record the cumulative time at this moment.

- immediately after returning from system call select() (file mtldaemon.C, method wait_on_input()), display the reason of the wake up (**TMOUT**, **USERREQ**, or incoming **PACKET**), and take again a timestamp.

The outcome of these monitoring changes led to the confirmation of the anticipated but unsatisfactory behavior as traced in Section 5.3. Here follows some trace excerpts that confirm this unsatisfactory behavior:

```
...up on TMOUT 146 ms later, now=2659892808

(26) 0x3780 RTIMER expired now=2659892808

->Bubbling up shortest to select()<-
check_timers()-returning shortest_orig=-4, now=2659892809
(27) 0x3780 Sending packet from send FIFO, seq = 2880

->RTIMER - start_rtimer()
 RTIMER=144, r_timer=2659892953, now=2659892810

(28) 0x3780 DATApacket BEING SENT at 2659892811

satisfy() - returning shortest_orig=50, now=2659892812

->shortest (top level do { }while(!ds) )= 50, now=2659892812
timeout (before select(...,&timeout))= 50, now=2659892812
daemon going to sleep.......
```

Indeed, on a TMOUT case (with rate=10, i.e., RTIMER = 144), a negative value for **shortest** is being returned, and 50 is eventually bubbled up to select().

Whereas the hypothetical scenario was halted at this point, the execution of the program goes on. At worse, the 50ms would eventually time out, and control would return to the deamon. In the present case, the deamon is awakened much earlier than after 50ms, whether by one XTP_SEND user request or an incoming control packet. On this later pass, the updated value of r_timer is properly taken into consideration, and method check_timers() returns a non-negative value.

6.2 Imprecision on select() return timeout / MAXANTICIPATION

The changes introduced to monitor the behavior of the algorithm and test the anticipated unexpected scenario led to the observation that select() would often return after a shorter timeout period than the one specified. For instance, the daemon could yield to select() with a `timeout` value of 50ms, but control would come back to the daemon after a shorter period of time, say 45ms. Naturally, a check reveals that the timer has not expired and control would return back to the operating system via the select*() system call with possibly a timeout value of 50ms. In this case, the discovery that the timer has expired could be made well after it has expired, depending on the timing of the next event that provokes a return of control to the daemon. Here follow traces that display this imprecision effect on the timeout value of select():

```
->shortest (top level do { }while(!ds) )= 78, now=2659892582
timeout (before select(...,&timeout))= 78, now=2659892582
daemon going to sleep.......
...up on TMOUT 76 ms later, now=2659892658
```

A similar situation could happen when the daemon is serving many contexts. Once the daemon has finished acting on behalf of the context targeted by the user request, and before it yields to the operating system, the daemon cycles through all active contexts for work to do. A check on a timer that is soon to expire, say 4 ms later, would reveal that it has not expired, and control could return to the O.S. via the select() system call with possibly a timeout value of 50ms. For both cases, the discovery that a timer has expired could be made well after it has effectively expired.

To care for the imprecision on the select() return timeout value, and also for those *soon to expire* timers, the idea of allowing for an *anticipation margin* is introduced. With the original code, a timer is considered to have expired only when the current cumulative time (nŏw) at the moment of the check is greater than `r_timer`. Here follows code excerpts that illustrate the nature of the checks made and the criterion used (file XTPcontext_gen.C, method check_timer()):

```
    // Check the RTIMER
```

```
  if (r_timer_armed) {
    if (!lt32(now, r_timer)) {
        stop_rtimer();
    }
     shortest = min((int)(r_timer - now), shortest);
  }
```

With the revised code, and taking into consideration the anticipation margin, r_timer is considered to have expired if the time duration left before it effectively expires is less than a MAXANTICIPATION margin. The basic philosophy is to accept the limitations of the implementation architecture with regards to timing considerations, but attempt to ease its implications. In effect, it is equivalent to discharging now some *soon to mature* tasks, or else they might get done much too late anyway. A tentative value of 10ms for the MAXANTICIPATION margins used to compensate for all cases of imprecision on select() return timeout value, and yet maintain reasonable rate control. Here follows code excerpts that illustrate the changes:

```
   //  Check the RTIMER
   if (r_timer_armed) {

     //---icici---
#define MAXANTICIPATION 10
     //it has expired || it will expire soon
     if ( (!lt32(now, r_timer)) ||
  ((r_timer > now) && ((r_timer-now) <= MAXANTICIPATION))) {
       //---icici--

         stop_rtimer();
     }
      shortest = min((int)(r_timer - now), shortest);
   }
```

6.3 Select() minimal timeout value / SELECT_FLOOR

Given the original implementation, a lower limit of 50ms is hardcoded to guarantee that the daemon never yields to the operating system via the select() system call with a `timeout` value of less than 50ms. For instance, if a timer will expire in 15ms, the daemon yields to select() with a `timeout` value of 50ms, and the discovery that the timer has expired could be made only about 35ms after it has expired. The consequence of this practice is a decrease in the quality of the type of rate control effectively exercised by the SandiaXTP implementation. However, the designer has justified this choice on the ground that with a lower `timeout` value than 50ms, there might be instances when the select() system call never returns, thereby stalling the execution of both the client and the daemon processes; which is certainly a greater evil than some decrease in the quality of the rate control exercised.

As an intermediary step towards possibly converting this low limit into a tunable parameter, the symbolic constant `SELECT_FLOOR` is being introduced. Programming wise, this change is trivial, but the shift in approach is considered conceptually meaningful. The intent is to experiment with different `SELECT_FLOOR` threshold values, starting with a larger value than 50ms, and then using much lower values than 50ms. The expectations are: (1) a better mapping of the repercussions of this limit on the quality of rate control that can be achieved given the SandiaXTP user level implementation strategy; (2) to determine in what specific circumstances system call select() would never return when given a lower `timeout` value than 50ms. Here follows code excerpts showing the situation before and after the change (file XTP-context_manager.C, method satisfy()):

Original implementation:

```
return(max(shortest, 50));
```

Modified code (the other changes are explained later):

```
#define SELECT_FLOOR 50

...

shortest = max( (dueTime - now), SELECT_FLOOR);
```

6.4 Linked list of timers - principles

The fact that method check_timers() often returns a negative value for variable *shortest* has the consequence that the 50ms non-timer related related `timeout` value is passed to the select() system call. The least inconvenience of this occurence is a spurious call to select(), as the 50ms soon expires anyway. However, the consequences could also become more devastating, a exposed through the following scenario:

- method check_timers() returns a negative value, and 50ms is bubbled up to select();
- select() returns about 50ms later, but the timer has not expired yet (say because RTIMER is slightly larger than 50ms, ex: 1440/23=62ms);
- the daemon yields again to select() for another 50ms.

In this case, there is not only a spurious call to select, but the task of sending the packet is also much delayed. This type of bad scenario cannot be cured with the MAX-ANTICIPATION margin, unless the value of this margin is incremented. Clearly, the solution is preferably not on the side of a larger anticipation factor, as this tactic would amount to discharging too many tasks too early.

Primarily to help understandability of the SandiaXTP rate control algorithm, but also secondarily to care for the type of scenarios as exposed above, it was decided to implement a linked list of timers, including all XTP timers (i.e., `CTIMEOUT`, `RTIMER`, `STIMER`, `WTIMER`, `CTIMER`), sorted in ascending order of cumulative due time (`r_timer`) to fire in the future. The idea is similar to the UNIX callout queue implemented in the kernel, and which records functions that the kernel must invoke at a later time.

At the origin of the SELECT_FLOOR problem lies the fact that method check_timers() uses the expired value of `r_timer` to compute shortest and thus return a negative value for shortest (`shortest = r_timer-now`). Subsequently, and before the daemon yields to select(), an updated value for `r_timer` is most likely computed, but it has no effect on the value of `shortest` taken into consideration immediately before returning from method satisfy(). With a linked list of timers, this problem disappears, as the head of the linked list of timers is read only immediately before returning from method satisfy(), and the updated value for r_timer is taken into consideration.

Though this linked list of timers could mean a whole re-architecture of the code, it was done to intermesh as seamlessly as possible with the existing code and with minimal changes, taking advantage of the existing situation. For instance, methods are called when a timer is being started or stopped (ex: methods start_rtimer and stop_rtimer(), file XTPcontext_gen.C). When those method are invoked, a call is made to enqueue or to dequeue a timer. Here follows examples of such calls (file XTPcontext_gen.C, method start_rtimer()):

```
//---icici---
DAEMON->d_cm->enq_timer(key(), RTIMER, r_timer);
//---icici---

//---icici---
DAEMON->d_cm->dq_timer(key(), RTIMER, r_timer);
//---icici---
```

The linked list of timers is consulted only at the end of method satisfy(), immediately before returning a timeout value to be used in the scope of the main loop. This ensures that timer related values are taken into consideration before yielding to the operating system, and thereby breaking the unsatisfactory circle described previously.

As mentioned, most of the original behavior of the algorithm remains intact, with shortest being returned as usual up until the end of method satisfy(), where it is bypassed by reading from the linked list of timers. Here follows some trace displays that show how the original and the updated mechanisms intermesh, and where the reading from the linked list result in returning a different timeout value at the scope of the main loop:

```
->Bubbling up shortest to select()<-
check_timers()-returning shortest_orig=-5, now=2731988849

(35) 0x5580 Sending packet from send FIFO, seq = 5800
```

```
->RTIMER - start_rtimer()
 RTIMER=57, r_timer=2731988907, now=2731988850

->enq_timer() called
key=0x5580, timer_type=RTIMER, due_time=2731988907

->print_timers_onq() called:
# of timers onq=2  now=2731988850
key=   type=    due_time=  derived delta time to fire (ms)=
0x5580 RTIMER   2731988907 57
0x5580 CTIMER   2735588531 3599624

(36) 0x5580 DATApacket BEING SENT at 2731988851

satisfy() - returning shortest_orig=50, now=2731988852

->SATISFY() - bypassing shortest_original

->read_first_timer_onq() called
timers_list_head->due_time=2731988907
time left to fire: 54 ms

->satisfy() - returning shortest_NEW=55, now= 2731988852

->shortest (top level do { }while(!ds) )= 55, now=2731988853
timeout (before select(...,&timeout))= 55, now=2731988853
daemon going to sleep.......
```

The code at the end of method satisfy that produced the value returned is as follows:

```
File XTPcontext\_manager.C
...
word32 XTPcontext_manager::satisfy() {
    ...
```

```
//if (shortest != -1)
//   return(max(shortest, 50));
//return(shortest);

//---icici---old way
int shortest_orig;
if (shortest != -1) {
   shortest_orig=max(shortest, 50);
} else {
  shortest_orig=shortest;
}
if (TRACE && trace) {
  log_print("\t satisfy() - returning shortest_orig=%d, now=%u \n",
    shortest_orig, (word32) DAEMON->timestamp());
}
//---icici---old way

//if (shortest != -1)
//   return(max(shortest, 50));
//return(shortest);

//---icici---new way

now = (word32) DAEMON->timestamp();
if (TRACE && trace) {
  log_print("\n->SATISFY() - bypassing shortest_original\n");
}
// read_first_timer_onq() returns the next absolute clock
// due_time, we have to substract now to get interval

if ((word32) (dueTime = read_first_timer_onq()) == 0) {
  // meaning there is no timer onq
  shortest = -1; // i.e., impose POOL_FREQ
```

```
    } else if ((int) (dueTime - now) < 0) {
      //it has a negative value, return select_floor for now!!
      shortest = select_floor;
    } else {
      //it has a positive value
      shortest = max( (dueTime - now), select_floor);
    }

    //print_timers_onq();
    if (TRACE && trace) {
      log_print("->satisfy() - returning shortest_NEW=%d, now= %u \n\n",
      shortest, now);
    }
    return (shortest);
    //---icici---new way

} // end satisfy()
=====================
File XTPcontext_manager.h
  ...
  int select_floor;
  ...
====================
File XTPcontext_manager.C
...
XTPcontext_manager::XTPcontext_manager(int num_contexts)
      : context_manager(num_contexts) {

    //---icici---
    select_floor = SELECT_FLOOR;
    //---icici---
    ...
====================
XTPtypes.h
...
#define SELECT_FLOOR 50 //for example
...
```

As one can observe, the logic used to return a timeout value at the scope of the main loop, from the end of method satisfy(), still considers the possibility that a timer could have expired (i.e.,dueTime-now is negative) when reading from the head of the linked list of timers. Such an occurence would indicate that a timer has expired, but the corresponding stop_?timer() method has not been called yet. It could happen in rare circumstances when a daemon is very busy serving many contexts, with many timers coming to maturity at short interval of time (say 1 or 2ms).

As per the design of the algorithm, the daemon checks the timers of all contexts for work to be done. It could happen that when a timer is being checked, it has not expired, but by the time the other timers of other contexts have been checked, and the corresponding work done, that it has expired as revealed when reading from the head of the linked list of timers.

At this point, i.e., at the end of method satisfy(), the natural flow of the algorithm is to return to the scope of the main_loop and yield to the operating system via the select() system call. Rather than doing so, the daemon could remain in control and method satisfy() could be called again recursively (so to speak), as if select() had returned on a TMOUT case, to serve the context with the expired timer. However, at the end of this next pass, another timer may have expired, and this recursive process could linger for long.

Besides the programming difficulties involved by this recursive approach, it was felt preferable not to disrupt the logical flow of the algorithm and let the operating system regain control as designed as soon as possible. For the proper working of the daemon, control must switch seamlessly between the operating system that is responsible for event monitoring and notification, such as incoming user requests or incoming packets, and the daemon who is reponsible to discharge the tasks consequent to the occurence of these events.

As will become evident later, the SELECT_FLOOR effect is greatly attenuated by these continuously upcoming events, with returning from select() well before the specified timeout periods. Consequently, SELECT_FLOOR is returned at the scope of the main loop. Finally, let us mention that such a case is not even visible with the unmodified algorithm.

To conclude this subsection, we summarize with two points:

1. Given the fact that select() often returns well before the specified timeout period, the difference between the original algorithm, with check_timers() *often* returning negative values for shortest and 50ms being bubbled up to select(), and the linked list of timers *sometimes* returning negative values and SELECT_FLOOR being bubbled up to select(), is one of degree only; the latter fostering understandability of the algorithm and also improving efficiency and the quality of the type of rate control exercised.

2. The idea of not disrupting the flow of the algorithm and let the operating system regain control as soon as possible for event notification also greatly precludes another idea which would consist of using *busy-wait* to let the daemon stand idle until *a soon to expire* timer would expire. This approach would also disrupt the normal working of the daemon, by postponing the handling of other events, and could also lead to abuse as the end of a *busy-wait* period could also be followed by another one. The previously explained decision to use a MAXANTICIPATION margin is in accordance with this "least disruption of event notification and handling" principle.

6.5 Linked list of timers - Implementation details

The goal of this subsection is to explain with more detail how the additional source code needed to implement the linked list of timers is integrated with the source code of the MTL base classes and SandiaXTP derived classes.

All the changes made to the source code are indicated with the keyword `//---icici---` appearing before and after the changed code. As such an expression is unlikely to appear in the regular code, it can be used as a regular expression with a text editor to search for the changes rapidly in a large file. Preferably, the changes are made in the form of additions; leaving intact, but commenting out the original source code segment.

Though the initial goal was to introduce changes only to the SandiaXTP derived classes, it soon became evident that some changes also needed to be made to the MTL source code, essentially virtual methods added to the context_manager base class.

Because the linked list of timers is for all contexts, and this is precisely the outlook of the context manager, the data structure for the linked list of timers, and the methods needed to manage it are part of the XTPcontext_manager subclass, with some virtual methods (the ones that need to be called from another class) added to the context_manager base class. The fact that method satisfy() belongs to the XTPcontext_manager subclass, and that the linked list is consulted at the end of method satisfy() before returning to the main_loop is another reason to attach the linked list of timers to the XTPcontext_manager subclass.

However, many of the methods needed to manage the list are called from the XTPcontext subclass, as timers are context specific items and methods start_?timer() and stop_?timer() belong to class XTPcontext. The manner used for insertion of timers in the queue, or extraction are explained in Section 6.4. Hence, most methods related to the management of the linked list of timers are public and are called from subclass XTPcontext (such as enq_timer() or dq_timer()).

There are also some linked list management methods that are kept private to subclass XTPcontext_manager. One instance is method read_first_timer_onq(), which is called at the end of method satisfy() to help determine the timeout value to be eventually bubbled up to system call select().

For the time being, the nature of the changes are not considered to be optimized. Should such an approach using a linked list of timers be considered for including in a standard SandiaXTP release, some changes could be done differently. Suggestions are presented in the following paragraphs.

6.5.1 Timer types

```
(file XTPtypes.h)
//---icici---
// For the timer linked list operated by the context manager
typedef enum {
  CTIMEOUT, //CTIMEOUT
  RTIMER,   //RTIMER
  STIMER,   //wtimer
  WTIMER,   //WTIMER
  CTIMER   //CTIMER
}timerType;
//---icici---
```

6.5.2 Timer items

```
(file XTPcontext\_manager.h)
//---icici---
typedef struct timer_item {
  word64 c_key;               // the key of the context
  timerType timer_type;       //CTIMEOUT, RTIMER, STIMER, WTIMER, CTIMER
  word32 due_time;            //a timer to expire in the future
  struct timer_item* next;    //the next item in the list
} timer_struct;
//---icici---
```

6.5.3 Virtual methods

```
(file context\_manager.h)
public:
   ...
//---icici---
virtual void enq_timer(word64 key, word32 type, word32 time) = 0;
virtual void dq_timer(word64 key, word32 type, word32 time) = 0;
virtual void cleanup_context_timers_onq(word64 key) = 0;
virtual word32 test_if_timer_onq(word64 key, word32 type, word32 time)=0;
//---icici---
```

6.5.4 Redefinition of virtuals...

```
(file XTPcontext\_manager.h)
private:

//---icici---
timer_struct* timers_list_head=NULL;
int timer_items_onq=0;

word32 read_first_timer_onq();
void print_timers_onq();
char* timer_type_to_string(word64 tt);
//---icici---

public:

//---icici---
void enq_timer(word64 key, word32 type, word32 time);
void dq_timer(word64 key, word32 type, word32 time);
word32 test_if_timer_onq(word64 key, word32 type, word32 time);
void cleanup_context_timers_onq(word64 key);
//---icici---
```

Methods print_timers_onq() and timer_type_to_string() are convenience ones that become effective when the daemon is started with the **trace** option. The first is called after inserting a timer item in the linked list, or after removing one, to show the state of the queue at this particular moment. It is most useful to understand what is going on. The second method is a complement of the first one and is used to convert the numeric representation of a timer to its string equivalent (ex: timer type 0 is displayed in the trace as CTIMEOUT). Here follows a display example of these methods:

```
->print_timers_onq() called:
# of timers onq=3  now=2831287449
key=   type=    due_time=  derived delta time to fire (ms)=
0x1c0 RTIMER   2831287503 54
0x1c0 WTIMER   2831287866 363
0x1c0 CTIMER   2834887368 3599502
```

The purpose of method test_if_timer_onq() is explained in the next subsection. We now explain the raison d'être of method cleanup_context_timers(). As a result of using method print_timers_onq(), it was observed that when the daemon is kept alive and many application program instances are started in sequence (i.e., many contexts are created), then the last WTIMER would remain in the linked list even when one particular instance of the application program has halted. The inference was that method stop_wtimer() is not called on this last instance of the WTIMER.

The solution was to invent method cleanup_context_timers(), which is called when a context becomes zombie (i.e., it is called from method start_zombie(), file XTPcontext_gen.C). It ensures that all timers that belong to the context, with the exception of the CTIMEOUT timer, are removed from the linked list of timers.

6.5.5 Enqueuing a timer of type WTIMER

```
(file XTPcontext\_gen.C)
void XTPcontext::start_wtimer(word32 factor) {
...
//---icici---
// make sure we do not end up with 2 or more WTIMER onq
if (DAEMON->d_cm->test_if_timer_onq(key(), WTIMER, w_timer)) {
  DAEMON->d_cm->dq_timer(key(), WTIMER, w_timer);
}
//---icici---

word32 now = DAEMON->timestamp();

w_timer = now + duration;
w_timer_armed = 1;

//---icici---
DAEMON->d_cm->enq_timer(key(), WTIMER, w_timer);
//---icici---

} // end start_wtimer()
```

Again as a consequence of using method print_timers_onq(), it was observed that more than one instance of the WTIMER for one context could be present in the linked list of timers. Typically, one would never be removed; it would expire, but remain at the head of the queue and wreck the operation of the queue. Upon further inquiry, it was observed that occasionally, such as when the output data rate is high (30 Bpms and above), a subsequent WTIMER for the same context would be started without stopping the previous one. The inference is that method start_wtimer() is called twice, but without any intervening call to method stop_wtimer(). Eventually, the same *test-and-set* solution had to be adopted for the CTIMEOUT timer also (file XTPcontext_gen.C).

With the unchanged implementation, this occurence creates no apparent difficulty, as the value to the w_timer is just overwritten. However, with the linked list approach, each insertion has to be matched with a removal. The purpose of method test_if_timer_onq() is part of the tactic used to solve this problem. Before a timer of type WTIMER is inserted in the queue, a check is made (as shown in the above code excerpt) to determine if another one is already in the queue. If so, it is removed before inserting the new one.

Currently, the test is made from the XTPcontext subclass, which means that method test_if_timer_onq() that belongs to subclass XTPcontext_manager has to be made public. However, this test could become a private affair to the XTPcontext_manager subclass. The reason for the current state of affairs is that method dq_timers() needs three parameters: the key, the type of the timer, and the timer itself. If only method enq_timer() was called from the XTPcontext subclass, knowledge of the previous timer would be lost, and method dq_timer() as currently implemented would not work.

Given a confirmation by the XTP designers that only one timer of a type could be active for any context at any given moment, method dq_timer() could be reimplemented with two parameters only: i.e., the key and the type. These two arguments only are neded to conduct the test and remove a timer item from the queue. This is one example of code optimization that could be introduced in case of longer term interest for the linked list approach for SandiaXTP.

6.6 Changes specific to multicast experiments

As explained at the beginning of the report, the evolving horizon of the work underlying this report has already been the realization of multicast rate control experiments using the Internet/Mbone environment. To perform this task, and as a result of unsuccessful attempts using the Internet, a special purpose multicast testing program was developed and also some changes implemented to SandiaXTP, which are reported in this section.

The realization of the live multicast data transfer experiments from the Concordia University HSP Lab was a long term endeavor with many preparatory steps. For instance, the Sun Solaris2.5 Operating System supporting IP multicast at the network level was installed, and some multicast testing programs were developed (**udip**, a C++/UDP based testing program is presented in Section C.4 at the end of this report).

The TTL, or Time-To-Live, is a settable key factor for performing multicasting. It defines how far out from the sender an IP multicast addressed packet will be propagated by the mrouters to reach awaiting receivers. As the early multicast attempts on the Internet were not successful, first doubts went to the TTL factor, which was set to a default value of 10 in MTL file MTLtypes.h. After some trials to effect a TTL change from the user level testing program only, conveyed to the daemon through the xtp_config structure and the xi.reg() method, a decision was made to change the default value, and have the effective O.S. used TTL value displayed when the daemon starts. Here follows some excerpts from the source code outlining these changes:

`File MTLtypes.h`

```
#define MCAST_DIAMETER       127   // Multicast transmission diameter
                                   // was 10 before LH Nov. 1997
=========
File udp_del_srv.C
setup()
#ifdef IP_ADD_MEMBERSHIP

  byte8 ttl = (byte8)MCAST_DIAMETER;

  setsockopt(daemon_id, IPPROTO_IP, IP_MULTICAST_TTL,
                 (char*)&ttl, sizeof(ttl));

  //-------------------------------------------------
  byte8 ttlos;                           // added LH Dec. 1997
                                         // to check effective O.S. ttl
  int optlen;
  optlen = sizeof(ttlos);

  if (getsockopt(daemon_id, IPPROTO_IP, IP_MULTICAST_TTL,
                (char*)&ttlos, &optlen) < 0) {
    fprintf(stderr, "ERROR: getsockopt(ttl) failed in udp_del_srv\n");
    return(0);
  }
  fprintf(stderr, "with TTL: %d\n", ttlos);
  //-------------------------------------------------
#endif /* IP_ADD_MEMBERSHIP */
```

After recompiling the MTL libary and SandiaXTP, the daemon now starts with the following type of message:

```
Starting XTP Daemon (3519) on forest Sat Dec 20 20:12:56 1997
with TTL: 127
```

7 Organization of the experiments

7.1 Task: reliable data transfer

The global task consists of: (a) conducting unicast and multicast data transfer experiments; (b) collecting "meaningful" monitoring data that will hopefully help devising a rate control policy for XTP. The data transfer part consists of sending a large amount of data (one megabyte). More detailed descriptions of the subtasks needed to discharge this global task are provided in the next subsections.

7.2 Topology

Figure 28 shows the logical network connectivity including only the subnetworks 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.

7.3 Experiments structuring for data organization

To conduct the experiments in an orderly fashion, and also to organize the data gathering and presentation tasks, some structure is needed.

Data transfer job

At the most primitive level, we have the activity of reliably sending the specified amount of data from one sender to possibly multiple receivers. Timewise, this activity extends from the moment the application program transfers the first buffer full of data, until the moment the sender has received the last acknowledgment from the last receiver. The send rate is specified at the beginning and remains fixed for the whole data transfer job.

Data transfer session

A data transfer session consists of many data transfer jobs; in fact as many as we wish to include discrete values in our range of test send rates. The task to discharge

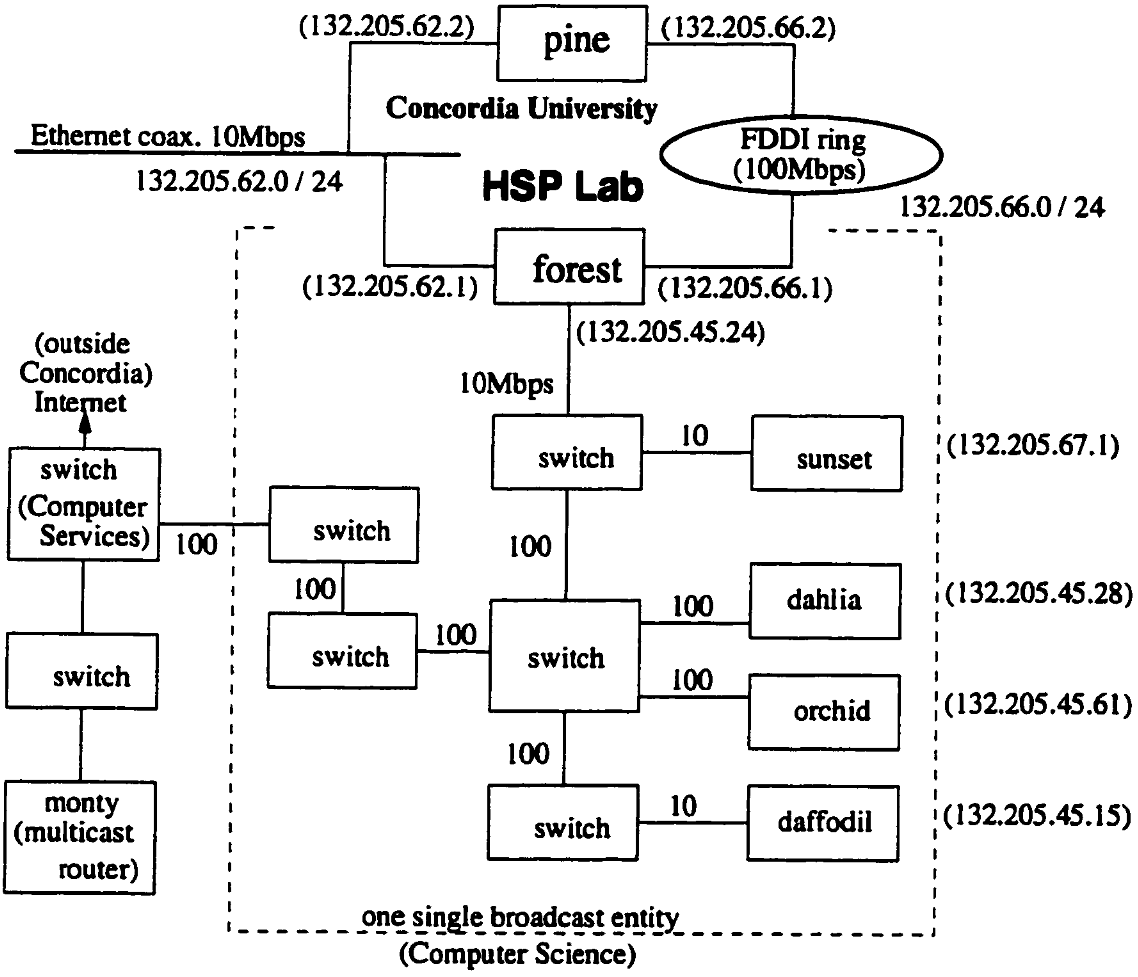


Figure 28: Logical network connectivity

within a data transfer session is to repeatedly transfer the specified amount of data, starting with the first data transfer job at the lowest send rate, then the next one at the next higher send rate, and so on until the last data transfer job gets completed at the highest send rate.

The delay between the moment a data transfer job gets completed (say at rate=r) and the moment the next data transfer job gets started (say at $rate = r_{prev} + \Delta r$) should be as small as possible. The output of a successful data transfer session (also called an **experiment unit**) produces all the data necessary to plot one saturation curve, such as the one shown in Figure 30; with every point of the curve being the result of one "in session" data transfer job. The window size remains constant throughout.

7.4 Data gathering and presentation

Naming scheme: For data gathering and referencing purposes, a labeling scheme is used to uniquely identify the data related to each experiment unit. For the reliable multicast experiments, the slowest receiver governs and the data for all members are organized into one logical unit identified with a single label.

The generic label used is:

u|mxSN where:

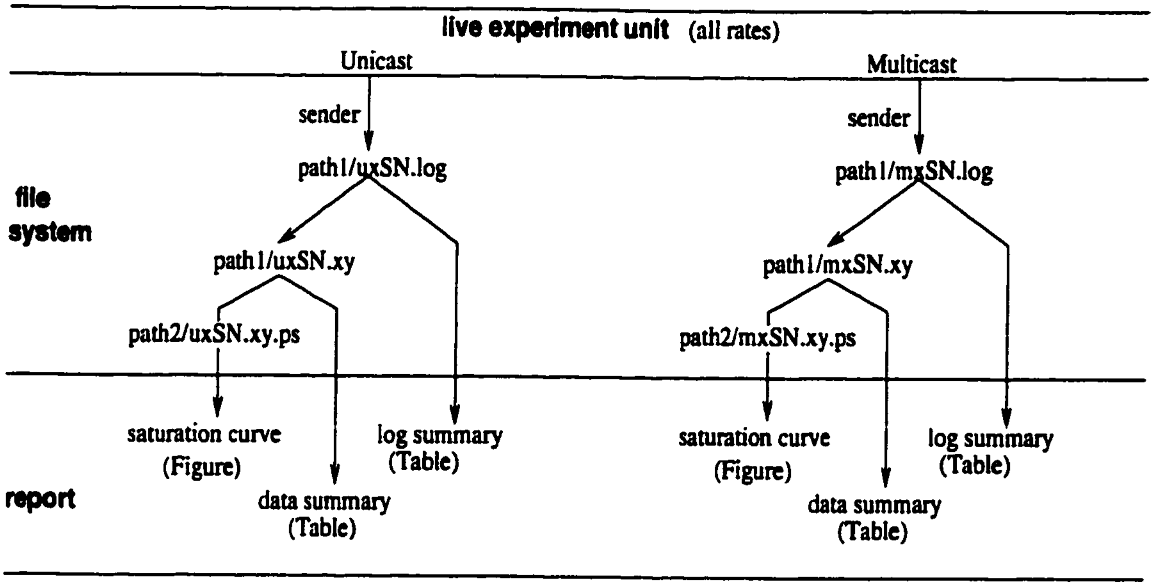
ux - stands for *unicast experiment unit;*
mx - stands for *multicast experiment unit;*
SN - is a unique 2 digit sequence number, such as 00;

Example:

ux00 -a prefix used to organize and refer to the data for one reliable unicast experiment
ux00.log -the log file for experiment unit ux00
ux00.xy -the files containing the xy coordinates of the saturation curve (xtractXY ux00.log > ux00.xy)
ux00.xy.ps -the ps file for the saturation curve (xgraph ux00.xy -saved as ps)
Table ux00.log - log summary -derived from ux00.log and included in this report
Table ux00.xy - data summary -derived from ux00.xy and included in this report
Figure z: ux00 saturation curve

Use of the naming scheme

Figure 29 illustrates the naming scheme at work for the purpose of gathering and presenting the data. For one experiment unit, the basic source file is the **log** file, out of which all the other files, Tables and Figures documenting the experiment are derived. These consist mostly of a log summary and data summaries in tabular as well as graphical forms (saturation curve) presented in Chapter 8.



Notes: path1=/mnt/jwa/jwa1/grad/harveyl/experiments/withSXTP/logNxy/
path2=/mnt/jwa/jwa1/grad/harveyl/report/expResults/

Figure 29: Use of the naming scheme

The log file is a plain ASCII file and is not included as such in the report. For the most part, it is filled automatically from instrumentation messages through redirection of standard output of the testing program. The details of the instrumentation messages and the redirection are covered later in the report. Here follows a verbatim excerpt to illustrate the type of information contained in a log file.

```
Thu Dec 18 09:22:54 1997
mmetric starting (5441) on orchid
mode=MULTICAST, off_load=10 Bpms, WinSiz=102400 bytes,
mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440
                                        -a 51200 -o 250 -W 102400 -c 10

Transmitting to 239.159.100.40
51200 bytes using buffers of size 1440 bytes

Timing: 5908 ms
Throughput: 8.666 Bpms - Bytes per ms
xy: 10 8.666
```

```
Throughput: 0.069 Mbits/sec
Number of calls: 36
Latency: 164.111 ms/call

Sent 51200 bytes
Thu Dec 18 09:23:01 1997
=====================================
```

Table 5: ux00 log summary

Table ux00.log - log summary (at sender)

Date: 97.12.18 | **Start:** 09:22:54 | **End:** 09:23:01 | **Duration: 07 min:** | **Mode:** Multicast
Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 51200 -o 250 -W 102400 -c 10.. mmetric -R 239.159.100.40 -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

A typical sample of a log summary is shown in Table 5. The log summary is subdivided into four parts: (1) the top part gives contextual details about the experiment unit, such as the date it was done, the time of day, etc.; (2) the second part provides a synopsis of the main parameters of the experiment unit, such as whether the unmodified or modified version of SandiaXTP was used, the values of the `SELECT_FLOOR` and `MAXANTICIPATION` daemon parameters, whether harmonization of burst and rate values was used or not, and any other pertinent detail regarding the development of the experiment; (3) the third part lists the commands used for starting the testing programs (i.e., the "user") at the sender and at the receiver(s); (4) finally, the fourth part lists the machines used for the experiments.

Table 6 shows a typical tabular presentation of the data for one experiment unit. The **load** label corresponds to the *offered load* (i.e., the outrate at the sender), and is presented in the native unit of the SandiaXTP implementation for rate (i.e., in bytes per millisecond - Bpms). The offered load (load) corresponds to the **x** axis of

Table 6: ux00 data summary

tput(y) (Bpms)	7.111	10.222	12.333	15.444	25.555
load(x) (Bpms)	10	15	20	25	30

its corresponding experiment saturation curve. The **tput** label corresponds to the throughput and transposes on the y axis of its corresponding saturation curve. In Section A, the **tput** and the **load** labels are not repeated but can be inferred easily from the relative position of the data, and the number of precision digits (x - 10, y - 7.111).

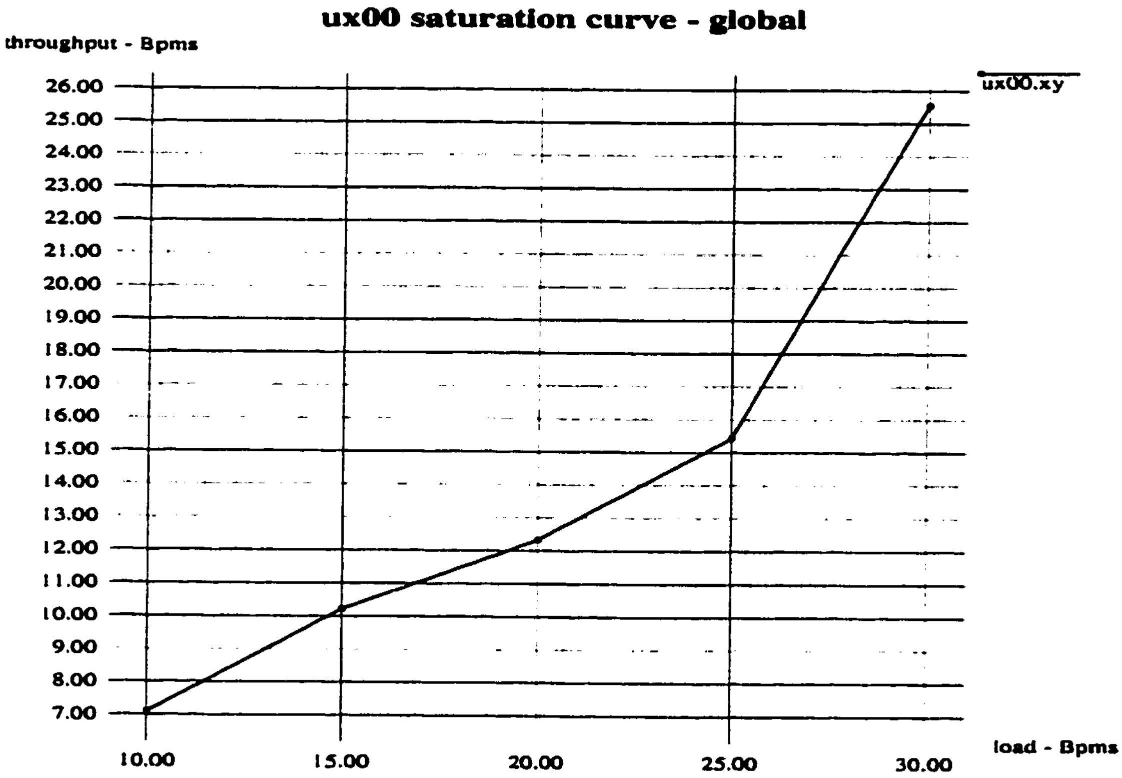


Figure 30: ux00 saturation curve

The throughput is computed at the sender. A typical sample of a saturation curve corresponding to the hypothetical list of **xy** coordinates for the reliable experiment shown in Table 6 is presented in Figure 30.

