From Specification to Realization: Implementing the Express Transfer Protocol

by

Philip M. Irey IV

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Dr. Richard E. Nance

Dr. Dennis G. Kafura

Mr. David T. Marlow

Blacksburg, Virginia

From Specification to Realization: Implementing the Express Transfer Protocol

Philip M. Irey IV

Naval Surface Warfare Center
Dahlgren, VA 22448

Committee Chairman: Dr. Richard E. Nance
Computer Science

(ABSTRACT)

The research described in this thesis deals with effective protocol specification. The primary question addressed is whether the Express Transfer Protocol (XTP), a "real-time" Transport layer protocol, is sufficiently specified or are there "holes" in its specification? A new protocol evaluation process is formulated and applied to XTP in order to answer this question. The evaluation process combines a detailed analysis of the XTP specification with an attempt to implement parts of the protocol from the specification. Special attention is given to those aspects of the protocol that affect "real-time" naval tactical communications.

The detailed analysis of the specification and its effect on the specification revision process are presented. The analysis is described in formal comment papers and electronic mail transmitted to the protocol designer, Dr. Greg Chesson.

Elements of the protocol most applicable to tactical communication are selected for implementation. A number of assumptions are made so that an implementation can be built. The design of the prototype implementation and the assumptions made to build it are discussed. Both the hardware and software being used to build the implementation are presented.

The protocol evaluation process is found to be appropriate for evaluating XTP. A comparison is made between this technique of protocol evaluation and existing techniques (i.e. simulation, complete implementation, and protocol verification).

The principal conclusion of the protocol evaluation process is that a number of areas of the XTP Protocol Definition (prior to Revision 3.3) are not sufficiently specified. These areas must be properly specified before a complete implementation can be built. Without a complete specification, the communications support intended for XTP is unlikely to be realized.

Acknowledgements

The author would like to sincerely thank the chairman of this committee, Dr. Richard E. Nance, for the guidance provided to make this thesis possible. Appreciation is also extended to Dr. Dennis D. Kafura for his service on my committee.

Dr. Greg Chesson and Protocol Engines, Incorporated must also be thanked for bringing XTP into the world. As documented in Appendix F, Dr. Chesson patiently answered my questions in great detail. In a number of meetings, Dr. Chesson carefully considered my concerns about the protocol.

Special appreciation must be given to of the Naval Surface Warfare Center who served on my committee. not only made this project possible but ensured that it proceeded in a direction that would satisfy the needs of my graduate committee and would provide an endproduct that is useful to the Navy. must be commended for his infinite patience in dealing with a very trying and tired graduate student.

Finally, I must thank my family for the warm support they offered when it was needed most. They provided the encouragement I needed to complete this thesis.

Table Of Contents

Appendices

List of Figures

Chapter 1.

1. Introduction

Traditionally, Naval Tactical Data Systems (NTDS) have performed real-time data communications using fast, parallel point-to-point connections. This is no longer feasible. Today's tactical systems utilize an ever increasing number of computers necessitating a higher number of interconnections yet individual connections require very high communications bandwidths. These requirements cannot be easily achieved using traditional point-to-point connections. The Navy is looking toward Local Area Networks (LANs) to meet these more demanding requirements.

1.1. Methods of Computer Communications

The point-to-point interconnection of systems requires a communication channel to connect each pair of systems that desire to communicate. A point-to-point interconnection scheme with complete interconnection is shown in Figure 1. Such a scheme has a number of advantages over other interconnection methods. Systems using this technique are highly reliable. If a channel becomes inoperable, only communications over that channel are interrupted. Other channels may continue to communicate as they operate independently of the faulty channel.

Since a high degree of simultaneous communications can be achieved using many point-to-point links, the total communications bandwidth of the system can be quite high. In Naval Tactical Data Systems, a computer may have many point-to-point channels implemented using sixteen or thirty-two bit parallel interconnections with bandwidths approaching 10 megabits per second per channel.

Finally, the software required to communicate reliably over a point-to-point channel is several orders of magnitude less complex than that used in a typical LAN. Error detection and recovery, packet routing and addressing, buffer management, and flow control become trivial issues when compared to the methods used for LANs.

Point-to-point interconnections have a number of disadvantages, however. As the number of systems wishing to communicate increases, the number of channels required to enable communications increases. A system with a large number of channels is hard to maintain and is inflexible. Also, the number of channels required to provide the desired level of interconnection can be prohibitive.

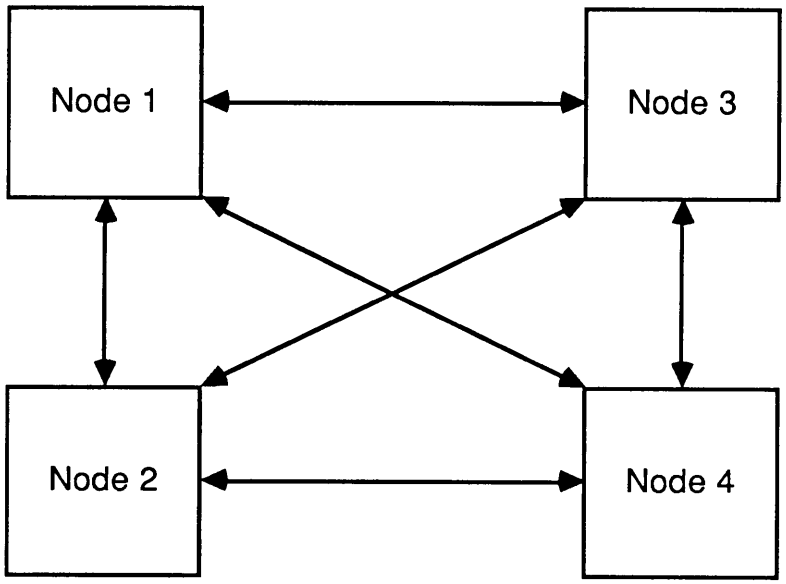


Figure 1. Point-to-Point Communication with Complete Interconnection

These problems are most evident in existing Navy computers. The Navy's standard 16-bit computer, the UYK-44, has only 16 I/O channels. With over forty computers in a modern combat system, the standard UYK-44 can not support full point-point connectivity. The problem becomes even more complex when redundant channels are introduced.

Weight and space are two important factors in the design of a combat system. Forty UYK-44 computers, each with sixteen communications channels, requires 640 channels for connectivity. Conceivably, the replacement of all point-to-point channels using copper cables by a single fiber optic LAN can save many tons in cable weight on a ship.

1.2. Introduction to Local Area Networks

LANs allow many systems to communicate using a single channel. Communications over the channel are multiplexed between all of the systems using the channel. LANS do not suffer from the limited connectivity typical of point-to-point channels.

LANs can be configured using a number of topologies. These topologies are usually characterized by a single data path connecting all of the components of a system. A ring and linear bus topology are shown in Figures 2 and 3 respectively. A number of communications standards have been based on these two LAN topologies.

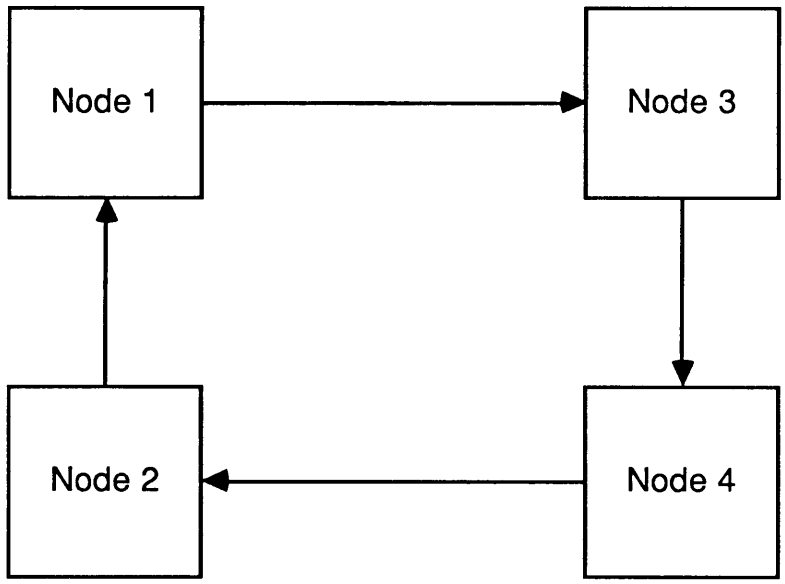


Figure 2. A Ring Topology

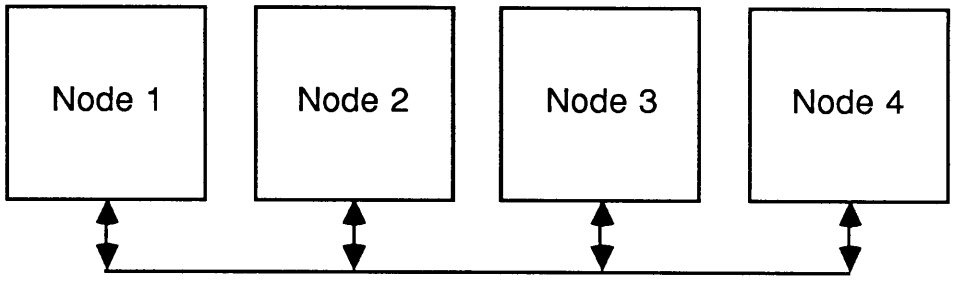


Figure 3. A Linear Bus Topology

The Ethernet standard [DIX82] is based on the linear bus topology. Ethernet is being used for a wide variety of applications ranging from inter-office communication to factory automation. The 802.5 standard [IEEE85] is based on the ring topology. Using 802.5 LANs has become popular in connecting microcomputers. FDDI [ANSI86] is also based on the ring topology. Using a high speed fiber optic communications media, FDDI is expected to dominate future high performance LAN communications.

1.3. The International Organization for Standardization Open Systems Interconnection Reference Model

The introductory period for LAN technology is marked by proprietary hardware and software. The consequent problems both for LAN vendors and customers are obvious: 1) small markets for vendor products since the enormous development efforts can not be justified for a small market, 2) customers became "enslaved" to a particular vendor since selection of a particular LAN forced the acquisition of products which interfaced to the LAN from that vendor. In this period the availability of products is limited to the vendor of the proprietary LAN.

Efforts are being made to produce open communications standards. Standards benefit both the communications vendor and customer. The vendor benefits because development costs are reduced. Open products tend to be more marketable today than proprietary products. The customer benefits because products can be purchased from any vendor who complies with the desired standard. One organization coordinating such standards on an international basis is the International Organization for Standardization (ISO).

ISO has defined a communication model, called the Open Systems Interconnection (OSI) Reference model [ISO84], which divides network communications into seven logical layers as shown in Figure 4. Each of the layers operates without knowledge of the inner workings of the other layers. Protocol specifications define the interactions between peer entities while Service Definitions are defined between adjacent layers. The ISO/OSI Reference Model provides a framework for developing the Protocol Specifications and Service Definitions ISO is defining for all seven layers.