7.5 Programming for implementation of experiments

User level testing programs (mmetric & mbulk)

The user level testing programs used for the data transfer experiments (multicast as well as unicast) are an extension of the ones used for the HSP Lab May 1997 unicast experiments ([SUL]). These are in turn derived from the example program suite provided with the SandiaXTP implementation, namely metric.C, bulk.C and common.h.

As we inherit much of their functionalities, further indications are provided about the design of these testing programs, as originally conceived by the designers of SandiaXTP. Program **bulk** is compiled from source file bulk.C and common.h. Here follows some excerpts of the README file concerning program **bulk**:

> This program is the sender and receiver (issue "bulk -?" to see the flags to set) for a bulk data transfer. Information is printed to the screen as the data is transferred, so the timing has more to do with standard I/O than data transfer. Multicast is allowed. Since bulk does reliable multicast, it is important to remember to send an SREQ in the FIRST packet, so at least the "-f" switch must be used.

Program **metric** is compiled from source files metric.C, bulk.C and common.h. The additional code in file metric.C consists of very few lines that have the effect of silencing the numerous messages sent to the screen (stdout) by bulk and also to trigger the computation of throughput statistics. Here follows what the README file states about program **metric:**

> This derivative of "bulk" removes any of the print statements, so the timing is more accurate.

Though retaining the same global functionality, the source code underlying programs metric and bulk have been greatly modified to suit the peculiar needs of the experiments done at the HSP Lab. To retain the inheritance link, but still mark the difference, the names of the source files have been prefixed with letter **m** (for **multicast**) respectively to mmetric.C, mbulk.C and mcommon.h, and their executable versions to **mmetric** and **mbulk**. The in-extenso listings of the source code for these files are presented in some of the appendices. Some indications of the changes introduced in the code are presented in the following paragraphs.

Instrumenting the code

The types of data needed are the `offered load` and the `throughput`. The offered load is given as a command line parameter and presents no measurement difficulty. Essentially then, throughput measurements are needed. To compute throughput, two components are needed: the `amount` of data sent and the `time interval`. Again, amount presents no measuremnt difficulty, as the actual quantity is provided by the user and can be displayed quite easily at both the sender and receiver sides. The task for measuring the time interval is more complicated though, and accuracy has to be bargained with.

The viewpoint adopted for the timing measurements is from a user level perspective. Therefore, the computations are made from the user level testing programs used. For reliable experiments, a timestamp marks the beginning of the time interval, immediately before the test program engages in its send loop. Similarly, another timestamp marks the end of the time interval, immediately after the testing program exits its send loop. The difference between the two timestamps is the time interval used as the denominator to compute the throughput (throughput=amount/ts1-ts0). Here follows some excerpts from the code outlining how the timing computations are made and the throughput is measured.

`in mcommon.h`

```
#ifdef TIMING
    timer tm;
    word32 start = tm.timestamp();
#endif

    do {
        if ((res = xi->send(buf, len, blk, &soptions)) < 0) {
...
    } while (!done);
#ifdef TIMING
    /* sent is in bytes */

    word32 stop = tm.timestamp();
    word32 diff = stop - start;

    fprintf(stderr, "Timing: %d ms\n", diff);
    printf("Timing: %d ms\n", diff);

    fprintf(stderr, "Throughput: %.3f Bpms - Bytes per ms\n",
        (double)((double)sent/(double)diff));
    printf("Throughput: %.3f Bpms - Bytes per ms\n",
        (double)((double)sent/(double)diff));

    /*
     * now the xy coordinates for the log file
     *
     */
    fprintf(stderr, "xy: %d %.3f \n",
        outrate, (double)((double)sent/(double)diff));
    printf("xy: %d %.3f \n",
        outrate, (double)((double)sent/(double)diff));

    fprintf(stderr, "Throughput: %.3f Mbits/sec\n",
           (double)(sent*8)/(double)(((double)diff)*(double)1000.0));
    printf("Throughput: %.3f Mbits/sec\n",
            (double)(sent*8)/(double)(((double)diff)*(double)1000.0));

    fprintf(stderr, "Number of calls: %d\n", pkts);
    printf("Number of calls: %d\n", pkts);

    fprintf(stderr, "Latency: %.3f ms/call\n",
          (double)diff/(double)pkts);
    printf("Latency: %.3f ms/call\n", (double)diff/(double)pkts);

#endif /* TIMING */
```

Therefore, for the measurement of the time interval, there is at least a half round trip time interval inaccuracy (at the end). Given the large amont of data transferred, this does not introduce a large error though. Furthermore, a greater number of bytes than the number of bytes used for the throughput computations get effectively handled, as there are the overheads of the various headers for XTP, UDP, IP and Ethernet. Both factors contribute to yield a reported throughput slightly lower than the effective throughput.

Capturing instrumentation messages

While conducting the live experiments, it was felt that two main observation requirements had to be satisfied: (1) one was to keep the progression of the experiment continually under the monitoring eyes of the experimenter; (2) the other was to update as quickly as possible the log file for later evaluation and curve plotting. To meet these two requirements, the technique used consists of duplicating most of the instrumentation messages (i.e., one is sent to standard output - the screen unless redirected, and the other is sent to standard error -the screen). This message duplication has the form:

```
fprintf(stderr, “message”);
print(“message”);
```

as is illustrated in the previous code excerpt. When conducting the numerous simple mock experiments, the test program can be run with all messages sent twice to the screen, or redirection could be used as follows to get rid of one stream of messages:

```
mmetric -args > /dev/null
```

When conducting an experiment with the intention of completing it (i.e., covering all send rates), the testing program can be run as follows:

```
mmetric -args >> m|xSN.log
```

to automatically update the log file and still be able to monitor the ongoing live experiment.

7.6 Option settings for the experiments

For convenience, the xtp_config structure (included in XTPtypes.h) used by client programs to convey to the daemon the terms of a communication association is included verbatim. Many of the command line arguments used with **mmetric** to conduct the experiments result in the setting of fields that belong to this structure.

```
typedef struct {
   short16 options;              // preset options
   short16 yes_mask;             // what options MUST be set for incoming
   short16 no_mask;              // what options MUST NOT be set
   short16 sort;                 // priority value
   short16 edge_freq;            // how frequently to change the EDGE bit
   short16 mcast_diameter;       // how far out to send multicast packets
   short16 mcast_max_act_rcvrs;  // max number of active receivers
   short16 mcast_min_act_rcvrs;  // min number of active receivers
   word32 excess_alloc;          // optimistic send buffer credit
   word32 w_timer_limit;         // limit on how large wtimer can get
   word32 c_timeout_interval;    // time spent in synchronizing handshake
   word32 retry_count;           // number of retries for sync handshake
```

```
    word32 init_rtt;            // initial round trip time setting
    word32 maxspans;            // number of selective retransmission spans
    word32 pdu_size;            // PDU size
    word32 rate;                // outgoing rate value
    word32 burst;               // outgoing burst value
    word32 extra_modes;         // user-controlled modes of operation
    word32 snd_buf_size;        // send buffer size
    word32 rcv_buf_size;        // receive buffer size
} xtp_config;
```

Typical command line arguments used for the experiments are as follows:

- Unicast experiment, sender side;

```
-t 132.205.45.24 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10
```

- Unicast experiment, receiver side;

```
-r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10
```

- Multicast experiment, sender side;

```
-T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -O 127 -W 102400
-c 10
```

- Multicast experiment, receiver side.

```
-R 239.159.100.40 -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10
```

where :

-t or -r –are respectively used to designate a unicast transmitter or receiver.

132.205.45.24 –is the IP class B address designating the receiver; in this case machine forest.cs.concordia.ca.

-T or -R –are used respectively to designate a multicast transmitter or receiver. Consequently, the MULTI bit is set (xcf.options |= MULTI), and a flag is also set (is_trans=1 or is_recv=1).

239.159.100.40 –is an arbitrary IP class D multicast group address used for the experiments. All members of a multicast group must share the same IP group address and port number. (see mcommon.h for the arbitrary port number used).

-a 1048576 (*bytes*) –is used to convey the **amount** of data to be sent. This quantity is used only within the mmetric program to make repeated calls using a buffer of buffersize (-b 1440) for sending the data (xtpif::send(buffer,...)).

-b 1440 (*bytes*) –defines the user level buffer size (i.e., the service data unit-SDU) when making send calls to transfer the data in the send shared memory area (xtpif::send(buffer,...)). If we add 32 bytes for the xtp header to 1440, then we obtain the PDU size of 1472 bytes.

-c 10 (*bytes per ms - Bpms*) –is used to convey the output rate at the source (*outrate* - Traffic Field), here shown at its minimal value of 10 Bpms used for the rate control experiments. The output rate (also called *send rate* or *offered load* elsewhere in the report) is progressively increased during one unicast or multicast experiment unit to cover all test rates. (xcf.rate = 10, ...).

-C 1440 (*bytes*) –is the suggested output burst value. With reference to XTP4.0, section 4.5 Rate Control, "The *burst* value specifies the maximum number of bytes to be sent in a burst of packets". The goal of matching the burst size and the user level buffer size (both at the same value, here at 1440 bytes) is to force an evenly distributed outflow of packets in time, and thus achieve a better queuing discipline with regards to the operating system. It is worth mentioning that the burst value of 1440 is conceived from a user level perspective (as well as the send rate (offered load) and throughput), implying that more bits are effectively dealt with.

-f –is used to set SREQ in the FIRST packet, which forces a synchronizing handshake and key exchange procedure (see also comments about the -g argument).

-g –is used to block the sender on acknowledgements. For all practical purposes, it means that the first xtpif::send() call is made with the value of block set to BLOCK and the options to SREQ (xtpif::send(...,BLOCK,&soptions=SREQ)), which results in blocking the transmitter awaiting a control packet response. For the following xtpif::send() calls, the returned &soptions conditions are likely to be different, therefore the call becomes non-blocking. This approach is used to prevent needless filling of the send shared memory area.

-j 10 (*bytes per ms - Bpms*) –is used to convey the input rate for the incoming data stream (tspec.ts1.inrate=input_rate). The input rate and the output rate (-c at the receiver side) are kept in synchrony all along the experiment unit.

-J 1440(*bytes*) –is used to set the input burst (tspec.ts1.inburst = input_burst), which is kept in synchrony with the output burst.

-o 250 (ms) –is the initial round trip time (xcf.init_rtt = 250). The particular value of 250 ms is the default value used by SandiaXTP.

-O 127 –is used to set the multicast diameter (xcf.mcast_diameter), here set at its maximum value of 127.

-p 1472 (*bytes*) –is used to define the PDU size (xcf.pdu_size = 1472) at the transport level. A particular PDU size of 1472 bytes prevents packet fragmentation at the physical level.

-S –is used to specify selective retransmission (rather than the default go-back-N), which was felt more appropriate for long distance, large data transfers. (xcf.extra_modes |= SELRETRANS).

-w 102400 (*bytes*) -is used to convey the receive window size (xcf.rcv_buf_size = 102400); i.e., the size of the client/daemon shared memory area when the client is receiving. The window sizes are kept in synchrony both at the sender and at the receiver side.

-W 102400(*bytes*) -is used to define the send window size (xcf.snd_buf_size = 102400), i.e., the size of the shared memory area when the client is sending.

7.7 Typical data transfer session scenarios

Tables 7 and 8 summarize the main steps of typical data transfer sessions. For the sake of simplicity, we show only one receiver, but the steps would be the same if there are many receivers. We also presume that the send operation is done from the experimenter's machine, which is not a necessity. Any receiver is started first of course, awaiting a slight delay before starting the sender in case of a multicast experiment (to allow IP_ADD_MEMBERSHIP message sent from receivers to propagate to the mrouters). Actually, those steps are automated with PERL programs (called **play and playm**) presented as appendices.

Table 7: Typical data transfer session scenario - Unicast

```
SENDER
cd_play #to conduct experiment in a temporary subdirectory
# recompile the daemon for SELECT_FLOOR and MAXA if needed
# use configPlay to fix the user parameters (presuming prefix=ux00)
...play> xtpd -d udp #to start the daemon
#make sure receiver is started before issuing next command !
# use 'play -ut' to start the sender and cover all ranges
# sample command launched by play:
mmetric -t $DST -S -g -f -p 1472 -b $UBUF -C $BUR -a $AMT
    -o $RTT -W $WS -c $rate >> $log_file
...
Throughput: 9.607 Bpms - Bytes per ms
xy: 10 9.607
...
Sent 1048576 bytes
#make sure receiver is started before issuing next command !
...
#automatically repeated by play for each remaining send rate
...
# the following are automatically performed by play at the end:
...play> xtpdrm #experiment session terminated, remove daemon
...play> xtractXY $log_file > $xy_file #build XY file
...play> xgraph $xy_file & # plot curve and save in ux00.xy.ps
...play> xtabLOG $log_file > $data_file #build Table ux00.log
...play> xtabXY $xy_file >> $data_file #build Table ux00.xy
# possibly relocate files permanently
...play> mv $log_file $xy_file $xy_file.ps $data_file files (y/n)?:
# update $log_file with receiver command, etc.
```

```
RECEIVER
...play/jeux> telnet receiver.cs.concordia.ca
...>
play/jeux> xtpd -d udp
play/jeux> play -ur
...
Received 1048576 bytes
...
#automatically repeated by play for each remaining send rate
...
# the following is automatically performed by play at the end:
play/jeux> xtpdrm
```

Table 8: Typical data transfer session scenario - Multicast

SENDER

```
cd_play #to conduct experiment in a temporary subdirectory
# recompile the daemon for SELECT_FLOOR and MAXA if needed
# edit playm to fix the experiments parameters (presuming prefix=mx00)
# playm automatically starts and stops the daemon for each data transfer job
# a separate file with all data rates is needed
...play> playm -init #to reset rates file
#make sure receiver is started before issuing next command !
#the following command must be restarted interactively until r=1250
...play> playm -mt
# sample command launched by playm:
mmetric -T $DST -S -g -f -p 1472 -b $UBUF -C $BUR -a $AMT
   -o $RTT -W $WS -c $rate >> $log_file
...
Throughput: 9.607 Bpms - Bytes per ms
xy: 10 9.607
...
Sent 1048576 bytes
...
# the following are automatically performed by playm when rate=1250:
...play> xtractXY $log_file > $xy_file #build XY file
...play> xgraph $xy_file & # plot curve and save in mx00.xy.ps
...play> xtabLOG $log_file > $data_file #build Table mx00.log
...play> xtabXY $xy_file >> $data_file #build Table mx00.xy
# possibly relocate files permanently
...play> mv $log_file $xy_file $xy_file.ps $data_file files (y/n)?:
# update $log_file with receiver command, etc..
```

RECEIVER(s)

```
...play/jeux> telnet receiver.cs.concordia.ca
...>
#the following command must be restarted interactively until r=1250
play/jeux> playm -mr
...
Received 1048576 bytes
...
```

8 Experimental results and interpretation

8.1 Synopsis of the experiments

Table 9 shows the planning used for conducting the experiments. There are seven groups of experiments, and the group labels are also used later as section headings for the purpose of presenting and interpreting the results. Each line provides the main parameters that characterize each experiment unit, and that were used for recompiling the daemon (if needed), for conducting the experiment and for recording its results. The key identification label for each experiment unit is given in column "Prefix".

Experiments shown in Table 9 follow a logical order. For example, the fact that ux18 follows ux04 simply indicates that ux18 was executed later in time than ux17, and that its raison d'être was not apparent at the begining but became apparent later as a result of executing other experiments. The order used in Section 8.3 for the purpose of presenting and interpreting the results is the same logical order as used for Table 9. However, a purely sequential order based on the prefix is used for presenting the data and the saturation curves in Appendix A.

8.2 Elements for interpretation

8.2.1 Limits on physical resources

Table 10 shows some physical characteristics of the machines used for the experiments. The data for the second column (**Mean delay**) were derived from a set of special purpose experiments done with SandiaXTP. Packets (1440 bytes of data each) were sent from a faster machine to a slower machine, with burst set to 14400 bytes so that about 10 back-to-back packets are sent in a burst of packets.

The indicators used are traces (with timestamps added) reported by the daemon once it has sent a packet. Ex:

```
(10) 0x67c0 DATApacket BEING SENT at 2924694589
(11) 0x67c0 DATApacket BEING SENT at 2924694591 etc.
```

Table 9: Synopsis of Experiments

Using	SEL[2]	MAX[3]	Har[4]	WS[5]				
SXTP-1.5.1	(ms)	(ms)	(y/n)	(bytes)	Sender	Receiver(s)	Mode	Prefix
Basic unicast experiments/curves:								
Unmodified	50		no	102400	orchid	forest	Unicast	ux01
Unmodified	50		no	102400	dahlia	orchid	Unicast	ux02
Unmodified	50		yes	102400	orchid	forest	Unicast	ux03
Unicast - impact of using a MAXANTICIPATION margin ..:								
Modified	50	10	no	102400	orchid	forest	Unicast	ux04
Modified	50	5	no	102400	orchid	forest	Unicast	ux18
Modified	50	10	yes	102400	orchid	forest	Unicast	ux05
Unicast - impact of varying the SELECT_FLOOR threshold:								
Modified	100	0	no	102400	dahlia	orchid	Unicast	ux06
Modified	25	0	no	102400	dahlia	orchid	Unicast	ux07
Modified	10	0	no	102400	dahlia	orchid	Unicast	ux08
Unicast - impact of SELECT_FLOOR & MAXANTICIPATION..:								
Modified	100	10	no	102400	dahlia	orchid	Unicast	ux09
Modified	25	10	no	102400	dahlia	orchid	Unicast	ux10
Modified	10	10	no	102400	dahlia	orchid	Unicast	ux11
Modified	10	5	no	102400	dahlia	orchid	Unicast	ux19
Unicast - lack of bombardment effect:								
Unmodified	50		no	102400	orchid	forest	Unicast	ux12[6]
Modified	50	10	no	102400	orchid	forest	Unicast	ux13
Modified	10	10	no	102400	orchid	forest	Unicast	ux14
Modified	10	0	no	102400	orchid	forest	Unicast	ux16[7]
Modified	0	0	no	102400	orchid	forest	Unicast	ux17
Multicast experiments/curves:								
Unmodified	50		no	102400	orchid	daffodil	Multicast	mx01
Unmodified	50		no	102400	orchid	dahlia	Multicast	mx02
Unmodified	50		no	102400	orchid	dah/sunset	Multicast	mx03
Unmodified	50		no	102400	orchid	dah/sun/daf	Multicast	mx04
Modified	50	10	no	102400	orchid	dah/sun/daf	Multicast	mx05
Other experiments/curves:								
Unmodified	50		no	51200	orchid	forest	Unicast	ux15
Modified	0	5	yes[8]	102400	orchid	dahlia	Unicast	ux20
Modified	0	5	yes	102400	orchid	dah/sun/daf	Multicast	mx06

1 SELECT_FLOOR threshold value

2 MAXANTICIPATION margin value

3 Harmonization of rate and burst values

4 Window size

5 User buffer size = 14400 bytes instead of 1440 bytes, ux12, ux13 & ux14

6 User buffer size = 28800 bytes (ux16 & ux17)

7 Burst = rate * 1000 * 0.1 (ux20 & mx06)

Table 10: Machine characteristics

Name	Mean delay[1] (ms)	Derived Capacity (Bpms)	Features	O.S.
dahlia	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 bytes

[2] Solaris2.5

Each line of the traces ends with a timestamp. The difference between two sequential timestamps yields an interval of time (2 ms for the example given above), which is interpreted as the time required to send a packet. The average of many such intervals is reported in column **Mean delay** of Table 10.

Based on the mean delay, it is then possible to extrapolate and derive the apparent capacity of a machine. For instance, given that 0.5 ms are needed by machine dahlia to send a 1440 byte data packet, then it can send 2880 bytes per millisecond. The derived machine capacities are reported in the third column **Derived capacity** of Table 10.

There is no claim to absolute precision with these machine characteristics, but they can be of relative use when interpreting the saturation curves. Even if the data were obtained for sending back-to-back packets, it can also indicate the relative speed of the machines when receiving. For instance, for a scenario where orchid is used for sending, and forest for receiving; it is to be expected that the bottleneck won't be orchid (at 1440 Bpms), or Ethernet (at 1250 Bpms), but forest with an indicative capacity of 206 Bpms. The mean delay could also supply ammunition for interpretation when considering the *bombardment effect* as explained later.

Table 11: Evolution of RTIMER when burst = 1440 bytes

RTIMER (ms):	144	96	72	57.6	48	41.1	28.8	19.2	14.4
Rate (Bpms):	10	15	20	25	30	35	50	75	100
RTIMER (ms):	11.5	9.6	8.2	7.2	5.7	4.8	4.1	3.6	2.8
Rate (Bpms):	125	150	175	200	250	300	350	400	500
RTIMER (ms):	2.4	2.0	1.8	1.6	1.4	1.3	1.2	1.1	
Rate (Bpms):	600	700	800	900	1000	1100	1200	1250	

Finally, it may appear strange that the send capacity of dahlia at 2880 Bpms is higher than the underlying capacity of the Ethernet network technology at 1250 Bpms. However the puzzle disappears when considering two explanations: (1) those are extrapolated data from sending about 10 back-to-back packets; in reality, a machine will not send back-to-back packets all the time; (2) SandiaXTP is a user level implementation, and the timestamps reported by the daemon are probably the rate at which the data is moved from shared memory/user space to kernel space. The O.S. probably allows enqueuing many packets in its internal buffer space to be sent asynchronously later.

8.2.2 SELECT_FLOOR effect

Trial experiments with traces have shown that, amongst all the XTP timers and if RTT is large, the RTIMER is the one that dominates most of the time (i.e., has lowest absolute value and thus has most effect on the timeout value passed to the select() system call).

Table 11 shows the evolution of the values of RTIMER, given a fixed burst size of 1440 bytes, and the various discrete data rates used for the experiments. One can observe that the values of RTIMER (computed as burst/rate) steadily decrease as the values for the rate increase.

At a particular rate value of 30 Bpms, the value of RTIMER is 48 ms, which is close to the threshold value of 50 ms allowed by the unmodified SandiaXTP-1.5.1 implementation to be passed as the timeout parameter to the select() system call. Given the particular combinations of command line arguments used for many of the experiments conducted at the HSP Lab, it is as if the SandiaXTP implementation did not allow a data rate higher than approximately 30 Bpms. Our colleague, Mr. Pierre Falcot, has first thrown light on this phenomenon (call it the **SELECT_FLOOR effect**), which is frequently invoked later in this report to explain parts of the shapes of the curves.

Other events limit however the impact of the **SELECT_FLOOR effect**, as explained later. Here follows edited trace excerpts to illustrate the SELECT_FLOOR effect (for rate = 100 Bpms), when RTIMER equals 14 ms, but 50 ms is the value of the timeout parameter conveyed to the select() system call:

```
->RTIMER - start_rtimer() RTIMER=14

(87) 0x4180 DATApacket BEING SENT at 3255285002

satisfy() - returning shortest_orig=50, now=3255285003

->shortest (top level do { }while(!ds) )= 50, now=3255285003
timeout (before select(...,&timeout))= 50, now=3255285003
daemon going to sleep.......
```

8.2.3 Bombardment (of XTP_SEND requests) effect

The SELECT_FLOOR effect described in the previous subsection dominates only within a subset of the range of send rates. Throughout however, what we label as the "**bombardment phenomenon**", comes into play to mitigate slightly at low data rates, and dramatically at high data rates, the impact of the SELECT_FLOOR effect.

The **bombardment phenomenon** can be defined as the incessant upcoming of events, such as incoming packets or incoming XTP_SEND requests issued by the user, with the consequence that the select() system call returns well before the specified timeout value. Recall from Figure 25 that the daemon could get awakened as a consequence of these two events also.

For instance, here follows edited trace excerpts that illustrate the daemon being awakened prematurely because of an incoming CNTL packet received from the other party:

```
->shortest (top level do { }while(!ds) )= 138, now=3274857552 icici
timeout (before select(...,&timeout))= 138, now=3274857552 icici
daemon going to sleep.......
....up on INCOMING PKT 2 ms later, now=3274857554 icici
```

```
(446) 0x7f80 CNTLpacket RECEIVED from 132.205.45.24
```

For the rate control experiments done at the HSP Lab, the sizes of the user buffer, PDU and burst have been matched, with the (probably intuitive) consequence that there would be approximately as many XTP_SEND requests issued as there were packets to send. This scenario really creates a **bombardment effect** on the daemon with incessant XTP_SEND user requests.

At low rate values, implying high values for RTIMER, the daemon gets awaken prematurely (i.e., well before the timeout expiration given to select()), but as r_timer has not expired, the packet is just inserted on the send FIFO queue and the daemon returns to inactivity. If in the proper range of send rates, the SELECT_FLOOR effect still dominates.

From observation of many saturation curves and of traces, the bombardment/boiling effect seems to become effective when the value of RTIMER has about the same value as the time needed by a machine to send a packet. The scenario of the bombardment effect can be summarized as follows:

1. The daemon gets awaken prematurely, probably on a XTP_SEND request, and discovers that RTIMER has expired, a new RTIMER (say of value 3 ms) is computed and armed before sending a packet;

2. A packet is sent (suppose the time needed is about 3 ms). When this is done,

whether r_timer has expired or will expire soon is of no consequence, as it is not checked yet;

3. The daemon yields to select() with a timeout value of 50 ms;

4. Select() returns immediately (or almost) as a consequence of one XTP_SEND request;

5. The r_timer is checked, found that it has expired, and the pattern repeats itself resuming again at step 1.

Though it conveys the essence of the phenomenon, let us emphasize that this scenario is slightly simplified in the following manner. When the daemon resumes from select(), there could be some packets on the send FIFO queue, and one could be sent. In this case, the one corresponding to the send request could be enqueued. Then, as shown on Figure 25, the daemon would proceed with a check of timers for all contexts, and depending whether r_timer has expired, another packet for the same context could again be sent before the daemon yields to select(). So, instead of one packet being sent on a pass, two could be sent. But even with this revised scenario, select() would again return immediately (or almost), and the resulting pattern is not changed significantly. What happens exactly depends on the exact value of RTIMER, and probably other factors, such as process scheduling.

When the bombardment effect starts occuring, the SELECT_FLOOR effect is completely wiped out, and the daemon effectively jumps to sending packets presumably at a send rate closed to the machine capacity. The transmitter will be periodically halted, because of the flow control mechanism and window size, with the net result tempered also by the capacity of the receiver. Whatever the new throughput achieved, we can expect a sudden surge in the saturation curve.

Here follows trace excerpts that illustrate the bombardment effect at high data rates:

```
Note : rate=1250
(102) 0x5480 DATApacket BEING SENT at 3181192617
satisfy() - returning shortest_orig=50, now=3181192618
```

```
->shortest (top level do { }while(!ds) )= 50, now=3181192618
timeout (before select(...,&timeout))= 50, now=3181192619
daemon going to sleep.......
...up on USERREQ 0 ms later, now=3181192619

(103) XTPdaemon XTP_SEND request for 0x5480
(104) 0x5480 Putting packet on send FIFO, seq = 15880
(105) 0x5480 RTIMER expired now=3181192619 icici

(106) 0x5480 Sending packet from send FIFO, seq = 15880

RTIMER=1, r_timer=3181192622, now=3181192621

(107) 0x5480 DATApacket BEING SENT at 3181192623
satisfy() - returning shortest_orig=50, now=3181192624

->shortest (top level do { }while(!ds) )= 50, now=3181192624
timeout (before select(...,&timeout))= 50, now=3181192624
daemon going to sleep.......
...up on USERREQ 1 ms later, now=3181192625
etc...
```

8.2.4 Harmonization of burst and rate values

The SELECT_FLOOR effect has important negative impacts on the quality of the rate control exercised by SandiaXTP only when the value of RTIMER falls below the SELECT_FLOOR threshold (i.e., below 50 ms with the SandiaXTP-1.5.1 unmodified release). More particularly, given the particular combination of command line arguments used previously at the HSP LAb for many rate control experiments, this too low RTIMER value occurred as a result of maintaining the burst value constant, but progressively increasing the rate value.

One approach that can be used to circumvent the SELECT_FLOOR effect problem, in fact one that has already used in for previous study done at the HSP Lab ([FALCOT]),

Table 12: Harmonizing burst and rate values

Rate (Bpms) (1)	Bytes to send/s (2) =(1)*1000	#packets to send/s (3) =(2)/1440	Appr.send time (ms) (4) =(3)*4 ms	Idle time (ms) (5) =1000-(4)	Burst (bytes) (6)	Ratio burst/PDU (7) =(6)/1440	# gaps (8) =(3)/(7)	inter-bursts gap (ms) (9) =(5)/(8)
10	10000	7	28	972	1440	1	7	139
15	15000	10	40	960	1440	1	10	96
20	20000	14	56	944	1440	1	14	67
25	25000	17	68	932	2880	2	8.5	109
30	30000	21	84	916	2880	2	10.5	87
35	35000	24	96	904	2880	2	12	75
50	50000	35	140	860	4320	3	11.6	74
75	75000	52	208	792	5760	4	13	61
100	100000	69	276	724	8640	6	11.5	63
125	125000	87	348	652	11520	8	10.8	60
150	150000	104	416	584	15840	11	9.4	62
200	200000	139	556	444	28800	20	6.9	64
250	250000	174	696	304	43200	30	5.8	52
300	300000	208	832	168	103680	72	2.9	58
350	350000	243	972	28	103680	100	-	-
400	400000	278	1112	-	103680	-	-	-
500	500000	347	-	-	103680	-	-	-
600	600000	417	-	-	103680	-	-	-
700	700000	486	-	-	103680	-	-	-
800	800000	556	-	-	103680	-	-	-
900	900000	-	-	-	103680	-	-	-
1000	1000000	-	-	-	103680	-	-	-
1100	1100000	-	-	-	103680	-	-	-
1200	1200000	-	-	-	103680	-	-	-
1250	1250000	-	-	-	103680	-	-	-

consists of varying both the rate and the burst such that RTIMER does not fall below the SELECT_FLOOR threshold. Table 12 shows some detailed calculations used to derive combinations of burst and rate values to meet this criterion, given the range of rates used for the current experiments. The key columns are: (a) column (1) listing the rate; (b) column (6) listing the corresponding burst; (c) column (9) listing inter-bursts gaps between the end of sending one burst of packets and beginning of the next one. Roughly, column (9) corresponds to the value of the timeout parameter passed to the select() system call. The purpose of Table 12 is to make sure that the inter-bursts gap (column (9)) never gets below the SELECT_FLOOR threshold (here presumed at 50 ms). We now comment on the details of Table 12, with particular emphasis on the first data row.

1. **Columns (1) & (2):** Given that column (1) shows the rate in Bpms; column (2) indicates how many bytes per second need to be sent to satisfy the rate specified in column (1). As there are 1000 milliseconds in one second, and for a rate of 10 Bpms, then 10 000 bytes would need to be sent in one second.

2. **Column (3):** Presuming that each packet contains 1440 bytes of data, column (3) shows the approximate number of packets that would need to be sent, i.e., 7 packets for the first data row (10 000/1440=6.9).

3. **Column (4):** Again presuming that 4 ms of host processing time are needed per packet (probably a too high value), column (4) indicates the total processing time, i.e., 28 ms for the first data row (7*4=28 ms).

4. **Column (5):** Given that there are 1000 ms per second, and that column (4) gives the total processing time, then column (5) shows the total waiting time per second, i.e., 972 ms for the first data row (1000-28=972).

5. **Column (6):** The purpose of the remaining columns (6), (7), (8), and (9) is to adjust the burst value such that the inter-burst gap is not less than the SELECT_FLOOR threshold (here at 50 ms). Column (6) lists the tentative burst value, i.e., 1440 bytes for the first data row.

6. **Column (7):** For a given burst value and a given PDU size of 1440 bytes (in actuality 1472 bytes), column (7) shows the ratio burst/PDU (or burst size expressed in number of packets), i.e., 1 for the first data row (1440/1440).

7. **Column (8):** Given the ratio burst/PDU, column (8) supplies the implied number of gaps by dividing the number of packets to send by the burst size, i.e., 7 for the first data row (7/1=7).

8. **Column (9):** Finally, column (9) supplies the inter-burst gap by dividing the total waiting time (column (5)) by the number of gaps (column (8)), i.e., 139 ms for the first data row (972/7=139). As this is well above the SELECT_FLOOR threshold, there is no problem with this mix rate=10/burst=1440.

With a data rate of 30 Bpms, and keeping the burst value constant to 1440 bytes, the inter-bursts gap would reach 43.6 ms, which is below the SELECT_FLOOR threshold. This is the reason why the burst value is increased to 2880 bytes, producing an inter-burst gap of 87 ms. The remaining of Table 12 does about the same kind of harmonization of burst and rate values.

Again, let us emphasize that these data are not meant to be rigorously precise with respect to the reality of actual machines, but to "improve" on the SELECT_FLOOR effect. More particularly, the higher values of Table 12 reveal the appearance of physical limits, which would be different with real machines. For example, with a processing time of 4 ms per packet, and a rate of 400 Bpms, it would take more than one second for a presumed one second duration task (1112 VS 1000). Also, the burst size has reached a greater value than the window size used for the experiments (103680 VS 102400). The higher data rates have been maintained to keep in perspective this hard Ethernet physical limit of 1250 Bpms, though the actual maximum throughputs expected are likely to be well below 1250 Bpms. Given the mixes of rate and burst values shown on Table 12, additional rate control experiments can be conducted, and the results compared with experiments done with other parameters.

8.3 Presentation and interpretation of the results

To conserve space, a decision was made to separate the present Section from the experimental data and the saturation curves, which are included in Section A. The format used for presenting the data and the saturation curves is *one experiment unit/one sheet*, with the top part being the log summary, the middle part being the data summary (x,y coordinates), and the bottom part being the saturation curve.

8.3.1 Basic unicast experiments/curves

ux01 - SXTP-1.5.1 unmodified, orchid to forest

Table ux01.log, Table ux01.xy (refer to Section A), and Figure 31 show respectively the log summary, the data summary, and the saturation curve for the ux01 unicast experiment unit. The experiment was done with SandiaXTP-1.5.1 unmodified (i.e., with a threshold value of 50 ms passed to system call select() as the minimum timeout parameter). Machine orchid is acting as the transmitter and machine forest is acting as the receiver, hence a mix of fast and slow machines.

As shown on Figure 31, the resulting saturation curve can be subdivided into four segments for interpretation purposes:

1. **Low rising segment (up to load=30 Bpms):** This part of the curve displays the expected behavior of a system under light loading conditions, i.e., there is some loss due to overhead, but the throughput is proportional and very close to the offered load (about 96%). With reference to Table 11, the value of RTIMER is above or about equal to the SELECT_FLOOR threshold of 50 ms, and the select floor effect is of no apparent consequence.

2. **The SELECT_FLOOR plateau** (from load=30 to load=700 Bpms) For this part of the curve, RTIMER has decreased below the SELECT_FLOOR threshold of 50 ms (check again Table 11) and the throughput remains stable and equivalent to a rate of 30 Bpms (or RTIMER of about 50 ms). The SELECT_FLOOR effect is in full control.

3. **The steep rise (from load=700 to load=800 Bpms:)** For this part of the curve, the bombardment effect described previously has taken over. Given the time needed to send a packet, and the fact that RTIMER is small enough, the daemon would yield to select() but select() returns immediately (or almost) due to incessant XTP_SEND user requests. Hence the steep rise in the curve. Here follows edited trace excerpts at rate=800 Bpms to substantiate this interpretation:

```
...up on USERREQ 0 ms later, now=3357170611

(56) XTPdaemon XTP_SEND request for 0x4dc0
(57) 0x4dc0 DATApacket BEING SENT at 3357170612

->RTIMER - start_rtimer()
 RTIMER=1, r_timer=3357170614, now=3357170613

(58) 0x4dc0 RTIMER expired now=3357170614

->dq_timer() called
  key=0x4dc0 timer_type=RTIMER due_time=3357170614

->print_timers_onq() called:
  # of timers onq=1  now=3357170615
  key=   type=    due_time= derived delta time to fire (ms)=
  0x4dc0 CTIMER   3360770447 3599832

->Bubbling up shortest to select()<-
check_timers()-returning shortest_orig=0, now=3357170615
satisfy() - returning shortest_orig=50, now=3357170615

->SATISFY() - bypassing shortest_original

->read_first_timer_onq() called
timers_list_head->due_time=3360770447
time left to fire: 3599831 ms

->satisfy() - returning shortest_NEW=3599831, now= 3357170616

->shortest (top level do { }while(!ds) )= 3599831, now=3357170616
timeout (before select(...,&timeout))= 10000, now=3357170616
daemon going to sleep.......
...up on USERREQ 0 ms later, now=3357170616 icici

(59) XTPdaemon XTP_SEND request for 0x4dc0
(60) 0x4dc0 DATApacket BEING SENT at 3357170617
etc..
```

This trace was obtained by running the modified version of SandiaXTP-1.5.1. Using the modified version is of no consequence on the data reported on Table ux01.xy, as the goal is to prove the working of the bombardment effect. However, one could feel puzzled by the greatly varying timeout value bubbled up at the level of the main loop with the 2 methods. With the unmodified version, 50 ms would be bubbled up; with the modified version, the value of the CTIMER (359983) is bubbled up, and eventually POOL_FREQ (10000 ms - the maximum allowed timeout value) is passed to select(). This appears an awful long duration of possible inactivity for the daemon. However, such is in the nature of things.