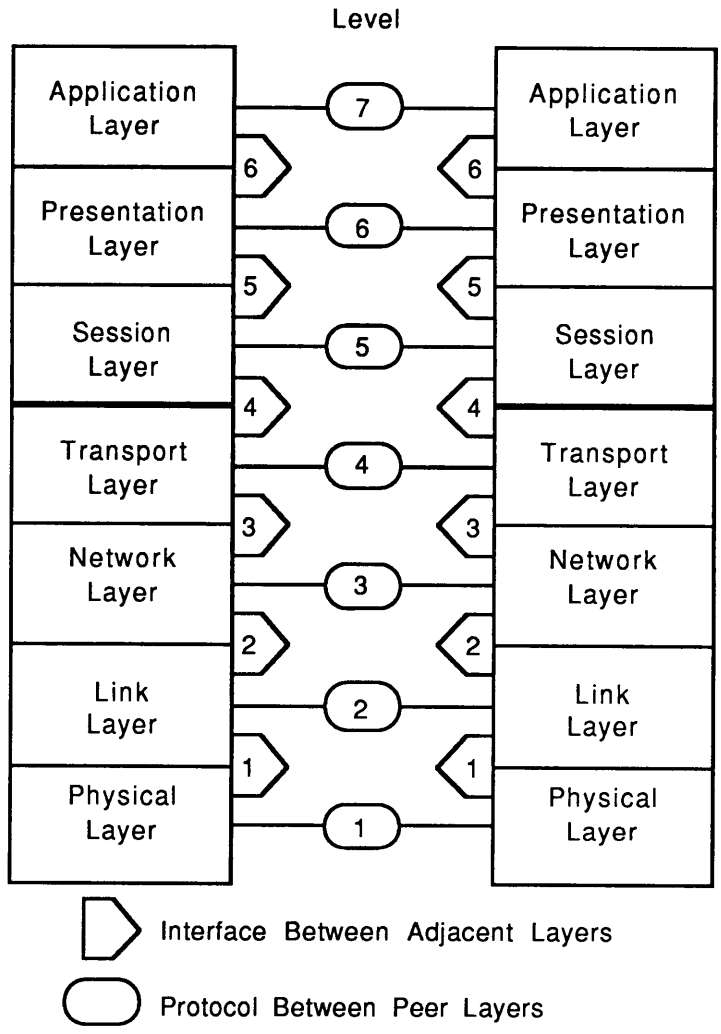


Figure 4. The ISO Reference Model

1.3.1. Application Layer

Layer seven, the Application layer, serves as a window between application processes that are exchanging data. It is the only layer in the OSI model that provides an application with direct access to the services provided by the model.

Functions provided by the Application layer can be divided into three categories: user specific service elements, application specific service elements, and common service elements. User specific service elements provide the functions needed by a specific user (i.e. an airline reservation system, an order tracking system, etc.). Application specific elements provide the mechanisms needed for a particular application (i.e. virtual terminal service, file transfer service, etc). Common service elements provide the services needed for OSI communications regardless of the application.

1.3.2. Presentation Layer

Layer six, the Presentation layer, is concerned with ensuring that the data transferred between Application layer entities is in a format understood by those entities. This is done by syntax translation protocols. Other translations such as data compression and encryption can be accomplished at this layer.

In a heterogeneous network environment, problems arise when exchanging data structures. Different machines, languages, and application programs typically use different storage formats for data structures. For example, consider the representation of an integer. A sixteen-bit microprocessor might use 16 bits for the representation while a 32-bit microprocessor might use 16 or 32 bits or allow both. Some microprocessors store integers in memory in a swapped byte format [COHEN81]. Presentation layer protocols such as Sun Microsystem's XDR [SUN86] and ISO's ASN.1 define a common format to exchange data structures.

1.3.3. Session Layer

Layer five, the Session layer, provides the services necessary for Presentation layer entities to synchronize their dialogue and to manage requisite data exchanges. Typical services of the Session layer are to provide normal and expedited data exchange, token management, dialogue control, synchronization and resynchronization, and exception reporting.

Token management allows Presentation layer entities to exchange a token. The token could be used to grant exclusive access to a certain function. Dialogue control allows full duplex or half duplex communications to be selected. Synchronization provides a mechanism to allow checkpoints to be placed in the data stream to mark events. If an error is detected, resynchronization allows the communicating Presentation layer entities to "roll-back" to a synchronization checkpoint.

Many OSI implementations do not implement the full functionality of the Session layer. When a Session layer connection is established, most of the Session layer features can be disabled through option negotiation. Implementations not requiring all of the features of this layer ignore these features when requesting a connection and do not grant these features when requested by another Session layer entity.

1.3.4. Transport Layer

The main service provided by the Transport layer, layer four, is to provide a means for Session layer entities to exchange data. The Transport layer can provide both connection-oriented and/or connectionless service. The selection of which type of Transport layer is used may be dictated by the type of service provided by the underlying Network layer.

Other services provided by the Transport layer include: the packetization and reassembly of messages, the detection and correction of errors which occur at the Network layer or below, and multiplexing transport connections onto network connections.

As data flows from the Session layer to the Transport layer it is segmented into packets. The packets are padded with sequencing information. Error detection information, usually a

checksum or Cyclic Redundancy Check (CRC), may be attached to the packets. The packets are passed to the Network layer. If the Transport layer supports flow control, a sender may have to suspend passing packets to the Network layer over a particular transport connection until an enabling signal from the receiver permits transmission to resume.

When a packet is passed from the Network layer to the Transport layer, it is checked to detect any errors in transmission. If an error occurs, the remote transport entity may be requested to resend the packet. Otherwise, the packet is placed in the appropriate place in a "receive buffer" area. The receiving transport entity must ensure that duplicate packets are discarded. If rate control is supported, the receiver may throttle the transmitter by specifying the amount of data transmitted over a given period of time.

1.3.5. Network Layer

Layer three, the Network layer, performs operations concerned with routing information between transport entities within a network environment. Two types of services are provided at this layer, virtual circuits and datagrams.

Virtual circuits require the explicit establishment of a connection between Network layer entities. During connection establishment, the full address of the destination must be given. A route is recorded as the connection is established. Subsequent packets, use the computed route. Virtual circuits have the advantage that packets arrive at the receiver in the order of original transmission. A disadvantage of this scheme is the overhead required to establish a virtual circuit.

Datagrams contain all the addressing information needed for a packet to reach its destination. They are routed by the network to the destination. Datagrams require little transmission overhead when compared to a virtual circuit, but can suffer from complications caused by receiving packets out of order.

1.3.6. Data Link Layer

The Data Link layer, layer two, provides the mechanisms necessary to exchange data between Network layer entities. This layer detects errors which occur at the Physical layer. The

Data Link layer is subdivided into sublayers, the Media Access Control (MAC) sublayer and the Logical Link Control (LLC) sublayer. This was done because it was found that the control of the media could not be separated from the physical attributes of the media.

1.3.6.1. The Logical Link Control Sublayer

The LLC layer, like the Network layer, provides both virtual circuit and datagram services. The three types of services offered by the IEEE 802.2 LLC sublayer [ISO87] are unacknowledged connectionless service, connection-oriented service, and acknowledged connectionless service.

Unacknowledged connectionless service is analogous to datagrams, while connection-oriented service is analogous to virtual circuits. Acknowledged connectionless service allows individual packets to be acknowledged without a connection establishment. This service can be useful for real-time applications since it doesn't suffer from the overhead associated with connection establishment but still allows packets to be acknowledged.

1.3.6.2. The Media Access Control Sublayer

The MAC layer defines how the communications medium is to be shared among the entities attached to it. Two types of MAC protocols used on LANs are Carrier Sense Multiple Access with Collision Detection (CSMA/CD) and token passing.

In a CSMA/CD MAC protocol, a transmitter samples the transmission medium to see if a carrier signal is available. If it is, transmission proceeds, otherwise the medium is sampled until it becomes available. If transmission proceeds and becomes garbled, as a result of two or more transmitters simultaneously trying to transmit, each transmitter must "back-off" for a random amount of time before sampling for transmission again.

In a token passing MAC protocol, a ready transmitter emits a message only after successfully capturing a token which is circulated among all transmitters on the medium. A ready transmitter, on receipt of the token, removes it from the network and begins transmitting. When the data transmission is complete, it transmits the token on the network so that another transmitter

can claim it. If a transmitter wishes to send data and does not have control of the token, it must wait until it gains control of the token.

The MAC layer defines the format of the frames transmitted over the communications medium. Fields of a frame include an addressing field, data field, error detection field, and control fields. MAC protocols usually provide three types of addressing, point-to-point addressing, multicast or group addressing, and broadcast addressing.

1.3.7. Physical Layer

The Physical layer is concerned with issues of the communication medium: the transmission of raw data over the communications medium, the mechanical properties of interconnection devices, voltage levels, etc. A number of communications media are popular today, e.g. Ethernet uses coaxial cable; token ring uses twisted pair cable; and FDDI uses fiber optic cable.

1.4. The Survivable Adaptable Fiber-optic Embedded Network Effort

The Survivable Adaptable Fiber-optic Embedded NETwork (SAFENET) committee was formed by the United States Navy to investigate applying LAN concepts to tactical systems. The committee is selecting standards at all levels of the ISO/OSI model. In addition, the special requirements of the Navy's tactical environment are being considered.

The SAFENET protocol [SAFENET87] calls out existing standards whenever possible and specifies new protocols when needed. The Manufacturing Automation Protocol/Technical Office Protocol (MAP/TOP) [MAP87] effort is a pioneer in this approach. By using existing standards, the protocol development and testing time is reduced. Since new protocols are developed to accommodate applications for which existing protocols are not appropriate, these applications are not forced into using the unsuitable protocols defined by the standard. Protocols which meet the application's requirements are developed.

The SAFENET effort is divided into three main committees: the Physical Media Subgroup, the Communications Services Subgroup, and the User Services Subgroup. In addition, a Communication Management Subgroup has recently been formed. The Physical Media Subgroup is concerned with issues at the Physical layer. The Communications Services Agent Subgroup is concerned with issues at the Data Link, Network, and Transport layers. The User/Application Interface Subgroup is concerned with issues at the Session, Presentation, and Application layers. Finally, the Communication Management Subgroup deals with network management issues.

The work being done by the Communications Services Subgroup is directly applicable to this research. One topic of study by the committee is the selection of a Transport layer protocol. A standard transport protocol is needed for interoperability, and a "real-time" transport protocol is needed for time critical applications.

TP/4 is the transport protocol used for interoperability. Applications such as port to ship communications, North Atlantic Treaty Organization (NATO) exercises, and applications not requiring real-time response use this protocol. TP/4 is gaining wide acceptance as a transport protocol. It is the Transport layer protocol selected for use in the Government Open Systems Interconnect Profile (GOSIP). TP/4 is replacing TCP as the Transport protocol used for the ARPANET.

The eXpress Transfer Protocol (XTP) is selected for detailed study as a candidate for the real-time transport protocol used in SAFENET. The XTP protocol provides both Transport and Network layer functionality. This combined layer architecture is referred to as a transfer layer.

The Logical Link Control (LLC) layer of the SAFENET architecture is used for protocol multiplexing between the non-real-time TP/4 stack and the real-time XTP stack, as show in Figure 5.

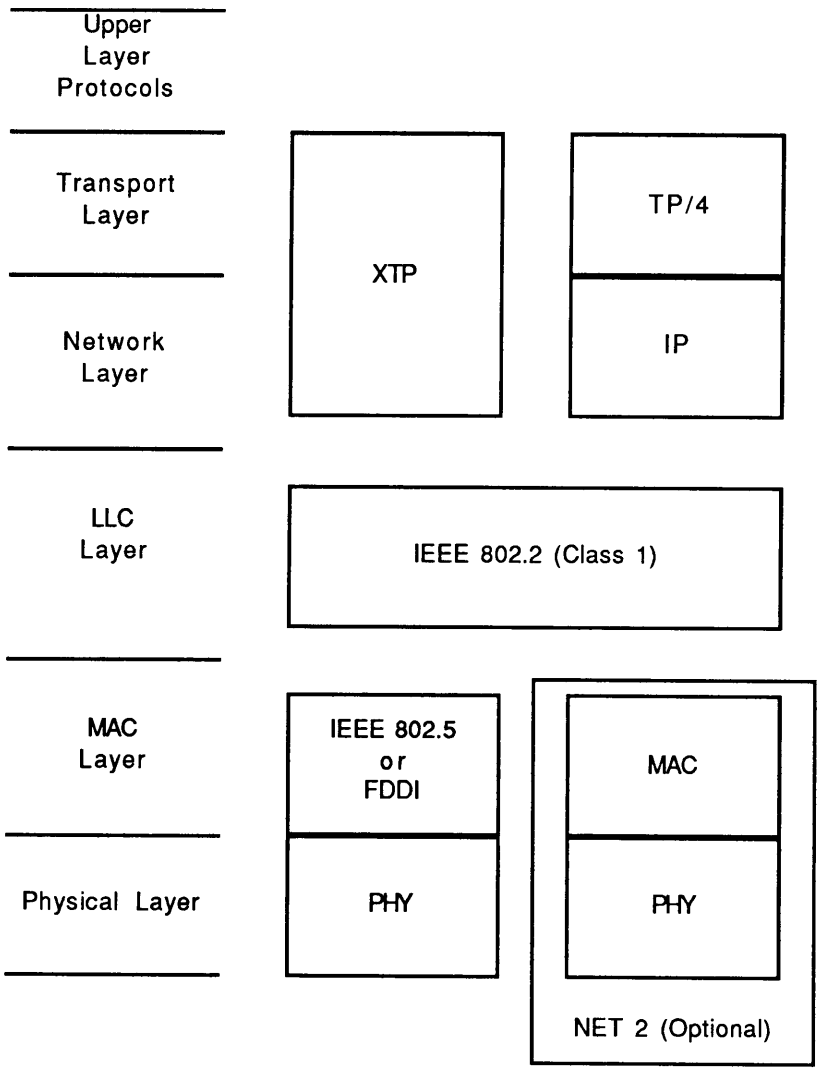


Figure 5. The SAFENET Protocol Architecture

1.5. The Express Transfer Protocol

XTP is a new protocol being developed by Greg Chesson, the chief scientist at Silicon Graphics. It is designed to provide much better performance than existing reliable transport protocols such as TCP or TP/4. By reducing the amount of control information which must be transmitted between nodes in order to ensure reliable data transfers, XTP may reduce the latencies inherent in current non-real-time protocols.

XTP is a full function transport protocol designed specifically to support implementation in Very Large Scale Integration (VLSI) circuits. The first silicon implementation of XTP is being realized in the Protocol Engine project [CHESSON87] by Chesson and a team from Protocol Engines, Inc. (PEI). The Protocol Engine is designed to provide real-time communications at the 100 Mbit/sec speeds required by Fiber Distributed Data Interface (FDDI) networks.

1.6. The Information Transfer Architectures Project

The Information Transfer Architectures (ITA) project is an effort at the Naval Surface Warfare Center (NSWC). The project is under the leadership of David T. Marlow, the chairman of the middle layer protocol committee of SAFENET. The ITA project is investigating SAFENET concepts to demonstrate their validity. One area of investigation is building a LAN testbed which conforms to the SAFENET standards. The testbed is intended to test and verify the SAFENET standards to ensure that they can be implemented and are unambiguous, non-proprietary, yet provide the functionality required by SAFENET.

The LAN testbed consists of SAFENET-compatible communications controllers which interface to UYK-44 computers. SAFENET protocol stacks are implemented on top of these controllers. The two transport protocols called out in the SAFENET specification, TP/4 and XTP, are being implemented as part of this effort. The implementation of the XTP is of special interest to the ITA project, the SAFENET committee, and to the Navy.

Since the XTP is not yet an internationally accepted and proven standard, as TP/4 is, the SAFENET committee has declared that the XTP Protocol Definition must be given a detailed study before it can become part of the SAFENET specifications. The three goals of this study are:

(1) to verify that a valid XTP implementation can be built from the XTP Protocol Definition,

(2) to show that there are no proprietary aspects of the protocol, and

(3) to show that valid implementations can meaningfully communicate.

This work is the subject of this thesis.

1.7. The Evaluation Process

Two evaluative approaches comprise the XTP study: detailed reviews of the XTP Protocol Definitions [PEIX.Y], and an implementation of XTP from the XTP Protocol Definition. Each approach serves to mutually confirm the results of the other.

The detailed reviews provide feedback to the protocol designer, who uses the comments to clarify the XTP Protocol Definition. The implementation process uncovers intricate problems in the XTP Protocol Definition not revealed in the detailed review process. These detailed points are highlighted for examination in a subsequent detailed review.

A number of revisions of the XTP Protocol Definition released during the course of this study support the work, specifically revisions 2.0, 3.1, 3.2, and 3.25. Each is studied in detail and thoroughly reviewed in subsequent sections.

1.7.1. Detailed Reviews of the Express Transfer Protocol Definition

The goals of the XTP Protocol Definition Reviews are to verify lack of completeness, perceived ambiguities, areas requiring added detail, and to locate superfluous or unnecessary functionality. SAFENET must ensure XTP has the mandatory real-time capabilities, and is reliable, completely non-proprietary, and can be independently implemented.

Each of the detailed reviews are formally commented and delivered to the protocol designer and PEI. The reviews are part of the PEI document registry and are presented in the Appendices A, B, C, and D.

Questions which impede the progress of the XTP implementation after a review has been completed and delivered to PEI are sent to Chesson via electronic mail. A log of the message exchanges to date is presented in Appendix F.

1.7.2. Implementation of the Express Transfer Protocol

An attempt is being made to independently build a working XTP implementation. The implementation differs from others in that the XTP Protocol Definition is used to produce an original "C" source code implementation. Others are porting a "C" source code implementation which is written by PEI to their systems. Many other implementations are targeted to UNIX-based systems, but the author's implementation is targeted for the NSWC LAN testbed. The NSWC testbed runs a real-time operating system called VRTX.

The goals of the implementation are to verify that:

(1) a working XTP implementation can be built from the XTP Protocol Definition,

(2) XTP can operate using existing LAN chipsets (i.e. FDDI, 802.5, and Ethernet),

(3) a software only implementation of XTP can be built, and

(4) XTP nodes can interoperate.

Finally, this effort should provide detailed notes on implementing XTP for a real-time system.

1.8. A New Approach to Protocol Evaluation

The protocol evaluation process outlined in Section 1.7. represents a new approach in protocol evaluation. This approach combines a formal Protocol Definition review process with implementation of selected parts of the protocol. No references describing such a technique have surfaced during the course of the research. This technique is of value for any protocol which is still in the process of being developed. The approach defined is especially valuable in the incremental development of a protocol.

The XTP protocol evaluation is done is two phases. In the first phase, protocol definitions are given formal reviews in an attempt to locate problems in them prohibiting implementation of XTP features. The reviews generally focus on problems which deal with a single aspect of the protocol. In the second phase, parts of the protocol which have passed the first phase, and on

the surface appear complete, are implemented. In this phase, much more complex issues are addressed ,after stressing the integration of protocol functions.

A new protocol evaluation methodology is needed for an XTP evaluation because of the state of development of the XTP Protocol. Since the XTP Protocol Definition is incomplete, formal protocol evaluation techniques are not applicable. "Well-established" transport protocols, such as TP/4 or TCP, are typically evaluated using protocol verification techniques, simulation, or complete implementation to draw conclusions about them. These techniques are not yet appropriate for the XTP protocol.

Protocol verification techniques rely on a standard reference (i.e. a protocol definition), against which all implementations of the reference are judged. Simulations are typically used to measure the performance of the protocol under test. Protocol performance is not a prime issue under consideration in achieving the goal of this thesis: to determine whether there are gaps (i.e. ambiguous descriptions, missing elements, etc.) in the XTP Protocol Definition. Complete implementation of a verified protocol would be accelerated with this partial implementation evaluation.

Chapter 2

2. Background Information

In order to understand the rationale for XTP implementation decisions, it is necessary to understand the environment in which the XTP implementation is built. This chapter is intended to provide the needed background information.

2.1. The Information Transfer Architectures Node Hardware

An ITA node is a complete communications subsystem that allows a host computer, referred to as a CPU in this thesis, to interconnect and communicate on a SAFENET-compatible LAN. The ITA node hardware consists of a Data Bus Controller (DBC), Data Bus Adapters (DBAs), a Central Processing Unit, a Central Services Module (CSM), and a COMMunications STORage (COMMSTORE) module interconnected using a Multibus-II backplane as shown in Figure 6.

2.1.1. Multibus-II

A Multibus-II backplane is used to interconnect all of the modules in the ITA node. Multibus-II is selected over other backplane buses because of its advanced message passing capabilities and its VLSI interface solution: the Message Passing Coprocessor (MPC).

Multibus-II supports Transport-layer functionality at the backplane bus level implemented in a VLSI bus interface integrated circuit (IC). Both unsolicited messages (datagrams) and solicited messages (virtual circuits) are available as methods of communications on the Multibus-II backplane. Rate-based flow control and credit-based flow control are available.

Two constituent buses provided by the Multibus-II are the Parallel System Bus (PSB) and the Local Bus eXtension (LBX). The LBX is generally used for intermodule communications and data transfer. Message passing is done over the PSB with a throughput of 40 Mbytes/sec. The LBX is generally used for high speed communications (i.e. Direct Memory Access (DMA) from a

processor to memory). Message passing is not allowed over the LBX. Its throughput is 48 Mbytes/sec.

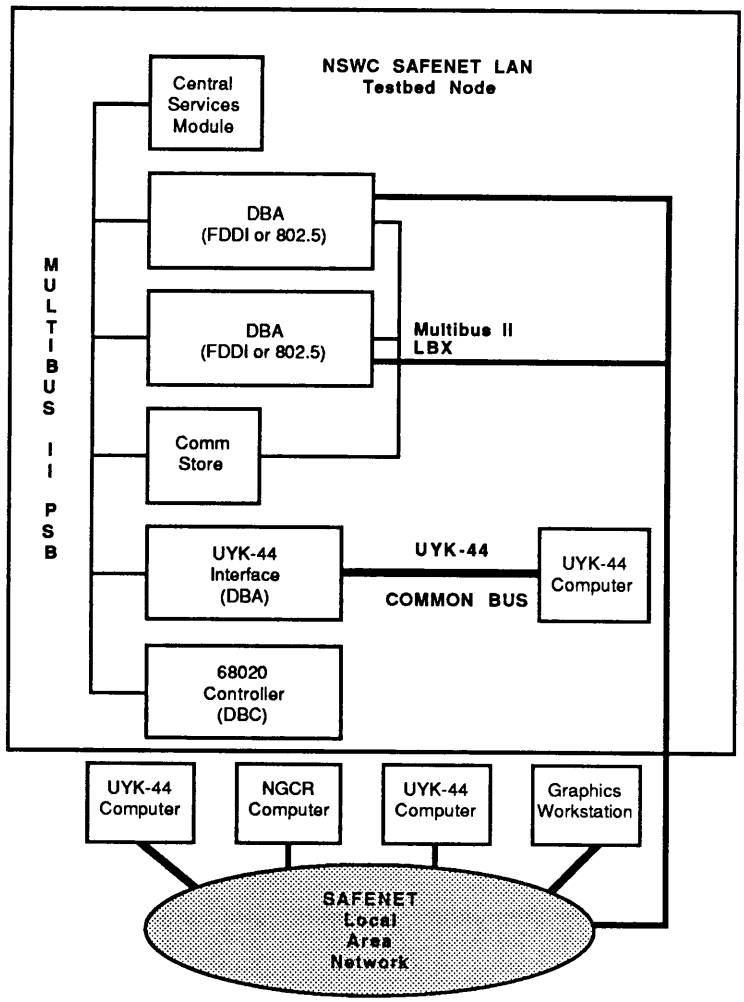


Figure 6. An ITA Node and Network

2.1.2. The Data Bus Controller

The DBC is an autonomous processor whose purpose is to relieve much of the burden of LAN communications from the CPU. It serves as the interface between the CPU and the network. The main functions performed by the DBC are executing the middle layer protocol functions (i.e.

Transport, Network, and some Data Link layer functions), managing the COMMSTORE, and controlling the LAN hardware in failure modes such as node reconfiguration.

Presently the basic DBC is a Heurikon M220/68020 controller board (M220). The M220 board is configured with a 20 Mhz 68020 microprocessor, 512 Kbytes of Read Only Memory (ROM), 1 Mbyte of Random Access Memory (RAM), 2 RS-232 serial ports, a Serial Bus eXtension (SBX) bus interface, a Small Computer Systems Interface (SCSI) bus interface, multiple timers, and an MPC.

The DBC can logically be composed of multiple processors, but its functionality from the CPU perspective remains the same. Since the CPU interface and the DBA interface each have a 68020 microprocessor with RAM, ROM, an MPC, timers, and RS-232 ports, the DBC functionality could be shared between these processors; however this sharing is not to be discussed further.

2.1.3. The Data Bus Adapter

The DBA serves as the PHY/MAC interface for the ITA node. VLSI network controllers implement much of this interface. Two DBAs are used in each ITA node to implement redundancy. Currently three types of DBAs are used in the ITA node: Ethernet, IEEE 802.5, and FDDI.