The packet that was sent was the one accompanying the XTP_SEND request (i.e., as shown on Figure 24, *acting on behalf of THE context* that made the request, loop 2). Then the daemon goes to check the timers of all contexts. At line (58), this check is reported and it is discoverd that RTIMER has expired. However, as there is presumably no packet on the send FIFO queue, nothing is done and r_timer is not started. The daemon has to yield to select(), and not knowing if there is anything more to send, the CTIMER value is bubbled up. However long a value, it is of no consequence because of the bombardment effect and select() returns immediately.

4. **The high plateau** (above load=800 Bpms: The maximum throughput reached is about 120 Bpms, which is roughly 60% of the recognized 206 Bpms capacity of the receiver (see Table 10) forest. The throughput fluctuates (for unknown reasons) little and remains high, as the bombardment effect once started remains active.

ux02 - SXTP-1.5.1 unmodified, dahlia to orchid

Experiment unit ux02 is meant to validate the results of ux01. A degree of variation is introduced in the design of the experiment by sending from presumably the fastest machine (dahlia) to the second fastest machine (orchid). For experiment ux01, the receiver was comparatively very slow (forest - refer to Table 10). Again, SandiaXTP-1.5.1 unmodified is used.

The results of the experiment are presented on Table ux02.log, Table ux02.xy, and Figure 32. When comparing the two saturation curves (Figure 31, Figure 32), one can observe a close concordance between the shapes of the two curves, and the data for the first two segments. The remaining thoughputs obtained for the ux02 (rate = 800 Bpms and above) saturation curve (Figure 32) are much lower. For instance, at rate = 800 Bpms, the throughput for ux02 is about 47 Bpms (Table ux02.xy), when it is 95 Bpms for ux01 (Table ux01.xy). Another experiment was done immediately after with similar results. Another experiment done the following day (thurs. 10 sept 98 around 14:30-14:45) gave very similar low results; with a jump to 46 throughput at rate=800.

Such throughput results for the offered loads above 800 Bpms are rather against expectations, as both dahlia and orchid are very fast machines with a high capacity 100 Mbps network connection (forest has a 10 Mbps network connection). Given that the shapes of the two curves are in close concordance, which is the important criterion for the present study, questions of comparative maximum throughput are not considered worth of further inquiry at this point. Finally, the explanations regarding the shape of the saturation curve for ux02 are the same than the ones presented for ux01.

ux03 - SXTP-1.5.1 unmodified, orchid to forest, rate/burst harmonized

The intent for conducting experiment unit ux03 is to show the consequences of harmonizing the burst and rate values such that the value of RTIMER does not fall below the 50 ms threshold. With reference to Figure 33, the expectation is to extend as much as possible the first linearly rising segment of the saturation curve, and thus eliminate the SELECT_FLOOR plateau where throughput is constant at about 30 Bpms due to the SELECT_FLOOR effect. The parameters used for experiment ux03 are the same as the ones used for ux01, except for adjusting the burst values as per Table 12. Table ux03.log, Table ux03.xy, and Figure 33 show respectively the log summary, the data summary, and the saturation curve for ux03.

As one can observe on Figure 33, the expectations are reasonably well satisfied. What is a four part curve shown on Figure 31 has now changed into a two part curve made up of one linearly rising segment (from load=10 to load=250 Bpms) and one relatively flat segment above load = 250 Bpms, probably indicating that the the maximum processing capacity of the receiver forest has been reached (Table 10 shows 206 Bpms for forest).

When comparing with ux01 (Figure 31), the maximum throughput of ux03 is about twice better (120 for ux01 - 220 for ux03). Presumably, this large increase in maximum thoughput is accountable not only to greatly reducing the consequences of the SELECT_FLOOR effect, but also to performing the work more efficiently through handling many packets in a row at higher offered load.

With the *one packet* burst size policy of ux01, the intent is to ease the strain on the network as much as possible, even if at the expense of more overhead processing (more stop and go) at the host. With harmonizing the burst and rate values, there is a remarkable and observable improvement of the quality of the rate control exercised by SandiaXTP (through elimination of the SELECT_FLOOR plateau), accompanied by less overhead processing work at the host, but with paradoxically more strain on the network. For the current experiments, which were performed in a closed LAN environment, an increase in throughput may appear as a boon, but it may also have its drawbacks on a WAN such as the Internet. In some respects, the SELECT_FLOOR effect is a throttling mechamism, and it has its good sides for preventing abuse of a shared facility. However, this mechanism is not under the control of the user.

8.3.2 Unicast - impact of using a MAXANTICIPATION margin

The goal of the next few experiments is to evaluate the impact of a less stringent criterion for considering that RTIMER has expired. For doing so, the modified SandiaXTP-1.5.1 release is being used. With unmodified SandiaXTP-1.5.1, a timer (any timer) is considered to have expired only if the timestamp at the moment of making a check is greater than or equal to r_timer. With modified SandiaXTP-1.5.1, the RTIMER (only) is considered to have expired if the lapse of time left before it expires is less than or equal to a *MAXANTICIPATION margin* (abbreviated to *MAXA margin* hereafter in the text) when r_timer is being checked. The reasons for introducing the MAXA margin are outlined in Section 6.

ux04 - SXTP-1.5.1 modified, orchid to forest

A value of 10 ms is used for the MAXA margin. Table ux04.log, Table ux04.xy (refer to Section A), and Figure 34 show respectively the log summary, the data summary, and the saturation curve for the ux04 unicast experiment unit.

As one can observe, the shape of the saturation curve for ux04 (Figure 34) has a hybrid form halfway between the shape of the curve for ux01 (Figure 31), where the SELECT_FLOOR effect is in full control for the second part of the curve, and the curve for ux03 (Figure 33), where the SELECT_floor effect is practically non-existent.

There is a definite improvement in the sense that the length of the SELECT_FLOOR plateau is greatly reduced. Again the throughput tapers off at load = 30 Bpms, but it remains stable up to rate=100 Bpms only (it extended until load=700 Bpms for ux01). Then, there is a quick rise to a maximum throughput of about 210 Bpms, which is in the same range as for ux03, for which harmonization of the burst and rate values was used.

One expected implication of using the MAXA margin is doing work a bit prematurely, to prevent its being done much later becuase of the SELECT_FLOOR effect. The consequences of this policy becomes apparent because the throughput is more than the offered load for two points of the curve (188 > 150, 209 > 200). Per se, this excess is quite acceptable.

However, the early rise to maximum throughput, with few intermediary points, is questionable. The change starts occuring at rate= 125 Bpms, which corresponds to RTIMER value of 11.5 ms (see Table 11). This RTIMER value of 11.5 ms is very close to the 10 ms MAXA margin. Given that the bombardment effect is always active, it would appear that, because of the MAXA margin, the r_timer is always found to have expired whenever it is being checked, producing a too early jump to maximum throughput. With respect to the SELECT_FLOOR effect, which maintains arbitrarily the offered load too low, there could be a reverse of the situation once the RTIMER value is about equal to or smaller than the value used for the MAXA margin. Beyond this upper threshold, effective rate control is no longer being exercised. Let's call this scenario the *MAXANTICIPATION effect* (abbreviated to MAXA effect hereafter in the text), and the set of points included the *MAXANTICIPATION plateau* (abbreviated to MAXA plateau thereafter in the text) for later referencing purposes.

Given the bombardment effect, a burst size of one packet, and a MAXA margin of 10 ms, this upper threshold would correspond to an offered load of about 150 Bpms; if 5 ms is used for the MAXA margin, then the upper threshold would correspond to an offered load of about 300 Bpms (Table 11, rate=300, RTIMER=4.8); etc. This newly hypothesized *MAXA effect* at high offered load, whereby no more rate control would be effectively exercised, cannot be fully verified here with ux04, as the maximum throughput is well below the offered loads for the highest half points of the saturation curve.

Refering again to Figure 5 discussed in Section 3.3, this phenomenon implies that rate control becomes problematic once RTIMER reaches low values. Given that the tapering off of the throughput may occur at quite low offered loads (say below 100 Bpms) on the Internet, then this MAXA effect may not be of much consequence for such an environment.

ux18 - SXTP-1.5.1 modified, orchid to forest (MAXA=5)

The purpose of conducting experiment ux18 is to verify the proposition put forward in the description of ux04 postulating an interruption of effective rate control for RTIMER values in the vicinity and below the value of the MAXA margin (called MAXA effect). The MAXA margin is now reduced by 50% to 5 ms; machine orchid is used for sending, and dahlia for receiving.

Table ux18.log, Table ux18.xy (refer to Section A), and Figure 48 show respectively the log summary, the data summary, and the saturation curve for the ux18 unicast experiment unit.

As shown on Figure 48, the proposition is verified. There is a huge jump in throughput to a value of 1434 Bpms corresponding to a rate of 250 Bpms. Refering to Table 11, a rate of 250 Bpms corrresponds to a RTIMER value of 5.7 ms, and the MAXA effect appears to be verified.

ux05 - SXTP-1.5.1 modified, orchid to forest, rate/burst harmonized

For ux05, both a 10 ms MAXA margin and the harmonization of the burst and rate values are being used. Table ux05.log, Table ux05.xy (refer to Section A), and Figure 35 show respectively the log summary, the data summary, and the saturation curve for the ux05 unicast experiment unit.

The net result is a very interesting curve, with about half of the points (rate=10 to rate=250) nicely distributed in the left rising part of the curve, and the other half of the points (rate=300 to rate=1250 Bpms) slightly fluctuating at the maximum throughput. The fact that the lower half points have a much better distribution than for ux04 can be attributed to the elimination of the SELECT_FLOOR effect through harmonizing the burst and rate values. The fact that the resulting throughput values, also for the lower half points of the curve, are much closer to the offered loads than for ux03 can be ascribed to the anticipated smoothing effect of using a 10 ms MAXA margin.

There is still the possibility of this MAXA effect described for ux04 at high offered loads, but it is again not verifiable as the range of highest throughput remains relatively low.

8.3.3 Unicast - impact of reducing the SELECT_FLOOR threshold

The goal of conducting the next few experiments is to map more precisely the pattern of the SELECT_FLOOR effect. Very often, select() returns much more prematurely than the timeout value that it receives as a parameter, and this is caused by the bombardment of XTP_SEND requests that continually pop up. At low data rates, the bombardment is of no consequence as the data packets implied by the XTP_SEND requests are simply inserted on the send FIFO queue.

The exact value of the SELECT_FLOOR threshold matters for those circumstances when only a small lapse of time is left before RTIMER expires, and the daemon is about to yield to select(). For example, supposing RTIMER = 57.6 ms (rate = 25, RTIMER = 1440/25), and only about 7.6 ms are left. If SELECT_FLOOR = 50 ms (as the hardcoded default for SandiaXTP-1.5.1), then the next waiting round may last up to about 50 ms. Obviously, a lower SELECT_FLOOR value than 50 ms is going to decrease the excess gap between the duration that is left before RTIMER fires, and the actual incurred waiting period. It also matters when RTIMER is smaller than SELECT_FLOOR, and we have not reached the start of the bombardment effect

(which occurs about at load=800Bpms)

For the following experiments, we cover a range of values for SELECT_FLOOR varying from as high a value as 100 ms to as low a value as 10 ms; 10 ms being quite a precise resolution for select(). The 50 ms case is not covered, as ux01 already describes it. Modified SandiaXTP is being used, but with a 0 ms MAXA margin (i.e., as if non-existent). Machine dahlia is used for sending, and orchid for receiving. Using the same two machines for ux02 yielded a similarly shaped saturation curve as the one for ux01 (where orchid and forest were used), but with lower throughputs for higher offered loads.

ux06 - SXTP-1.5.1 modified, dahlia to orchid (SEL=100, MAXA=0)

The SELECT_FLOOR threshold is set to 100 ms. Given the *one packet* burst size used and the SELECT_FLOOR effect, the expectation is that throughput should taper off for a long while at 14.4 Bpms, i.e., once RTIMER reaches a value equal to or less than the SELECT_FLOOR threshold. It is as if SELECT_FLOOR equaled RTIMER or vice versa (hence rate or offered load = 1440/100, because RTIMER = burst/rate).

Table ux06.log, Table ux06.xy (refer to Section A), and Figure 36 show respectively the log summary, the data summary, and the saturation curve for the ux06 unicast experiment unit.

As one can observe on Figure 36, the expectations are well satisfied; throughput indeed tapers off at 14.4 Bpms for many points of the curve. In conformity with the saturation curve for ux01, the throughput also jumps to much higher values at load=800 Bpms because of the bombardment effect. There is no apparent surprise with this ux06 saturation curve (Figure 36), whose shape is very similar to the one for ux01.

ux07 - SXTP-1.5.1 modified, dahlia to orchid (SEL=25, MAXA=0)

SELECT_FLOOR is set to 25 ms, i.e., half the hardcoded value of unmodified

SandiaXTP-1.5.1. Given the *one packet* burst size used and the SELECT_FLOOR effect, the expectation is that throughput should taper off for a long while at 57.6 Bpms (1440/25; see ux06 for an example of more detailed calculations), which is another way of stating that the software maintains the same offered load for a while.

Table ux07.log, Table ux07.xy (refer to Section A), and Figure 37 show respectively the log summary, the data summary, and the saturation curve for the ux07 unicast experiment unit.

As shown on Figure 37, the results are slightly disappointing, as throughput tapers off at 48 Bpms rather than 57.6 ms. The number of points that belong to the SELECT_FLOOR plateau (not so low now) is less than for ux06.

A throughput of 48 Bpms in a plateau also means a constant offered load (rate) of about 50 Bpms, which in turn means that the SELECT_FLOOR value effectively obeyed by select() is about 29 ms (1440/50), and this would explain why we get this plateau at 48 Bmps. Here follows edited trace excerpts that support this interpretation (rate=150):

```
(186) 0x2fc0 Sending packet from send FIFO, seq = 41760

->RTIMER - start_rtimer()
 RTIMER=9, r_timer=3865788947, now=3865788938

->enq_timer() called
key=0x2fc0, timer_type=RTIMER, due_time=3865788947

->print_timers_onq() called:
# of timers onq=2  now=3865788938
key=   type=    due_time=  derived delta time to fire (ms)=
0x2fc0 RTIMER   3865788947 9
0x2fc0 CTIMER   3869388069 3599122

(187) 0x2fc0 DATApacket BEING SENT at 3865788939
```

```
Destination: 132.205.45.61

 satisfy() - returning shortest_orig=50, now=3865788939

->SATISFY() - bypassing shortest_original


->read_first_timer_onq() called
timers_list_head->due_time=3865788947
time left to fire: 8 ms

->satisfy() - returning shortest_NEW=25, now= 3865788939

->shortest (top level do { }while(!ds) )= 25, now=3865788940
 timeout (before select(...,&timeout))= 25, now=3865788940
 daemon going to sleep.......
 ...up on USERREQ 0 ms later, now=3865788940

(188) XTPdaemon XTP_SEND request for 0x2fc0
(189) 0x2fc0 DATApacket BEING SENT at 3865788940

(190) 0x2fc0 Putting packet on send FIFO, seq = 43200

->Bubbling up shortest to select()<-
 check_timers()-returning shortest_orig=6, now=3865788941
 satisfy() - returning shortest_orig=50, now=3865788941

->SATISFY() - bypassing shortest_original

->read_first_timer_onq() called
timers_list_head->due_time=3865788947
time left to fire: 6 ms
```

```
->satisfy() - returning shortest_NEW=25, now= 3865788941

->shortest (top level do { }while(!ds) )= 25, now=3865788941
 timeout (before select(...,&timeout))= 25, now=3865788941
 daemon going to sleep.......
 ...up on TMOUT 26 ms later, now=3865788967

(191) 0x2fc0 RTIMER expired now=3865788967

->dq_timer() called
key=0x2fc0 timer_type=RTIMER due_time=3865788947

->print_timers_onq() called:
# of timers onq=1  now=3865788967
key=   type=    due_time=  derived delta time to fire (ms)=
0x2fc0 CTIMER   3869388069 3599102

->Bubbling up shortest to select()<-
 check_timers()-returning shortest_orig=-20, now=3865788968
(192) 0x2fc0 Sending packet from send FIFO, seq = 43200
```

For a rate of 150 Bpms, the value of RTIMER is 9.6 ms (see Table 11), and this is the value shown at the beginning of the trace (truncated with no decimal shown). Note that the last digits of the timestamp when RTIMER is started are 88938. Then the SELECT_FLOOR value of 25 ms is bubbled up to select(), but select() returns immediately because of an incomimg USERREQ. The packet underlying this XTP_SEND request is inserted on the send FIFO queue for the context to be sent later (line (190)). At this point, the duration left for RTIMER to expire is 6 ms, and again 25 ms is bubbled up to select(). Now, select() returns only after a timeout period of 26 ms, and the lasts digits of the timestamp are 88967. The difference between the two timestamps is 29 ms (88967-88938), as declared before showing the trace. Because of a particular sequence of events, a RTIMER of 25 ms is effectively acted upon 29 ms later.

ux08 - SXTP-1.5.1 modified, dahlia to orchid (SEL=10, MAXA=0)

For ux08, the SELECT_FLOOR threshold is set to 10 ms. Given the *one packet* burst size used and the SELECT_FLOOR effect, the expectation is that throughput should taper off for a long while at 144 Bpms (1440/10; see ux06 for an example of more detailed calculations).

Table ux08.log, Table ux08.xy (refer to Section A), and Figure 38 show respectively the log summary, the data summary, and the saturation curve for the ux08 unicast experiment unit.

As one can observe on Figure 38, there are a few surprises; instead of only one plateau, the saturation curve displays what appears to be three plateaux. The most important one (though with a smaller number of points than ux07) is the SELECT_FLOOR plateau at throughput = 144 Bpms as expected. A smaller plateau (with 3 points) is shown at throughput = 72 Bpms. Finally, another embryonic plateau (2 points) can also be detected at throughput = 28 Bpms. Let's try to analyse the reasons for the appearance of the most important unexpected plateau, the one appearing at throughput = 72 Bpms using the same approach as for ux07 with daemon execution traces.

Here follows trace excerpts for a scenario where rate = 100 Bpms (i.e., throughput = 71.9 - see Table ux08.xy) and RTIMER = 14.4 ms:

```
(66) 0x1000 Sending packet from send FIFO, seq = 12960

->RTIMER - start_rtimer()
 RTIMER=14, r_timer=3860249626, now=3860249612

->enq_timer() called
key=0x1000, timer_type=RTIMER, due_time=3860249626

->print_timers_onq() called:
# of timers onq=2  now=3860249612
```

```
key=   type=     due_time=  derived delta time to fire (ms)=
0x1000 RTIMER    3860249626 14
0x1000 CTIMER    3863849423 3599797

(67) 0x1000 DATApacket BEING SENT at 3860249613
Destination: 132.205.45.61

 satisfy() - returning shortest_orig=50, now=3860249613

->SATISFY() - bypassing shortest_original

->read_first_timer_onq() called
timers_list_head->due_time=3860249626
time left to fire: 13 ms

->satisfy() - returning shortest_NEW=13, now= 3860249613

->shortest (top level do { }while(!ds) )= 13, now=3860249613
 timeout (before select(...,&timeout))= 13, now=3860249614
 daemon going to sleep.......
 ...up on USERREQ 0 ms later, now=3860249614

(68) XTPdaemon XTP_SEND request for 0x1000
(69) 0x1000 DATApacket BEING SENT at 3860249614

(70) 0x1000 Putting packet on send FIFO, seq = 14400

->Bubbling up shortest to select()<-
 check_timers()-returning shortest_orig=11, now=3860249615
 satisfy() - returning shortest_orig=50, now=3860249615

->SATISFY() - bypassing shortest_original
```

```
 ->read_first_timer_onq() called
 timers_list_head->due_time=3860249626
 time left to fire: 11 ms

 ->satisfy() - returning shortest_NEW=11, now= 3860249615

 ->shortest (top level do { }while(!ds) )= 11, now=3860249615
  timeout (before select(...,&timeout))= 11, now=3860249615
  daemon going to sleep.......
  ...up on TMOUT 16 ms later, now=3860249631

 (71) 0x1000 RTIMER expired now=3860249631

->dq_timer() called
key=0x1000 timer_type=RTIMER due_time=3860249626

->print_timers_onq() called:
# of timers onq=1  now=3860249631
key=   type=    due_time=  derived delta time to fire (ms)=
0x1000 CTIMER   3863849423 3599792

->Bubbling up shortest to select()<-
 check_timers()-returning shortest_orig=-5, now=3860249632
(72) 0x1000 Sending packet from send FIFO, seq = 14400

->RTIMER - start_rtimer()
 RTIMER=14, r_timer=3860249646, now=3860249632
```

The explanation is similar to the one provided for ux07. When RTIMER is first being computed in the above trace, the last four digits of the timestamp are 9612. Later on, after handling one XTP_SEND request, select() returns on a TMOUT case 5 ms later (16 vs 11), and the last digits of the timestamp are 9631. The difference between the two timestamps is 19 ms (9631-9612). If the value of RTIMER was 19 ms, then the value for rate would be about 76 ms (19=1440/rate), which is reasonably close to the

72 ms thoughput plateau being analysed. The essential reason would then be some imprecision on exactly when select() returns on a TMOUT case, which in this range of timeout values would appear to be about 5 ms late.

8.3.4 Unicast - impact of SELECT_FLOOR & MAXANTICIPATION

The impact of varying the value of the SELECT_FLOOR threshold value only was explored in the previous Section 8.3.3. The reasons for conducting the following few experiments are to evaluate the impact of also incorporating the MAXA margin (do work somewhat earlier, otherwise it might be done much later). Hence, modified SandiaXTP is being used with machine dahlia used for sending and machine orchid for receiving.

Because of the additional loosening effect of a MAXA margin, we would expect better throughputs than the ones obtained for the experiments described in Section 8.3.3.

ux09 - SXTP-1.5.1 modified, dahlia to orchid (SEL=100, MAXA=10)

For ux09, the SELECT_FLOOR threshold and the MAXA margin are respectively set to 100 ms and 10 ms. The expectation for conducting ux06 (a similar experiment, but without the MAXA margin) was a taper off plateau at throughput = 14.4 Bpms because of the SELECT_FLOOR effect.

Table ux09.log, Table ux09.xy (refer to Section A), and Figure 39 show respectively the log summary, the data summary, and the saturation curve for the ux09 unicast experiment unit.

As one can observe on Figure 39, the results of the experiments are quite astonishing. Record high throughput values are obtained, and 60% of the points of the saturation curve belong to this very high throughput category. There is indeed a taper off plateau at throughput = 14.4 Bpms, but it extends only up to load = 100 Bpms (for ux06, it extended up to load = 700 Bpms).

At load = 150 Bpms, there is a drastic jump to throughput = 1025 Bpms. In fact, throughput gets as high as 1296 Bpms at load = 1100 Bpms. This very high throughput of 1296 Bpms is higher than the 10 Mbps (i.e., 1250 Bpms) nominal capacity of 10 BASE-T Ethernet, and presumed to be high limit for choosing the send rates of the experiments.

In comparison with the other experiments where maximum throughputs are much lower and the same two machines are being used (for example, ux02, where maximum throughput is 46.7 Bpms at load = 800 Bpms), a maximum throughput of 1296 Bpms is more acceptable, given that these two machines have a 100 Mbps (i.e., 12500 Bpms) network connection. After analysing this sudden jump to very high throughputs, we formulate a tentative explanation of these record high throughputs.

Why this sudden rise in throughput at load = 150 Bpms? An offered load of 150 Bpms implies a RTIMER value of 9.6 Bpms (see Table 11). We presumably have a verification of the *MAXA effect* described for ux08. With a RTIMER value of 9.6 ms, and a MAXA margin of 10 ms, we are in the area where each check made to r_timer is going to yield the conclusion that RTIMER has expired (MAXA > RTIMER). The implication of this phenomenon is that, at load = 150 Bpms, the throughput is 6.8 times (1025/150) more than load, and we are clearly no longer exercising rate control. This phenomenon is similar to the SELECT_FLOOR effect, but at the other/excessive end of the polarity. Apparently, what was meant to be an improvement for anomalies such as select() returning early, or a *soon to expire* RTIMER, turns out in another excess at high offered loads.

For ux09, both the MAXA and the bombardment effects are at work for high offered loads. RTIMER is no longer a constraint, as it has always expired (because MAXA > RTIMER). Surely, each XTP_SEND request passed by the client must be replied to by the daemon. When the daemon yields to select() at some point, select() returns immediately (or almost) because of another incoming XTP_SEND user request. As RTIMER has expired, the packet is also sent immediately, and this cycle repeats itself resulting in a batch of back-to-back packets being sent. Here follows edited trace excerpts that support this interpretation (rate=200 Bpms, dahlia to orchid):

```
(30) 0x3000 DATApacket BEING SENT at 411146031
```

```
->RTIMER - start_rtimer()
 RTIMER=7, r_timer=411146039, now=411146032

->enq_timer() called
key=0x3000, timer_type=RTIMER, due_time=411146039

->print_timers_onq() called:
# of timers onq=2  now=411146032
key=   type=    due_time=  derived delta time to fire (ms)=
0x3000 RTIMER   411146039 7
0x3000 CTIMER   414745999 3599960

(31) 0x3000 RTIMER expired now=411146033

->read_first_timer_onq() called
timers_list_head->due_time=414745999
time left to fire: 3599966 ms
->satisfy() - returning shortest_NEW=3599966, now= 411146033

->shortest (top level do { }while(!ds) )= 3599966, now=411146033
 timeout (before select(...,&timeout))= 10000, now=411146034
 daemon going to sleep.......
 ...up on USERREQ 0 ms later, now=411146034

(32) XTPdaemon XTP_SEND request for 0x3000
(33) 0x3000 DATApacket BEING SENT at 411146034

->RTIMER - start_rtimer()
 RTIMER=7, r_timer=411146042, now=411146035
......

(34) 0x3000 RTIMER expired now=411146035
```

```
->satisfy() - returning shortest_NEW=3599963, now= 411146036

->shortest (top level do { }while(!ds) )= 3599963, now=411146036
 timeout (before select(...,&timeout))= 10000, now=411146036
 daemon going to sleep.......
 ...up on USERREQ 1 ms later, now=411146037

(35) XTPdaemon XTP_SEND request for 0x3000
(36) 0x3000 DATApacket BEING SENT at 411146037

->RTIMER - start_rtimer()
 RTIMER=7, r_timer=411146044, now=411146037
etc..
```

For a data rate of 200 Bpms, RTIMER value is 7.2 ms, which is shown at the beginning of the trace. When the daemon resumes execution (line (30)) as a result of one incoming XTP_SEND user request, the underlying data packet is being sent immediately and r_timer is armed. Then the daemon proceeds to check all timers and realizes that RTIMER has expired (necessary, because MAXA > RTIMER), but has to yield to select() as there are no more data to send. However, select() returns immediately because of another incoming user request and a packet is again sent immediately. This pattern repeats itself probably until all data have been sent.

ux10 - SXTP-1.5.1 modified, dahlia to orchid (SEL=25, MAXA=10)

For ux10, the SELECT_FLOOR threshold and the MAXA margin are respectively set to 25 ms and 10 ms. Similar to experiment ux07 (done without a MAXA margin), we could expect a taper off plateau at throughput = 57.6 Bpms (1440/25).

Table ux10.log, Table ux10.xy (refer to Section A), and Figure 40 show respectively the log summary, the data summary, and the saturation curve for the ux10 unicast experiment unit.

The taper off plateau occurs at throughput = 48 Bpms (rather than 57.6 Bpms, and probably for the same reasons as the ones invoked for ux07). It extends for only 3 points (50, 75, and 100 Bpms). Recall that it extended up to load = 700 Bpms for ux07. Then the throughput jumps to still higher values than for ux09. For example, at load=150 Bpms, throughput = 1272 Bpms, i.e., throughput is about 8.5 times more than the offered load. Clearly again, no more rate control is being exercised and the MAXA effect described previously for ux09 has taken over. The following trend now seems to emerge clearly:

1. The SELECT_FLOOR effect produces a constant throughput plateau (or a constant offered load producing the same throughput); the value of the SELECT_FLOOR threshold determines the "altitude" of the plateau (or its throughput value).

2. The value of the MAXA margin determines the length of the SELECT_FLOOR plateau (or how many points of the curve are included), or stated otherwise, at what offered load the throughput starts to explode to record high values (here at offered load=150 Bpms for a MAXA margin of 10 ms).

ux11 - SXTP-1.5.1 modified, dahlia to orchid (SEL=10, MAXA=10)

For ux11, the SELECT_FLOOR threshold and the MAXA margin are both set 10 ms. Similar to experiment ux08 (done without a MAXA margin), we could expect a taper off plateau at throughput = 144 Bpms (1440/10) because of the SELECT_FLOOR effect (recall that there was some disappointment for ux08).

Table ux11.log, Table ux11.xy (refer to Section A), and Figure 41 show respectively the log summary, the data summary, and the saturation curve for the ux11 unicast experiment unit.

The shape of the saturation curve for ux11 (Figure 41) is very similar to the one for ux10 (Figure 40). Similar to ux08 (and probably for the same reasons), there is a short taper off plateau at throughput = 72 Bpms (2 points, when it was 3 points for ux08). Then there is a drastic surge to again record high throughput values.

Indeed, the saturation curve for ux11 (Figure 41) is very similar to the one obtained for ux10 (Figure 40), and the same analysis applies. The impact of reducing the SELECT_FLOOR threshold value seems to consist only of increasing the "altitude" of the SELECT_FLOOR plateau (here at 48 Bpms).

ux19 - SXTP-1.5.1 modified, dahlia to orchid (SEL=10, MAXA=05)

Recall that experiment ux11 produced grossly a two part saturation curve (Figure 41) with each part separated by a huge gap in throughput presumably attributable to the MAXA effect. The purpose of experiment ux19 is to repeat experiment ux11, but with a 50% reduction in the value of the MAXA margin from 10 to 5 ms. As per Table 11, this change should result in shifting at most four points from the high plateau to the linearly increasing part of the curve.

Table ux19.log, Table ux19.xy (refer to Section A), and Figure 49 show respectively the log summary, the data summary, and the saturation curve for the ux19 unicast experiment unit.

As shown on Figure 49, the jump in throughput starts at rate = 250 Bpms (throughput = 989 Bpms). As compared to ux11, a 50% increase of the MAXA margin yields a "shift" of only two points (150 & 200), and even those two points are on a plateau by themselves with about the same throughput (144 Bpms). Consequently, the "return on investment" for reducing the MAXA margin is not very promising, given the current mix of command line arguments at least.

8.3.5 Unicast - lack of bombardment effect

Recall that most unicast experiments done with unmodified SandiaXTP-1.5.1 so far, such as experiment ux01, get higher throughputs than the SELECT_FLOOR plateau (parts 3 & 4 described for ux01) for a reason that we attributed to the bombardment effect. In turn, this bombardment phenomenon is presumed to be created through matching the user buffer size and the PDU size such that there would be approximately as many XTP_SEND user requests as there are packets to send, which creates an incessant flow of disturbances causing the select() system call to return much earlier than scheduled.

The goal of the following few experiments is to evaluate the impact of greatly reducing the frequency of the bombardment phenomenon, by specifying a user buffer size 10 times larger (14400 vs 1440 bytes used when talking of the bombardment effect). The expectations are that: (1) the throughput should not increase much beyond the SELECT_FLOOR plateau; (2) as the whole dynamics of the interactions client/daemon might get changed, that there might also be implications even for the rising part of the curve before the SELECT_FLOOR plateau, and also to the SELECT_FLOOR plateau itself.

Unmodified SandiaXTP is used, with machine orchid for sending and forest for receiving.

ux12 - SXTP-1.5.1 unmodified, orchid to forest (-b 14400)

The parameters used for ux12 are identical to ux01, except that the size of the user buffer is 10 times larger at 14400 bytes. For ux01, the taper off SELECT_FLOOR plateau was expected at throughput = 28.8 Bpms.

Table ux12.log, Table ux12.xy (refer to Section A), and Figure 42 show respectively the log summary, the data summary, and the saturation curve for the ux12 unicast experiment unit.

As one can observe on Figure 42, the resulting saturation curve is less than amenable to dynamic rate control because of its erratic shape. There is indeed a plateau at throughput = 28.8 Bpms, but the preceding and the following points show no relative consistency. Also as expected, the throughput does not rise significantly beyond the SELECT_FLOOR threshold for the later points.

Many trace inquiries and analysis (a time consuming process) would be needed to track down more precisely the ups and downs of this saturation curve. As the global expectation is verified, there is no further finer grained interpretation attempted.

By using a larger buffer size, one would normally expect that a better quality rate control be exercised, as there is less time consumed for interprocess communications between the daemon and the client, and also less boundary crossing between user level proceses and the operating system. However, the result is quite the opposite. For the particular mix of user level arguments used for ux12 (which are not odd at all), it would appear that unmodified SandiaXTP does a less than acceptable job at exercising rate control.

ux13 - SXTP-1.5.1 modified, orchid to forest (-b 14400)

Experiment ux13 is similar to ux12, but with the addition of a 10 ms MAXA margin. Hence modified SandiaXTP-1.5.1 is used, with the same mix of machines (orchid and forest).

Table ux13.log, Table ux13.xy (refer to Section A), and Figure 43 show respectively the log summary, the data summary, and the saturation curve for the ux13 unicast experiment unit.

As one can observe on Figure 43, the shape of the resulting saturation curve is much more regular than the one for ux12 (Figure 42). The rise to the 28.8 Bpms SELECT_FLOOR plateau is very linear with throughput close to offered load (about 93%). However, once reached, the SELECT_FLOOR plateau extends for only five points, from load=30 up to load=100 Bpms. Then there is a slight rise to a not so high second plateau where throughput fluctuates slightly around 32 Bpms, and includes about 66% of the points of the whole curve.

Let us try to explain the reasons of this low and stable second plateau. At a rate of 125 Bpms, where the higher plateau begins, the value of the RTIMER is 11.5 ms (1440/125). Given a burst of one packet, a 10 ms MAXA margin, and the particulars of the SandiaXTP rate control algorithm (see Figure 24), each execution pass of the algorithm by the daemon results in at most two packets being sent. Once a pass is completed, the daemon yields to select() for a 50 ms specified timeout period. As there are many fewer XTP_SEND user requests, select() is likely to return only after about the scheduled timeout period, resulting in this much lower throughput.

E'en if the rate is increased, say to 200 Bpms with a corresponding 7.2 ms RTIMER, a 10ms MAXA margin, and the *one packet* burst size, each execution pass of the algorithm by the daemon will just result again in no more than two packets being sent. This recurring pattern explains why the throughput remains stable for the greater part of the ux13 saturation curve.

Being very regular, the only apparent disadvantage of this saturation curve seems to be its low maximum throughput range.

ux14 - SXTP-1.5.1 unmodified, orchid to forest (-b 14400, SEL=10)

For experiment ux14, the SELECT_FLOOR threshold and the MAXA margin are both set to 10 ms. Hence, unmodified SandiaXTP-1.5.1 is used, with machine orchid for sending, and forest for receiving. With the SELECT_FLOOR threshold set to 10 ms (rather than 50 ms for ux13), the expectation is that the throughput should increase for many rate/loads.

Table ux14.log, Table ux14.xy (refer to Section A), and Figure 44 show respectively the log summary, the data summary, and the saturation curve for the ux14 unicast experiment unit.

As one can observe on Figure 44, the shape of the saturation curve is very similar to the one obtained for ux13 (Figure 43), but with many higher throughput values, as expected.

The "altitude" of the higher plateau is approximately at throughput = 164 Bpms (it was 33 Bpms for ux13). The first point of the higher plateau corresponds to load=150 Bpms, and a RTIMER of 9.6 ms (see Table 11). As explained for ux13, this marks the beginning of the MAXA effect where RTIMER has always expired when checked, and no increase of the offered load will improve the throughput significantly.

When compared to ux13, the "altitude" of the higher plateau is greater than the one for ux13 because of the much shorter waiting period between two consecutive execution passes of the algorithm by the daemon, which is caused by the much shorter 10 ms SELECT_FLOOR threshold (it was 50 ms for ux13).

ux16 - SXTP-1.5.1 unmodified, orchid to forest (-b 28800, SEL=10, MAXA=0)

The purpose of conducting experiments ux16 and ux17 is to test the behavior of the daemon under very different dynamic conditions. Recall that many experiments, such as ux01, were done with a minimal user buffer size of 1440 bytes, creating an incessant flow of XTP_SEND user requests. The size of the user buffer size is now set to 28800 bytes, thereby reducing the number of XTP_SEND user requests to one twentieth of what it was for many other experiments, such as ux01.

As for ux14, the SELECT_FLOOR threshold value for ux16 is set to 10 ms, which implies a taper off SELECT_FLOOR plateau at throughput = 144 Bpms (1440/10). The MAXA margin is set to 0 ms.
Table ux16.log, Table ux16.xy (refer to Section A), and Figure 46 show respectively the log summary, the data summary, and the saturation curve for the ux16 unicast experiment unit.

As shown on Figure 46, there is indeed a plateau corresponding approximately to the SELECT_FLOOR threshold value (i.e., 144 Bpms). However, there is no rise beyond this plateau, as such is the case for the experiments done with a user buffer size matched to the PDU size (see ux01 - Figure 31). The relatively large user buffer size justifies the particular shape of the ux16 saturation curve.

ux17 - SXTP-1.5.1 unmodified, orchid to forest (-b 28800, SEL=0, MAXA=0)

For ux17, the SELECT_FLOOR threshold value is set to 0 ms. The goal of conducting the experiment is to test the behavior of the daemon under limit conditions. Whether the daemon crashes because of the nil value conveyed to select(), or whether the daemon does a very good job through enlisting the cooperation of the operating system for incoming user requests or packets, and still checking the timers for expiration with a much improved time granularity and also sending the packets timely.

Table ux17.log, Table ux17.xy (refer to Section A), and Figure 47 show respectively the log summary, the data summary, and the saturation curve for the ux17 unicast experiment unit.

As shown on Figure 47, the daemon survived well. However, the shape of the resulting saturation curve is quite irregular, and does not fulfill the expectation formulated of an "ideal" rate control. More particularly, and despite the nil SELECT_FLOOR threshold value implying that there should be no SELECT_FLOOR plateau, there still appears to be one (though not so stable as when the SELECT_FLOOR threshold has low values - such as for ux01) approximately at throughput = 144 Bpms.

8.3.6 Multicast experiments/curves

The goal of the following few experiments is to test the behavior of SandiaXTP-1.5.1 (mainly unmodified SandiaXTP-1.5.1) for rate control experiments using the multicast mode of communication. The data underlying the saturation curves being gathered at the transmitter, and the task consisting of reliable data transfer, the global expectation is that the shapes of the multicast saturation curves should be very similar to the ones obtained for the unicast mode, such as for ux01.

Intuitively, two factors should reduce the throughput values of the resulting saturation curves: (1) the capacities of the slowest receiver should impose its pace to the whole group of receivers; the greater processing time at all hosts inherent to the greater complexity of the multicast mode.

Machine orchid is generally used for sending. The evolution of the set of multicast experiments consists of first producing a basic reference multicast saturation curve including only the slowest machine (daffodil) in the group of multicast receivers. Then, the next experiments include only the fastest machine (dahlia) in the group of receivers, and subsequently the two fastest machines (sunset and dahlia), and finally the full group of multicast receivers (machines dahlia, sunset, and daffodil).

Proceeding gradually in this manner, the hypothesis to be verified is that there should be a gradual decrease in the throughput values, particularly towards the end of the curves where the SELECT_FLOOR effect has no impact.

Some multicast experiments needed a much longer period of time for their realization. Whereas the task of conducting the unicast experiments could be automated with a shell script, the same method did not work for multicast experiments with more than one receiver in the group of receivers. Using the same shell script as for the unicast experiments, the most frequent problem encountered was that the transmitter would stall while in progress, say at the beginning of the data transfer job for load=400 Bpms, with the message:

```
reg: EXCNTXT: Exceeded the maximum number of contexts in daemon
```

The interpretation given to this message was the presence of zombie processes that accumulated and could not be released before the next data transfer job started, eventually reaching the default number of allowed contexts. The solution adopted consisted of starting and stopping the daemon for each experiment (each point of the saturation curve), with interactive intervention of the experimenter to start the daemon, each receiver and the transmitter client programs for each data transfer job.

mx01 - SXTP-1.5.1 unmodified, orchid to daffodil

As stated in the general introduction about the set of multicast experiments, the goal of experiment mx01 is to provide a basic reference multicast saturation curve with only the slowest receiver included in the group of multicast receivers. Unmodified SandiaXTP-1.5.1 is used, with machine orchid for sending and daffodil for receiving.

Table mx01.log, Table mx01.xy (refer to Section A), and Figure 51 show respectively the log summary, the data summary, and the saturation curve for the mx01 multicast experiment unit.

In conformity with the expectations, the shape of the multicast saturation curve for mx01 shown on Figure 51 is very similar to the shape of the unicast saturation curve obtained for ux01 (see Figure 31). Only the end parts of the two curves differ, particularly with regards to the throughput values of mx01 being higher than ux01. Daffodil being a faster machine than forest (see Table 10), this fact does not come as a surprise.

mx02 - SXTP-1.5.1 unmodified, orchid to dahlia

As compared to mx01, the only change introduced for mx02 consists of using machine dahlia for receiving. Table mx02.log, Table mx02.xy (refer to Section A), and Figure 52 show respectively the log summary, the data summary, and the saturation curve for the mx02 multicast experiment unit.

Again the resulting multicast saturation curve obtained for mx02 and shown on Figure 52 is very similar to the one obtained for ux01 (Figure 31) and no further comments are needed.

mx03 - SXTP-1.5.1 unmodified, orchid to dahlia/sunset

For mx03, the group of multicast receivers includes machines dahlia and sunset. Table mx03.log, Table mx03.xy (refer to Section A), and Figure 53 show respectively the log summary, the data summary, and the saturation curve for the mx03 multicast experiment unit.

As expected and stated previously, a trend seems to emerge for lower throughput values after the SELECT_FLOOR plateau. For instance, for rate=800 Bpms, the throughput for mx02 is 144 Bpms; it is 90 Bpms for ux03.

mx04 - SXTP-1.5.1 unmodified, dahlia to orchid/sunset/daffodil

For mx04, the group of multicast receivers includes three machines: dahlia, sunset, and daffodil. Table mx04.log, Table mx04.xy (refer to Section A), and Figure 54 show respectively the log summary, the data summary, and the saturation curve for the mx04 multicast experiment unit.

With mx04, the expected decrease in throughput corresponding to an increase in the number of receivers does not clearly materialize. The saturation curve for mx04 (3 receivers - Figure 54) shows a decrease in throughput when compared to mx02 (1 receiver - Figure 52), but an increase in throughput when compared to mx03 (2 receivers - Figure 53).

The trend that seems to emerge is the drop in throughput after a maximum has been reached for mx03 and mx04. This phenomenon is in accordance with the thrashing effect observed in other areas of study, such as for demand paging as introduced at the beginning of the report (Section 2). With mx03 and mx04, the so-called thrashing phenomenon starts at load=900 Bpms. For such a rate, RTIMER equals 1.6 ms.

mx05 - SXTP-1.5.1 modified, dahlia to orchid/sunset/daffodil

Modified SandiaXTP-1.5.1 is used for experiment mx05. The MAXA margin is set to 10 ms. The same set of machines is used as for mx04, i.e., orchid for sending; dahlia, sunset, and daffodil for receiving. The unicast equivalent (with the same 10 ms MAXA margin) is ux04, which yielded a very regular saturation curve (see Figure 34) having a low and a high plateau where throughput values are stable.

Table mx05.log, Table mx05.xy (refer to Section A), and Figure 55 show respectively the log summary, the data summary, and the saturation curve for the mx05 multicast experiment unit.

As one can observe on Figure 55, the shape of the resulting saturation curve is very irregular, and correlates very little with the expectations. The beginning of the curve, up to load=100 Bpms, has the familiar shape and expected throughput values of a unicast curve, such as ux01. Thereafter, the throughput jumps from low to high values without apparent pattern.

8.3.7 Other experiments/curves

ux15 - SXTP-1.5.1 unmodified, orchid to forest, WS=51200

Other unicast studies conducted at the HSP Lab. ([SUL], [FALCOT]) in a WAN/Internet environment have shown that a window size of 102400 bytes offers excellent throughput results. Consequently, a window size of 102400 bytes was used for most of the experiments reported in the present study.

The goal of experiment ux15 is to verify the expectation that a reduced window size should not significantly change the shape of the saturation curve, except for lower throughput values for the points situated beyond the SELECT_FLOOR plateau. The reason being that a smaller window size implies more restriction on the sender, and hence a longer period of time should be needed to complete the data transfer task. For the points of the curve included in the SELECT_FLOOR plateau and the previous ones, the rates are so low that there should be no influence at all.

Unmodified SandiaXTP-1.5.1 is used, with machine orchid for sending and forest for receiving.

Table ux15.log, Table ux15.xy (refer to Section A), and Figure 45 show respectively the log summary, the data summary, and the saturation curve for the ux15 multicast experiment unit.

As one can observe on Figure 45, the shape of the resulting saturation curve for ux15 is very similar to the one obtained for ux01 (with WS=102400 bytes - Figure 45). As expected, the higher throughput values for ux15 are lower than for ux01. For example, corresponding to load=800 Bpms, the resulting throughput value is 69 Bpms for ux15, while it is 96 Bpms for ux01.

8.4 Synthesis of the results and global evaluation

The goals of this subsection are twofold: (1) to review the problem of exercising rate control with SandiaXTP; (2) to evaluate the success of the changes introduced to the algorithm.

The experimental method consisted of recompiling the source code of the daemon as needed for the MAXA margin and the SELECT_FLOOR threshold, and also of varying command line parameters conveyed to the daemon from a user level program and contrast the results of both unmodified and modified SandiaXTP-1.5.1.

Rate control with SandiaXTP - Global problem structure

As any other software tool, SandiaXTP has capacities but also limitations that became quite apparent when concentrating on its rate control facilities. At the beginning of the report, Figure 5 led to anticipating that there would be difficulties with exercising proper rate control below a lower RTIMER threshold value. Table 13 shows the global problem structure that emerges as a result of having analysed the rate control mechanisms used by SandiaXTP, and also testing its behavior through experimentation.

One problem that caused concern was understandability of the algorithm (item 1 on Table 13) used when bubbling up a timeout value to the select() system call after a timer (RTIMER) has expired. Normally, when a timer expires, one would expect that the next timeout value bubbled up to select() would be a timer related timeout value, or at least that a timer (if any ongoing) be taken into consideration. Such is not the case with unmodified SandiaXTP-1.5.1. When a timer has expired, the discovery is naturally made after the fact. Consequently, the difference between r_timer and *now* yields a negative value for variable shortest. As no negative timeout value is allowed to be passed to the select() system call, a hardcoded minimum timeout value (50 ms) is passed to select(). The next execution pass of the algorithm is likely to bubble up a proper timer related timeout value though, but at the cost of one possible spurious call to select(), and another longer wait period if the lapse of time before RTIMER expires is short (say 3 ms).

Mainly to foster understandability of the rate control algorithm, a linked list of timers (for all timers) was introduced. Though tested only within the context of these rate control experiments, the operation of this linked list appears to be correct.

Table 13: Rate control with SandiaXTP - Global problem structure

<table>
<tr><td colspan="3"><b>Implementation architecture characteristics:</b> Being a user level implementation, the daemon has to yield to the operating system to be notified of incoming user requests, incoming packets, or after a timeout period, whichever comes first. The select() system call is used for this purpose. Each time that select() returns, the timers are checked for expiration.</td></tr>
<tr><th>Problem areas:</th><th>Elements of solution:</th><th>Evaluation/implications:</th></tr>
<tr><td>1. <b>Algorithm</b></td><td>Use a linked list of timers</td><td>Appears successful</td></tr>
<tr><td colspan="3">2 <b>SELECT_FLOOR threshold</b>: As per the design, select() should not be passed a timeout value below some threshold; otherwise it might never return (hardcoded to 50 ms with SXTP-1.5.1). Note: this constraint was not effectively observed during the experiments with Solaris2.5.</td></tr>
<tr><td>2.1 <b>SEL_FLOOR effect:</b> Given a fixed burst size, if the rate increases, then RTIMER decreases to a point where it becomes smaller than or equal to the SELECT_FLOOR threshold. When this occurs, the offered load remains constant, producing a plateau in throughput (SELECT_FLOOR plateau).</td><td>Use <b>harmonization</b> of rate and burst values such that RTIMER remains well above the SELECT_FLOOR threshold.
Reduce the SELECT_FLOOR threshold value as much as possible (could also limit RTIMER to a low value of 10 ms).</td><td>Works well. Somewhat conflicting objectives; when rate is high, burst should decrease so as to ease the strain on the network, whereas it increases when using harmonization.
Partly successful if done alone. There is still a plateau, but with less and less points included in it.</td></tr>
<tr><td>2.2 <b>Delay due to select() imprecision:</b> Often, select() returns a few ms before the specified timeout period, implying possibly another wait period.</td><td>A MAXA margin (say 5 ms).</td><td>Partly sucessful; no more rate control is possible when RTIMER ≤ MAXA. Could limit RTIMER to a low value (say 10 ms)</td></tr>
<tr><td>2.3 <b>Delay for soon to expire timers:</b> a timer to expire shortly (say 3 ms after being checked) could be acted upon much later because of the SELECT_FLOOR threshold.</td><td>Same</td><td>Same</td></tr>
<tr><td>3 <b>Hard limit:</b> In principle, no more rate control is possible when the time required to process a packet is equal or greater than RTIMER.</td><td>Nothing to be done, except use a faster machine.</td><td>Another reason to fix a lower limit to RTIMER.</td></tr>
</table>

Another important problem area that has much impact on the rate control quality that can be exercised with the SandiaXTP implementation, and that stems from the anticipated low limit conditions on RTIMER values, concerns the interaction between the operating system and the daemon through the select() system call. The description of these problems occupy most of the space of Table 13 (items 2, 2.1, 2.2, and 2.3) and are explored in further detail in the following subsections.

Finally, Table 13 (item 3) shows that there is a hard limit to rate control possibilities, which occurs when the time needed to process a packet is about equal to or exceeds RTIMER. On a relatively slow machine such as forest.cs.concordia.ca, this would imply that no rate control above the relatively low offered load of about 150 Bpms (see Table 11) can be exercised.

Pertinence of a MAXANTICIPATION margin

A MAXA margin was introduced as an attempt to compensate for: (1) the observed fact that select() sometimes returns slightly before the specified timeout period, resulting in another wait period (ex: suppose select() returns 3 ms early, then the daemon would yield for another 50 ms); (2) *soon to expire* timers, also resulting in another wait period (ex: when checked, suppose 7 ms is left before RTIMER expires, then the daemon would yield for possibly another wait period of about 50 ms). As indicated on Table 13, the results of this change are only partly successful, and we should consider two cases for evaluation purposes:

1. **When the bombardment is at work:** When comparing the throughputs obtained for ux01 (Table ux01.xy), where no MAXA margin is being used, and for ux04 (where a 10 ms MAXA margin is used), one can observe only a very slight throughput improvement of the order of 1%. For example, corresponding to load=10 Bpms, the throughput obtained for ux01 is 9.528 Bpms, and it is 9.623 for ux04. Such a meager improvement can be attributed to the fact that at low offered loads, most of the time is spent waiting for sending the next burst of packets anyway.

   On the other hand, analysis of the results for other experiments (such as ux09,

Table 14: Degree of precision of the select() system call on a TMOUT case

SELECT_FLOOR	Observed values on orchid (solaris2.5)	Average	Imprecision gap
(ms)	(ms)	(ms)	(ms)
150	149 150 152 147 150 148 150 151 149 149	149.5	-0.1
125	125 126 125 130 125 125 125 123 125 125	125.4	+0.4
100	095 097 095 096 095 095 096 093 095 096	095.3	-4.7
075	076 075 076 076 076 075 075 078 076 076	075.9	+0.9
050	046 047 045 046 046 046 046 046 046 047	046.1	-3.9
040	036 036 037 039 035 036 036 036 036 036	036.3	-3.7
030	027 025 026 025 025 025 026 026 026 026	025.7	-4.3
025	026 026 027 026 025 026 026 026 026 026	026.0	+1.0
020	016 016 016 016 016 016 016 016 016 015	015.9	-4.1
015	016 017 016 016 016 016 016 017 017 016	016.3	+1.3
010	004 002 006 006 010 006 006 009 006 006	006,1	-4.9
005	005 006 005 004 005 006 006 010 006 006	005.9	+0.9
000	000 001 000 001 000 000 000 000 000 000	000.2	+0.2

ux10, and ux11) have led to observing a nasty side effect of using a MAXA margin. When the offered load reaches high values such that RTIMER $\leq$ MAXA (as per Table 11, this occurs when load=150 Bpms and RTIMER=9.6), then there is a sudden jump to very high throughput values and effective rate control is probably not being exercised.

2. **When there is a lack of bombardment effect:** For those experiments (ux13 and ux14), the user buffer is much larger than the PDU size and the daemon does not benefit from this incessant upcoming of user requests (identified as the *bombardment effect*). For such a scenario, select() is likely to trigger on the specified `timeout` value much more often than for the previous case.

As one can observe on Figures 43 and 44, there is a noticeable improvement over the results obtained for ux01 (where a MAXA margin is not used). This suggest a potential use of a MAXA margin, given certain conditions.

Mainly as result of observing the side effect of the MAXA margin, special experiments were conducted with the intent of mapping more precisely the lapse of time that occur between the specified timeout value conveyed to select() and the moment that select()

Table 15: Benefits of lowering the SELECT_FLOOR threshold value

Experiment Number	SELECT_FLOOR threshold (ms)	SELECT_FLOOR plateau (Bpms)	# of points up to SF plateau
ux06	100	014.4	02
ux01	050	028.8	05
ux07	025	048.0	07
ux08	010	144.0	11
ux17	000	144.0	11

actually returns on a TMOUT case. Table 14 shows the results of such special purpose experiments, using machine orchid for sending. The discrete SELECT_FLOOR values cover the range of RTIMER values used for the experiments. A sample of ten actual observed return values are being recorded. As one can notice, the imprecision gap is never greater than 5 ms, and this suggests a more meaningful value (than 10 ms) for the MAXA margin when being used, which would also postpone the beginning of the MAXA effect.

Pertinence of varying the SELECT_FLOOR threshold value

Table 15 summarizes the benefits of lowering the SELECT_FLOOR threshold value. The point of comparison consists of observing the "altitude" at which the SELECT_FLOOR plateau occurs, and also the number of points that are included in the growing linear part of the saturation curve (i.e., part 1 described for ux01). As one can observe on Table 15, the results obtained respond well to the changes introduced to the SELECT_FLOOR threshold value.

Pertinence of harmonizing the rate and burst values

The SELECT_FLOOR effect problem is circumvented if RTIMER is not allowed to have a lower value than the SELECT_FLOOR threshold value. As shown by the ux03 and ux05 (Figures 33 and 35) saturation curves, harmonizing the rate and burst values such that the SELECT_FLOOR effect does not occur when the rate value increases is a winning tactic in terms of curve regularity. A curve that has the shape of

Figure 35 is highly amenable to dynamic rate control, as throughput keeps increasing regularly up to a point where it yields and does not significantly improve thereafter.

Intuitively though, there appears to be a drawback to using this policy for other purposes than simply conducting rate control experiments for testing the behavior of a network. The whole purpose of exercising rate control on a shared medium, such as the Internet for instance, would appear to ease the strain placed on the network while at the same time taking advantage of whatever residual capacity there might still be available. Burst control carries this self-constrait possibility one step further by providing for pacing as evenly as possible whatever data needs to be sent per unit of time. If the rate is increased, the burst value should be presumably be kept as low as possible in order to relieve the underlying network. Contrary to this philosophy, harmonization is done by increasing the burst value in some proportion to the increase in the rate value.

ux20 - SXTP-1.5.1 modified, orchid to forest, rate and burst harmonized

Given the lessons learned as result of conducting and analysing the previous experiments, the goal of conducting experiment ux20 is to obtain as a regular and linearly increasing saturation curve as possible (i.e., avoiding the idiosyncrasies of the SandiaXTP software as much as possible). The daemon is recompiled and run with a 0 ms SELECT_FLOOR threshold value and a 5 ms MAXA margin. Harmonization of rate and burst values is used as per a formula that approximates the burst values shown on Table 12 (i.e., burst=rate*1000*0.1).

Table ux20.log, Table ux20.xy (refer to Section A), and Figure 50 show respectively the log summary, the data summary, and the saturation curve for the ux20 unicast experiment unit.

As shown on Figure 50, the shape of the saturation curve satisfies the expectations very well, with some slightly higher throughput values than the offered load attributable to the 5 ms MAXA margin.

mx06 - SXTP-1.5.1 modified, orchid to dah/sun/daf, rate and burst harmonized

The reason for conducting experiment mx06 is similar to the ones explained for ux20, but for the multicast mode of communication. Table mx06.log, Table mx06.xy (refer to Section A), and Figure 56 show respectively the log summary, the data summary, and the saturation curve for the mx06 multicast experiment unit.

As shown on Figure 56, the shape of the saturation curve is also very regular and highly amenable to dynamic rate control. Consequently, the optimal mix used for the ux20 unicast experiment also appears to yield very interesting results for the multicast case.

At a value of 191 Bpms, the throughput starts to taper off corresponding to an offered load of 200 Bpms. A throughput of 191 Bpms corresponds to about half the maximum capacity recognized to machine daffodil on Table 10 (i.e., 400 Bpms). For mx06, daffodil is the slowest receiver. In addition to expecting the slowest receiver to pace the entire data transfer task, it is also expected that multicast be slower than unicast, the gain consisting of serving many receivers from a unique data transmission source.

9 Concluding remarks

Throughout the period of time devoted to the investigatory work and to the writing of this report, the two most important themes have been: (1) earliest in time, the *multicasting* aspect; (2) then later, increasingly, the *rate control* aspect. For various reasons, such as the non-feasibility of conducting multicast rate control experiments using the Internet/Mbone environment and also the desire of a better framework for interpreting former as well as current rate control experiments results, there was a gradual shift of emphasis from the multicast aspect to the rate control aspect.

The outlook of the present section is to take some distance from the details of the experiments and formulate tentative prescriptions for the future, which are presented around the following themes: (1) Significance of the report for XTP4.0; (2) Significance of the report for the SandiaXTP implementation; (3) Significance of the report for more rate control experiments with SandiaXTP.

9.1 Significance of the report for XTP4.0

Providing mechanisms - a blessing and a curse

As perceived by the author, one basic tenet underlying the design of XTP consists of "providing orthogonal mechanisms to the user, who is presumed to be in the best position to decide about the policies to use". As shown by this report, with the *one packet* fixed burst size choice and no boundaries or minima stated by the specification for any implementation to meet, this approach may lead to time consuming disappointments. Not only does the "provide mechanisms" design approach greatly complicate the work for designing and implementing the specification, the great variety of possible mixes creates a very complex situation to the "user" who then has the burden of mapping the ground and discovering the pitfalls that may arise. The feeling is that the specification should "limit" (or at the very least circumscribe) the degree of variety left to the "user" to explore with some bottom line criteria to be satisfied by any implementation. This does not imply a static state of affairs as some implementations could claim to achieve more, and the criteria given in the specification could be progressively upgraded.

Quantitative criteria for rate control

Rate control is one area where it is felt that the specification, by not providing measurable objectives and minima, can lead to time consuming disappointments. Ideal rate control is a nice selling argument, but it would appear as a result of these studies that it is doable only to some extent and within some limits of precision, even for a kernel level implementation because of the many other needs that a kernel has to support. For example, one hypothetical future version of the XTP specification could state some more specific/partly quantifiable requirement such as:

> "As tested on a LAN with the SoAndSo test, the quality of the rate control shall not decrease significantly for offered loads up to 150 Bpms and for a low limit RTIMER value of 10 ms."

Starting WTIMER again before receiving a reply

Another aspect, that may not concern the specification itself, consists of the observed fact that using SandiaXTP and a linked list of timers there could be more than one WTIMER inserted in the list. The inference is that a new WTIMER is started before a reply to a previous one was obtained. The pragmatic solution consisted of checking and removing from the queue a timer of a given type before inserting another one of the same type. The same solution eventually had to be adopted for the CTIMEOUT timer.

9.2 Significance of the report for SandiaXTP-1.5.1

Linked list of timers

The task of trying to understand the rate control mechanisms used by SandiaXTP-1.5.1 was delayed because of the peculiar use of variable *shortest*, which has three levels of visibility, and especially by the fact that shortest regularly returns a negative value (i.e., a non-timer related value) each time a timer expires. To foster understandability of the rate control algorithm, a linked list of timer was introduced, with timers sorted from earliest to latest to fire.

Though it does not have much impact on efficiency, this change being more in accordance with timer management techniques (such as call outs) could be permanently integrated to some later release of SandiaXTP. The main resistances to change will probably consist of: (1) more thorough testing needed to check the behavior of the management of the linked list for all timers and for multiple contexts and a busy daemon; (2) the multiple changes needed to the source code with regards to variable shortest. When the daemon is awakened on a timeout event, instead of cycling through all contexts for work to be done, the possibility of using the linked list of timers could be evaluated.

The SELECT_FLOOR threshold

As there is some noticeable gain, and no bad side effects were observed with Solaris2.5 and as low a value as 0 ms, creating the symbolic variable name *SELECT_FLOOR*, using it in the code as suggested in Section 6 and making it a tunable parameter could also be made a permanent feature of a future release of SandiaXTP. There could be a default value set in some header file (say XTPtypes.h - #define SELECT_FLOOR 50), but that could be overwritten with some command line arguments when starting the daemon (similar to the -d argument used for choosing the data delivery service).

MAXANTICIPATION margin

The experiments have also shown that there could be some usefulness of incorporating a MAXA margin as a criterion for declaring if a timer has expired when choosing a proper mix of options at the user level. Such a change could also be introduced permanently with some future release of SandiaXTP. It could only be done through the introduction of the symbolic *MAXANTICIPATION* variable name in some header file (say XTPtypes.h - #define MAXANTICIPATION 0), but set to a default value of 0 ms. The code itself would use a logic based on existence of a margin (as indicated in Section 6). A change for a recommended value of 5 ms could be activated through recompilation of the daemon's code.

9.3 Significance of the report for more rate control experiments with SandiaXTP

Interpretation framework

The report, through its presentation of the structure of the SandiaXTP implementation and more particularly the exposure of the rate control mechanisms, provides a valuable framework for the interpretation of the data obtained when conducting rate control experiments. Furthermore, by providing a model of "what is", it can also serve as a base for introducing changes of the type "what ought to be" for later dynamic rate control investigations.

The many additional trace statements with timestamps, and available with the -t option when starting the daemon, can also be used as such for later experiments, or serve as a base for introducing additional traces as needed. A warning though ! The size of the trace file can get very large very quickly. Poking into this data is a time consuming process that may lead to no insight unless "one has some idea of what one is looking for". Consequently, it should be used very carefully, such as for transmitting a relatively small amount of data.

Receipe for conducting rate control experiments

The many settings of daemon variables and mixes of application program command line arguments explored with the numerous experiments provide a mapping of what can be expected. More particularly, the settings for experiments ux20 and mx06 provide a sample for using the SandiaXTP software avoiding many of its idiosyncrasies and really testing the behavior of an underlying network.

Better interpretation of multicast rate control experiments results

The interpretation and experimentation efforts of the present study have finally concentrated more on the unicast results. The multicast mode was only explored from the angle of some high level expectations, such as that the slowest receiver should impose its pace to the whole multicast group. Some unicast findings, such as extending the conditions of ux20 to mx06, were also extrapolated to the multicast mode and proved quite successful.

However, even if SandiaXTP appears to work reasonably well for conducting multicast rate control experiments, it is felt that the quality of the interpretation of the results would gain from a more thorough investigation of the ways used by SandiaXTP to implement multicast. This work could be done using the same (or similar) object modeling techniques as the ones applied and presented earlier in the report for understanding the global structure of SandiaXTP and its rate control mechanisms. Such an effort could be part of another study, building on the current one.

A Experimental data and saturation curves

A.1 ux01

Table ux01.log - log summary (at sender)

Date: 98.09.08 | **Start:** 19:03:47 | **End:** 19:19:46 | **Duration: 16 min:** | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux01.xy - xy data summary (at sender)

9.528	14.289	17.574	23.782	28.719	28.805	28.754	28.850	28.843
10	15	20	25	30	35	50	75	100
28.824	28.925	29.082	29.569	29.486	29.779	29.493	31.093	31.500
125	150	200	250	300	350	400	500	600
31.631	95.109	111.102	100.796	95.282	108.966	127.254		
700	800	900	1000	1100	1200	1250		

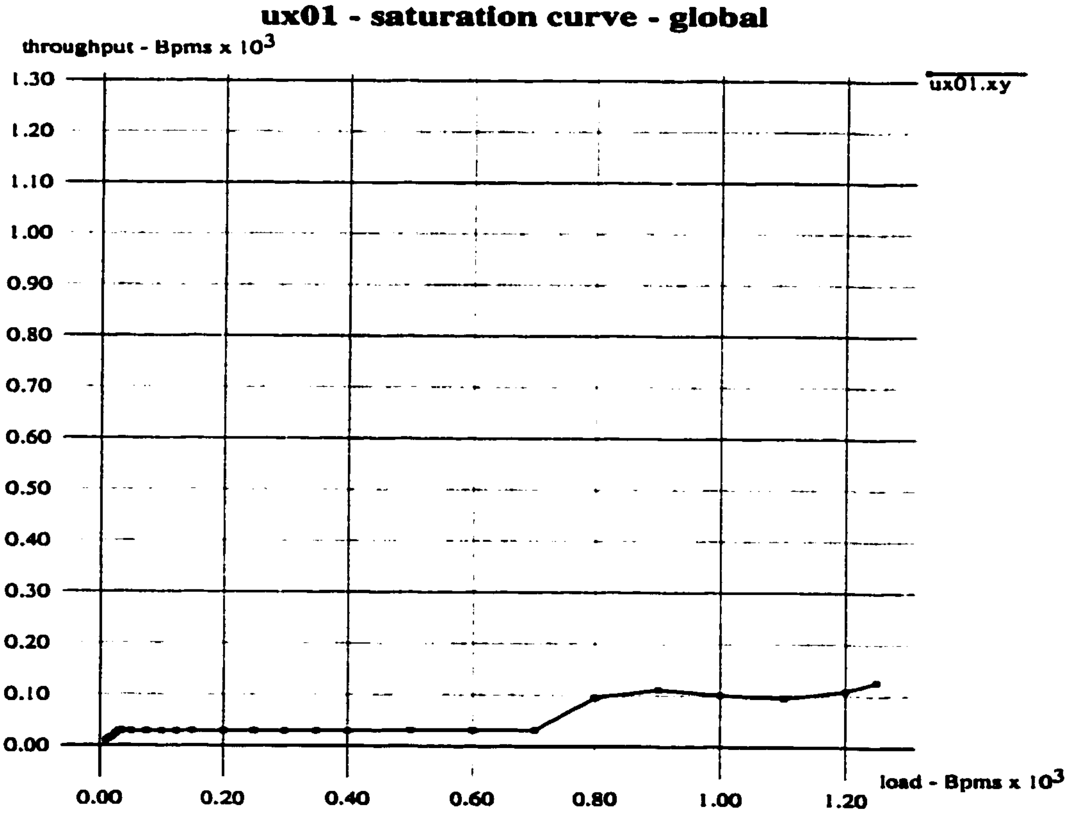


Figure 31: ux01 - saturation curve - global

A.2 ux02

Table ux02.log - log summary (at sender)

Date: 98.09.09 | **Start:** 16:44:43 | **End:** 17:02:08 | **Duration: 18 min:** | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.61 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: dahlia | **Receiver(s):** orchid 132.205.45.61

Table ux02.xy - xy data summary (at sender)

9.603	14.405	17.907	24.007	28.776	28.808	28.800	28.808	28.816
10	15	20	25	30	35	50	75	100
28.808	28.816	28.807	28.855	28.886	28.808	28.807	29.168	29.169
125	150	200	250	300	350	400	500	600
29.169	46.707	46.498	45.195	45.003	45.391	46.293		
700	800	900	1000	1100	1200	1250		

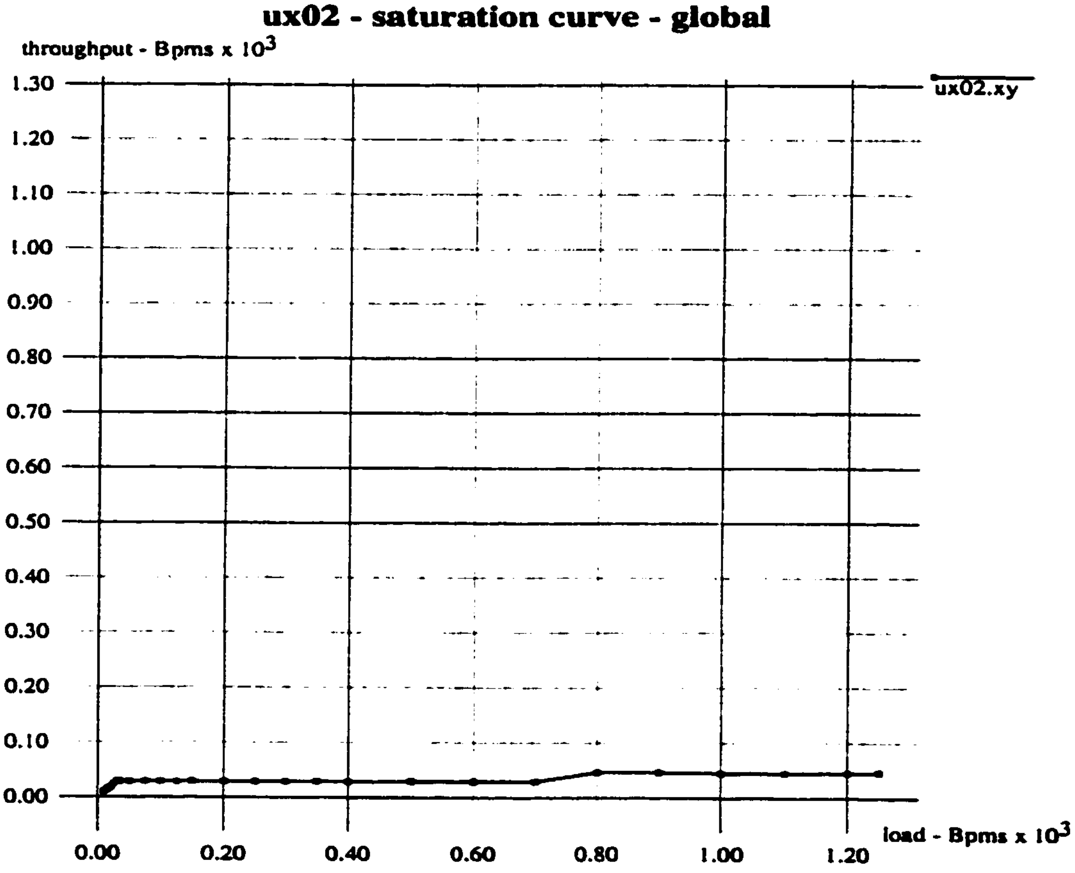


Figure 32: ux02 - saturation curve - global

A.3 ux03

Table ux03.log - log summary (at sender)

Date: 98.09.10 | **Start:** 14:58:02 | **End:** 15:07:53 | **Duration: 10 min:** | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = Yes
Commands: mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 103680 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux03.xy - xy data summary (at sender)

9.521	14.257	17.421	23.551	28.224	22.573	46.581	68.134	91.635
10	15	20	25	30	35	50	75	100
85.815	111.090	170.667	221.079	208.423	209.297	211.066	217.186	215.225
125	150	200	250	300	350	400	500	600
216.424	216.737	212.693	224.294	214.564	213.255	218.044		
700	800	900	1000	1100	1200	1250		

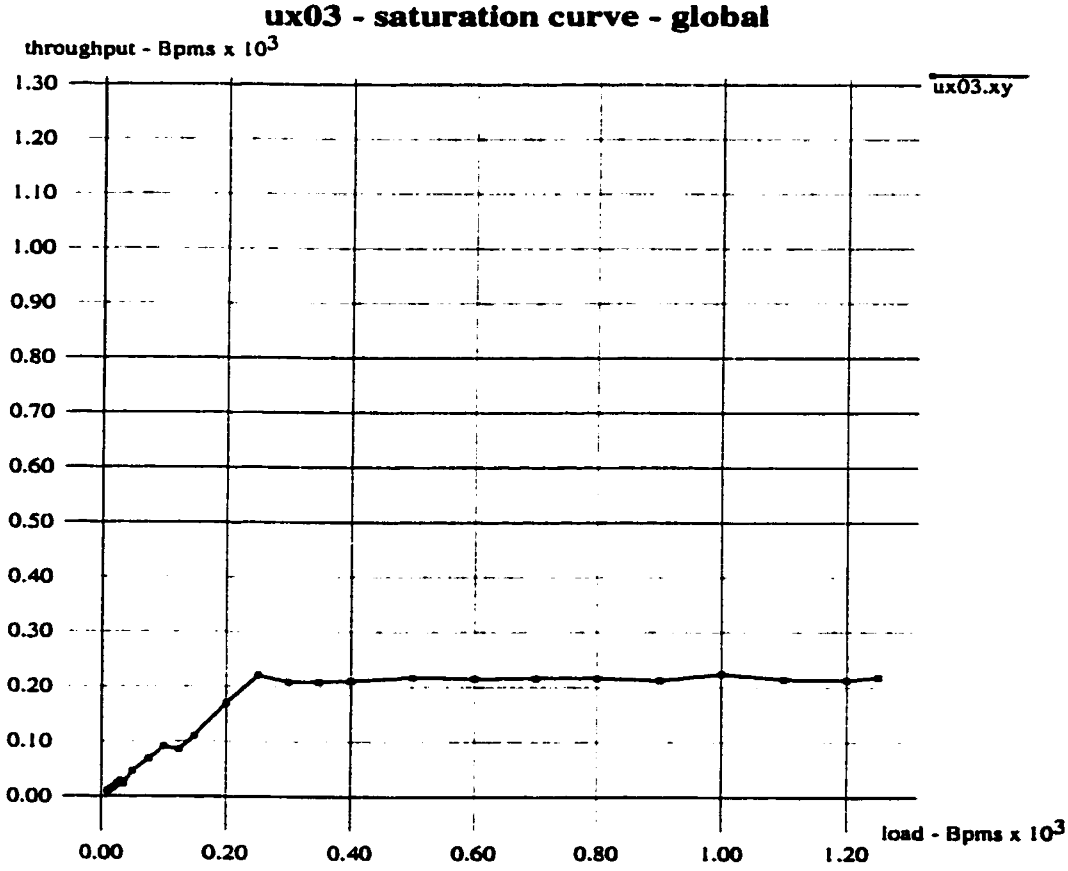


Figure 33: ux03 - saturation curve - global

A.4 ux04

Table ux04.log - log summary (at sender)