The Ethernet DBA uses the iSBX586 Ethernet SBX module manufactured by the Intel Corporation. The iSBX586 module is controlled by the 82586 Ethernet chipset also manufactured by Intel. The module is configured as a Multibus SBX module which is piggy-backed to the DBC. The module has 16 Kbytes of onboard private RAM for MAC communications storage. Communications with the DBC are done over the DBC's local SBX bus. Communications with COMMSTORE must be done through the DBC.

The IEEE 802.5 DBA uses the TMS380 802.5 chipset manufactured by Texas Instruments, Incorporated. The module is configured as a standalone Multibus-II board. It has 32 Kbytes of onboard private RAM used by the MAC for communications storage. Communications with the DBC are done using the PSB. Communications with COMMSTORE can be done over either the PSB or the LBX.

The FDDI DBA uses the AMD SuperNet FDDI chipset which is manufactured by the AMD Corporation. The module is configured as a standalone Multibus-II board. It uses 256 Kbytes of onboard private RAM for MAC for communications storage. Communications with the DBC are done using the PSB. Communications with COMMSTORE can be done over either the PSB or the LBX.

2.1.4. The Central Processing Unit

The CPU serves as the computational element for application tasks wishing to communicate over the LAN. A custom interface to the ITA node is required for each type of CPU, consequently the CPU can be virtually any type of computer or processor. Currently interfaces are being built for UYK-44 computers and Heurikon M220 boards.

2.1.5. The Central Services Module

The CSM serves as the master Multibus-II backplane bus management entity for the ITA node. During node initialization, it assigns slot numbers to each board in the node. After initialization, it detects errors which occur on the bus and collects bus statistics. All Multibus-II backplanes require one CSM to be connected to the bus. Although the CSM is implemented as a Multibus-II module, it has recently been incorporated as an MPC function in a new version of that component.

2.1.6. Communications Storage

The COMMSTORE is a 4-Mbyte global memory which is shared by all modules in the ITA node and is managed by the DBC. Because of the ITA node's tightly coupled microprocessor architecture, shared data structures, generally used for interprocessor communication (i.e. host to DBC, DBC to host, DBA to DBC, etc.), are stored in this memory.

2.2. The Information Transfer Architectures Node Software

Over the years, the ITA effort has represented a large software investment in designing and implementing an operating system kernel called DBOS [DBOS84]. The kernel consists of scheduler and a set of operating system calls which provided the minimum functionality required for the communications experiments run on the ITA testbed. DBOS is designed to keep the kernel processing overhead to a minimum and to allow interrupts to be serviced as often as possible. A unique feature of DBOS is that the only critical section in the kernel is just 2 assembly language instructions long. DBOS is the conception of Mark Powers and David T. Marlow of NSWC.

DBOS admirably achieves its design goals, but suffers from some unforeseen weaknesses. As microprocessor technology progresses, DBOS does not. Being written in 68000 assembly language, DBOS is not portable to other microprocessors (except for possibly the 680xx family) and lacks many of the kernel features needed by the applications software (i.e. semaphores, intertask communication mechanisms, etc.). Finally, DBOS is inflexible. A major effort is required to add an application task.

Based on the consideration of several alternatives, an "off-the-shelf" kernel, called VRTX/32, was procured to serve as the kernel around which a new DBOS operating system can be built. The kernel must meet the design goals of DBOS, yet not suffer from its weaknesses.

VRTX32 is a high-performance real-time executive specifically designed for 32-bit microprocessors. It provides a number of mandatory DBOS features: multitasking support, event-driven priority-based scheduling, intertask communication and synchronization, dynamic memory allocation, real-time clock control, character Input/Output (I/O) support, and real-time responsiveness. VRTX32 is implemented as a silicon software component.

> "A silicon software component is a executable version of a microprocessor program that operates on all board-level microcomputers using the same type of microprocessor. Because a silicon software component does not have to be modified to make it work, it can be delivered in Read-Only Memory (ROM)" [1].

VRTX is currently available on over 30 microprocessors and is ported to nearly every major microprocessor available.

Another silicon component, RTSCOPE, is selected to work in concert with VRTX/32. RTSCOPE provides a system monitor function to VRTX/32, allowing the user to exercise control of the underlying microprocessor. Functions such as setting breakpoints, single stepping instructions, displaying microprocessor registers, modifying memory, displaying operating system data structures, etc. are provided by RTSCOPE.

The ITA node uses the protocol layering approach defined in SAFENET. Each layer is implemented as a collection of related tasks and modules. Five layers of the SAFENET architecture are selected for initial implementation: the Physical layer, the Data Link layer, the Network layer, the Transport layer, and the Application layer. The Session layer and the Presentation layer are not included in the initial implementation.

Applications software communicates directly with the Transport layer though the Applications layer. The Application layer decides whether to use the real-time or interoperability transport stack when transmitting. TP/4 is selected for interoperability while XTP is selected for real-time communications. A minimal IP subset is used as the Network layer for TP/4. Since XTP encompasses the functionality of both the Transport layer and the Network layer, XTP also performs the functions of the Network layer in the "real-time" stack. IEEE 802.2 LLC type 1 (unacknowledged datagrams) is implemented at the LLC layer for both protocol stacks. When packets are received, the LLC must select which transport stack is to receive them. The LLC interfaces with the MAC drivers to control the network chipsets.

2.3. The Express Transfer Protocol Test Node

The ITA node presented in the previous section is simplified for experimentation purposes. Simplifications are made in both the hardware and software design of the ITA node. The purpose of the simplifications is to reduce the scope of the implementation effort to manageable proportions.

2.3.1. The Express Transfer Protocol Test Node Hardware

The XTP test node consists of four boards, the CSM, the DBC, the DBA, and the COMMSTORE module. As in the ITA node, the boards are interconnected using Multibus-II. An XTP test node is shown in Figure 7. The DBA in the XTP test node is a single Heurikon M220 board. The M220 board also serves as the CPU for applications software. DBC and CPU functions are partitioned into separate processing tasks. The DBA is the iSBX586 Ethernet module. This DBA is used because the interoperability tests to be conducted with the PEI Reference Implementation require an Ethernet interface.

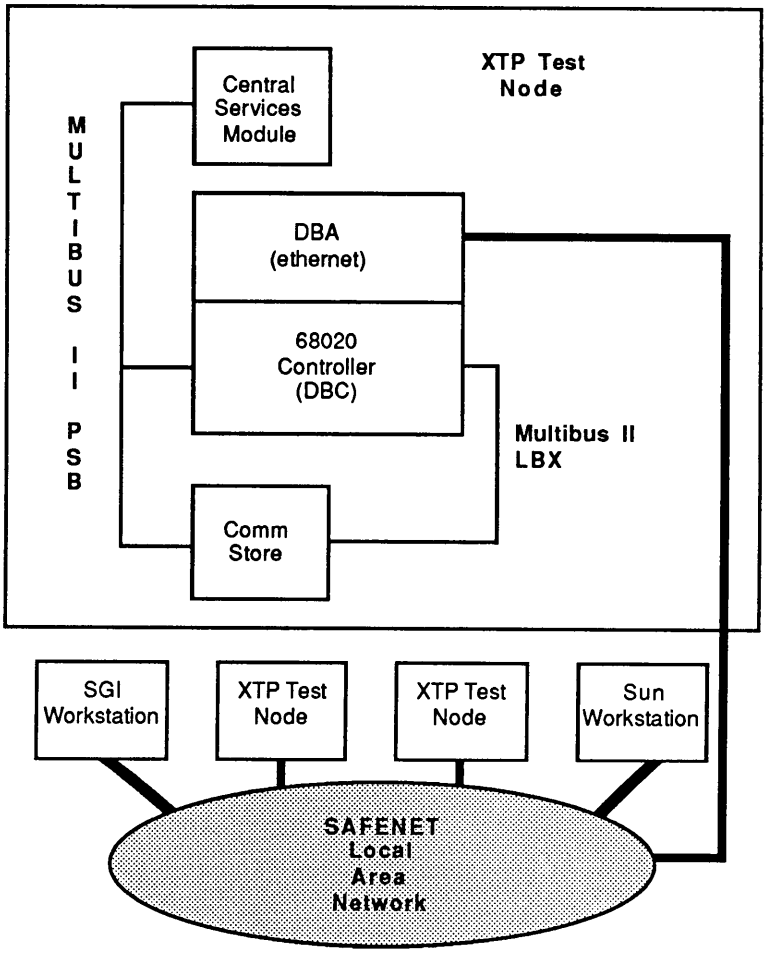


Figure 7. An XTP Test Node and Network

2.3.2. The Express Transfer Protocol Test Node Software

The XTP test node software implementation does not differ in philosophy from the ITA node implementation. The scope of the implementation is reduced, however.

As in the ITA node implementation, four layers of the SAFENET model are implemented. Applications communicate to the Transport layer through the Application layer. XTP is the only Transport layer implemented. XTP interfaces directly with an IEEE 802.2 LLC type 1 implementation being developed by David T. Marlow of NSWC. MAC drivers are written which interface directly to the LLC implementation.

2.3.3. Interoperability Testing

A port of the XTP Reference Implementation is to be made available to NSWC for interoperability testing. An NSWC owned Silicon Graphics workstation is used to run the Reference Implementation. The workstation is equipped with an Ethernet interface. The testing is modeled after the TCP/IP bakeoff procedures [RFC1025].

Chapter 3.

3. An Initial Evaluation of the Express Transfer Protocol

To limit the scope of the XTP implementation, the XTP protocol reviews are studied to determine which features of XTP are deserving of implementation. The protocol reviews reveal that some areas of the XTP protocol are not immediately applicable to the Navy's needs. Consequently, they are excluded from the XTP implementation. Other areas of the protocol are not sufficiently defined and also are not being included in the implementation. One area not adequately defined is reliable multicast operations.

3.1. Reliable Multicast Operations

Problems with reliable multicast operations are pointed out in the 2.0 Protocol Review (section 1.1), the 3.1 Protocol Review (Section 1.10), and in conversations with Chesson. One of the main areas of concern deals with how the protocol performs under degraded network operating conditions.

Real-time systems must perform adequately when conditions are at their worst. In a tactical situation, this usually corresponds to battle conditions. System performance under normal operating conditions may have to suffer so that schedules can be met under worst-case conditions. The analysis of the XTP reliable multicast protocol reveals that it does not meet the real-time requirements mandated by the Navy. In fact, the protocol performance is found to degrade dramatically under failure conditions.

When a packet is received out of sequence by a multicast receiver, it sends a reject message to the transmitter of the packet. Generally, an out of sequence packet received by one multicast receiver, is received by many others. If all receivers send reject packets to the transmitter, an avalanche effect occurs which degrades system performance. Furthermore, the transmitter is bogged down trying to process the received reject packets. In an effort to overcome this problem, a reject packet dequeueing operation is introduced in XTP.

The XTP Protocol Definition - Revision 3.25 states "Any REJ message queued for transmission by a multicast receiver must be dequeued if a REJ message arrives containing an x.r.rseq number less than or equal to (the) queued value for x.s.rseq"[2]. (REJ refers to a reject message multicast to all entities using a particular multicast address. All entities receive the REJ packet. This includes both the original multicast transmitter, and the multicast receivers. The reference to x.r.rseq is the highest sequence number received in a packet from the multicast transmitter by the multicast receiver while x.s.rseq refers to the sequence number of the REJ message queued by the multicast receiver.)

A multicast receiver monitors incoming messages using its multicast address. If it receives a REJ message with a sequence number less than or equal to one currently queued for transmission, it dequeues and discards its REJ message. Although this technique can potentially reduce REJ traffic overhead, it is not easily implemented in a SAFENET system. Initially, SAFENET implementations of XTP are being accomplished in software. These implementations are not able to process packets at the media rate as the Protocol Engine claims to do. If an XTP implementation must queue received packets before they are processed, the REJ dequeueing operation fails.