Date: 98.09.09 | **Start:** 21:14:42 | **End:** 21:25:16 | **Duration: 11 min:** | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 10 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux04.xy - xy data summary (at sender)

9.623	14.418	18.124	24.038	28.802	28.794	28.813	28.890	29.346
10	15	20	25	30	35	50	75	100
70.261	188.220	209.005	211.066	209.338	205.523	211.364	207.310	204.242
125	150	200	250	300	350	400	500	600
207.886	208.672	207.968	207.516	207.721	209.046	209.380		
700	800	900	1000	1100	1200	1250		

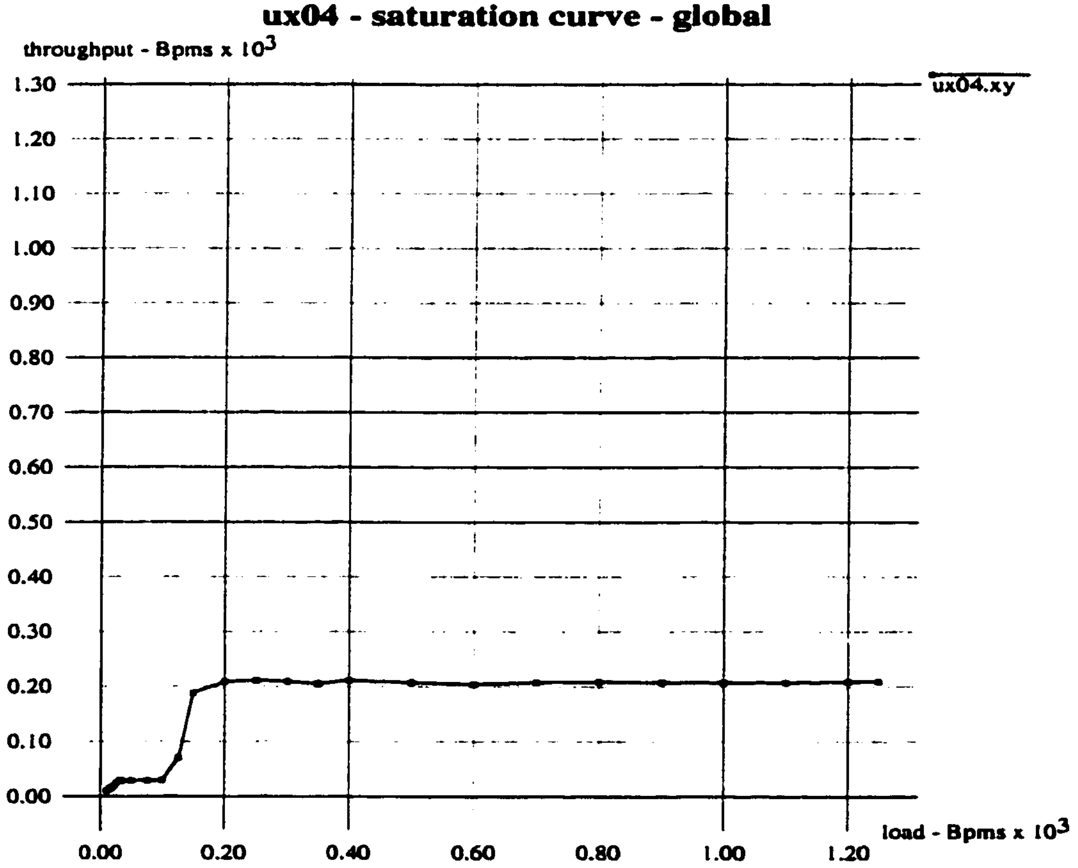


Figure 34: ux04 - saturation curve - global

A.5 ux05

Table ux05.log - log summary (at sender)

Date: 98.09.10 | **Start:** 15:35:16 | **End:** 15:44:35 | **Duration:** 09 min: | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 10 ms, Harmonization Burst/Rate = Yes
Commands: mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 103680 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux05.xy - xy data summary (at sender)

9.621	14.304	18.125	24.121	28.924	32.918	48.049	73.553	98.476
10	15	20	25	30	35	50	75	100
126.900	155.945	198.106	232.037	216.023	214.916	222.722	227.457	227.556
125	150	200	250	300	350	400	500	600
222.439	220.335	239.729	232.191	219.919	219.047	219.919		
700	800	900	1000	1100	1200	1250		

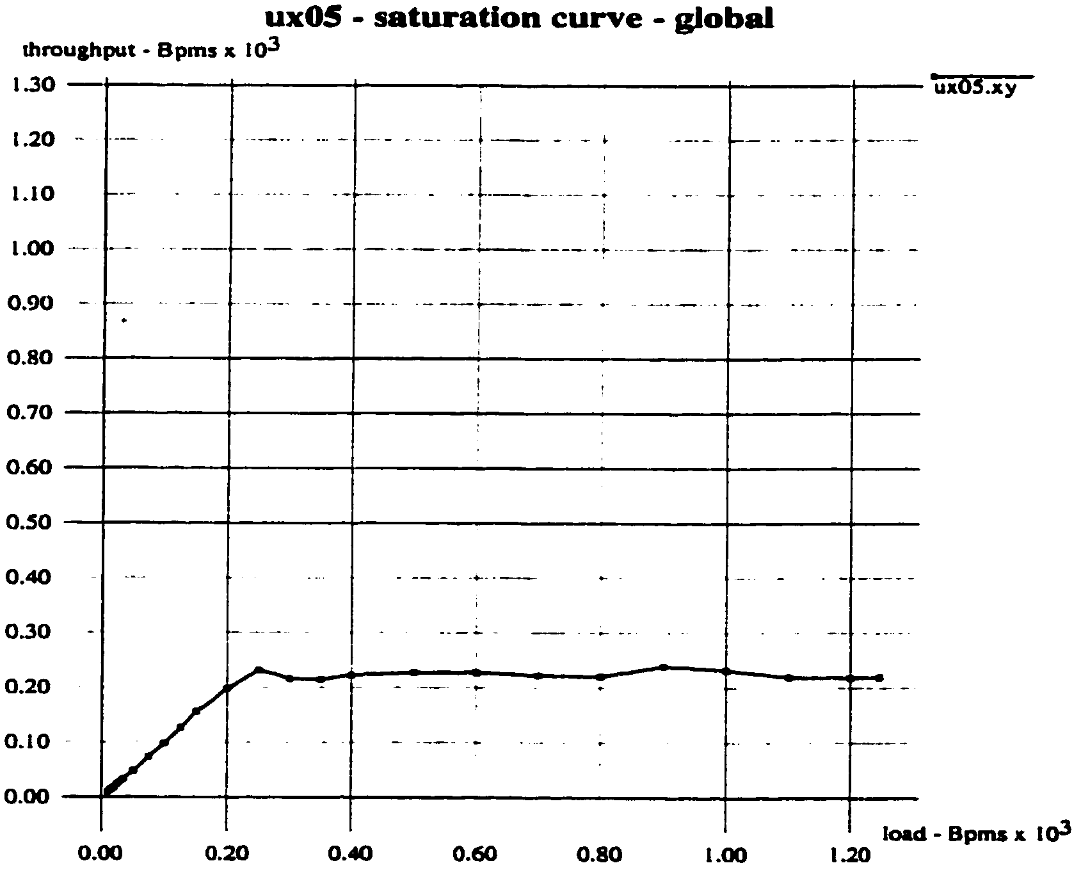


Figure 35: ux05 - saturation curve - global

A.6 ux06

Table ux06.log - log summary (at sender)

Date: 98.09.10 | **Start:** 16:02:04 | **End:** 16:31:23 | **Duration:** 29 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 100 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.61 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: dahlia | **Receiver(s):** orchid 132.205.45.61

Table ux06.xy - xy data summary (at sender)

9.603	14.404	14.405	14.402	14.400	14.404	14.405	14.404	14.405
10	15	20	25	30	35	50	75	100
14.404	14.402	14.404	14.404	14.405	14.404	14.424	14.564	14.605
125	150	200	250	300	350	400	500	600
14.605	25.390	26.215	25.955	22.147	21.892	22.311		
700	800	900	1000	1100	1200	1250		

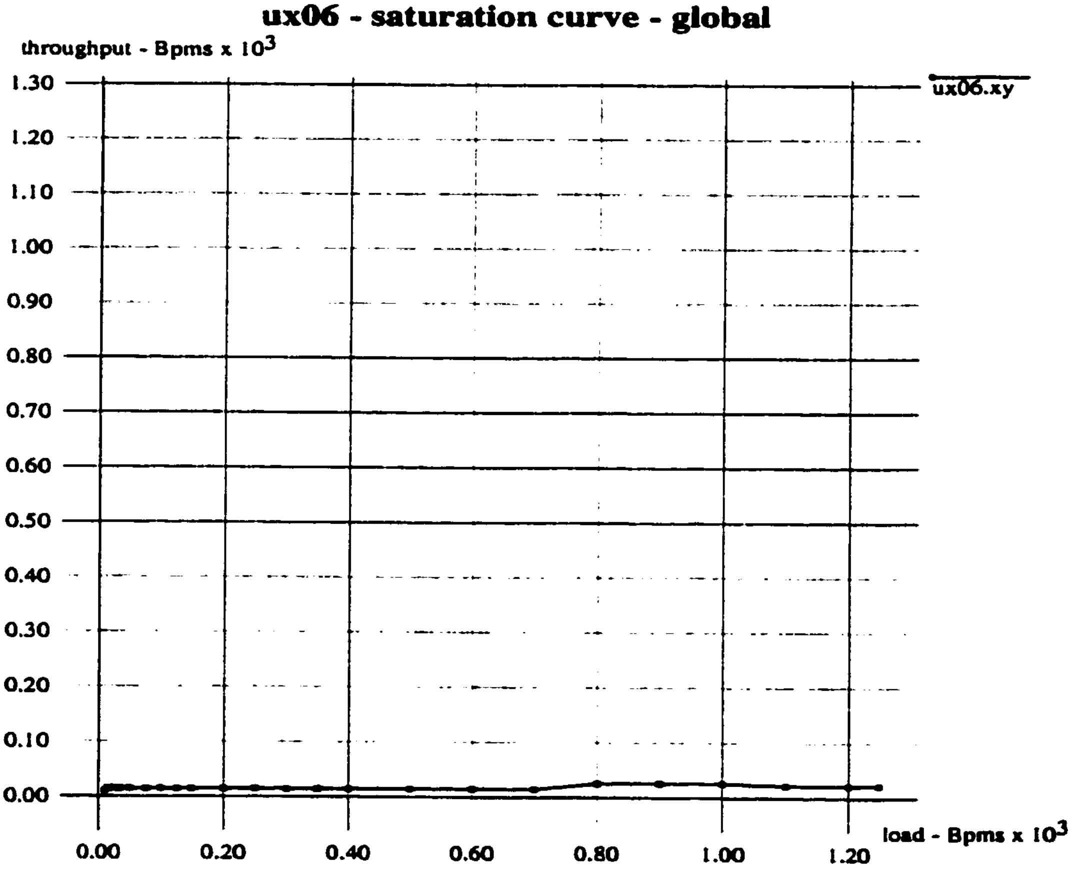


Figure 36: ux06 - saturation curve - global

A.7 ux07

Table ux07.log - log summary (at sender)

Date: 98.09.10 | **Start:** 17:08:49 | **End:** 17:22:21 | **Duration:** 14 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 25 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.61 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: dahlia | **Receiver(s):** orchid 132.205.45.61

Table ux07.xy - xy data summary (at sender)

9.601	14.401	17.941	23.996	28.776	24.123	47.878	47.988	48.140
10	15	20	25	30	35	50	75	100
47.968	47.988	47.966	48.117	48.098	48.120	48.120	48.386	48.769
125	150	200	250	300	350	400	500	600
48.520	76.199	74.357	75.048	73.838	73.112	72.914		
700	800	900	1000	1100	1200	1250		

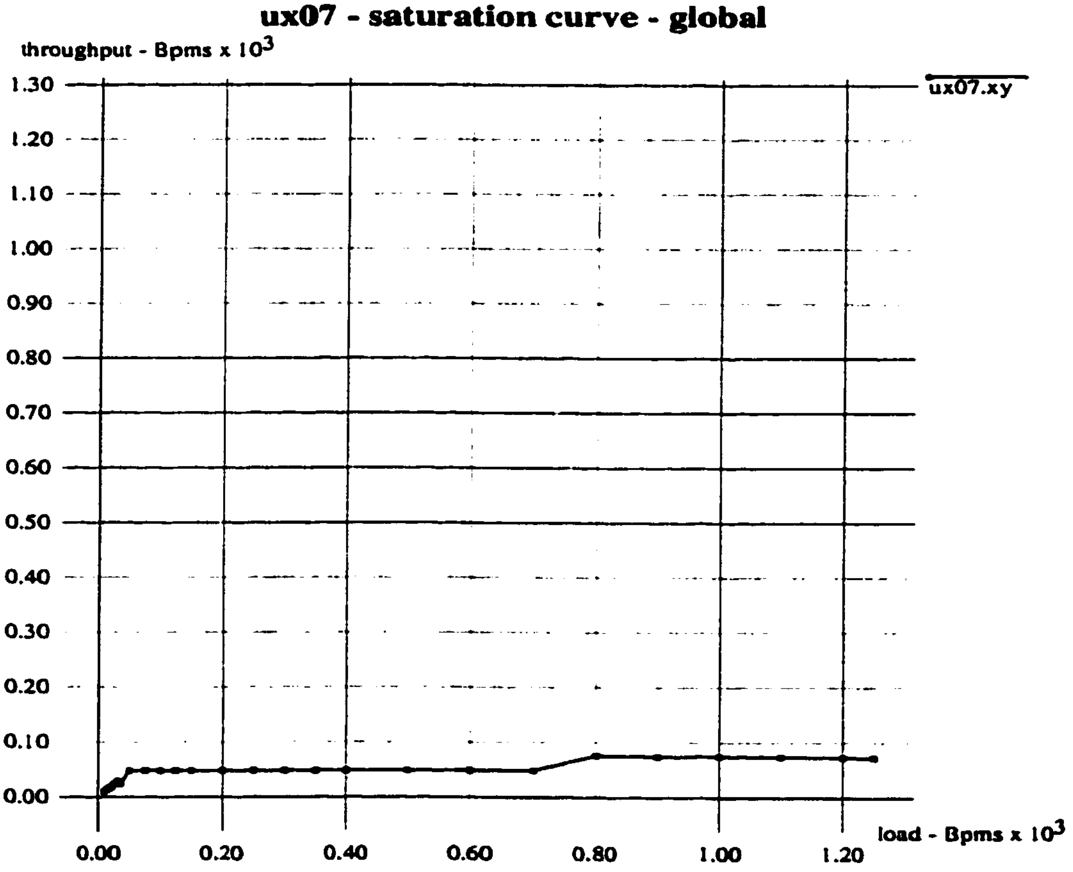


Figure 37: ux07 - saturation curve - global

A.8 ux08

Table ux08.log - log summary (at sender)

Date: 98.09.10 | **Start:** 18:21:00 | **End:** 18:30:58 | **Duration:** 10 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 10 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.61 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: dahlia | **Receiver(s):** orchid 132.205.45.61

Table ux08.xy - xy data summary (at sender)

9.603	14.405	18.006	24.006	28.809	28.831	47.856	71.712	71.909
10	15	20	25	30	35	50	75	100
72.062	143.582	143.582	143.385	143.582	143.582	143.582	144.174	144.771
125	150	200	250	300	350	400	500	600
144.971	151.682	209.213	295.874	171.001	166.891	263.925		
700	800	900	1000	1100	1200	1250		

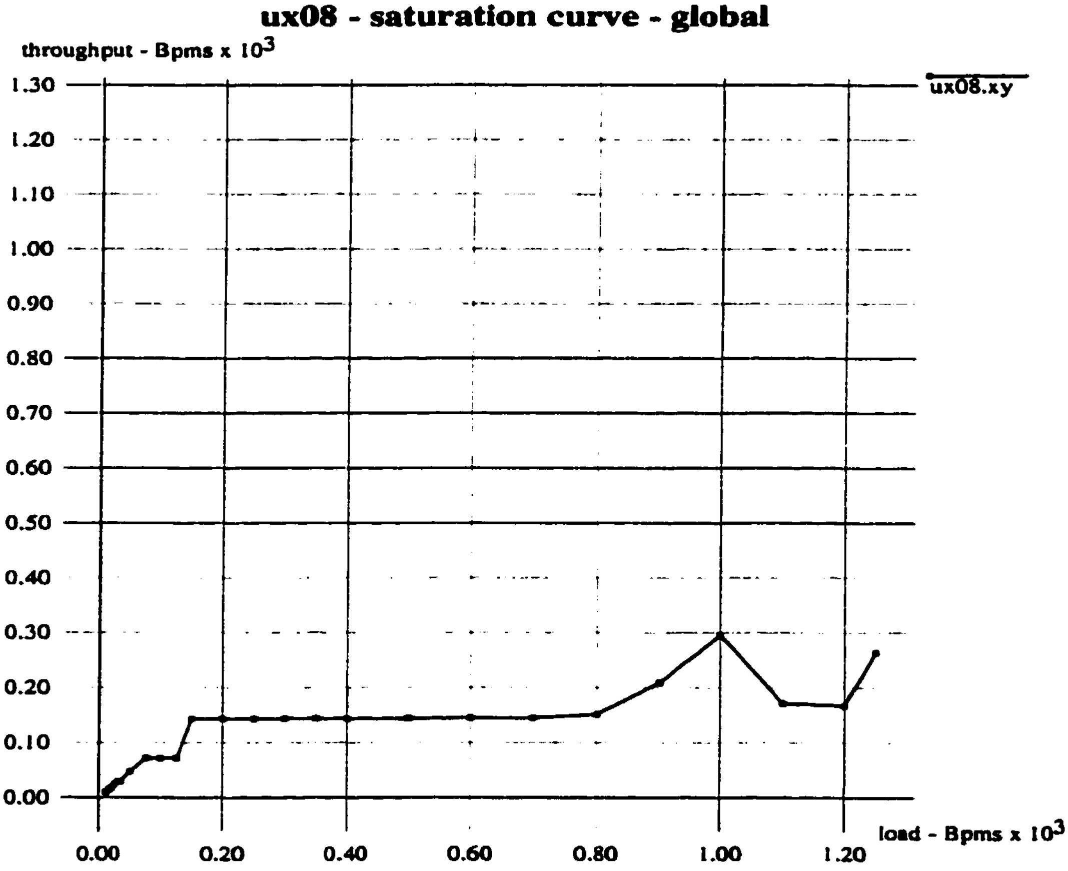


Figure 38: ux08 - saturation curve - global

A.9 ux09

Table ux09.log - log summary (at sender)

Date: 98.09.10 | **Start:** 18:50:43 | **End:** 19:04:34 | **Duration:** 14 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 100 ms MAXANTICIPATION = 10 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.61 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: dahlia | **Receiver(s):** orchid 132.205.45.61

Table ux09.xy - xy data summary (at sender)

9.603	14.405	14.405	14.405	14.405	14.405	14.406	14.424	14.425
10	15	20	25	30	35	50	75	100
21.756	1025.001	1022.004	1227.841	1197.005	1180.829	1205.260	1286.596	1248.305
125	150	200	250	300	350	400	500	600
1020.016	1156.093	1180.829	1192.919	1296.138	1183.494	1148.495		
700	800	900	1000	1100	1200	1250		

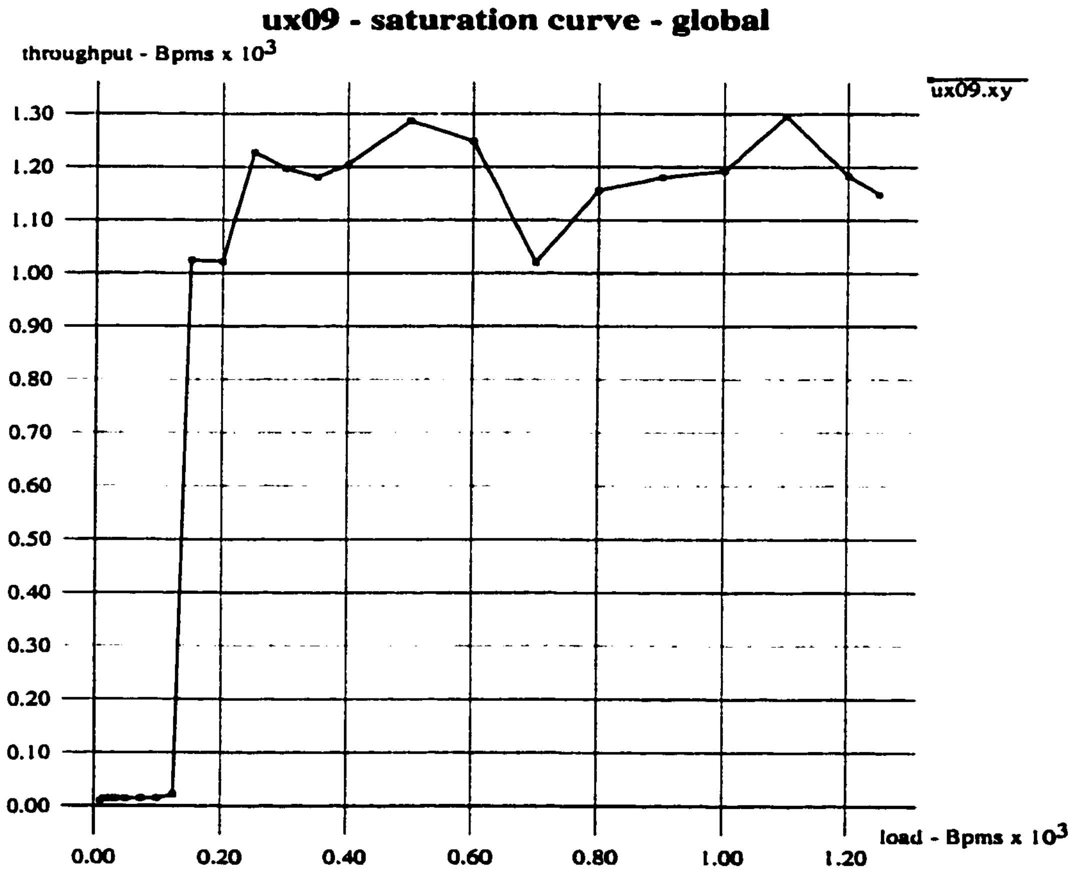


Figure 39: ux09 - saturation curve - global

A.10 ux10

Table ux10.log - log summary (at sender)

<table>
<tr><td><b>Date:</b> 98.09.10 | <b>Start:</b> 19:16:32 | <b>End:</b> 19:25:14 | <b>Duration:</b> 09 min | <b>Mode:</b> Unicast</td></tr>
<tr><td><b>Synopsis:</b> Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 25 ms
MAXANTICIPATION = 10 ms, Harmonization Burst/Rate = No</td></tr>
<tr><td><b>Commands:</b>
mmetric -t 132.205.45.61 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10..
mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..</td></tr>
<tr><td><b>Sender:</b> dahlia | <b>Receiver(s):</b> orchid 132.205.45.61</td></tr>
</table>

Table ux10.xy - xy data summary (at sender)

9.603	14.406	18.033	24.012	28.816	32.778	47.988	47.988	48.007
10	15	20	25	30	35	50	75	100
76.477	1285.020	1172.904	1263.345	1489.455	1232.169	1263.345	1239.452	1269.462
125	150	200	250	300	350	400	500	600
1227.841	1272.544	1191.564	1246.820	1305.823	1300.963	1172.904		
700	800	900	1000	1100	1200	1250		

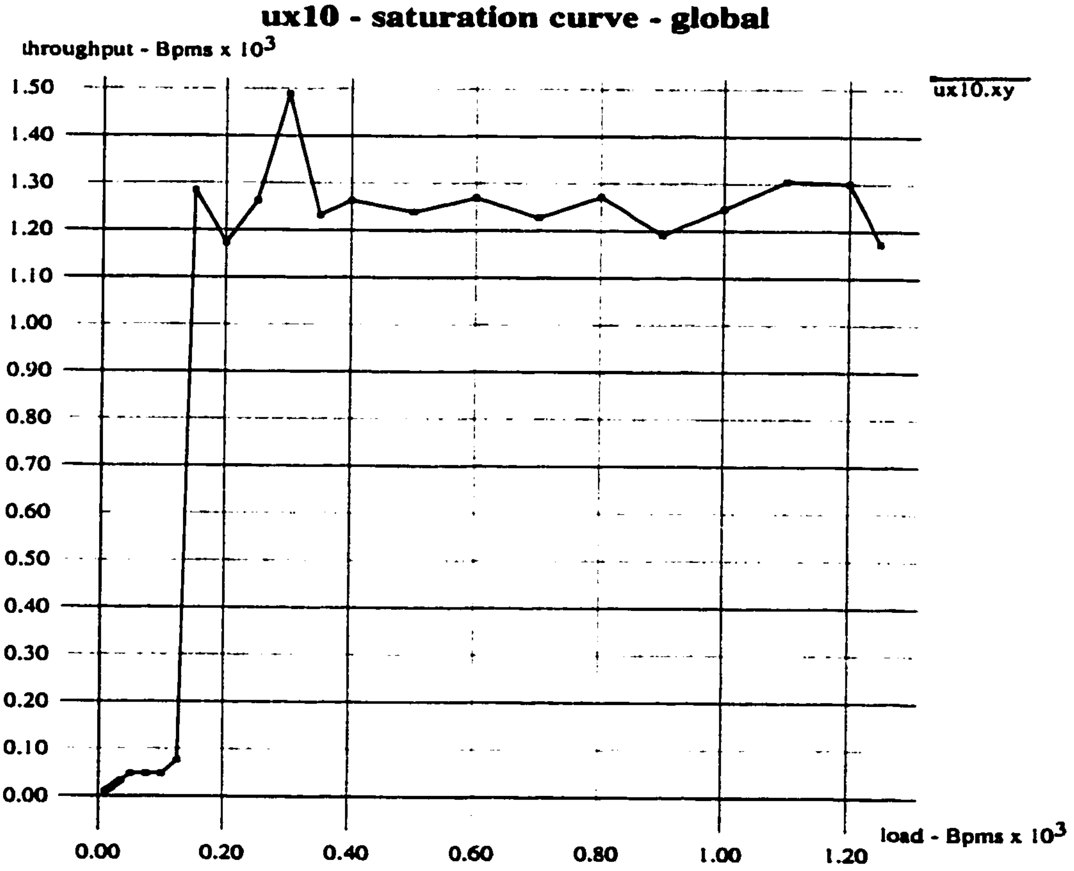


Figure 40: ux10 - saturation curve - global

A.11 ux11

Table ux11.log - log summary (at sender)

Date: 98.09.10 | **Start:** 19:31:39 | **End:** 19:40:02 | **Duration:** 09 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 10 ms MAXANTICIPATION = 10 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.61 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: dahlia | **Receiver(s):** orchid 132.205.45.61

Table ux11.xy - xy data summary (at sender)

9.603	14.405	18.039	23.610	28.808	33.026	47.985	71.953	71.958
10	15	20	25	30	35	50	75	100
110.179	1264.869	1235.072	1240.918	1255.780	1245.340	1242.389	1223.543	1245.340
125	150	200	250	300	350	400	500	600
1229.280	1261.824	1267.927	1272.544	1245.340	1305.823	1255.780		
700	800	900	1000	1100	1200	1250		

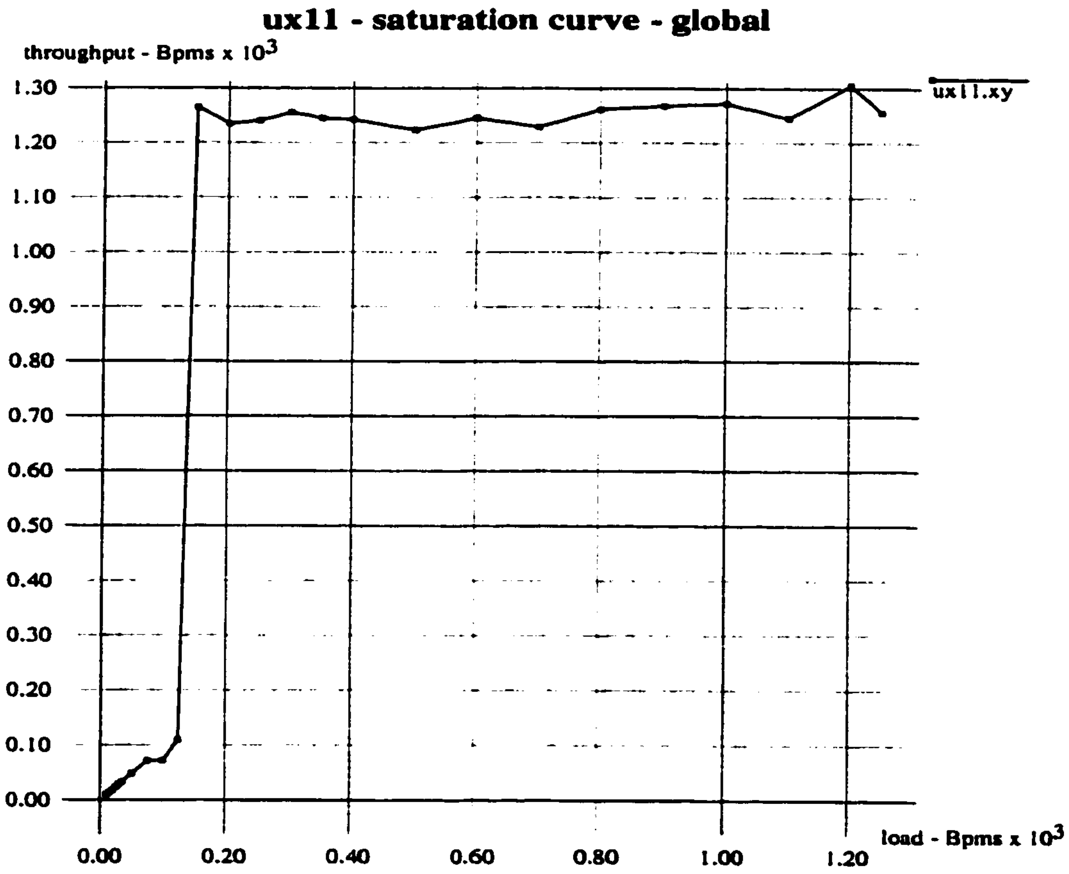


Figure 41: ux11 - saturation curve - global

A.12 ux12

Table ux12.log - log summary (at sender)

Date: 98.09.10 | **Start:** 19:48:33 | **End:** 20:13:57 | **Duration:** 25 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 14400 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 14400 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux12.xy - xy data summary (at sender)

8.756	14.338	14.233	13.086	13.752	28.799	28.772	28.798	28.824
10	15	20	25	30	35	50	75	100
28.798	28.910	13.824	29.323	12.976	11.012	17.178	23.127	25.255
125	150	200	250	300	350	400	500	600
29.875	20.153	24.763	16.888	20.115	16.888	16.931		
700	800	900	1000	1100	1200	1250		

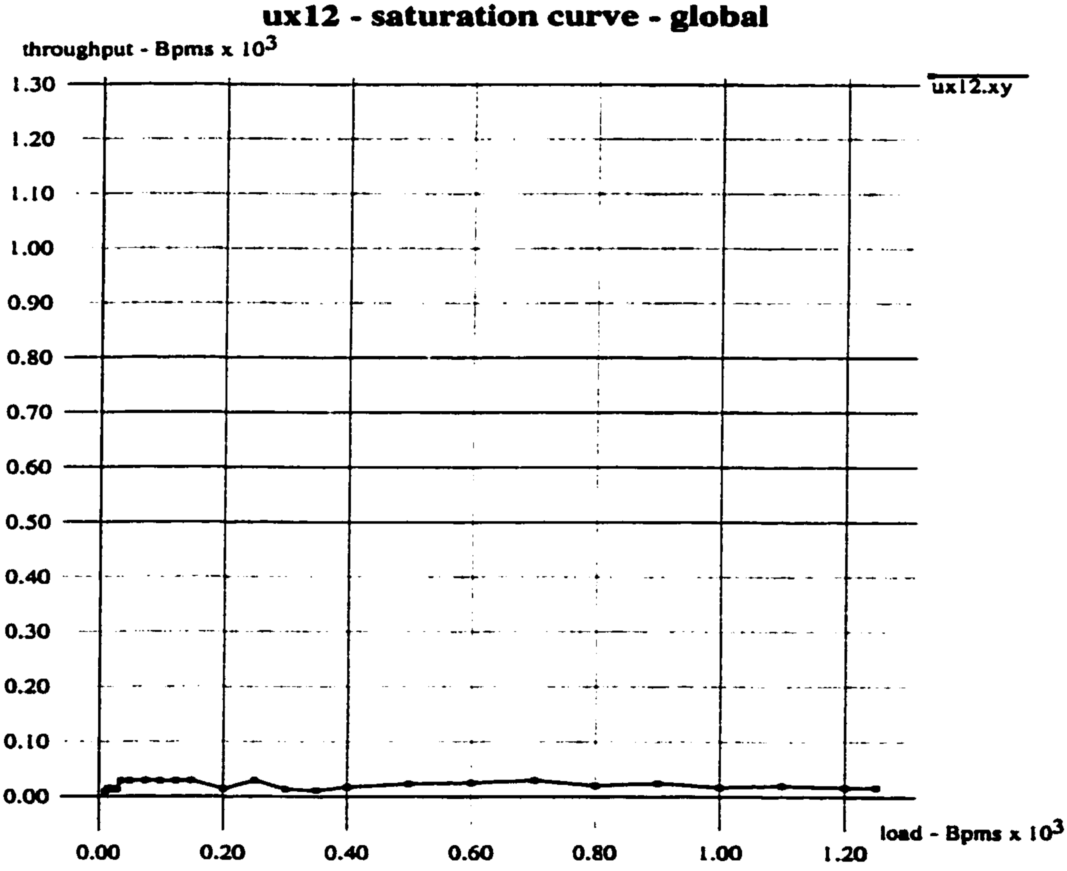


Figure 42: ux12 - saturation curve - global

A.13 ux13

Table ux13.log - log summary (at sender)

Date: 98.09.10 | **Start:** 20:23:44 | **End:** 20:41:19 | **Duration:** 18 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 10 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 14400 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 14400 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux13.xy - xy data summary (at sender)

9.607	14.409	18.132	23.984	28.786	28.759	28.726	28.894	29.660
10	15	20	25	30	35	50	75	100
32.645	32.705	32.933	32.900	32.899	32.901	32.889	32.740	32.914
125	150	200	250	300	350	400	500	600
32.554	32.818	32.911	32.859	32.918	32.879	32.889		
700	800	900	1000	1100	1200	1250		

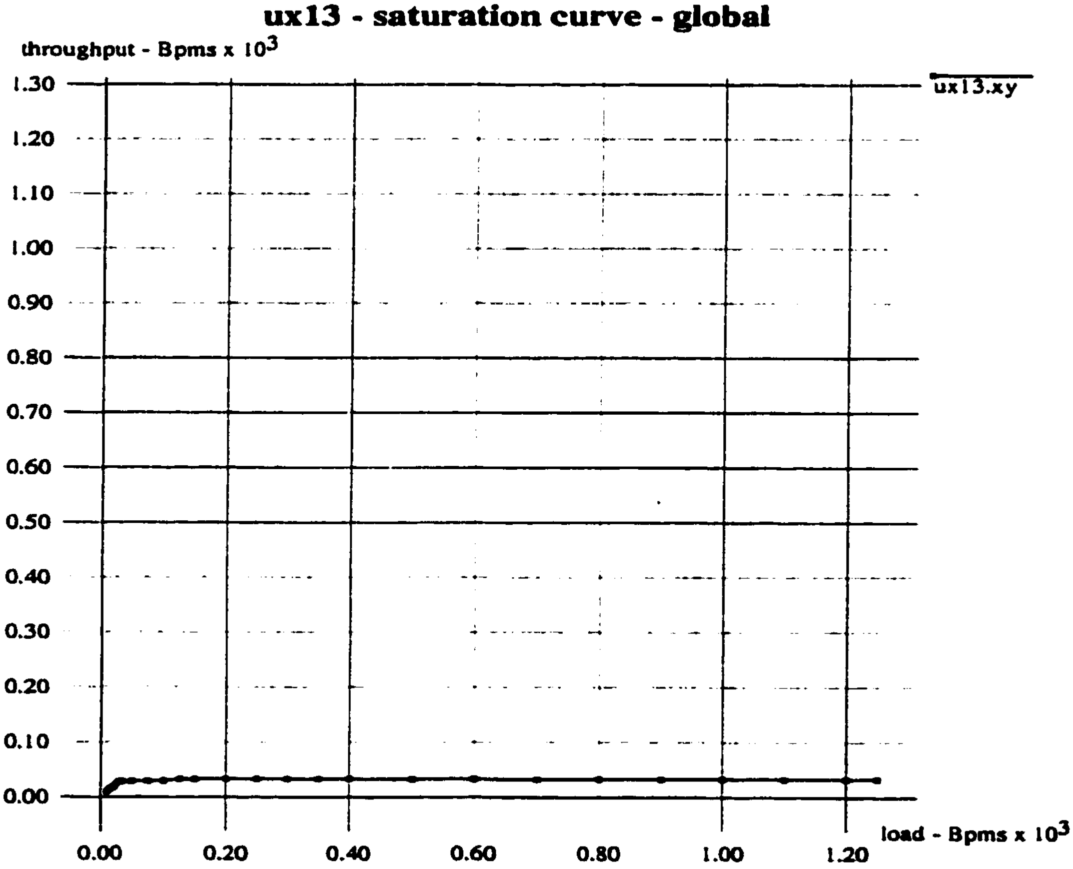


Figure 43: ux13 - saturation curve - global

A.14 ux14

Table ux14.log - log summary (at sender)

Date: 98.09.10 | **Start:** 21:02:30 | **End:** 21:12:24 | **Duration:** 10 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 10 ms MAXANTICIPATION = 10 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 14400 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 14400 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux14.xy - xy data summary (at sender)

9.608	14.375	18.098	23.996	28.737	29.546	48.069	71.963	73.693
10	15	20	25	30	35	50	75	100
81.665	164.845	163.968	163.712	164.019	164.328	164.767	164.328	164.096
125	150	200	250	300	350	400	500	600
163.814	164.019	163.482	163.891	164.251	163.000	164.276		
700	800	900	1000	1100	1200	1250		

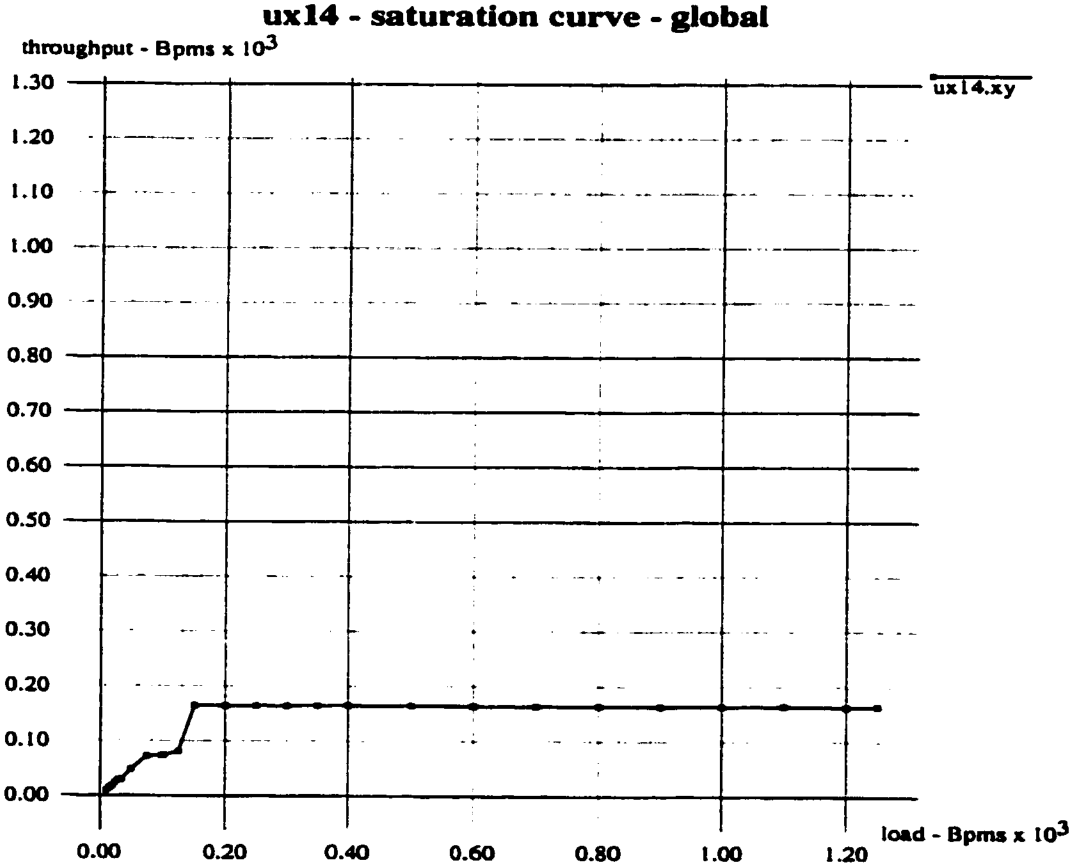


Figure 44: ux14 - saturation curve - global

A.15 ux15

Table ux15.log - log summary (at sender)

Date: 98.09.10 | **Start:** 21:18:07 | **End:** 21:34:31 | **Duration:** 16 min | **Mode:** Unicast

Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms
MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No, **WS=51200**

Commands:

```
mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 51200 -c 10..
mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 51200 -j 10..
```

Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux15.xy - xy data summary (at sender)

9.453	14.197	16.909	23.286	28.756	28.764	28.790	28.790	28.748
10	15	20	25	30	35	50	75	100
28.813	28.973	29.394	30.261	30.200	30.243	30.476	32.419	32.232
125	150	200	250	300	350	400	500	600
31.712	69.382	71.259	70.417	65.618	88.079	77.283		
700	800	900	1000	1100	1200	1250		

ux15 - saturation curve - global

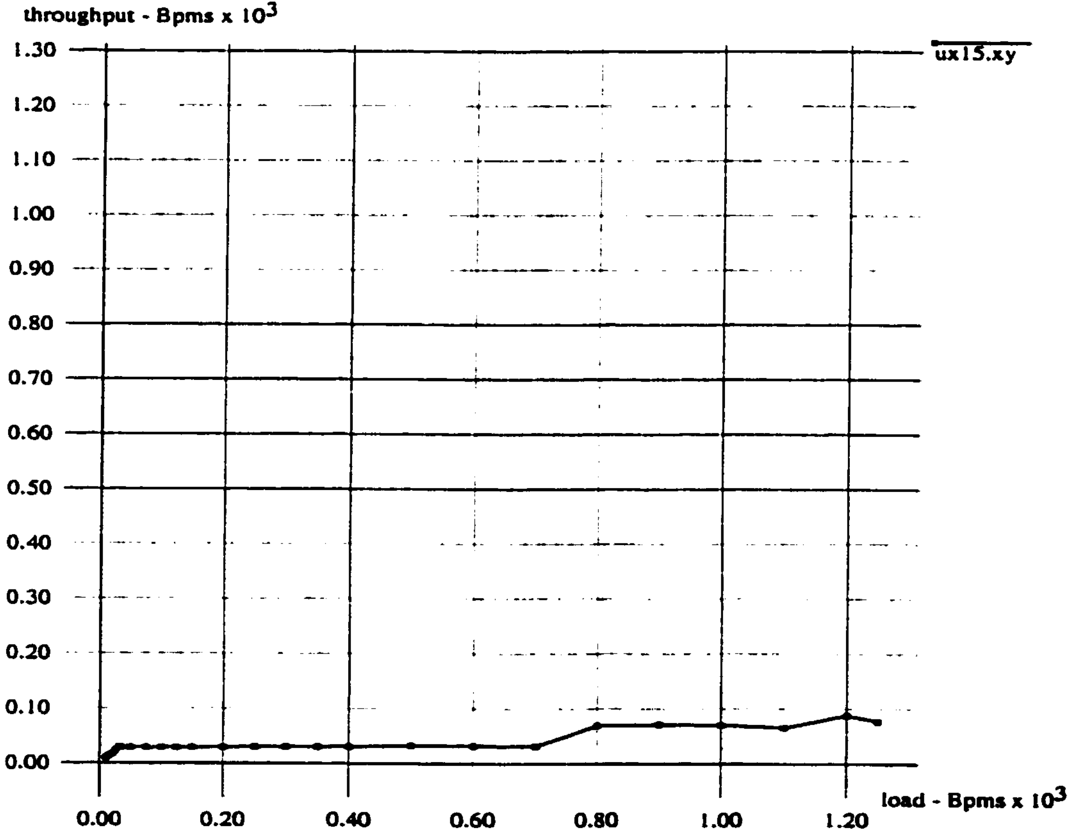


Figure 45: ux15 - saturation curve - global

A.16 ux16

Table ux16.log - log summary (at sender)

Date: 98.09.17 | **Start:** 23:07:46 | **End:** 23:17:57 | **Duration:** 10 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 10 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.24 -S -g -f -p 1472 -b **28800** -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 28800 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux16.xy - xy data summary (at sender)

9.586	14.397	18.004	23.973	28.800	29.094	47.898	71.619	71.958
10	15	20	25	30	35	50	75	100
72.949	141.318	143.542	144.392	140.390	146.429	130.534	135.056	146.001
125	150	200	250	300	350	400	500	600
146.449	151.946	153.054	154.863	148.755	152.343	133.849		
700	800	900	1000	1100	1200	1250		

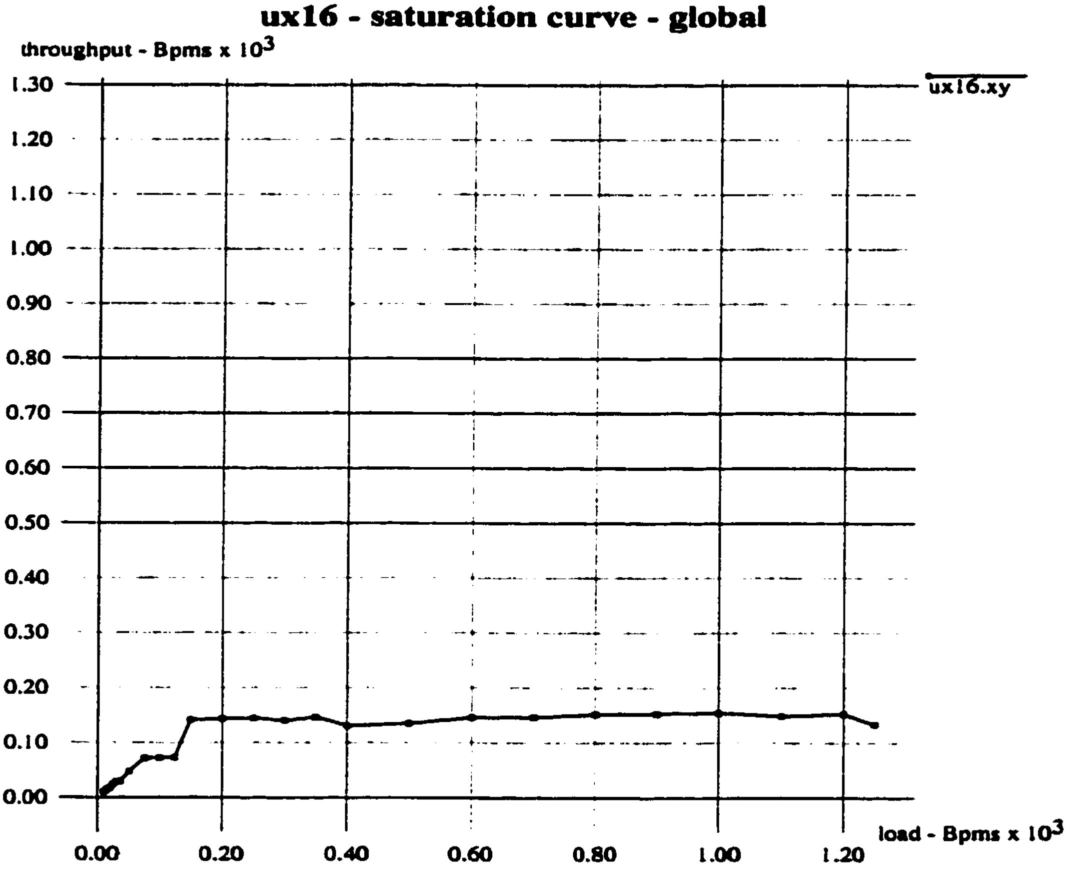


Figure 46: ux16 - saturation curve - global

A.17 ux17

Table ux17.log - log summary (at sender)

Date: 98.09.17 | **Start:** 23:38:07 | **End:** 23:48:12 | **Duration:** 10 min | **Mode:** Unicast

Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 0 ms
MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No

Commands:

```
mmetric -t 132.205.45.24 -S -g -f -p 1472 -b 28800 -C 1440 -a 1048576 -o 250 -W 102400 -c 10..
mmetric -r -S -g -p 1472 -b 28800 -J 1440 -o 250 -w 102400 -j 10..
```

Sender: orchid | **Receiver(s):** forest 132.205.45.24

Table ux17.xy - xy data summary (at sender)

9.598	14.402	18.003	23.995	28.771	28.855	47.917	71.512	67.045
10	15	20	25	30	35	50	75	100
72.365	143.641	143.818	145.011	145.980	136.979	118.456	147.438	146.880
125	150	200	250	300	350	400	500	600
149.157	190.304	179.428	182.266	157.799	198.707	200.034		
700	800	900	1000	1100	1200	1250		

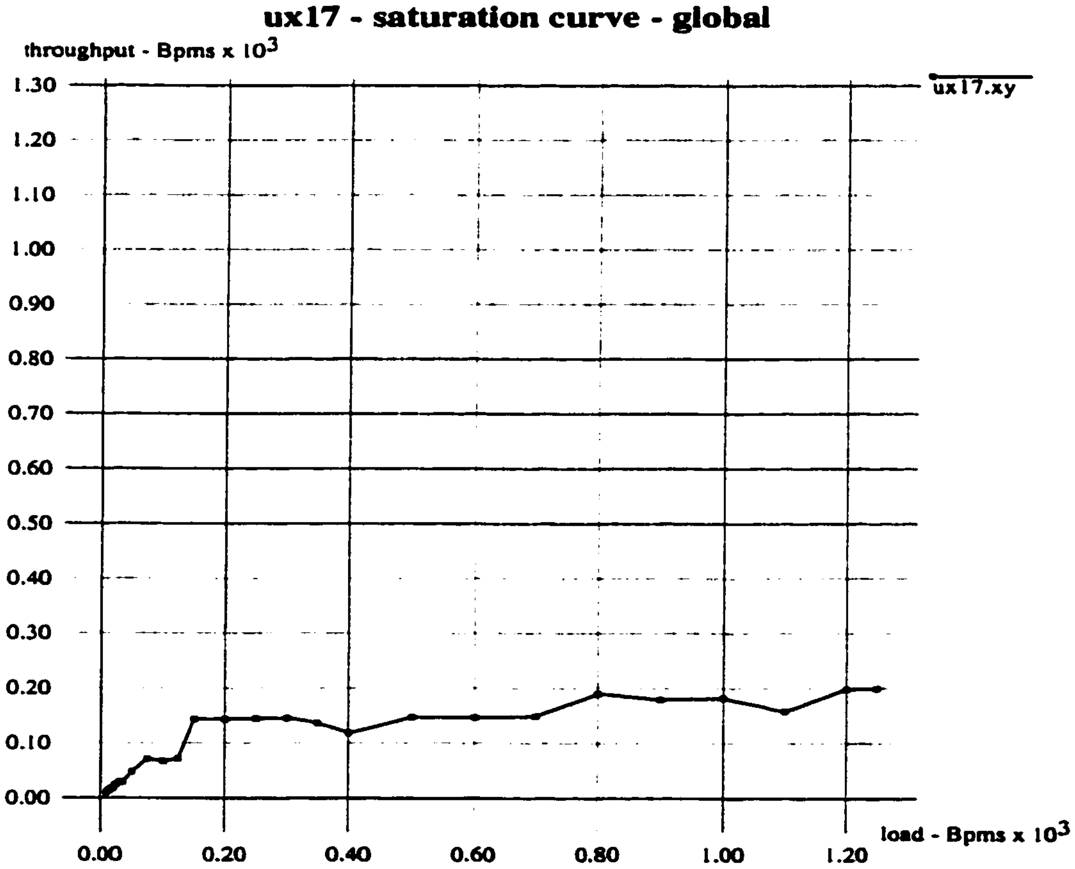


Figure 47: ux17 - saturation curve - global

A.18 ux18

Table ux18.log - log summary (at sender)

Date: 98.09.18 | **Start:** 15:42:44 | **End:** 15:53:45 | **Duration:** 11 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 5 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.28 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** dahlia 132.205.45.28

Table ux18.xy - xy data summary (at sender)

9.595	14.382	18.064	24.008	28.756	28.643	28.786	28.754	28.832
10	15	20	25	30	35	50	75	100
28.856	28.840	32.852	1434.440	1535.250	1648.704	1677.722	1620.674	1716.164
125	150	200	250	300	350	400	500	600
1528.536	1504.413	1082.122	1385.173	1574.438	1504.413	1659.139		
700	800	900	1000	1100	1200	1250		

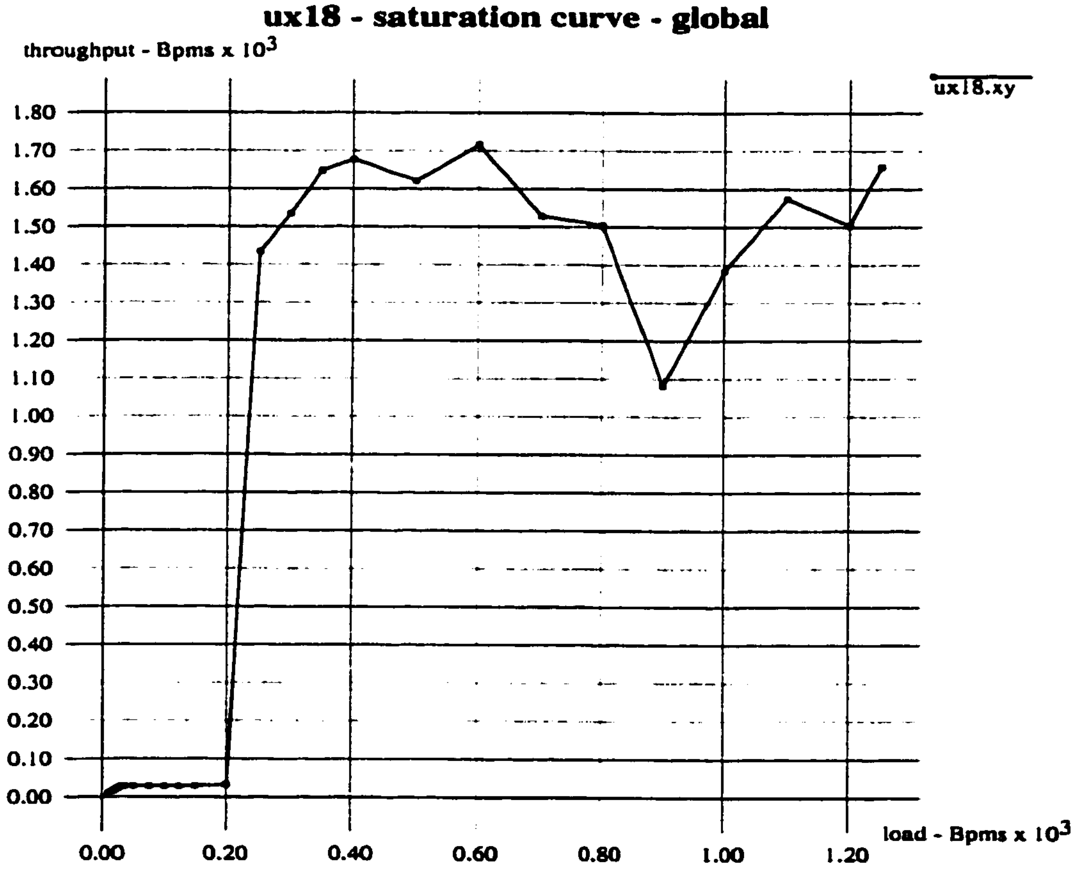


Figure 48: ux18 - saturation curve - global

A.19 ux19

Table ux19.log - log summary (at sender)

Date: 98.09.17 | **Start:** 22:51:20 | **End:** 22:59:56 | **Duration:** 08 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 10 ms MAXANTICIPATION = 5 ms, Harmonization Burst/Rate = No
Commands: mmetric -t 132.205.45.61 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: dahlia | **Receiver(s):** orchid 132.205.45.61

Table ux19.xy - xy data summary (at sender)

9.603	14.406	18.024	24.012	28.808	33.288	47.985	71.855	71.963
10	15	20	25	30	35	50	75	100
117.501	143.209	144.372	989.223	1245.340	1101.445	1045.440	1278.751	1203.876
125	150	200	250	300	350	400	500	600
1240.918	1360.021	1242.389	1126.290	1280.313	1093.406	1041.287		
700	800	900	1000	1100	1200	1250		

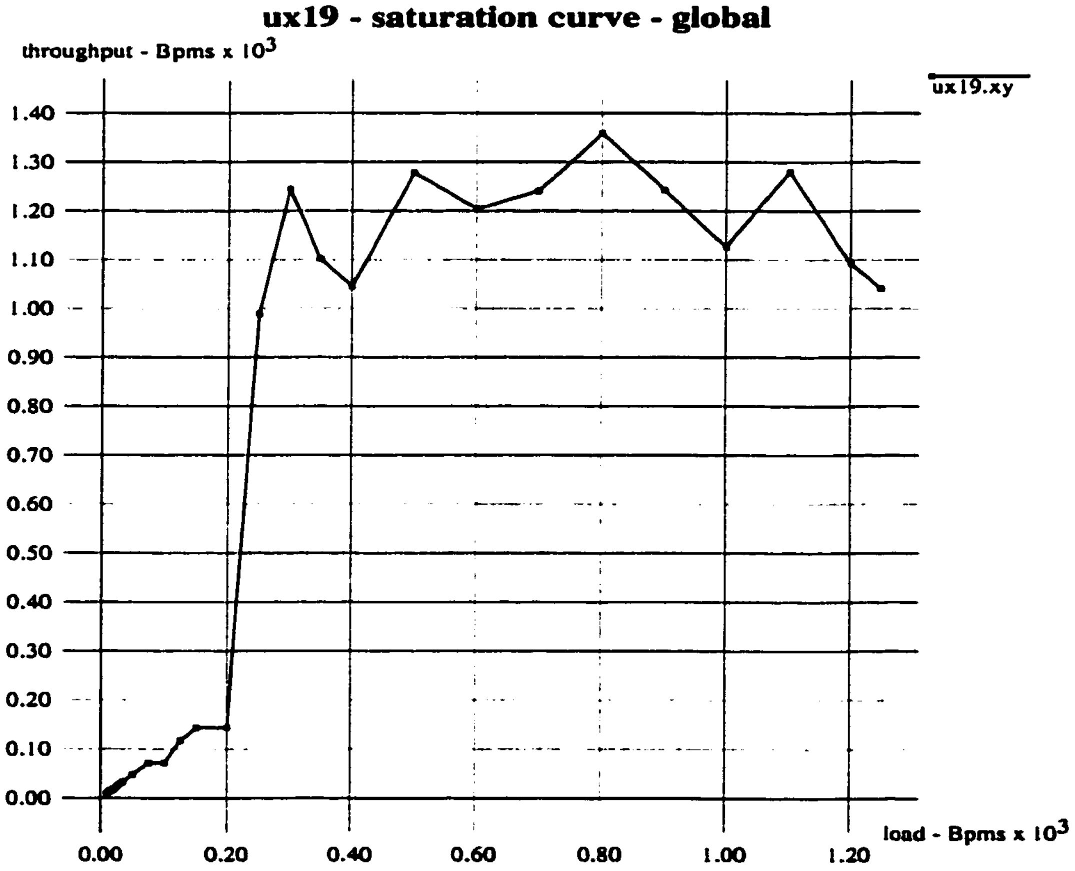


Figure 49: ux19 - saturation curve - global

A.20 ux20

Table ux20.log - log summary (at sender)

Date: 98.09.18 | **Start:** 00:09:01 | **End:** 00:17:26 | **Duration:** 08 min | **Mode:** Unicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 0 ms MAXANTICIPATION = 5 ms, Harmonization Burst/Rate = Yes (burst=rate*1000*0.1)
Commands: mmetric -t 132.205.45.28 -S -g -f -p 1472 -b 1440 -C 1000.. -a 1048576 -o 250 -W 102400 -c 10.. mmetric -r -S -g -p 1472 -b 1440 -J 1000.. -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** dahlia 132.205.45.28

Table ux20.xy - xy data summary (at sender)

10.005	14.984	20.012	24.880	29.956	34.955	50.412	74.743	100.737
10	15	20	25	30	35	50	75	100
125.759	151.748	201.688	255.439	307.500	359.101	401.907	518.071	610.347
125	150	200	250	300	350	400	500	600
734.297	802.277	934.560	1021.009	1138.519	1242.389	1283.447		
700	800	900	1000	1100	1200	1250		

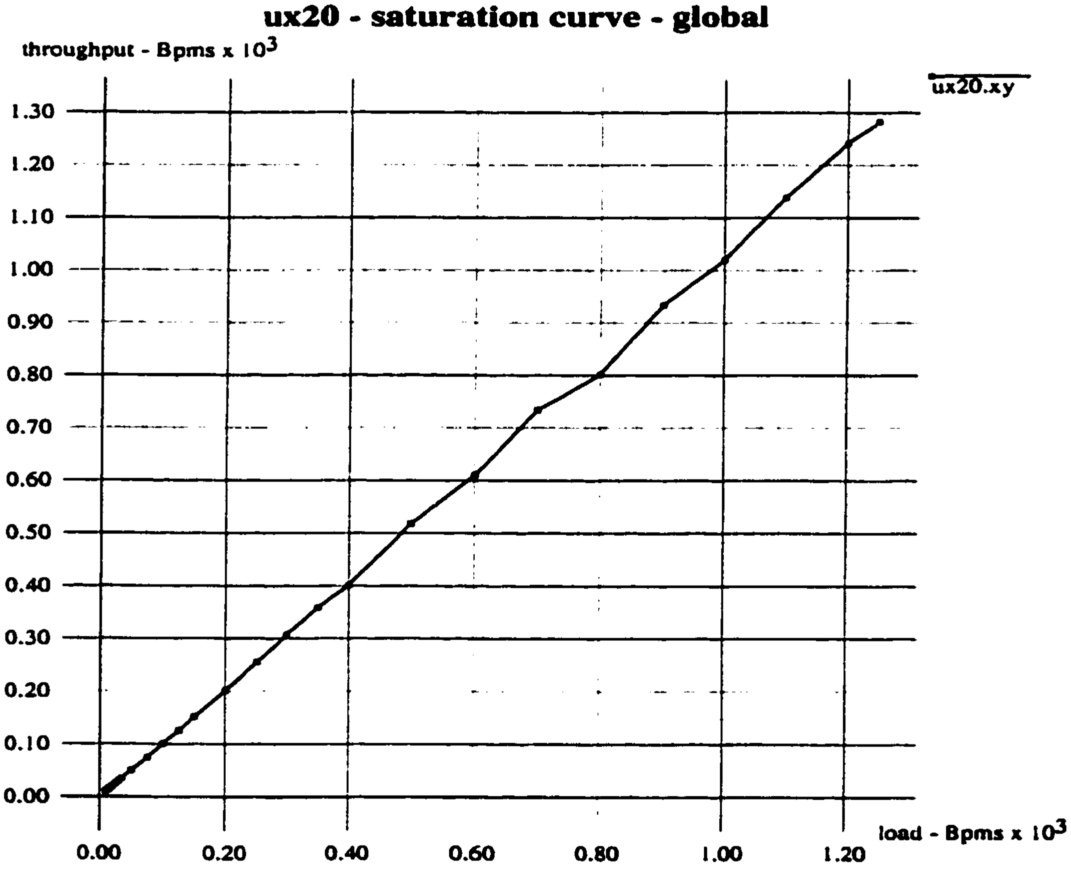


Figure 50: ux20 - saturation curve - global

A.21 mx01

Table mx01.log - log summary (at sender)

Date: 98.09.11 | **Start:** 10:46:02 | **End:** 11:01:52 | **Duration:** 15 min | **Mode:** Multicast
Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -O 127 -W 102400 -c 10.. mmetric -R 239.159.100.40 -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** 239.159.100.40 (daffodil)

Table mx01.xy - xy data summary (at sender)

9.496	14.315	13.872	23.831	28.601	28.495	28.609	28.625	28.674
10	15	20	25	30	35	50	75	100
28.744	28.845	28.705	29.357	29.765	29.569	29.220	31.292	37.766
125	150	200	250	300	350	400	500	600
32.597	147.251	126.167	167.853	164.354	210.515	198.069		
700	800	900	1000	1100	1200	1250		

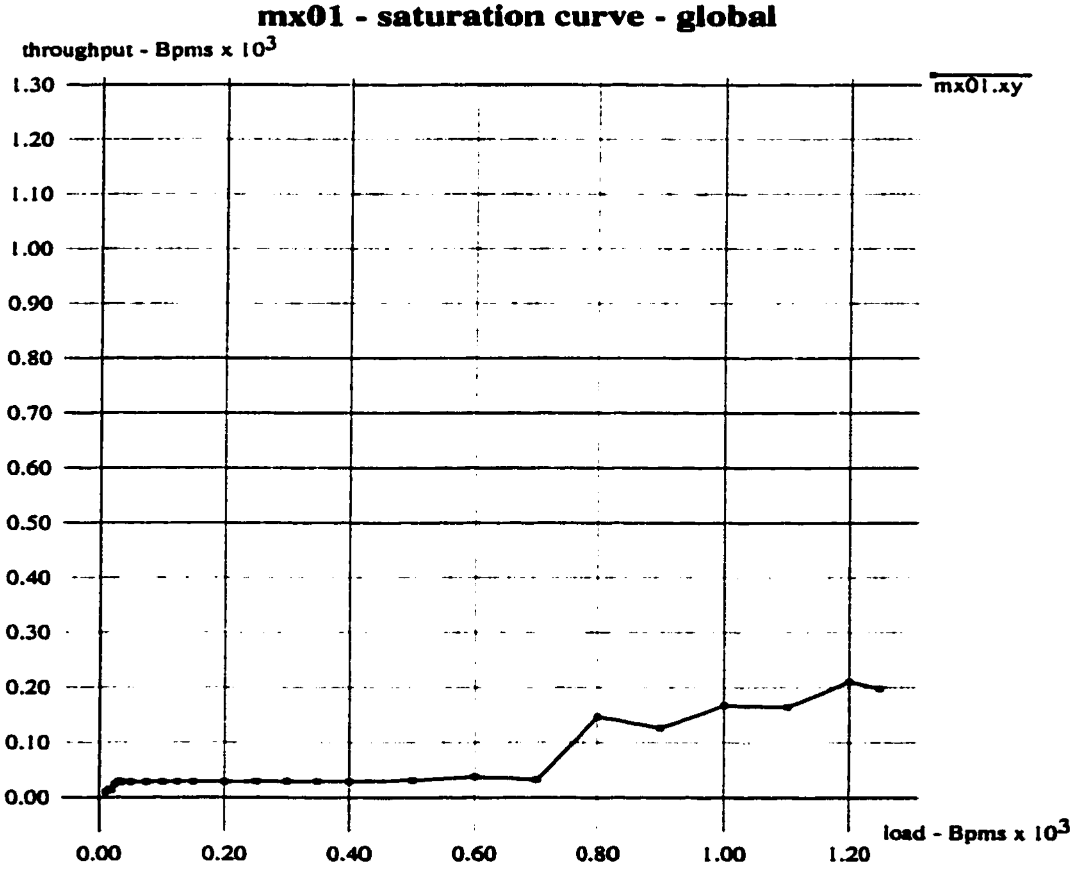


Figure 51: mx01 - saturation curve - global

A.22 mx02

Table mx02.log - log summary (at sender)

Date: 98.09.11 | **Start:** 13:07:12 | **End:** 13:22:51 | **Duration:** 15 min | **Mode:** Multicast
Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -O 127 -W 102400 -c 10.. mmetric -R 239.159.100.40 -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** 239.159.100.40 (dahlia)

Table mx02.xy - xy data summary (at sender)

9.526	14.286	14.505	23.725	28.511	28.574	28.558	28.600	28.559
10	15	20	25	30	35	50	75	100
28.680	28.683	28.864	29.138	29.597	29.597	29.891	37.483	46.670
125	150	200	250	300	350	400	500	600
33.460	144.831	156.925	128.031	128.691	163.917	171.785		
700	800	900	1000	1100	1200	1250		

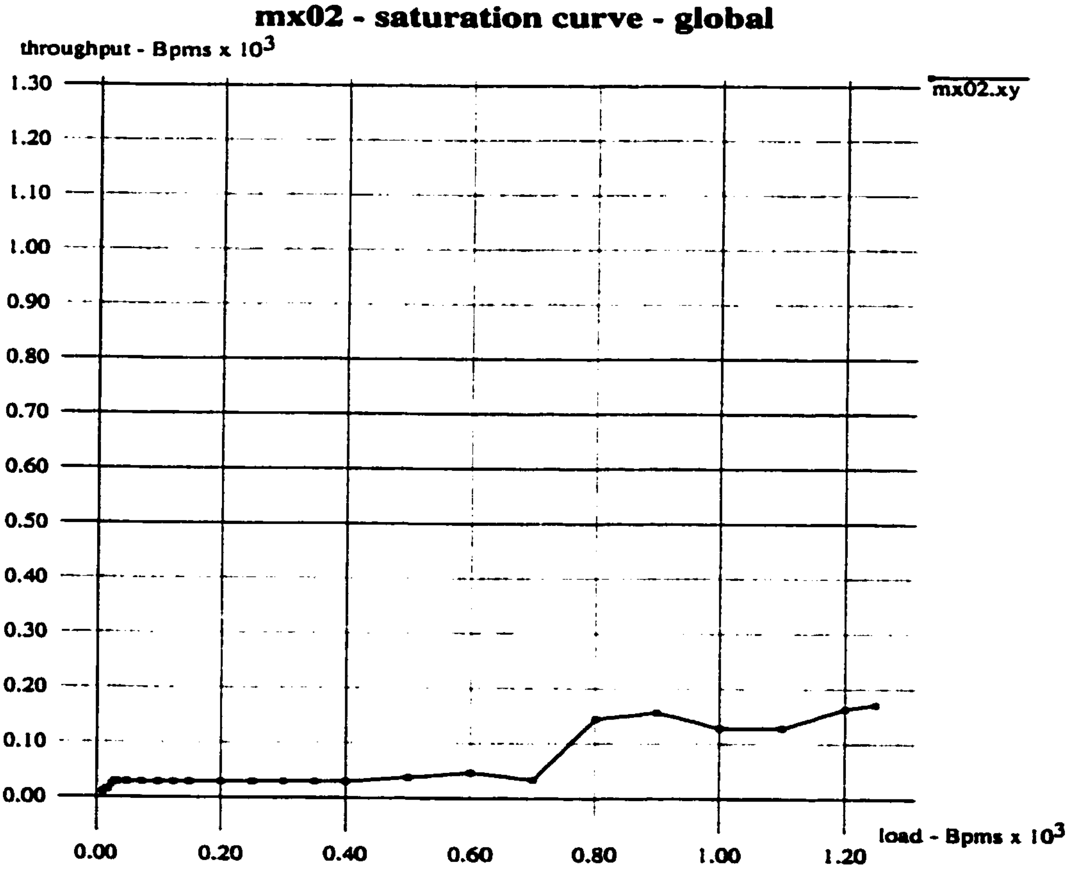


Figure 52: mx02 - saturation curve - global

A.23 mx03

Table mx03.log - log summary (at sender)

Date: 98.09.11 | **Start:** 21:59:23 | **End:** 22:19:00 | **Duration:** 20 min | **Mode:** Multicast
Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50 ms MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No
Commands: mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -O 127 -W 102400 -c 10.. mmetric -R 239.159.100.40 -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** 239.159.100.40 (dahlia/sunset)

Table mx03.xy - xy data summary (at sender)

9.536	14.341	16.971	23.897	28.643	27.589	28.234	28.627	28.565
10	15	20	25	30	35	50	75	100
27.867	28.643	28.628	28.785	29.186	29.170	29.398	29.910	29.831
125	150	200	250	300	350	400	500	600
30.210	90.488	104.910	82.383	94.885	79.036	65.832		
700	800	900	1000	1100	1200	1250		

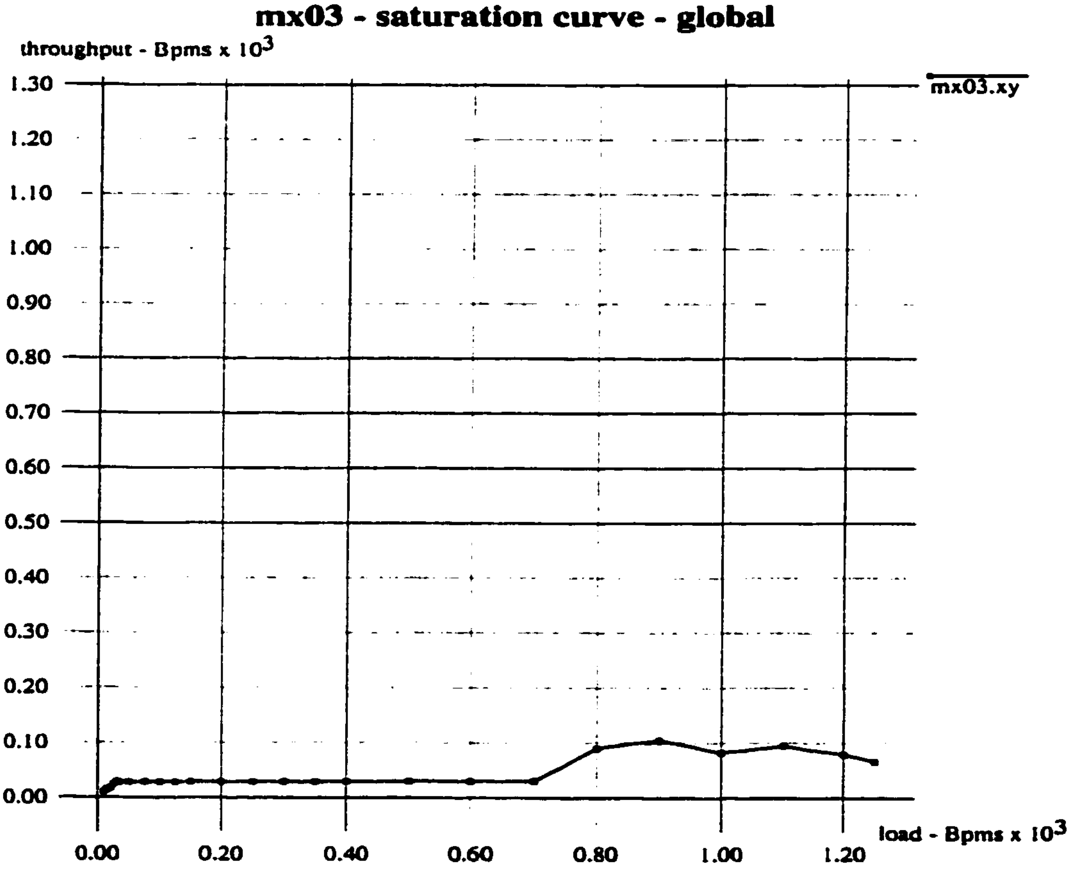


Figure 53: mx03 - saturation curve - global

A.24 mx04

Table mx04.log - log summary (at sender)