Suppose, for example, that reliable multicast messages are being exchanged on a homogeneous SAFENET network. Homogeneous configurations are common in NTDS because of the use of Navy Standard Computers. Furthermore, suppose that a multicast packet with sequence number X is lost on the network. The fact that X is missing is not detected by any of the multicast receivers until the next multicast packet, X+1, is sent. When X+1 is sent, it is received by all of the multicast receivers at approximately the same time. (The short separation between reception by multicast receivers is especially true in the 100 Mbit/sec FDDI network specified in SAFENET II.) The software overhead required to process the packet, and formulate and queue a REJ packet is virtually the same for all multicast receivers due to their homogeneity. Consequently, they pass their REJ messages to the LLC layer at approximately the same time. When the first of the REJ messages is received by a multicast receiver, it has already passed its duplicate REJ message to the LLC layer. Since messages can not be dequeued from the LLC layer by the XTP layer, the unwanted REJ message is sent anyway.

LLC packets can not be easily dequeued at the XTP layer. In both software and hardware implementations of this layer, no services are defined for dequeueing packets. In a specialized software implementation, the services provided by the LLC layer can be bypassed so that XTP has direct access to the LLC data structures. But this violates the layering concepts of ISO and should be avoided. Hardware implementations do not allow access to their internal registers and data structures, so the LLC services must be used directly.

3.2. Gateway Operations

Although gateway operations are of interest to the Navy, they are not selected for implementation. Gateways are not part of the initial ITA testbed implementation, but will be a part of future work. A cursory examination of the gateway protocols reveal that they are immature compared to other parts of the XTP Protocol and are not to be implemented until they reach a comparable level of maturity.

3.3. Priority Mechanisms

In order to meet schedules, operating systems order task execution using priority mechanisms. Many real-time operating systems use static priorities because of their deterministic nature. Dynamic priority mechanisms based on time-value functions are also used to schedule tasks to meet the deadlines required by such systems. The scheduling of communications resources and the ordering of message transmission and reception can also be based on the static or dynamic priority mechanisms. In order to use these mechanisms, however, the communications subsystem must be able to meaningfully interpret and act on the priorities passed to it. This is especially true of the XTP.

XTP encompasses the functionality of two critical communication layers, the Network layer and the Transport layer. If priorities can not be passed to and operated on by the XTP, they are meaningless. XTP must provide the mechanisms necessary to use priorities. One aspect of XTP where priorities are especially important is determining the order in which contexts are serviced.

A context may be thought of as a connection between communications entities. Messages are sent between the entities by referring to the context. Contexts should be serviced in their order of importance so that the most important messages are always sent first. The XTP Protocol Definition - Revision 3.2 provides several mechanisms, robin, separation, and duration, which affect the scheduling order of XTP contexts. While these mechanisms determine the scheduling order of packets to be transmitted, they do not ensure that the next packet transmitted by the sender is the "most important" packet. These mechanisms also fail to provide the receiver with a measure of the relative importance of a received packet.

A "tiger-team", including the author, has attempted to specify new mechanisms which overcome the shortcomings of those in the Revision 3.2 protocol definition. Two new mechanisms, SORT and ORDER, represent the result of the effort and XTP Protocol Definition - Revision 3.25 includes both.

Analysis of the new mechanisms, as specified in Revision 3.25, has raised questions by the author and others concerning their operation. A subsequent meeting of the members of the team has decided that the mechanisms should be reformulated. An analysis of the problems with the 3.25 mechanisms and a proposed redefinition are included in a paper which is presented in Appendix E. Consensus agreement is that further study is needed on the operation of the priority mechanisms through gateways. Because the priority mechanisms are in a state of change, they are studied but not implemented.

3.4. Checksum Algorithm

The checksum algorithm defined by XTP was studied and comments were submitted for the Revision 3.1 Protocol Definition (Section 1.2.2), the Revision 3.2 Protocol Definition (Section 1.5) and 3.25 Protocol Definition (Section 2.3).

Revision 3.1 of the protocol invokes the TCP ones-complement algorithm [TCP83] to calculate checksums for XTP. The XTP algorithm differs from the TCP algorithm by not including the header in the checksum calculation. This limits the extent of the checksum algorithm. "By including the extra psuedo-header information in the checksum, TCP protects itself from

misdelivery by the network protocol" [3]. This problem is partially corrected in Revision 3.25 as all of the header is included in the checksum. Including the entire XTP header can cause other problems, however. Certain fields in the XTP header change as a packet is routed through a network. When these fields change, the calculated checksum is no longer correct. The protocol definition must specify in detail which bits and fields are to be included in the checksum.

XTP includes the checksum at the end of a packet while TCP includes the checksum at the beginning of a packet. This gives XTP an important advantage over TCP in that checksums can be calculated "on the fly" as a packet is being transmitted. Using XTP's algorithm, the entire packet does not have to be buffered in memory, only the portion currently being examined. Consequently, checksums are easily computed in hardware. It is argued that TCP's placement of the checksum makes TCP difficult if not impossible to implement in hardware.

In an internetworked environment, the checksum algorithm should be able to detect errors caused by the fragmentation and coalescing of packets internal to a gateway or router. To do this, the checksum can be kept running over an entire message and reset at the end of a message. XTP introduced this feature in Revision 3.25, however the specification fails to specify when the checksum is started or reset.

The checksum algorithm is an important part of a protocol specification. During the TCP/IP bakeoffs [RFC1025], checksums had to be turned off so that different implementations could communicate. XTP must unambiguously specify such an algorithm if it is to be useful. Until this is done, the checksum algorithm is studied, but not implemented.

3.5. Out-of-Band Messages

XTP provides several fields to transmit out-of-band messages. In Revision 3.25, XTP defines the IESC and TESC fields. IESC allows out-of-band messages to be prefixed to the data field. TESC allows out-of-band messages to be appended to the data field of a packet. The TESC field evolved from the extension field defined in Revision 3.1.

The 3.1 Protocol Review (Section 1.2.1) notes that a variable length extension field renders XTP packets unparsable. This problem is eliminated in Revision 3.2 by making the

extension field a fixed length. The length of the field is not specified, however, which makes compatible implementations unlikely. This is stated in the 3.2 Protocol Review (Section 1.1). An observation is made that the field size must be a multiple of eight to meet the 0 mod 8 boundaries defined by XTP. The Revision 3.25 Protocol Definition fixes the size of the extension field to eight bytes (a multiple of eight) and renames it to TESC.

As pointed out in the 3.2 Protocol Review (Section 3.2) and in the 3.25 Protocol Review (Section 1.1), the buffering scheme for storing out-of-band messages must be defined. For example, should out of band data be part of the data stream passed to the transport user, or should it be stored in a separate message area? The buffering scheme is usually defined by the transport interface (i.e. a flag is returned to the user on a transport read indicating that the next N bytes read constitute an out-of-band-message).

Since XTP has yet to define a standard transport interface, the mechanisms to pass out-of-band messages to and from the user are studied but are not implemented. Supervisory out of band messages are implemented, however. These messages are needed to refuse connections and to close connections. Since they are not delivered to the user application, they do not have to be buffered.

3.6. Summary

To summarize, the XTP protocol reviews reveal five components of the protocol which do not meet the implementation requirements. These components are reliable multicast operations, gateway operations, priority mechanisms, the checksum algorithm, and out-of-band messages. Each of these components fails to meet an implementation criteria: components must be of interest to the Navy and must be sufficiently defined. Reliable multicast operations, priority mechanisms, the checksum algorithm, out-of-band messages and gateway operations are not specified to a level of detail that a valid implementation can be built.

Chapter 4.

4. The Express Transfer Protocol Implementation

The XTP implementation consists of three main parts: application processes, the XTP host, and the XTP subsystem. Figure 8 shows the implementation. Application processes are the users of the transport services defined by an XTP service definition. The XTP host provides the interface to these services through the service primitives. The XTP subsystem performs the protocol processing for the XTP, corresponding to the Protocol-Engine in Chesson's implementation.

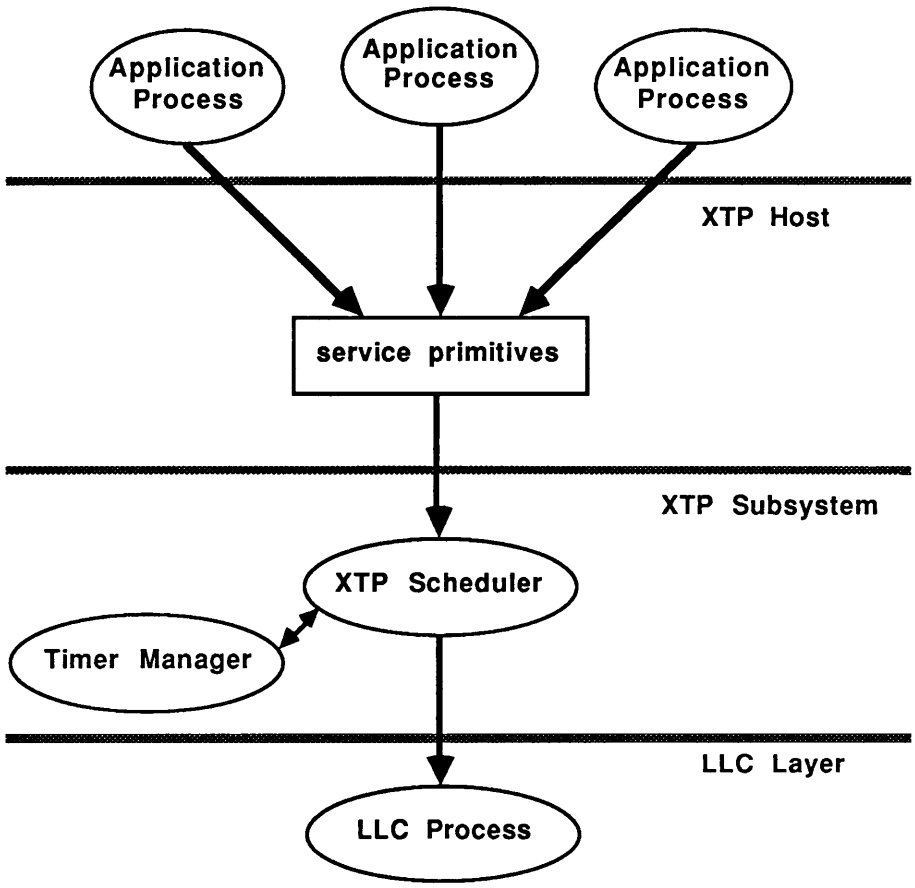


Figure 8. The XTP Implementation

4.1. The Express Transfer Protocol Host

The XTP host interacts with both application processes and the XTP subsystem. Its function is to serve as a communication liaison between these two components. Applications processes can not directly access the XTP subsystem and vice-versa. Instead, data structures, managed and manipulated by the XTP host are passed between them.

The data structures used for these communications, called control blocks, are hidden from application processes. This is done so that user applications do not rely on the details of the XTP host implementation. They only know about the services provided by the host. If the services remain unaltered, the XTP subsystem can be changed without impacting application processes. This interface helps reduce the complexity of applications wishing to use the services provided by XTP. Instead of being concerned with the intricate details of the XTP protocol, the application programmer can concentrate on using XTP services to communicate most effectively. Since the complexity of programs is reduced, applications are much simpler to write and easier to maintain.

Early in the XTP implementation, it was realized that no service definition existed for XTP. A number of people, including the author, questioned Chesson's choice of not providing a standard XTP service definition. In several PEI Technical Advisory Board (TAB) meetings, Chesson has stated that XTP is so flexible that it can support "any" service definition. While this may be true, although no one has yet to prove it, compatible implementations are not likely to be built without a service definition. Recently, Chesson has agreed to provide a standard XTP service definition. None has been released, yet, however.

The SAFENET committee has realized that a standard service definition must be formulated for XTP. Marc Cohn, a member of the Communications Services Subgroup, has written a document [COHN88] for the SAFENET committee which describes a "straw-man" set of service primitives for use with XTP. These primitives are modeled after the OSI layered service definition conventions [X.210].

The XTP host implements these primitives as a set of "C" function calls. An application passes all of the information needed to describe the requested operation to the XTP host as parameters to the service primitive function call. The XTP host gets a free control block and copies the parameters passed by the application to the appropriate fields in the control block. The host fills all fields in the control block needed to describe the requested operation to the XTP subsystem, not just those passed as parameters by the application.