Date: 98.09.11 | **Start:** 15:15 | **End:** 15:57 | **Duration:** 42 min. | **Mode:** Multicast

Synopsis: Using SandiaXTP-1.5.1 Unmodified, SELECT_FLOOR = 50ms, MAXANTICIPATION = 0 ms, Harmonization Burst/Rate = No

Commands:

```
mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250
                                                      -O 127 -W 102400 -c 10..
mmetric -R 239.159.100.40 -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
```

Sender: orchid | **Receiver(s):** 239.159.100.40 (dahlia/sunset/daffodil)

Table mx04.xy - xy data summary (at sender)

9.504	14.261	14.959	23.728	28.404	28.504	28.599	28.585	28.579
10	15	20	25	30	35	50	75	100
28.382	28.682	28.880	29.588	29.706	29.901	29.673	31.419	32.256
125	150	200	250	300	350	400	500	600
32.245	79.971	116.366	107.205	102.250	96.947	80.191		
700	800	900	1000	1100	1200	1250		

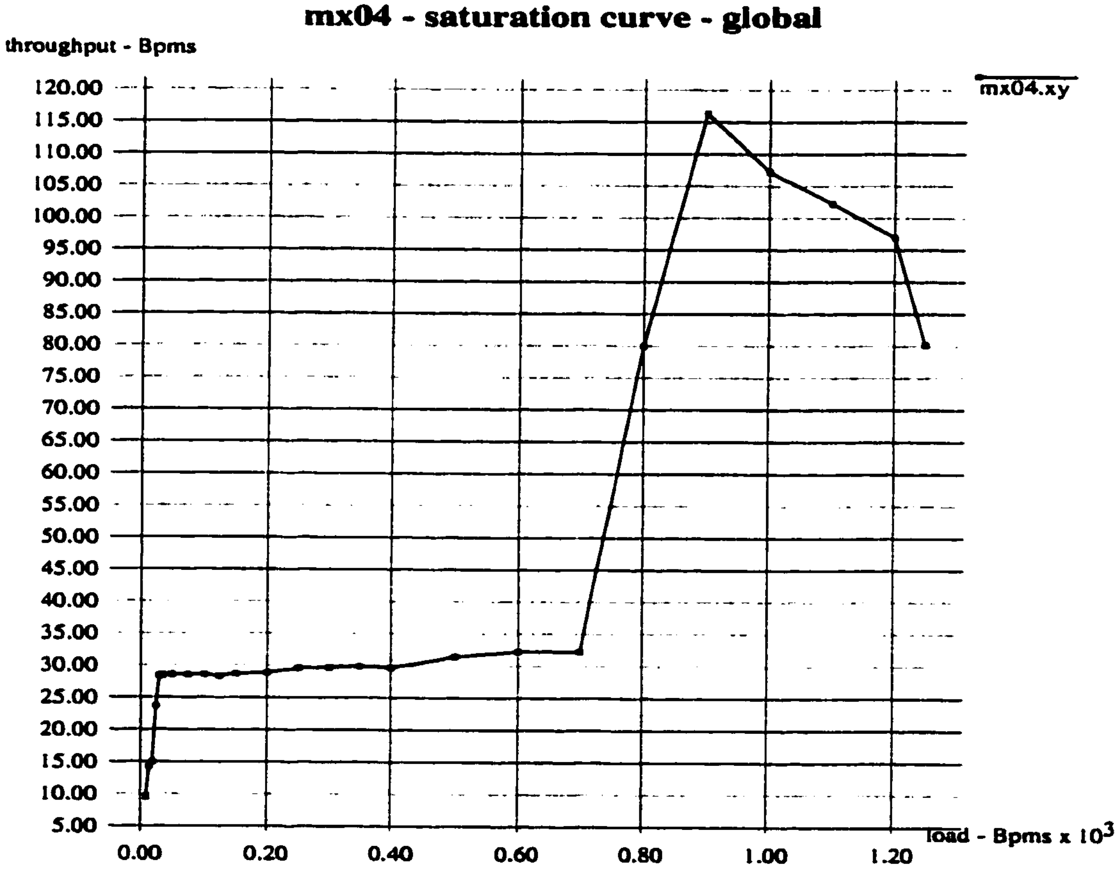


Figure 54: mx04 - saturation curve - global

A.25 mx05

Table mx05.log - log summary (at sender)

Date: 98.09.11 | **Start:** 16:33 | **End:** 17:09 | **Duration:** 36 min. | **Mode:** Multicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 50 ms, MAXANTICIPATION = 10 ms, Harmonization Burst/Rate = No
mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1440 -a 1048576 -o 250 -O 127 -W 102400 -c 10.. mmetric -R 239.159.100.40 -S -g -p 1472 -b 1440 -J 1440 -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** 239.159.100.40 (dahlia/sunset/daffodil)

Table mx05.xy - xy data summary (at sender)

9.593	14.261	18.735	23.918	28.580	28.434	28.534	28.549	30.150
10	15	20	25	30	35	50	75	100
89.676	129.470	7.354	224.342	27.994	242.670	261.751	176.677	144.771
125	150	200	250	300	350	400	500	600
141.834	287.675	144.452	241.942	231.066	257.762	266.069		
700	800	900	1000	1100	1200	1250		

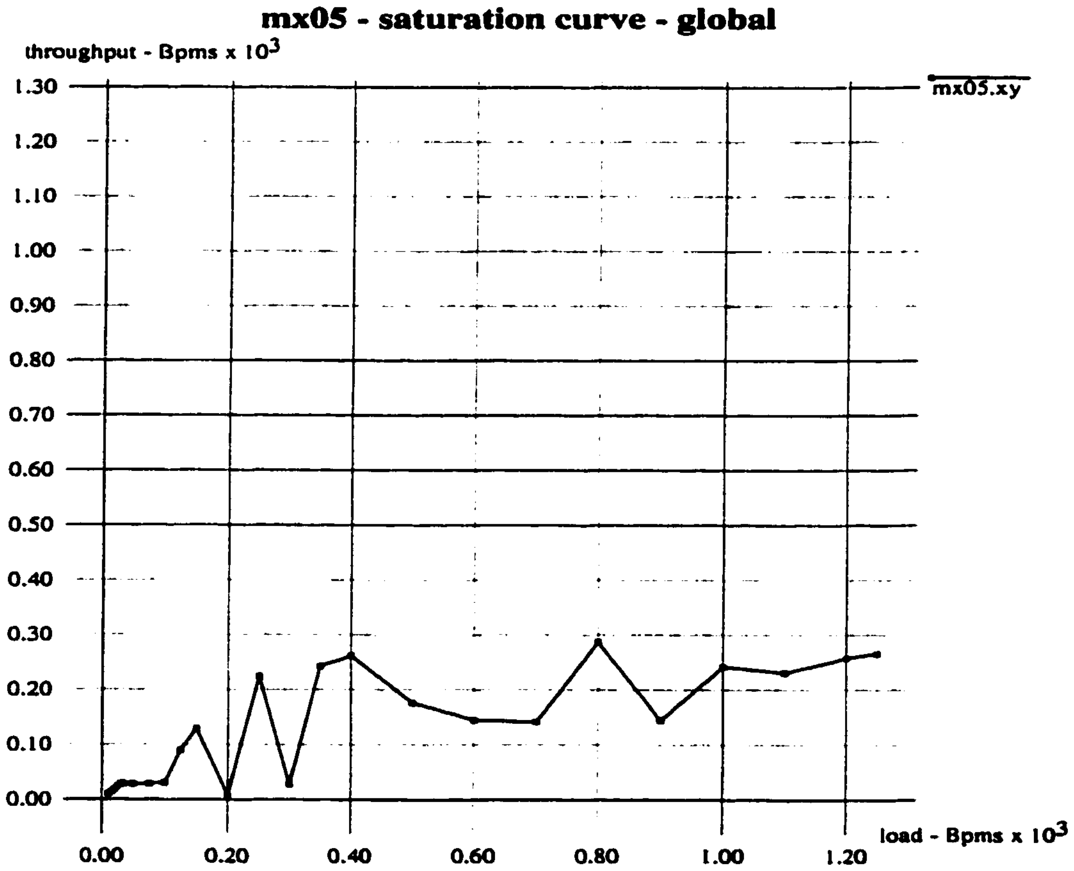


Figure 55: mx05 - saturation curve - global

A.26 mx06

Table mx06.log - log summary (at sender)

Date: 98.09.24 | **Start:** 17:44:53 | **End:** 18:01:32 | **Duration:** 17 min | **Mode:** Multicast
Synopsis: Using SandiaXTP-1.5.1 Modified, SELECT_FLOOR = 0 ms MAXANTICIPATION = 5 ms, Harmonization Burst/Rate = Yes (burst=rate*1000*0.1)
Commands: mmetric -T 239.159.100.40 -S -g -f -p 1472 -b 1440 -C 1000.. -a 1048576 -o 250 -O 127 -W 102400 -c 10.. mmetric -R 239.159.100.40 -S -g -p 1472 -b 1440 -J 1000.. -o 250 -w 102400 -j 10..
Sender: orchid | **Receiver(s):** 239.159.100.40 (dahlia/sunset/daffodil)

Table mx06.xy - xy data summary (at sender)

9.921	14.963	19.939	24.900	29.703	33.917	49.529	74.531	98.310
10	15	20	25	30	35	50	75	100
113.741	143.739	191.626	193.179	202.859	181.635	208.630	232.862	186.746
125	150	200	250	300	350	400	500	600
203.884	145.717	195.886	237.988	185.130	223.291	173.261		
700	800	900	1000	1100	1200	1250		

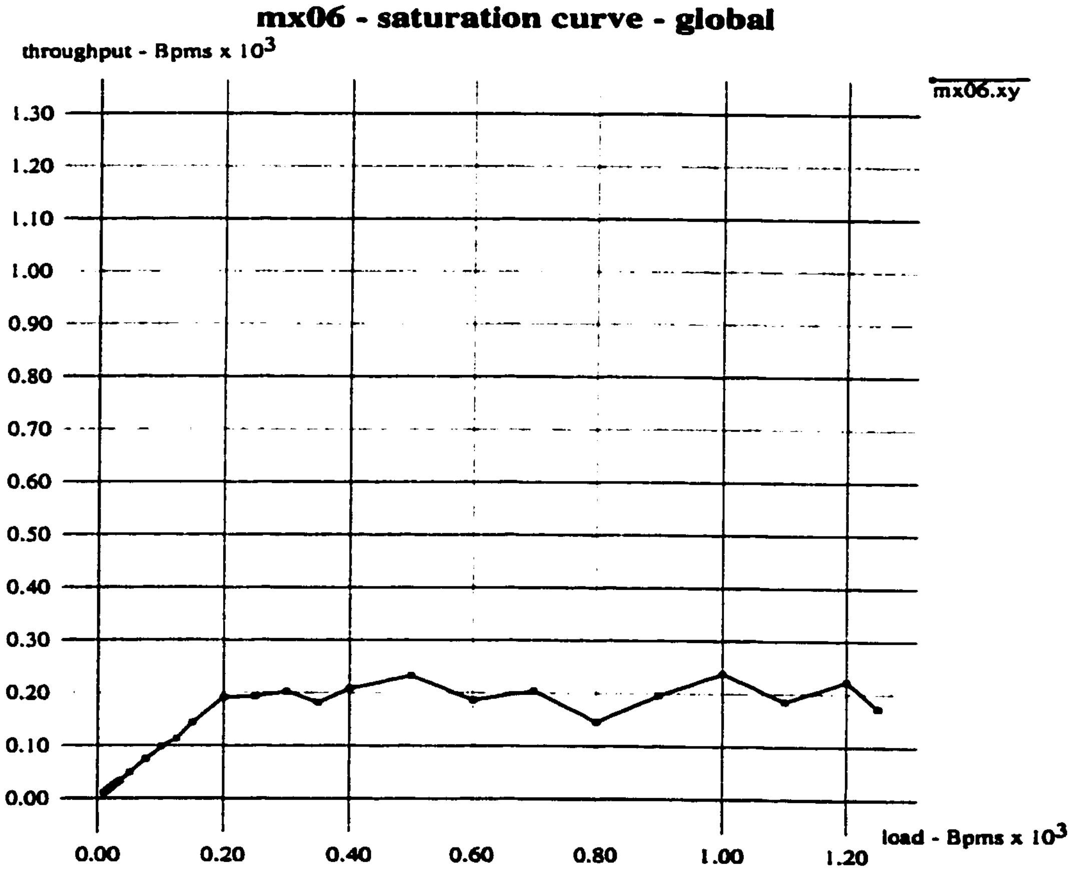


Figure 56: mx06 - saturation curve - global

B Review of recent studies at the HSP Lab

B.1 Work by Marie A. Wallace

Marie A. Wallace submitted her Major Report entitled **Error Control in XTP over the Internet** in December 1996. Marie conducted unicast experiments and used the Sandia XTP implementation of XTP written by Dr. Tim Strayer. The problem space that Marie investigated can be summarized as follows:

- What is the sequence of events when an error occurs (dammaged or lost packet)? How far out of synchronization do the sending and the receiving sides get?
- To successfully complete a synchronization handshake, the sending side must receive an echo from the receiving side within a fixed period of time (say T). Is this requirement too strong? Could we accept a late reply from the receiver?

Given this problem space, we could summarize the results of Marie's experiments as follows:

- **Synchronization handshake:** As shown in Figure 57 (a), Marie succeeded to expose a rather vicious scenario that was contributing to erode the performance of XTP. Let's outline this bad scenario as follows (please refer to Figure 57 (a) for the nomenclature):

  1. sending side emits an SREQ packet with sync=0, and starts a timer for T duration at time t0;
  2. T times out and sending side issues another SREQ packet now with sync=1, and starts again a timer for T duration at time t1;
  3. echo from receiving side reaches the sending side at time tr (t1<tr<t2); as per the definition of the protocol, this echo was not accepted by the sending side;
  4. again, T would time out at the sending side at time t2, sender would issue another SREQ packet with sync=2, and start another timer of larger duration 2T at time t2; now, it is likely that the echo will reach the sending side within the 2T duration, but still two attemps are lost.

  Marie demonstrated that this bad scenario was mainly the consequence of an underestimated round trip time (RTT - default at 50ms). This RTT had been derived from experiments done in a closed LAN environment. The 50 ms value was simply too low when XTP was being used in a WAN environment, such as the Internet where delays are much higher. Marie also came to the conclusion that late echos from the sender could also be accepted, up to a ...limit. With reference to Figure 57 (a), the first echo received at time tr could have been accepted, thereby completing the synchronization handshake at this point.

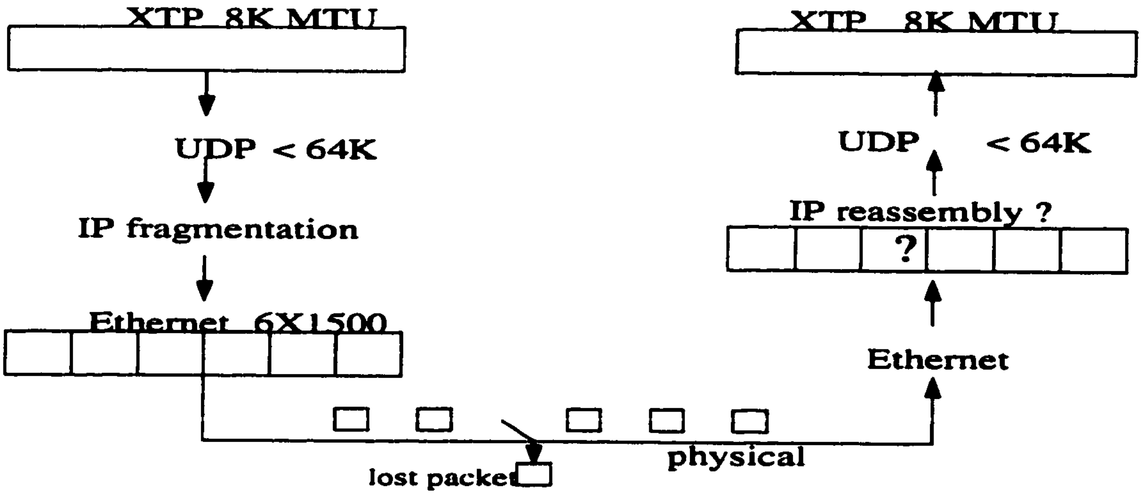


Figure 57: Marie's findings

- **MTU sizes:** The maximum transmission unit (MTU) is another area where Marie's experiments have benefited. Initially, the default XTP MTU size was fixed at 8192 (8K) bytes at the Transport level. When it reached the IP Network layer, with Ethernet being a very common technology with MTU sizes of 1500 bytes, it meant that one XTP packet could be split in up to 6x1.5K Ethernet packets. As can can be observed in Figure 57 (b), the Ethernet packets would be reassembled again at the receiving end by IP at the network level. Given the lossy behavior of the Internet, the chances that one of those Ethernet packets would be missing were non negligible. If this occurred, then some IP reassembly timer would fire, which would result in retransmitting the whole XTP packet (i.e. all 6 Ethernet packets even if five of them had reached destination). The consequence of Marie's findings was to adapt XTP MTU size to the underlying MTU sizes so that segmentation would be reduced as much as possible. Let us mention that a MTU size of 8192 bytes would create no difficulty on an ATM network, where loss rate is extremely low.

B.2 Work by Torsten Auerbach

Torsten Auerbach submitted his report [AUE] entitled **Using Error Status Information in the Xpress Transport Protocol** in March 1997. Torsten conducted unicast experiments between Concordia University in Montréal and his home University in Karlsrhuhe, Germany, using the SandiaXTP [SUG] implementation of XTP 4.0 written by Dr Tim Strayer. The research work was done in part concurrently, but can be considered logically posterior to Marie's work. In fact, although indirectly, Torsten addresses some of the problems that Marie exposed. The problem space that Torsten explored can be summarized as follows:

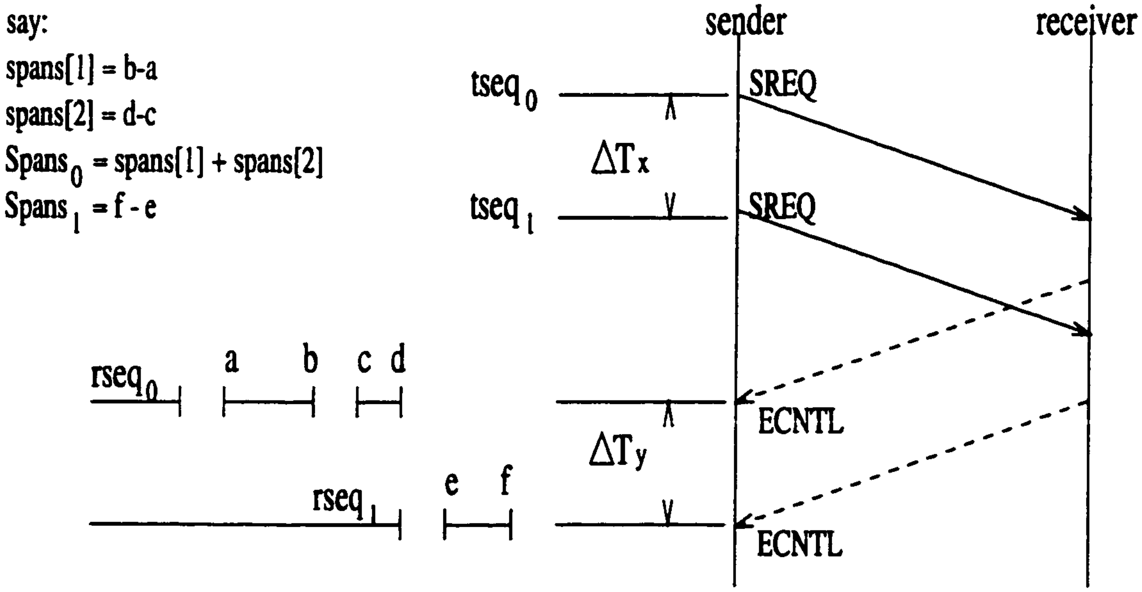


Figure 58: Torsten's rates computations

- How can wc dynamically adjust the data transfer rate to best suit the capacities of the underlying medium when XTP is used to transfer large files in the Internet environment (i.e. in a WAN environment)?
- Given the prolonged synchronizing handshake problem exposed by Marie, and the role played by a too low RTT, what mechnisms could we use to dynamically adjust the RTT for a particular communication association?
- If the dcfault MTU size is too large (at 8K bytes) when using XTP in an Internet environment, how could we dynamically optimize the MTU size for a particular communication association?

Given this problem space, the results of Torsten's work can be summarized as follows:

Rate Control Algorithm: This part of Torsten work addresses the data rate transfer adjustment problem. As a result of his experiments, Torsten devised many versions of his algorithm to adjust the send rate within an ongoing communication association. Here, we are only interested with the one that he declares the **The fifth and last version of the algorithm** (see [AUE]), which we shall thereafter refer to as **the fifth algorithm.** Before discussing the fifth algorithm itself, let's lay down some preliminaries (please, refer to Figure 58 for the following discussion).
Terminology:

bitrate (bits/sec) -optimized send rate for next round, taking into consideration the predicted send and receive rates, and their variance.

error (0..1) -computed error rate

E (0..1) -predicted error rate

rate (bytes/msec) -same as bitrate, but in bytes/msec.

rate_r (bytes/msec) -computed current receive rate

RATE_R (bytes/msec) -predicted receive rate

rate_s (bytes/msec) -computed current send rate

RATE_S (bytes/msec) -predicted send rate

RV_E -variance of the error rate

RV_R -variance of the receive rate

RV_S -variance of the send rate

Computing current send rate (rate_s):

- rate is computed by taking into consideration the amount of data sent, including retransmitted packets, within a short interval of time (ΔT_x in ms–milliseconds);
- the time interval (ΔT_x - difference between 2 timestamps) extends between the sending of two consecutive SREQ packets at the sending end;
- more precisely, the amount of data sent during the time interval is computed as:

$$data_{sent} = tseq_1 - tseq_0 \quad \text{(bytes)} \tag{1}$$

- then, the current send rate is computed as:

$$rate\_s = \frac{tseq_1 - tseq_0}{\Delta T_x} \quad \text{(bytes/ms)} \tag{2}$$

Computing current receive rate (rate_r):

- uses error status information in received ECNTL packets;
- rate is computed by taking into consideration the amount of data received within a short time interval (ΔT_y);
- the time interval (ΔT_y - difference between 2 timestamps) extends between the arrival of two expected ECNTL packets at the sending end;
- more precisely, the amount of data received is computed as:

$$data_{received} = (rseq_1 - rseq_0) + (Spans_1 - Spans_0) \tag{3}$$

- then, the current receive rate is computed as:

$$rate\_r = \frac{(rseq_1 - rseq_0) + (Spans_1 - Spans_0)}{\Delta T_y} \quad \text{(bytes/ms)} \tag{4}$$

Computing current error rate (error):

- the current error rate is deduced from the current amount of sent and received data, as follows:

$$error = \frac{DATA_{sent} - DATA_{receive}}{DATA_{sent}} \quad (0..1) \qquad (5)$$

The fifth algorithm: (refer to Figure 58)

Here follows a high level view of Torsten's **fifth algorithm:**

```
o every now and then, do the following:

  o sending side issues a SREQ packet, and takes a timestamp;

  o sometime later, sending side issues another SREQ packet,
    and takes a timestamp;

  o upon reception of the 1st ECNTL packet (matching 1st SREQ),
    sending side take a timestamp and do as the FIRST OPTIMIZATON
    PHASE of the algorithm to update send rate as follows:
```

```
if ((error >= 0) && (error <= 1)) {
  if (rate > (10 * minrate))
    outrate = (word32)((1-error) * (double)rate);
  else
    outrate = (word32)((1.4 - error) * (double)rate);
  E = error;
  if (outrate < minrate)
    rate = tspec.ts1.outrate = minrate;
  else
    rate = tspec.ts1.outrate = Word32)outrate;
} else { // Don't change rate
    RATE_R = rate * 8000;                        //Predicted receive rate
}
```

```
  o upon reception of the 2nd ECNTL packet (matching 2nd SREQ),
    sending side takes a timestamp and proceeds with the SECOND
    OPTIMIZATION PHASE of the algorithm to update send rate as
    follows:
```

```
RATE_R = RATE_R + (rate_r - RATE_R) / 2;        //Predicted send rate
E = E + (error - E) / 2;                        //Predicted error rate
```

```
RV_R = RV_R + (abs(rate_r - RATE_R) - RV_R) / 4; //Variance: receive rate

if ((diff = error - E) < 0)  // abs(error - E)
  diff = diff * (-1);

RV_E = RV_E + (diff - RV_E) / 2;                    //Variance: error rate
e_rate = E + RV_E / 2;

if ((quot = 2 * e_rate) < 0.5)
  quot = 0.5;

if (quot > 2)
  quot = 2;  // ???

bitrate = (word32)((double)((RATE_R + 2 * RV_R)) / quot);

if (bitrate <= (minbitrate = minrate * 8000)) {
  rate = tspec.ts1.outrate = minrate;
  RATE_R = minbitrate;
} else {
    rate = tspec.ts1.outrate = bitrate / 8000;
}
```

Comments on the fifth algorithm:

To make the data send rate adjustment, the new optimized send rate is derived by taking into consideration, inter alia, predicted send, receive and error rates for the next round. In turn, the predicted rates are established by taking into consideration the recent history (calculated rates) and the past history (the previous predicted rate). For instance, in $RATE\_R = RATE\_R + (rate\_r - RATE\_R)/2$;, the first RHS component of the equation RATE_R is the past history component, and (rate_r-RATE_R)/2 is the recent history component. Half the difference between the previous predicted rate and and the calculated rate is added to or substracted from the previous predicted rate (if RATE_R was 100, and rate_r = 80, then new predicted rate RATE_R = 90). For a more complete discussion of this type of method, as applied to CPU scheduling, see Appendix E, under the entry **Shortest-Job-First Scheduling.**

B.3 Work by Deric Sullivan

[Note October 1998: The core of this text was written in the Fall of 1997, i.e., before the work of analysis of the structure of SandiaXTP and of its rate control mechanisms presented respectively in Chapters 5 and 6. Consequently, this text has been greatly reduced to focus mainly on the impacts on throughput of varying the window size (i.e., the size of the shared memory send and receive areas).

Regarding the rate control aspect, Deric conducted his experiments using the Internet environment with unmodified SandiaXTP and the SELECT_FLOOR effect was at work. Accordingly, the saturation curves should display a taper off plateau at throughput=28.8 Bpms (1440/50=28.8). As shown on Figure 61, for the upper three saturation curves (WS=51200, 102400, 512000), there is indeed a taper off plateau at throughput=28 Bpms approximately. Given that few low range rate values were used, this plateau is not well mappped on the curves.

The end parts of these three curves are however different from the ones presented in Section A for a LAN environment; now there is a sharp throughput downfall which did not occur on Figure 31 for ux01 for instance. This behavior could confirm the anticipated saturation phenomenon when using the Internet environment.

The study of Deric's report has also been a source of information for selecting the user level command line arguments, the particular window size (102400), and the range of data rate values used for conducting the rate control experiments presented in this report.]

Deric Sullivan submitted his report [SUL] entitled **XTP PROJECT FOR COMP490** in May 1977. Deric conducted unicast file transfer experiments between Concordia University in Montreal and Karlsrhuhe University in Germany, using the SandiaXTP [SUG] implementation of XTP 4.0 written by Dr. Tim Strayer. The scenario of the experiments done by Deric can be summarized as follows:

- for rate = 10, 25, 50, 100, 500, 1000, 5000, 10000 bytes per millisecond (Bpms)
  - for window size = 1440, 5120, 10240, 51200, 102400, 512000 bytes
    - care for Round Trip Time (RTT) (0.5s)
    - care for XTP PDU size (1472 bytes)
    - do reliable 1MB file transfer
      - measure average data transfer time in ms (from the transmitter viewpoint)

Each file transfer can be considered as an independent session. Parameters, such as rate, window size, RTT and XTP PDU size are set at the beginning for the whole file transfer session.

RTT and XTP PDU size values

As a result of Marie's work, Deric was aware of the XTP packet fragmentation (default XTP PDU size of 8192 bytes get fragmented into 6x1526 bytes Ethernet packets),

Table 16: Sample of original Deric's data for rate=10

%packet loss 11 pings (%)	Rate (unknown units)	Initial Round Trip time (ms)	Send/Receive Window size (bytes)	Δ t run 1 (ms)	Δ t run 2 (ms)	Δ t run 3 (ms)	Average time (ms)
	10		1440				#DIV/0!
0 and 0	10	316 and 329	5120	173155	174935	178039	175376
0 and 0	10	349 and 352	10240	163662	159593	159759	161005
0 and 0(1)	10	315 and 332	51200	141045	142112	143137	142098
0 and 0	10	340 and 361	102400	139627	152041	141367	144345
0 and 0	10	324 and 324	512000	141054	140561	141101	140905
0 and 0	10	~tested	1048576	~tested	~tested	~tested	~tested

(1) 100 pings packets (ping -S -n 129.13.3.120 1472 100)

Source: XTP PROJECT FOR COMP490 by Deric Sullivan, May 1997 p14

and the prolonged synchronizing handshake (low default RTT) problems. Deric used a modified version of the network utility program called **ping** to find the path MTU, and then derive the XTP PDU size so as to avoid XTP packet fragmentation at the IP network layer. Program **ping** was also used to get an initial experimental value for the round trip time (RTT), which could be dynamically adjusted later by the XTP protocol itself. Command line arguments are then used to convey the values found with ping to the XTP daemon (such as -p 1472 for the PDU size, and -o 336 for the initial RTT).

Data gathered and analysis

Deric's data are presented in a series of eight tables, one for each rate value used for the tests. For instance, for rate=10, Table 16 shows a listing of the average data transfer times for all six window sizes. For the analysis part, Deric summarizes the data as follows ([SUL, p28]):

- *It is seen that a rate of 10 is not strongly affected by the different window sizes.*;
- *The higher rates (5000 and 10000) start to exibit strange patterns. They do not follow the same smooth curves that the middle rates show.*
- *The rates 25, 50, 100, and 500 exibit the behavior that would be expected with a theoretical rate control and flow control protocol.*
- *All round trip times lay between 300 and 399 milliseconds.*
- *All of the packet loss was between 0 and 10 percent.*

The reasons that led Deric devise his experiments along a window-size axis are not expressed, as well as why he chose the particular discrete values and ranges for window-sizes and data rates, varying respectively from 1440 to 512000 bytes, and from 10 to

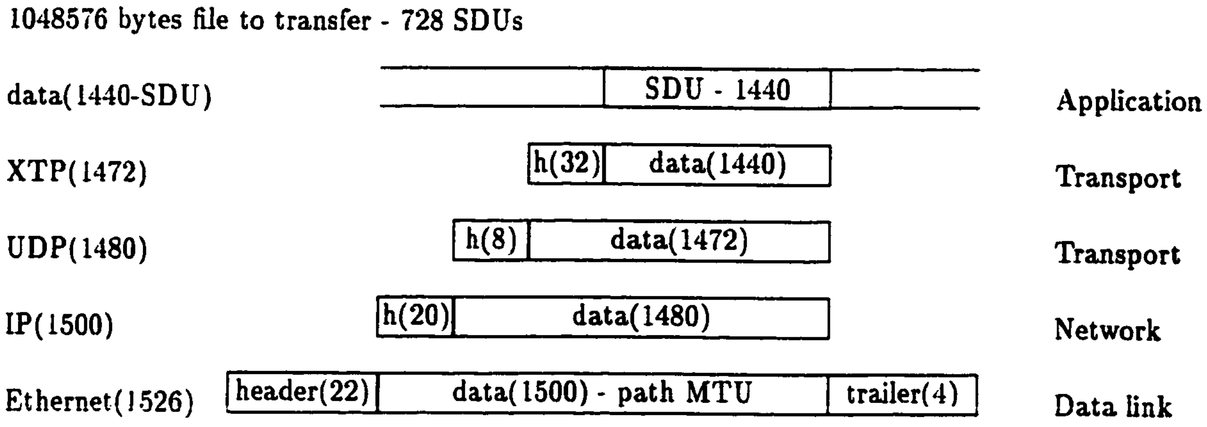


Figure 59: Data encapsulation for unicast experiments

10000 bytes per milliseconds (Bpms). The work of analysis and interpretation of the data produced could be further explored. In the following paragraphs, we venture some speculative observations.

Main parameters of the experiments (WS & data rate)

Questions of window size are generally considered within the realm of *flow control.* The goal of flow control is to prevent a fast sender from swamping a slow receiver with more data that it can handle. This goal is achieved by restricting the quantity of data sent to a limit not exceeding the residual buffer space available at the reception point. The receiver issues acknowledgements regularly to free up more buffer space.

Flow control and rate control are however related in the sense that both techniques are aimed at *regulating* the exchange of data between the sender and the receiver. Flow control regulates the sender through some sort of overdraft credit; rate control further regulates the sender by imposing time constraints on the consumption of those credits. Flow control is normally considered an end-to-end activity, whereas rate control is more considered to be path activity aimed at preventing congestion (called congestion avoidance) both along the intervening network (routers and data links) and also at the receiving end. It is now quite evident that both flow control and rate control may have impacts on throughput. Flow control may force the sender to stop and wait for acknowledgements from the receiver, with possible impacts on total file transfer time and throughput. Rate control will certainly impact on total transfer time by varying the amount of data "put on the wire" in a unit of time. It is therefore reasonable to have considered both aspects (rate and flow control) for these unicast file transfer experiments.

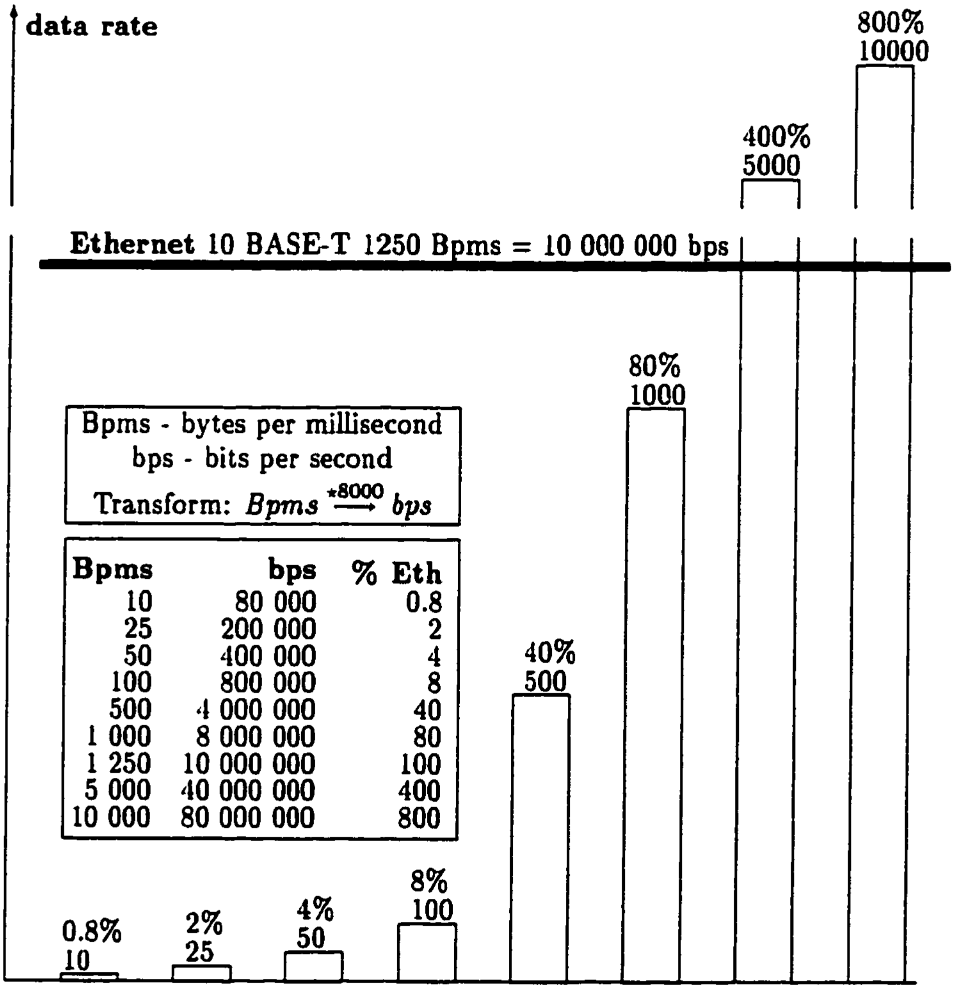
At this point, we introduce some differences in the terminology used in Deric's report and in this report. As per our interpretation of the unicast experiments, the term "window size" used in Deric's report corresponds to the term "send and receive (S/R) buffer size" (i.e., the shared memory area) in the context of the XTP protocol. From now on, we will use the more precise technical term "S/R buffer size" for our purpose, and the term "window size" with its usual meaning in the context of the sliding window protocol, i.e., as defining a subset of frames for the purpose of flow control.