As an example, consider the XTP_context.request() primitive. When an application process issues this primitive, the XTP host automatically sets the FIRST bit in the cmd[1] field of the control block. This is done to inform the destination XTP entity that this is the first packet of a connection. The fact that the FIRST bit must be sent in the initial packet of an XTP connection request is hidden from the application process as it should be.

4.2. An Express Transfer Protocol Service Definition

The service definitions used in this implementation are based on those presented in Cohn's paper. Since Cohn's primitives are based on Revision 2.0 of the Protocol Definition, they are extended to incorporate new features found in XTP. Features obsoleted by new revisions XTP are removed from the primitives.

While Cohn's "straw-man" primitives reflect the intuitive notion of the parameters necessary to effectively use the services provided by XTP, using them in building this implementation reveals that other parameters are also needed. These parameters are added to the primitives used for the implementation. The primitives presented here define the actual function calls made by an applications programmer and the parameters used by these functions. As a result of the analysis presented in Chapter 3, a subset of Cohn's primitives is selected for implementation. Specifically, multicast primitives are not included in the implementation.

Cohn's primitives are modeled after the ISO service primitives. There are four types of primitives defined in X.210: request, indication, response, and confirm. The request type

[1]In revision 3.1 of the protocol definition, the cmd field of an XTP packet contains control bits used during a connection.

primitive is used by a service-user to invoke a procedure (i.e. a procedure to establish a connection with a remote entity). The indication type is used by a service-provider to either invoke a procedure, or to indicate that a procedure by the service-user has been invoked at its service access point (i.e. a request has been made for a connection). The response type primitive is used by a service-user to complete a procedure invoked by an indication at that service access point (i.e. to inform the remote entity requesting a connection that the connection is accepted). The confirm type primitive is issued by a service-provider to complete a procedure previously invoked by a request at that service access point (i.e. the connection requested has been established with a remote entity).

The service primitives are named using the conventions defined in X.210. Three elements are used: 1) the initials which specify the layer where the primitive is used; 2) a name which describes the service provided; and 3) a name which describes the type of the primitive. Consider the name used to specify the service primitive which is used by an entity to request that a context be established with a remote machine. Here, the layer name is XTP, the service requested is context, and the type of service desired is a request, so XTP_context.request() is the service primitive used.

4.2.1. Service Primitive Overview

There are six fundamental types of service primitives defined for this implementation: context primitives, registration primitives, send primitives, datagram primitives, receive primitives, and disconnect primitives. Each fundamental primitive type is further categorized into one or more of the X.210 service primitive subtypes: request, indication, response, or confirm. A brief description of each of the service primitives is given in the subsections below. A detailed explanation of the service primitives and their parameters is presented in Appendix G.

4.2.1.1. Context Primitives

The context primitives are used to establish an XTP context between a client and a server. All four of the X.210 subtypes are defined. The request primitive is called by a client to establish a connection with a server. The indication primitive is called by a server's XTP subsystem to create a

mapping in the XTP translation map between a previously established port filter and its context. The response primitive is called by the server to generate a response packet to the client to indicate a successful or unsuccessful context establishment. The confirm primitive is called by the client's XTP subsystem to notify an application process that its context request call passed or failed.

Service Primitive	Called By	Called On
XTP_connect.request()	application	client
XTP_connect.indication()	XTP subsystem	server
XTP_connect.response()	XTP subsystem	server
XTP_connect.confirm()	XTP subsystem	client

Figure 9. XTP Connect Primitives

4.2.1.2. Registration Primitives

Only one registration primitive exists. The registration request primitive prepares a server to accept connections from clients. This is done by creating a port filter to accept incoming connections from clients.

Service Primitive	Called By	Called On
XTP_register.request()	application	server

Figure 10. XTP Registration Primitives

4.2.1.3. Send Primitives

Send primitives are used to send data and to confirm its receipt. Both a send request and a send confirm primitive are defined in this implementation. The send request initiates the transmission of data to a specified destination. The send confirm primitive is called by the XTP subsystem to indicate the completion of a send request primitive.

Service Primitive	Called By	Called On
XTP_send.request()	application	client or server
XTP_send.confirm()	XTP subsystem	client or server

Figure 11. XTP Send Primitives

4.2.1.4. Datagram Primitives

A single primitive is defined for sending datagrams. The datagram request primitive initiates the datagram transfer of data to a specified destination. Since datagrams are unacknowledged, a confirm primitive is not needed.

Service Primitive	Called By	Called On
XTP_datagram.request()	application	client or server

Figure 12. XTP Datagram Primitives

4.2.1.5. Receive Primitives

The receive primitives are used to receive data. The receive request primitive is called by an application process to define an area to store any information received on a connection. The

receive confirm is called by the XTP subsystem to indicate that the receive request operation has completed.

Service Primitive	Called By	Called On
XTP_receive.request()	application	client or server
XTP_receive.confirm()	XTP subsystem	client or server

Figure 13. XTP Receive Primitives

4.2.1.6. Disconnect Primitives

The disconnect primitives are used to close a connection. They may be called by either the XTP subsystem or an application process on clients and servers alike. A disconnect request initiates the closing of the connection while a disconnect confirm indicates that the connection is closed. A disconnect request, if used to deny a remote context request, causes a disconnect indication to be invoked by the remote XTP subsystem.

Service Primitive	Called By	Called On
XTP_disconnect.request()	application	client or server
XTP_disconnect.indication()	XTP subsystem	client
XTP_disconnect.confirm()	XTP subsystem	client or server

Figure 14. XTP Disconnect Primitives

4.2.2. Service Primitive Operation Overview

The service primitives defined above must be called in a specific sequence which is illustrated in the table shown in Figure 15. Upon initialization, all contexts are placed in the XTP primitive state called idle. When a service primitive is called, the next XTP primitive state is determined by finding the table entry corresponding to the service primitive being called (the row) and the current XTP service primitive state (the column). A number of table entries have more than one next state choice

If a connect.request is made while in the idle state, the next state is either the "connect waiting" or the "established" state. Entering the "connect waiting" state is done if a four-way handshake is desired. The advantages of this type of handshaking are pointed out in the definition of the confirm primitives in Appendix G. The "established" state is entered if the four-way handshake is not desired (i.e. time critical data exchanges).

If a send (or receive) confirm primitive is issued while in the sending (or receiving) state, the next state selected is either the established state or remains unchanged. The established state is entered if the send (or receive) confirm primitive issued acknowledges all outstanding sends (or receives). Otherwise, the next state is unchanged.

Time sequence diagrams are used to illustrate a number of typical XTP transactions in Figures 16, 17, 18, and 19. In these diagrams, the area between the two black lines represents the XTP subsystem. The areas to the left and right of the XTP subsystem represents two XTP hosts. An arrow from a service primitive to the XTP subsystem denotes the XTP host issuing the primitive while an arrow from the XTP subsystem to a service primitive denotes the XTP subsystem issuing the primitive. A dotted line through the XTP subsystem which connects two service primitives by arrows indicates that the service primitive issued by the XTP host causes the other service primitive to be invoked by the other XTP subsystem.

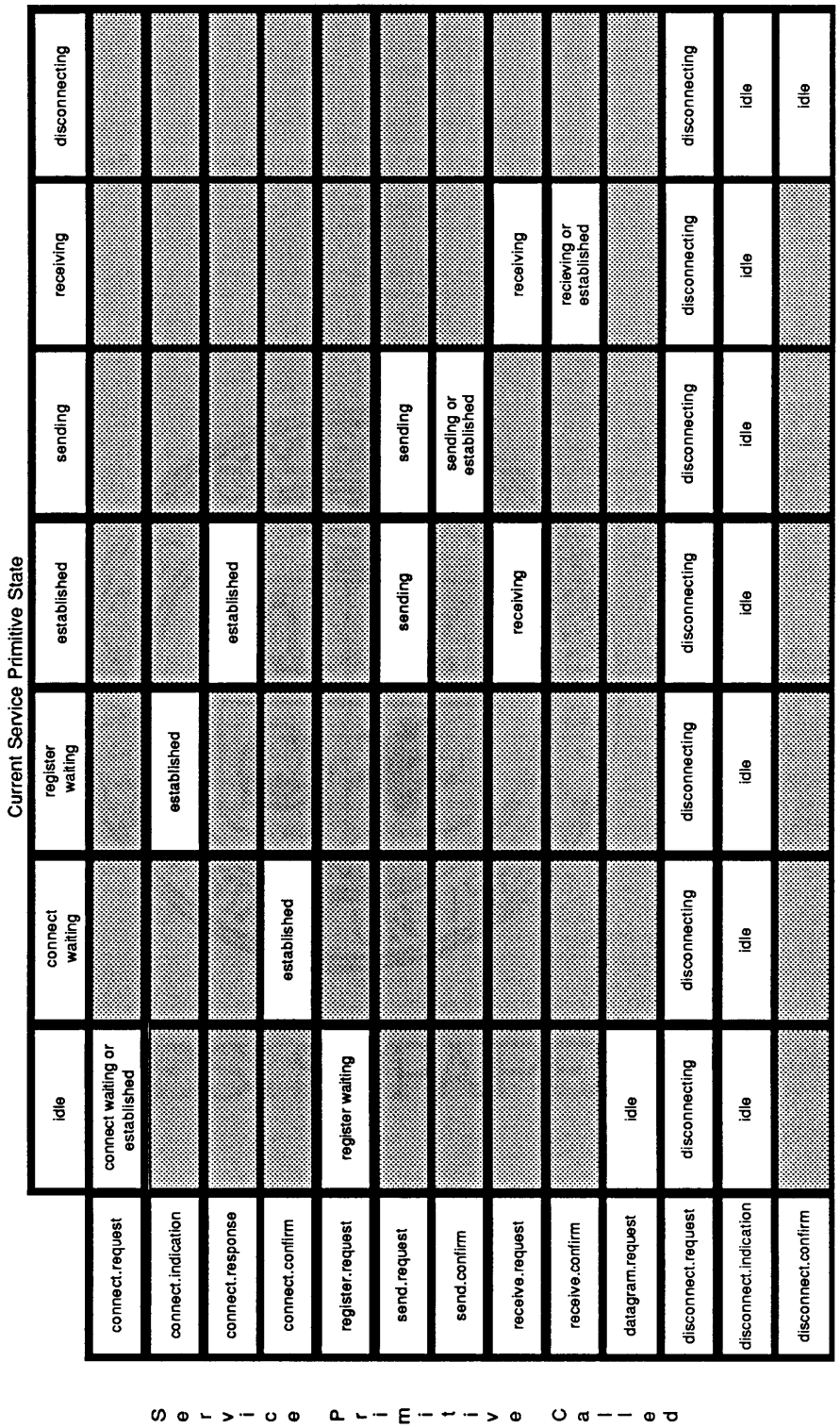


Figure 15. XTP Service Primitive State Machine

Figure 16 shows an XTP transaction using the four-way handshake. Although data may be sent as part of a context request primitive invocation, send request primitives are not honored until after a context confirm primitive is issued by the XTP subsystem.