Regarding the range of window size values and the inter-value gaps, we distinguish no particular predefined pattern, except the expectation that it should reveal the impacts on throughput. The default SandiaXTP default buffer size is 32768 bytes. Regarding the particular data rate values used for the unicast experiments, we detect no particular predefined scheme underlying the choice of the particular values. The goal was likely to trigger the underlying medium with a broad range of values so as to detect the extremes of behavior. To develop a sense of proportions, Figure 60 presents a bar diagram of the various data rates values used. At the top of Figure 60, the Ethernet 10 BASE-T nominal capacity (at 1250 Bpms or 10Mbps) is shown as a thick horizontal bar, and thus serves as the measuring stick against which we can compare the values used. On top of each vertical rectangle, we indicate the actual data rate values, both with the Bpms unit and as a percentage of the Ethernet nominal capacity.

Figure 60 clearly suggest a categorization of the values in two clusters: low and high data rates. Low data rates values range from 10 to 100 Bpms, or from 1% to 8% of Ethernet nominal capacity. High values range from 500 to 10000 Bpms, or from 40% to 800% of Ethernet nominal capacity. Given these percentages, one can already suspect that some data rates are "out of range" (such as 5000 and 10000), and the implications are further developed in the following paragraphs.

Significance of Deric's work

The outlook of present report is to correlate the offered load (sender) and the throughput. Because it suits better the saturation curve outlook, Deric's data have been reformated along a **window-size-first/rate-second** approach, with consistent units at both ends of the communication association (here in bytes/millisecond - Bpms). The recalculated data are presented in Table 17 for all window sizes. Corresponding to the data presented in Table 17, Figure 61 shows the saturation curves for all window sizes, ignoring the data rates of 5000 and 10000 Bpms. The speculative observations that follow relate to these reformulated data.



Bpms	bps	% Eth
10	80 000	0.8
25	200 000	2
50	400 000	4
100	800 000	8
500	4 000 000	40
1 000	8 000 000	80
1 250	10 000 000	100
5 000	40 000 000	400
10 000	80 000 000	800

Figure 60: Unicast data rate range of values

First, we abstract some very high level observations regarding the shapes of the curves, focusing on mega trends from curve to curve, and also within curves. Analysis of the changes from curve to curve should reveal the impacts of varying the window size, and analysis of the changes within curves should reveal the impacts of varying the offered load (i.e., the rate specified to XTP at the sender side).

Regarding the evolutionary changes of shape from curve to curve, one can observe a clear progression from a flat, undifferentiated curve shown in Figure 61 for WS=1440 bytes, to more vertical, better and better differentiated shapes that resemble the model Internet saturation curve discussed at the beginning of the report (see Figure 2). As the vertical axis represents the throughput, a more differentiated curve indicates more throughput responsiveness to changes in offered load. At least for the first part of the curves (say including points rate=10, 25, and 50 Bpms), it is as if an increase in window size was definitely resulting in an increased throughput, though

Table 17: Recalculation of Deric's data for unicast saturation curves

	Data rate (Bpms):	10	25	50	100	500	1000	5000	10000
WS=1440	time(ms):	-	301700	308245	270069	265137	260101	267950	-
	tput(Bpms):	-	3.47	3.40	3.88	3.95	4.03	3.91	-
WS=5120	time(ms):	175376	115870	110582	117388	107423	98638	127143	133162
	tput(Bpms):	5.97	9.04	9.48	8.93	9.76	10.63	8.24	7.87
WS=10240	time(ms):	161005	74523	68349	80712	66709	72845	71209	73547
	tput(Bpms):	6.51	14.07	15.34	12.99	15.71	14.39	14.72	14.25
WS=51200	time(ms):	142098	54443	43617	43253	36841	113203	186994	161122
	tput(Bpms):	7.37	19.26	24.04	24.24	28.46	9.26	5.60	6.50
WS=102400	time(ms):	144345	48522	40810	45639	37411	98690	90493	84982
	tput(Bpms):	7.26	21.61	25.69	22.97	28.03	10.62	11.58	12.33
WS=512000	time(ms):	140905	45499	38251	45161	36199	60242	99380	80185
	tput(Bpms):	7.44	23.04	27.41	23.21	28.96	17.40	10.55	13.07

Source: XTP PROJECT FOR COMP490 by Deric Sullivan, May 1997
WS = Window Size in bytes.
tput = throughput in Bpms; derived from time needed to receive the data & amount of data.
Ex: 1048576 / 140905 = 7.44 Bpms

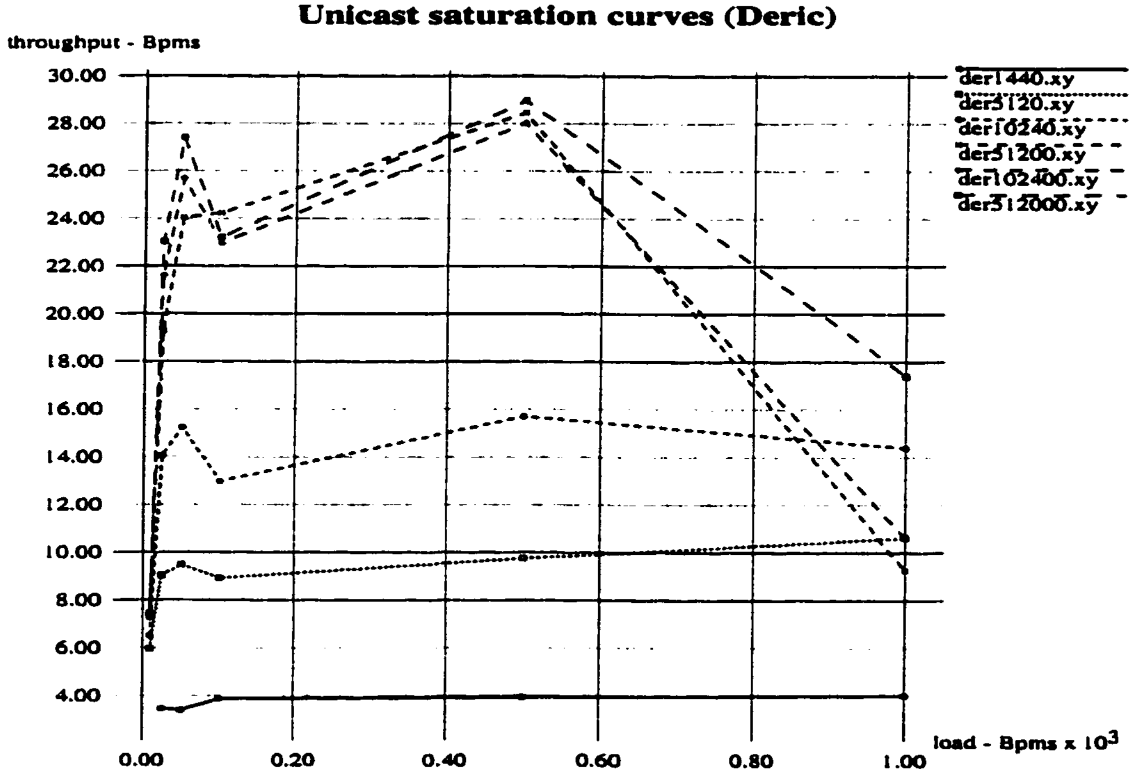


Figure 61: Unicast saturation curve for all window sizes

in a quickly diminishing fashion.

As the curves shown on Figure 61 are well differentiated for WS=51200, WS=102400, and WS=512000, we use them collectively to depict the changes occuring within a curve. For all three curves, there is a steep rise at the beginning of the curves, represented by the three first points (rate=10, 25, and 50). Then follows a relatively flat, or slightly down sloping portion from points for rate=50 to 100. Follows a slight rise again from points rate=100 to rate=500. Then, we have a dramatic fall in throughput from point rate=500 to point rate=1000. Beyond point rate=1000, the curves (not shown on Figure 61) are relatively flat or undifferentiated.

What do all those changes mean? To help formulate further and more detailed observations about the experimental data underlying the curves, we use a thematic/questioning approach. Such thematic questions consist of: (1) Why is the curve shown on Figure 61 for WS=1440 so flat or undifferentiated?; (2) Why are the end parts of all curves so undifferentiated?; (3) What are the impacts of varying the window size, and what appears to be the optimal window size (particular to those experimemts, of course)? Those issues are further discussed in the following paragraphs.

Why is the curve for WS=1440 so flat or undifferentiated

Short answer: because the window size is triviallly too small (in effect producing a stop-and-wait mode of exchange), given the PDU size and the long distance nature of the WAN link used.

The window size is 1440 bytes, and the PDU size used is 1472 bytes (see [SUL, p26]). With reference to textbooks on Data Communications and Computer Networks, such as [TAN, p239], this scenario corresponds to a window size of unity (WS=1), and implies a *stop-and-wait* protocol; sender sends one packet and waits for an acknowledgement from the receiver. As per the theory, the performance of the stop-and-wait protocol is very low; most of the time is spent waiting for acknowledgements.

As throughput is a derived quantity computed from the total time needed to effect the transfer of a file, and so much time is spent waiting, this would explain why we get such a low throughput. Furthermore, as only one packet is sent at a time, and it needs only one hardware packet, it is sent at the hardware rate. Increasing the XTP rate has no effect (more than one second will elapse before sending the next one anyway), and this explains why the throughput remains relatively constant, producing a flat curve.

(2) Why are the end parts of the curves so flat/undifferentiated?

Table 18: Unicast Time duration VS Window Size at point rate=50 Bpms

WS bytes	time ms	time s
1440	308245	308
5120	110582	110
10240	68349	68
51200	43617	44
102400	40810	41
512000	38251	38

Source: Table 17

Short answer: because, due to hardware limitations, offered loads (rates) of 1000Bpms and above are in effect all equivalent, thereby yielding approximately an equivalent throughput.

(3) What are the impacts of varying the window size, and what appears to be the optimum window size (at least for the particular link used)?

Short answer: From WS=52100 bytes (probably the optimum), the impact on throughput of increasing the window size is slight.

In his discussion about the performance of Sliding Window Protocols, [TAN, p242 ff] clearly shows that channel utilization increases with the window size (...up to a point). He goes on with such statement as: "If window size is large enough, then the sender can just keep going at full speed because the acknowkedgments get back before the sender's window fills up." or "If window size is at least one larger than the number of frames that fit on the cable, transmission can go continuously". The context of those statements is for relatively short cable lengths, i.e., for cable length of 1 to 5 frames, and for full occupancy of the channel.

Even if Tanenbaum is discussing about gain in global channel utilization, and for short length LANs, one can observe an equivalent general upwards trend in the throughput with increasing window sizes, for the present long distance unicast experiments, as shown on Figure 61.

From now on, the focus of the discussion will not be so much about the existence of such upward trends, as confirmed by our interpertation of the unicast experiments, but rather about the pecularities of the changes and the lessons that can be infered for further experiments. The facet that we develop is more of an economical nature; equating increase in window size as being a higher demand on a scarce resource (a

Table 19: Unicast window size - throughput incremental analysis

Ref Tab.	WS byte	WS #	MEM incremental cost bytes	throughput incremental benefits Bpms
17	1440	1	base case	base case
17	5120	3 (1)	add 3680 (2)	each add 1000B →≈ +1.24Bpms (3)
17	10240	6	add 5120	each add 1000B →≈ +0.74Bpms
17	51200	34	add 40960	each add 1000B →≈ +0.12Bpms
17	102400	69	add 51200	each add 1000B →≈ +0.02Bpms
17	512000	347	add 409600	each add 1000B →≈ +0.002Bpms

Examples of calculations:

(1) Window Size: $5120/1500 = \lfloor 3.47 \rfloor = 3$ (Ref: [TAN, p242])

(2) Main MEMory incremental portion: $5120 - 1440 = 3680$

(3) Incremental benefits:

$$prev = (3.47 + 3.40 + 3.88/3) = 3.58$$
$$cur = (5.97 + 9.04 + 9.48/3) = 8.16$$
$$improvement = cur - prev/3.68 = 1.24$$

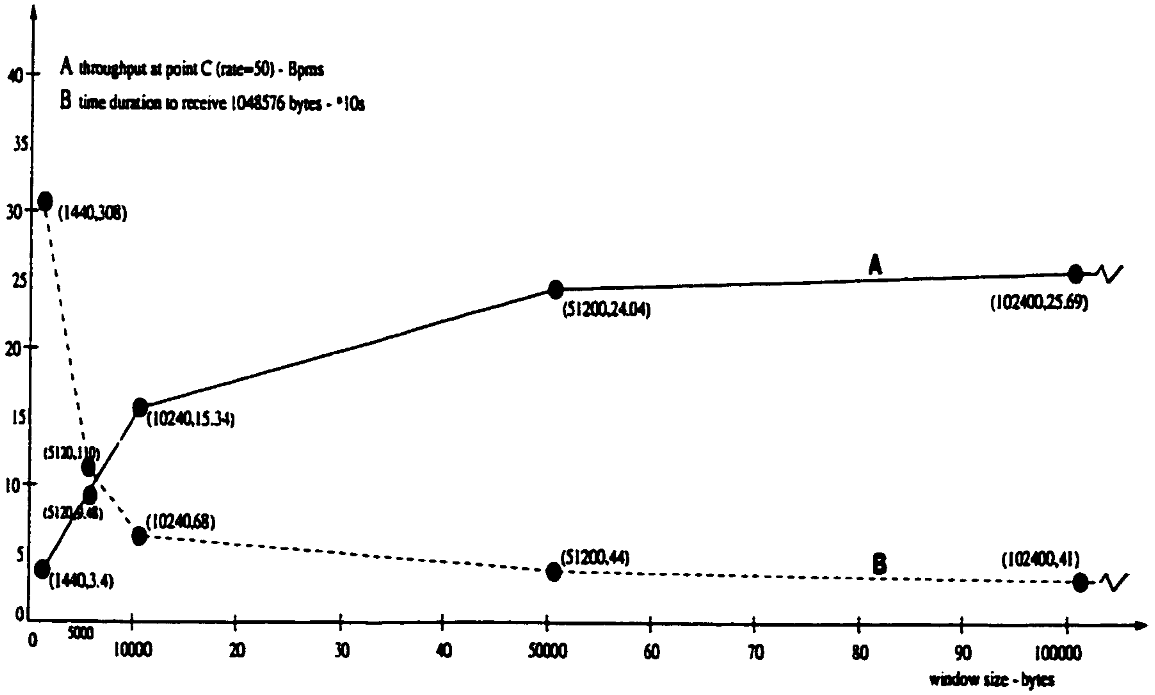


Figure 62: Unicast window size - throughput gain curve at point C (rate=50)

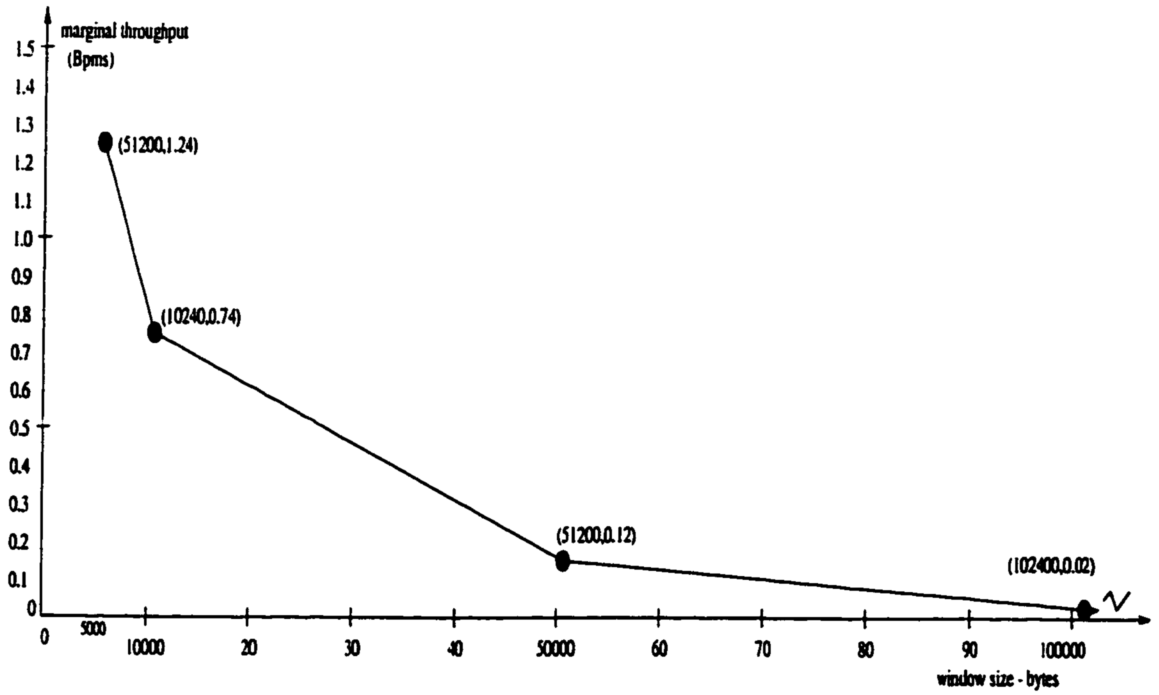


Figure 63: Unicast window size - throughput Diminishing Return curve

cost in terms of host main memory space, and also in contributed experimental complexity), and an increased in throughput as being a benefit in terms of reduced total duration time to transfer a large file. As the relation is far from being linear, the crucial questions becomes: is is worth complicating further experiments with many window sizes; and if not, what window size(s) could we select? The result of this economically minded analysis is presented in Tables 18 and 19, and in their corresponding "diminishing return" curves shown in Figure 62 and Figure 63. Here follows some observations based on the data and the curves.

First, let us discuss the curves shown in Figure 62; where curve A is built from point rate=50 only of the six unicast saturation curves, and curve B is built from data presented in Table 18. Point rate=50 was selected because of its key importance, representing the practical end of gain in throughput. For instance, to show the relationship between curve A shown on Figure 62 and the six unicast saturation curves, one should realize that the leftmost point of curve A (1440,3.4) corresponds to point rate=50 of the unicast curve shown on Figure 61 for WS=1440, next point of curve A (5120,9.48) corresponds to point rate=50 for WS=5120, and so on.

The purpose of curve A is to diplay the exact "diminishing return" nature of the gain in throughput VS increase in window size, from a commitment of system resources point of view. One can observe a steep gain in throughput for window size varying from 1440 to 51200. Thereafter, the shape of curve A is almost flat and the gains in throughput are very small despite massive incremental investments in window size (or amount of main memory devoted to buffering packets).

The purpose of curve B is to expose the exact "diminishing return" nature of the gain in total transfer time VS increase in window size, from a user point of view. As derived from Table 18, a 900% increase in buffer space (51200 to 512000) yields only a 14% (44 to 38s) reduction in total transfer time. Psychologically, given the long distance of the link (over 5 000 000 m), and the large file size (1MB), a waiting time of 44 seconds instead of 38 seconds should not be much of an inconvenience for the experimenter as well as for the user. The result is that the window size, for the time being at least, does not appear to be a sensible enough factor to invest much more investigation time.

Table 19 and its corresponding Figure 63 present results of a similar cost-benefits analysis, but using incremental concepts, and taking into consideration the first three points (rate=10, 25, and 50) of every unicast saturation curve. As this other analysis confirm the analysis done for point rate=10 only, no further comments are made here.

To conclude this part of the analysis, it would appear that a window size of 51200 bytes is the optimum in the sense that commitment of system resources are small, and impact on throughput is small, at least from the aspect of further file transfer experiments on the same link.

Lessons for further experiments?

The core value of Deric's experiments probably consist of indicating areas require further investigation. Globally, the list of parameters for further unicast experiments using the Internet environment is follows:

WS: 51200, 102400 bytes
Offered load: 10, 15, 20, 25, 30, 35, 50, 75, 100, 125 Bpms

C Programming and tools used for the experiments

C.1 Sandia source files modified

Table 20 shows a listing of the Sandia source files modified for conducting the experiments with the modified version of the daemon. The directory paths where these files are available are also indicated. As explained in Chapter 6, all changes introduced in the code are flagged with the keyword `//---icici---`.

Table 20: List of Sandia files modified

MTL		SandiaXTP	
header files[1]	C++ implem. files[2]	header files[3]	C++ implem. files[4]
MTLtypes.h context_manager.h mtldaemon.h	mtldaemon.C udp_del_serv.C	XTPcontext.h XTPcontext_manager.h XTPdaemon.h XTPtypes.h	XTPcontext_gen.C XTPcontext_manager.C XTPcontext_recv.C XTPcontext_send.C XTPdaemon.C XTPpacket.C xtpd.C

[1] **/mnt/jwa/jwal/grad/harveyl/pkg/sxtp/mtl-1.5.1/include/**
[2] **/mnt/jwa/jwal/grad/harveyl/pkg/sxtp/mtl-1.5.1/src/**
[3] **/mnt/jwa/jwal/grad/harveyl/pkg/sxtp/SandiaXTP-1.5.1/include/**
[4] **/mnt/jwa/jwal/grad/harveyl/pkg/sxtp/SandiaXTP-1.5.1/src/**

C.2 Client programs used for testing (C++ source code)

As explained in Section 7.5, the C++ source code used to compile program **mmetric** is included in files mmetric.C, mbulk.C and mcommon.h. Here follows the directory path where these files can be found:

/mnt/jwa/jwal/grad/harveyl/experiments/withSXTP/src/

C.3 Perl utility programs

Table 21 shows a summary of the Perl utility programs developed for conducting the experiments. The directory path where these files are available is also indicated.

C.4 Udip - a unicast/multicast testing program

Udip is a "small" testing program developed at the Concordia University HSP Lab. for experimenting with data communications in a UNIX environment. Through command line arguments, udip can be customized to behave as sender or receiver (but not both - udip is not duplex), and the unicast as well as the multicast modes of

Table 21: List of Perl utility programs

Name	Functionality
play[1]	Ex: `play -ur` to start the receiver for a unicast experiment unit covering all send rates. See Table 7
configPlay[1]	To configure program `play`; interactively calls for the settable parameters.
playm[1]	To conduct multicast experiments. See Table 8
xtractXY[1]	To format the xy_file for xgraph. See Table 7
xtabLOG[1]	To format the log summaries. See Table ux01.log
xtabXY[1]	To format the data summaries. See Table ux01.xy
1. `/mnt/jwa/jwal/grad/harveyl/bin/`	

communication are supported. udip is non-reliable and presently uses solely UDP as its underlying data delivery service (DDS), which allows it to run without superuser permission. Eventually, udip may offer the choice to use raw IP as the underlying DDS.

The networking socket interface is used. For multicasting, udip relies on IP multicast, which implies an Operating System supporting IP multicast, such as Sun Solaris2.5 that was used as its development testbed. Eventually, udip is meant to run on other UNIX platforms, such as Linux. udip has a class design loosely based on the SandiaXTP implementation of the Xpress Transport Protocol (XTP) and is implemented in C++.

The source code and the documentation can be obtained from the following directory paths:

```
/mnt/jwa/jwal/grad/harveyl/pkg/udp/udip/doc/
/mnt/jwa/jwal/grad/harveyl/pkg/udp/udip/include/
/mnt/jwa/jwal/grad/harveyl/pkg/udp/udip/src/
```

D SandiaXTP class description dictionary

The Class Dictionary is incomplete, and may be inaccurate. It was prepared progressively as an aid to help synthesizing the MTL and SandiaXTP source code, and no attempt was made update its accuracy as understanding of the software kept evolving.

This **Class Description Dictionary** covers all the classes shown on the global "SandiaXTP Class Diagram", though at a very greatly varying depth of coverage. The ordering principle used for presenting the classes is the alphabetical/dictionary arrangement, without consideration to the class inheritance relationships. For each class, a specific *three-areas* class diagram is shown, with "as significant as possible" samples of the actual variables and functions included in the header files. For each class, or grouping of related classes, a short textual description explaining their role is supplied, and possibly some *object diagram(s)* to illustrate a typical execution scenario.

The order of presentation of the classes is as follows (read horizontally):

buffer_manager
CNTLpacket
context
context_manager
dds_address
del_serv
DATApacket
DIAGpacket
ECNTLpacket
event_queue
FIRSTpacket
ip_dds_address
ip_del_serv
JCNTLpacket
lib_manager
mtldaemon
mtlif
packet
packet_fifo
packet_pool
state_machine
TCNTLpacket
timeout
timer
udp_dds_address
udp_del_serv
user_request
XTPcontext
XTPcontext_manager
XTPdaemon
xtpif
xtp_reg_msg
XTPstate_machine
xtp_state_msg
xtp_trans_msg

The LaTeX source file for the class description dictionary is available at the following directory path:

`/mnt/jwa/jwa1/grad/harveyl/report/appendix/sxtpClassDictionary.tex`

The xfig source files are also available at the following directory path:

`/mnt/jwa/jwa1/grad/harveyl/report/appendix/sxtpClassDict/`

Tables 22 and 23, and Figure 64 show the mapping of the class design to file organization and layout in the file system.

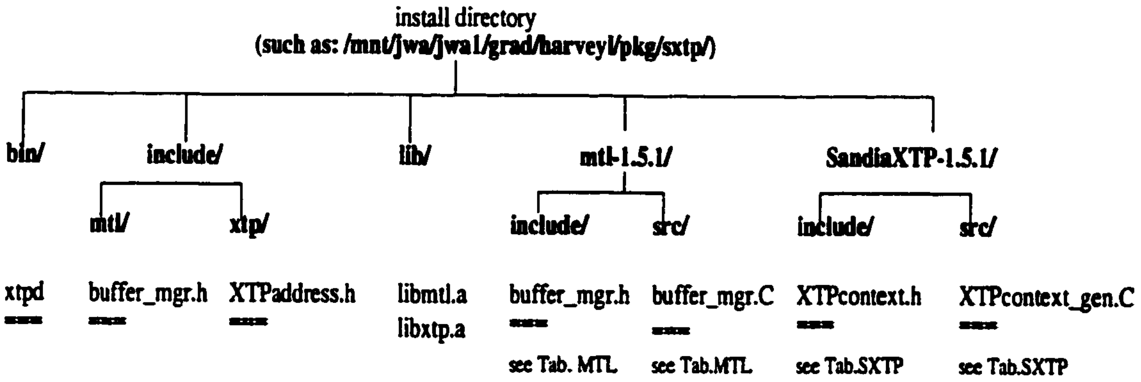


Figure 64: Software organization after installation

Table 22: MTL class/file organization

Class	header file	implementation file
buffer_manager	buffer_manager.h	buffer_manager.C (19302)
context	context.h	context.C (7279)
context_manager	context_manager.h	context_manager.C (13559)
dds_address	dds_address.h	none
del_srv	del_srv.h	none
event_queue	event_queue.h	event_queue.C (8390)
ip_dds_address	ip_dds_address.h	none
ip_del_srv	ip_del_srv.h	ip_del_srv.C (13537)
lib_manager	mtlif.h	none
mtldaemon	mtldaemon.h	mtldaemon.C (44109)
mtlif	mtlif.h	mtlif.C (16477 bytes)
packet	packet.h	packet.C (18304)
packet_fifo	packet_fifo.h	packet_fifo.C (3159)
packet_pool	packet_pool.h	packet_pool.C (6536)
state_machine	state_machine.h	none
timeout	timer_aides.h	none
timer	timer_aides.h	none
udp_dds_address	udp_dds_address.h	none
udp_del_srv	udp_del_srv.h	udp_del_srv.C (14582)
user_request	user_request	none
	MTLtypes.h	
	intr_handlers.h	intr_handlers.C (4189)
	mtlconf.h	
	mtlsignal.h	mtlsignal.C (2663)
	prototypes.h	
	word64.h	

Table 23: SandiaXTP subclass/file organization

Subclass	**header file**	**implementation file**
CNTLpacket DATApacket DIAGpacket ECNTLpacket FIRSTpacket JCNTLpacket TCNTLpacket	XTPpacket.h	XTPpacket.C (31952)
XTPcontext	XTPcontext.h	XTPcontext_gen.C (74698) XTPcontext_recv.C (71535) XTPcontext_send.C (72148)
XTPcontext_manager	XTPcontext_manager.h	XTPcontext_manager.C (62307)
XTPdaemon	XTPdaemon.h	XTPdaemon.C (27133)
xtpif	xtpif.h	xtpif.C (31365)
xtp_reg_msg	XTPuser_request.h	none
XTPstate_machine	XTPstate_machine.h	XTPstate_machine.C (6737)
xtp_state_msg	XTPuser_request.h	none
xtp_trans_msg	XTPuser_request.h	none
		xtpd.C (3790) (driver program)
	XTPaddress.h	
	XTPdiag_msg.h	
	XTPtraffic.h	
	XTPtypes.h	
		xtpdreset.C (2461)
		xtpdrm.C (2624)
		xtpds.C (4256)

E Keywords summary

Bpms (bytes per millisecond)

See **Data rate units**

Data rate units (bytes per millisecond - Bpms)

The data rate is a measure of the number of bits transferred per unit of time. In the literature, the data rate is expressed in *bits per second (bps)* (Ex: 10 Mbps for Ethernet 10 BASE-T; or 1.544 Mbps for Bell System T1 carrier). However, [XTP40, p66] specifies the data rate in *bytes per second.* The SandiaXTP implementation of XTP expresses the data rate in *bytes per millisecond* (abbreviated to Bpms), which is also the unit used in this report. For convenience, a table of conversion for many of the data rates used follows:

units	Data rate values						
Bpms:	10	25	50	100	500	1 250	4 250
bps:	80 000	200 000	400 000	800 000	4 000 000	10 000 000	34 000 000
Mbps:	0.08	0.2	0.4	0.8	4	10[1]	34

1. Ethernet 10 BASE-T nominal data rate - 10 Mbps

Transform: $B/ms \xrightarrow{*8000} bps$

HSP Lab High Speed Protocols Laboratory *Department of Computer Science, Concordia University, Montreal, Dr. J. W. Atwood (Director)*

The mission of the High Speed Protocols Laboratory (HSPL) consists in the specification, validation, testing, and performance evaluation of high speed data communication protocols. The specification and validation aspects are being investigated using formal descriptions methodologies such as Estelle, LOTOS, Valira, and Promela. The performance evaluation aspect is being investigated using simulation techniques, automatically-produced implementations (using Estelle), and hand-produced implementations (in C and C++).

The primary focus of activity since 1987 has been the Xpress Transport Protocol (XTP), a new transport level protocol designed for high-speed environments (100 Mb/s). XTP offers many innovating features such as multicasting and quality of service (QOS). With the publication of XTP 4.0 in March 1995, the multicasting functionality was considerably improved.

MBONE What is the MBONE? (excerpt from http://www.mediadesign.co.at/newmedia/more/mbone-faq.html)

The MBONE is an outgrowth of the first two IETF "audiocast" experiments in which live audio and video were multicast from the IETF meeting site to destinations around the world. The idea is to construct a semi-permanent IP multicast testbed to carry the IETF transmissions and support continued experimentation between meetings. This is a cooperative, volunteer effort.

The MBONE is a virtual network. It is layered on top of portions of the physical Internet to support routing of IP multicast packets since that function has not yet been integrated into many production routers. The network is composed of islands that can directly support IP multicast (e.g., multicast LANs such as Ethernet), linked by virtual point-to-point links called "tunnels". The tunnel endpoints are typically workstation-class machines having operating system support for IP multicast and running the "mrouted" multicast routing daemon. See also: ftp.isi.edu:mbone/faq.txt by Steve Casner, casner@isi.edu, 22-Dec-94

MTU Maximum Transmission Unit

The component handed over to the Data Link layer (i.e., a composite of the XTP packet plus the UDP and IP headers) is considered as "data" to encapsulate within a Data Link packet. The *MTU* is the upper limit on the number of bytes of data that the data link layer can encapsulate. Beyond this limit, IP performs fragmentation into many fragments. The *path MTU* is the smallest MTU of any data link between end points (as this path may vary, the path MTU may also vary). As per [SUL, p11], the experimental path MTU found was **1500 bytes**. This value corresponds to the maximum data field size of an Ethernet frame (IEEE 802.3), as shown in [SHAY, p330]. Including the Ethernet header and footer (max 26 bytes), the full size of a data link layer packet is thus **1526 bytes**. See also PDU and SDU entries.

PDU Protocol Data Unit

PDU is another name for "packet". More precisely, for this report, we are referring to the XTP/Transport layer packet (or TPDU). In [SUL, p11], the XTP PDU size used is derived to prevent fragmentation by Ip as follows:

XTP PDU size = path MTU - (UDP header + IP header) = **1472 bytes**

where:

path MTU = 1500 bytes;
UDP header = 8 bytes ;
IP header = 20 bytes

If the size of one XTP packet is 1472 bytes, including a 32 byte fixed size header, then the data portion is 1440 bytes (see SDU for more details).

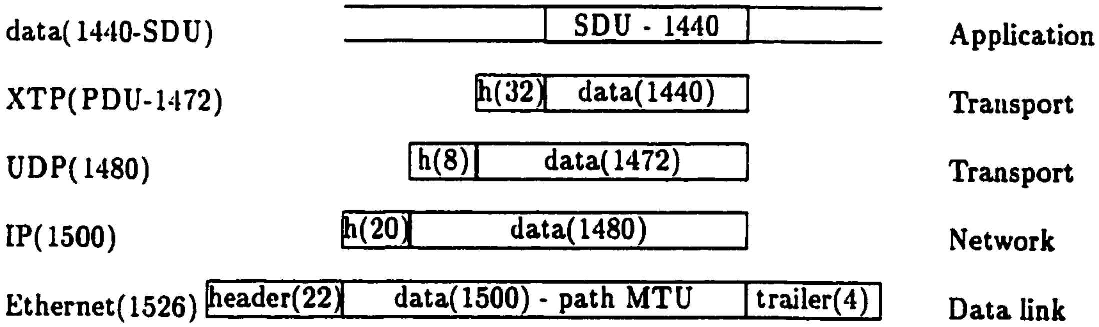
rseq see also [XTP40, p26]

-receive sequence number field- The *rseq* field holds the sequence number of the next in-sequence byte expected on this data stream, so it is one greater than the highest contiguously received data byte.

SDU Service Data Unit

As per [TAN, p21], "The SDU is the information passed across the network to the peer entity and then up to layer N + 1. The control information is needed to help the lower layer do its job, but is not part of the data itself". Here follows a diagram explaining the semantics of the SDU for the rate control experiments. The SDU corresponds to the **-b** (i.e., user buffer size) command line option of a client program such as mmetric.

1048576 bytes file to transfer - 728 SDUs



Shortest-Job-First Scheduling see also [SG, p139]

The following is an excerpt of Silberschatz & Galvin book: The next CPU busrt is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the nth CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α, $(0 \leq \alpha \leq 1)$, define

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

This formula defines an *exponential average.* The value of t_n contains our most recent information; τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient); if $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted.

tseq see also [AUE, p19]

-highest actually sent sequence number (context parameter *tseq*)- again an in-sequence byte number.

Variance The variance of n observations $x_1, x_2, \ldots, x_n$ measures esentially the average of their squared deviations from their mean, $\overline{x}$, and it is defined by the formula

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}$$

References

[AMZ] J. William Atwood, Anne Moreau, Yaolin Zhang. *A Service Interface for a Parametric Transport Protocol* ©1995 (an unpublished article)

[AUE] Torsten Auerbach. *Using Error Status Information in the Xpress Transport Protocol to Improve Performance* – A report in the Department of Computer Sience. Concordia University, March 1997

[BOG] David R. Boggs, Jeffrey C. Mogul & Christopher A. Kent. *Measured Capacity of an Ethernet: Myths and Reality.* in ACM SIGCOMM'95, pp124-137

[FALCOT] Pierre Falcot, *Traffic analysis with XTP in unicast environment*, Final Studies Project Report, Concordia University, May 25, 1998

[MTL] Sandia National Laboratories. *Meta-Transport Library User's Guide* – Meta-Transport Library, A Protocol Base Class Library, Release 1.4, Published by: Infrastructure and Networking Research, Sandia National Laboratories, P.O. Box 969 Mailstop 9011, Livermore, California 94551-0969

[SDW] W. Timothy Strayer, Bert J. Dempsey, Alfred C. Weaver *XTP: The Xpress Transfer Protocol* – ©1992 by Addison-Wesley Publishing Company, Inc.

[SG] Abraham Silberschatz & Peter B. Galvin *Operating System Concepts* – ©1994 by Addison-Wesley

[SHAY] William A. Shay *Understanding Data Communications and Networks* ©1995 by PWS Publishing Company

[STEV1] W. Richard Stevens. *TCP/IP Illustrated, Volume 1* – The Protocols. ©1994 by Addison-Wesley

[STEV2] W. Richard Stevens. *UNIX Network Programming* ©1990 by Prentice Hall PTR

[SUG] Sandia National Laboratories. *SandiaXTP User's Guide* – SandiaXTP An Object-Oriented Implementation of XTP 4.0 Derived from the Meta-Transport Library, Release 1.4, Published by: Infrastructure and Networking Research, Sandia National Laboratories, P.O. Box 969 Mailstop 9011, Livermore, California 94551-0969

[SUL] Deric Sullivan. *XTP Project* – A report for COMP 490. Concordia University, May 1997

[TAN] Andrew S. Tanenbaum. *Computer Networks* - Second edition ©1989 by PTR Prentice-Hall, Inc.

[VANJ] Van Jacobson. *Congestion Avoidance and Control* – in 1988 ACM 0-89791-279-9/88/008/0314, p314-329

[WAL] Marie A. Wallace *Error Control in XTP over the Internet* – A Major Report in The Department of Computer Science. Concordia University, December 1997

[XTP40] XTP Forum *Xpress Transport Protocol Specification* – XTP Revision 4.0 March 1995, Published by the XTP Forum, 1394 Greenworth Place, Santa Barbara, CA 93108, USA

Index