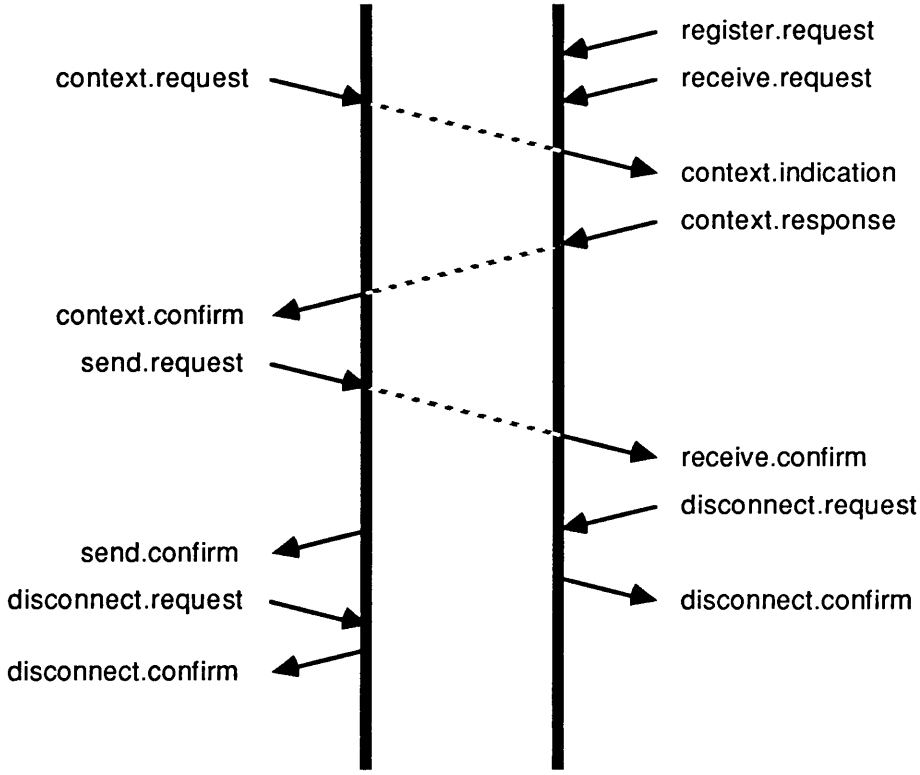


Figure 16. An XTP Transaction Using a Four-Way Handshake

Figure 17 shows an XTP transaction using XTP's fast connection facility. Using this feature, an application is able to issue send requests before a context confirm primitive is issued by the XTP subsystem.

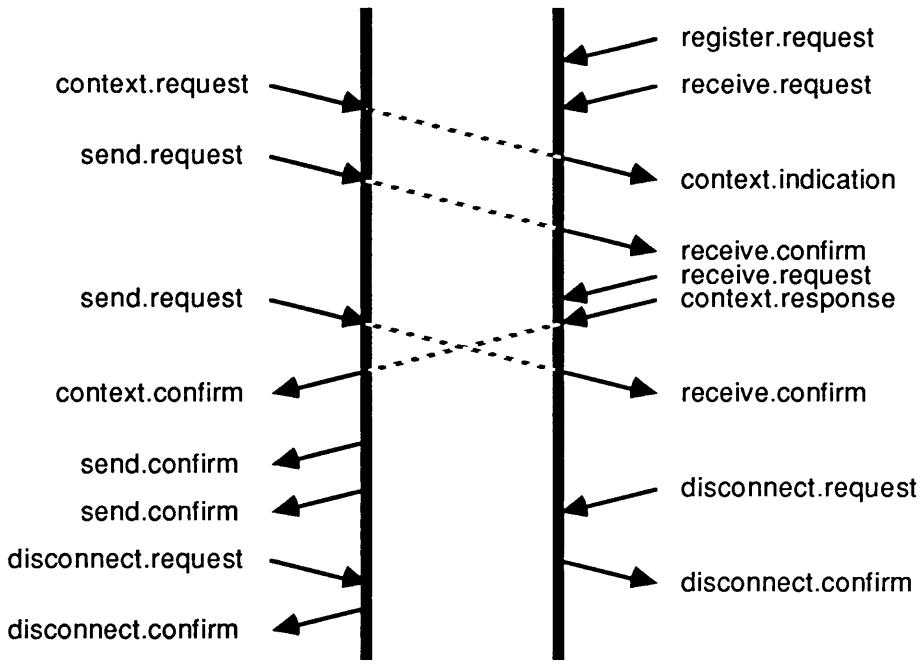


Figure 17. An XTP Transaction Using a Fast Connect

Figure 18 shows how the disconnect primitives are used to refuse a connection request. Suppose that an XTP host issues a context request and the XTP subsystem determines that there are no free contexts available. The XTP subsystem issues a disconnect indication primitive in this case to deny the context request.

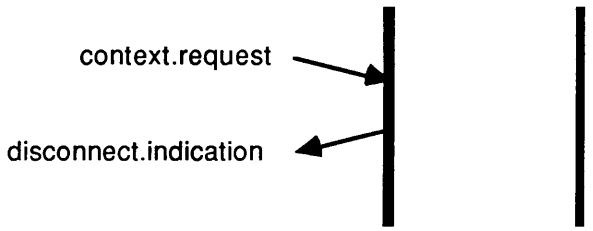


Figure 18. An XTP Subsystem Refusal of an XTP Context Request

A remote XTP subsystem may also refuse a connection request. This is illustrated in Figure 19. An XTP host issues a connect request primitive to its XTP subsystem which causes a context indication primitive to be invoked by the remote XTP subsystem. This generates a disconnect request on that system. The disconnection process is completed by a disconnect

indication being invoked by the XTP subsystem of the originating host and by a disconnect indication being invoked on the remote XTP subsystem.

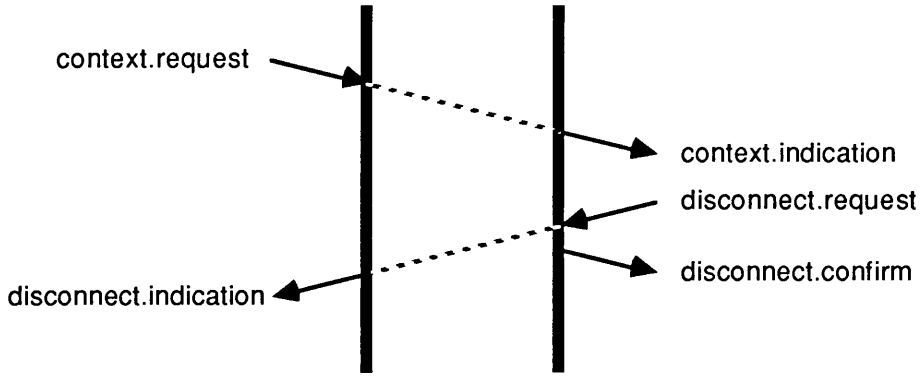


Figure 19. A Remote XTP Connection Refusal

4.2.3. Transport Protocol Data Unit Formats

Although a primitive XTP service definition is defined for this implementation, standard Transport Protocol Data Unit (TPDU) formats are not defined. This can not be done until the XTP packet format is finalized because each new release of the protocol definition changes the fields in XTP packets, their position, and their meaning. Trying to define such TPDUs before then is fruitless. Since the service definition presented here is defined independently of the PDU format definition, the transport interface in this implementation need not be changed when a standard PDU format becomes available.

A standard TPDU field describing the packet type is needed so that a correspondence can be drawn between a packet and the service primitive which caused it to be generated. TP/4 defines a TPDU code field and a number of TPDU packet types which may be placed in that field. The packet types correspond to the service primitives which caused the packets to be generated.

Some of this type of information can be inferred from XTP packets. If the FIRST bit is set in an XTP packet, one can deduce that this packet is generated by a context request primitive. If the IESC bit is set in the CMD field and the IESC field contains the REFUSED code, the packet is generated by an disconnect.request primitive. How a correspondence can be mode between

other service primitives and XTP packet formats is not clear. As can be seen, the existing correspondence is not uniform and is not straightforward.

The importance of a standard set of service primitives and a standard set of TPDUs can not be overemphasized. Interoperable implementations can not be built without such definitions. The XTP effort has chosen a bottom up approach to protocol design: first the protocol is designed and then the service primitives are built on top of the features provided by the protocol. A more top down approach would first define the services needed by applications processes and then define a protocol which can provide those services. The bottom up approach lends to protocol designs which are extremely flexible and general purpose while the top down approach leads to service definitions which provide a wide range of services with a protocol specifically designed to provide those services. It is unclear which approach is better. It is also unclear whether a bottom up design, such as XTP, might yield a protocol which can not perform the desired set of service primitives.

In this section a standard set of service primitives is presented. XTP appears to be capable of implementing this service definition effectively. The SAFENET committee should define a service definition which meets the requirements of their applications and see if XTP can perform these services. If it is determined that XTP can meet these requirements, standard TPDU formats should be defined for these primitives.

4.3. Data Structures

There are seven basic data structures defined in the XTP protocol definition: control blocks, context records, buffers, messages, packets, the translation map, and XTP tuning parameters. The Implementation of each of these data structures and the operations used to operate on them is explained below.

4.3.1. Control Blocks

A control block is a data structure which provides the command/response interface between the XTP subsystem and its host. Although control blocks are defined in the XTP

Protocol Definition, they are not part of the XTP protocol, but rather serve as a possible implementation model. The control blocks used in this implementation differ from those Chesson defines in the XTP Protocol Definition. The differences result from different implementation requirements.

A control block is created as a result of an XTP service primitive function call. The host acquires a free control block from VRTX, as described below, fills in the appropriate fields, and posts it to the XTP subsystem. Each XTP service primitive function returns a control block identifier to the application process. Functions are provided to determine when a control block operation has completed (i.e. when the service primitive function has completed). The results of the service primitive function are returned to the application in the control block. The fields of a control block may not be changed by an application process.

4.3.1.1. Control Block Format

The control blocks used in the ITA implementation are defined below:

```
struct  XTP_control_block  {
      int  status;
      int  op;
      int  mbox;
      int  xx;
      int  cmd;
      int  mode;
      int  priority;
      struct  XTP_address  *net_addr;
      DATA_TYPE  *msg_ptr;
      int  msg_len;
      int  rmsg_len;
      char  *nptr;
      char  *eptr;
      int  *next;
};
```

The status field is used to indicate who owns the control block, the application process, or the XTP layer (this includes both the host and the XTP subsystem). The value of status is initially set to CB_busy indicating that the control block is owned by the XTP layer and should not be accessed by an application process. When the control block is passed from the XTP host to the application process, state is set to CB_app.

The op field contains the command to be executed for this control block. A typical host command would be CBOP_send which tells the XTP layer to transmit the message pointed to by the control block.

The field, mbox, is a VRTX/32 mailbox which is set to empty during initialization. It provides a synchronization mechanism between the XTP host and its application process. When an application process issues a context request, the host gets a free control block and initializes it. The op field is set to CBOP_create_context. Next, the host pends on a free context record. When the context record returned, it is filled in by the host on behalf on the application process. The context record identifier, called a context id, is copied into the xx field of the control block. This is done because any XTP service primitives issued after a context request, such as send request, require a context to be specified as an input parameter.

The host pends on mbox. When the XTP subsystem creates a context in response to the service primitive, it posts the context identifier to mbox. (Since a mailbox containing zero is considered empty, a context identifier of zero is considered to be illegal). When the host is scheduled again, it retrieves the context identifier from the mailbox. The host copies the context identifier into the xx field of the control block. An application process can then retrieve the context identifier in one of two ways.

A busy wait loop on xx is issued until it changes to a non-zero value. The non-zero value is the context identifier. This is not be done in the ITA implementation because busy waits cause all tasks with priorities less than or equal to the priority of the task busy-waiting to be starved. Preferably, a task issues a cb_pend(), defined below, on the control block it issued. After the host posts the control block back to the application process, the context identifier may be accessed in the xx field of the control block.

When a packet is transmitted, the cmd field is used by the XTP host on transmission to tell the XTP which bits to set in the CMD field. The XTP subsystem copies the contents of the cmd field into each packet transmitted. When a packet is received, the XTP logically "or"s the contents of the received packet with the current value of the cmd field.

The mode field allows user defined data to be exchanged by XTP hosts. The application passes a value (i.e. connection management information) to the XTP host which copies it into the mode field of the control block. The XTP subsystem copies the mode field of a control block into the MODE field of the next XTP packet transmitted. How the MODE field in the packet is interpreted by the destination XTP is defined by the user of the field.

The priority field, like the mode field, is a user defined quantity that is delivered to a destination host in the ORDER field of a packet by the XTP subsystem. Again, the interpretation of the priority field is defined by the user of the field.

The XTP_address field is a pointer to an XTP_address structure consisting of all of the fields in a standard IP and TCP address encapsulated by an XTP address field. Type 1 XTP addresses, DARPA Internet Protocol addresses, are selected over ISO addresses so that the ITA implementation is compatible with the other XTP implementations used for interoperability testing.

On transmission, the msg_ptr field is passed from the application to the XTP host. It points to the area in memory where the message to be transmitted is located. When a message is to be received, the application passes the address at which to store the received data.

The XTP host uses the msg_len field to define the length of the current message being transmitted by the application process. The message length is passed down from the application to the XTP host which copies that value into the msg_len field. The XTP receiver uses this field to store the size of the receive buffer pointed to by msg_buf. The rmsg_len defines the length of a received message and is only valid on a receive operation.

For transmission, nptr points to the memory location where the next data to be transmitted for this control block is located. The field, called eptr, points to the end of the message contained in the current control block. Its value is calculated as follows:

```
eptr = msg_ptr + msg_len - 1;
```

For reception, nptr points to the next memory location where received data is to be stored for this control block. The field eptr is the last memory location where received data for this control block

may be stored. It is initialized as shown for transmission. The length of a received message is stored in rmsg_len before the control block is passed back to the XTP host.

The field, called next, is used to build linked lists of control blocks. It points to the next control block in a linked list. If the value of next is NULL (zero), the control block is the last in the linked list. A linked list of receive control blocks, transmit control blocks, and completed control blocks is maintained for each context.

4.3.1.2. Control Block Management

Control blocks are managed by VRTX. During XTP initialization, cb_init() builds a pool of free control blocks. The pool is created by posting the integer index of each control block to a VRTX queue called XTP_CBQ. The control block indices correspond to the array indices of the cb array, defined below:

```
struct  XTP_control_block  cb[NUM_CONTROL_BLOCKS];
```

The cb array consists of NUM_CONTROL_BLOCKS - 1 control block structures. cb[0] is an illegal control block. Its index is not be posted to XTP_CBQ during initialization. cb[0] is defined to be illegal because control block numbers are be posted to VRTX mailboxes and messages posted to VRTX mailboxes must be non-zero.

Control blocks may only be requested by the XTP host. When one is needed, the host invokes sc_qpend() on XTP_CBQ. If no control blocks are available, the host is put asleep until a control block is posted to XTP_CBQ. If a control block is available, its index , i, is returned to the host. The control block may then be accessed as the array element cb[i]. When the host is finished with a control block, it is returned to the system by posting it to XTP_CBQ using sc_qpost().

A user application requests control blocks indirectly by issuing an XTP service primitive. This is illustrated by the following example. When a user application issues a context request primitive, a control block index is returned by the call. The XTP host obtains a free control block, fills in the appropriate fields, posts the control block to the XTP subsystem, and returns the control

block index all on behalf of the application process. When the control block index is returned, the user application may continue processing (i.e. issuing more service primitives) or it may choose to pend on a previously issued control block by calling cb_pend() or cb_vpend(). These functions return after a designated control block has been completely processed by the XTP.

Both the cb_pend() and the cb_vpend() functions allow a single process to pend for a control block associated with single context. This implies that a context, in this implementation, may only be accessed by one task until it is closed by that task and reopened by another task. This restriction is placed on these functions to simplify the code required process contexts and control blocks. If the rules are violated, mutual exclusive access to the control block queues used by those functions is not guaranteed.

The function cb_post() is used to make control blocks available to the pend functions. It may only be called by the XTP subsystem (not the XTP host).

The functions cb_post(), cb_pend(), and cb_vpend() all rely on the XTP_cb_q structure shown below:

```
struct  XTP_cb_q  {
    U32  head;
    U32  tail;
    U32  sem;
    U32  post_flag;
};
```

This structure is used to build an array of XTP_cb_q structures:

```
struct  XTP_cb_q  cb_done_q[NUM_CONTEXT_RECORDS];
```

As in the cb array, there are NUM_CONTEXT_RECORDS - 1 elements in the cb_done_q array. There is an element corresponding to each legal context record. (As with control blocks, a context record with index zero is illegal in this implementation). The cb_done_q for context number five may be accessed using cb_done_q[5].

Four fields comprise the an XTP_cb_q structure; head, tail, sem, and post_flag. The head field denotes the first element in the control block queue, while tail denotes the last

element. A VRTX counting semaphore, called sem, stores the number of entries in the control block queue. The field, post_flag, is a VRTX mailbox which is posted to whenever a control block is posted to the control block queue.

Figure 20 shows the operation of cb_pend(). A user application issues a service primitive call (i.e. XTP_send.request()) which is executed by the XTP host. The host gets a free control block, fills in the necessary fields, and posts it to the XTP subsystem where it is processed. The host returns control to application which issues a cb_pend() call to the host. The host executes an sc_spend() call on the sem member of the cb_done_q structure associated with the current context. The host and consequently the application sleep until the XTP subsystem has processed a control block and issues an sc_spost() on the sem member which the host is pending on. Before the sc_spend() call returns, sem is decremented by VRTX. The XTP subsystem also issues an sc_post to the post_flag member of the cb_done_q. This is not used by cb_pend() but is used cb_vpend(), however. After the sc_spost() is executed by the XTP subsystem, the host is eventually rescheduled and returns to the application. The cb_pend() issued by the application returns when the next control block associated with the current context is processed by the XTP, not necessarily the last one issued.

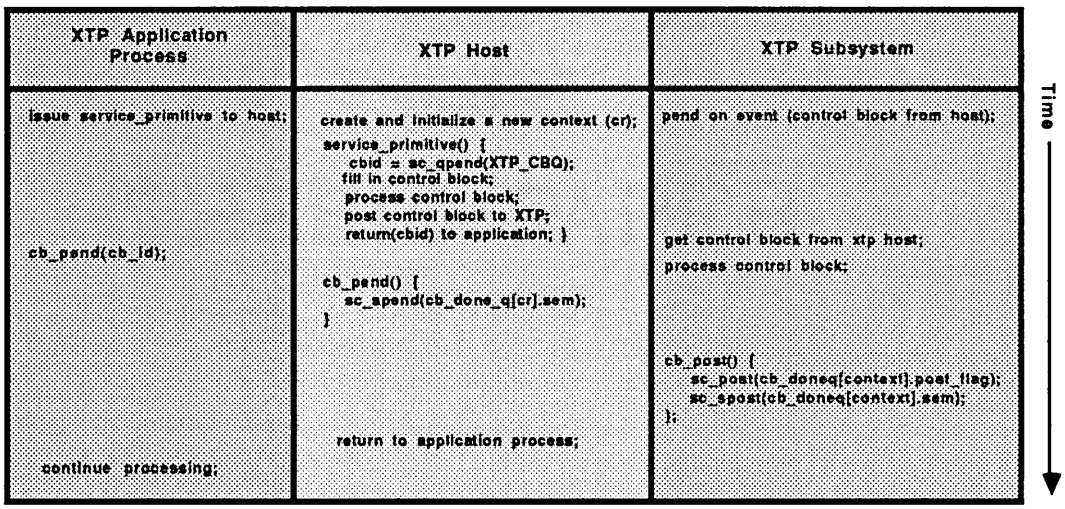


Figure 20. Time Sequence Diagram for cb_pend()

Figure 21 shows the operation of the cb_vpend(). The processing for this function is identical to Figure 20 up to the cb_vpend() call. cb_vpend() waits for a specific control block to be completed by the XTP. To do this, the application specifies which control block to wait for in the cb_vpend() call. Instead of pending on the sem member of the cb_done_q, the host does an sc_pend() on the post_flag mailbox. The post_flag indicates that a control block has been posted to the cb_done_q by the XTP subsystem. When the sc_pend() returns to the host, it indicates that there may be at least one element in the cb_done_q. The queue is searched for the desired control block. If it is found, the host removes it from the cb_done_q, sc_spend()s on sem to decrement the count of the number of entries in the queue, and returns to the application process. If the control block is not found, the host sc_pend()'s on post flag to see if any new control blocks have been posted, and repeats the above process until the control block becomes available.

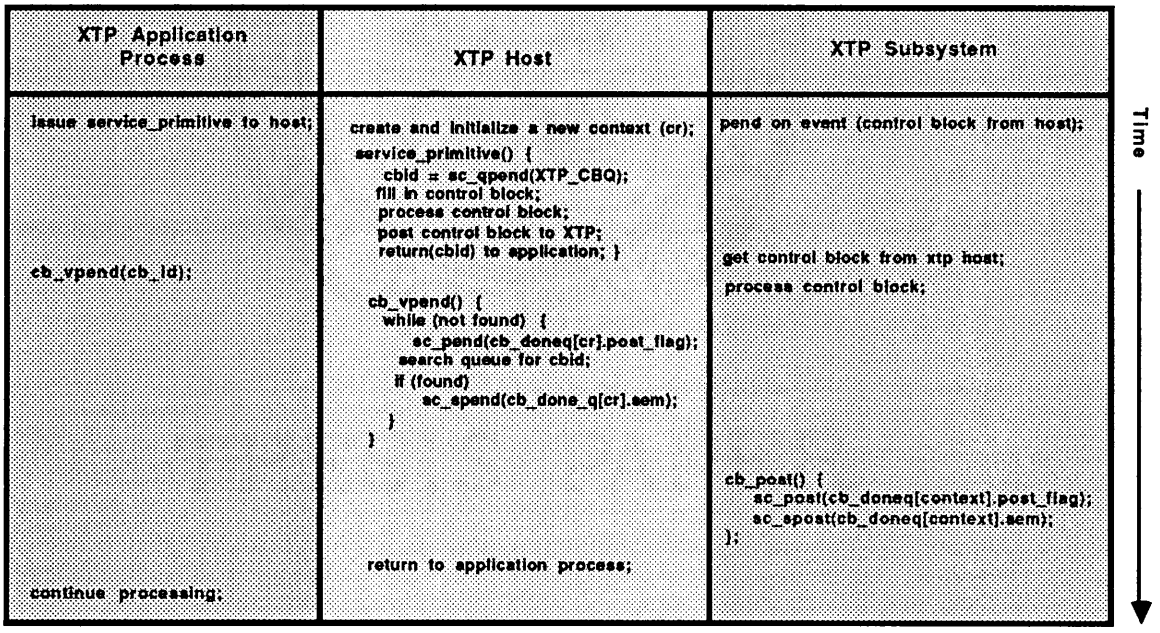


Figure 21. Time Sequence Diagram for cb_vpend()

The cb_pend() and cb_post() function call are similar to the select() call in BSD 4.X UNIX [LEFFLER89]. These calls, like select, allow multiplexing I/O on descriptors. In this implementation, descriptors are represented by linked lists of control block identifiers. Unlike

select(), the initial implementation of cb_pend() and cb_vpend() only allows event driven I/O. Non-blocking I/O is not implemented. Event driven I/O is preferred over non-blocking I/O in the ITA implementation because of the scheduling policy of VRTX. VRTX is completely event driven. If a busy wait is executed by a process, all processes of lesser priority remain blocked until the busy wait terminates, and the process sleeps. If non-blocking mechanisms are available to the user, they must be used with extreme care or process starvation will occur. Not implementing these functions eliminates the possibility of an application process starving other processes in this way.

4.3.2. Context Records

A context record maintains the state of an active connection or datagram. Context records are defined in the XTP Protocol Definition, but are not part of the XTP protocol. Like control blocks, they serve as a possible implementation model. The context records used in this implementation differ from those presented in the XTP Protocol Definition. The differences result from different implementation requirements.

4.3.2.1. Context Record Format

A context record is defined by the XTP_context_record structure as follows:

```
struct  XTP_context_record  {
    struct  XTP_cr_control  s;
    struct  XTP_cr_control  r;
    unsigned  int  s_state;
    unsigned  int  r_state;
    int  rcbq;
    int  rpbufq;
    int  rpbufqmax;
    int  tcbq;
    unsigned  int  sseq;
    unsigned  int  nseq;
    unsigned  int  eseq;
    unsigned  int  rcount;
    unsigned  int  robin;
    unsigned  int  rtrip;
    unsigned  int  ctimer;
    unsigned  int  ctimeout_flag;
    unsigned  int  wtimer;
```

```
    unsigned  int  wtimeout_count;
    unsigned  int  rtimer;
};
```

This structure contains two members which are XTP_cr_control structures. One of these structures is defined for sending data on the context and the other for receiving data. These structures correspond to the packet headers defined by the XTP. Two variables record the state of the sending and receiving portions of the context. The portion of the XTP subsystem responsible for transmitting data is referred to collectively as the transmitter. The receiver refers to the receiving portion of the XTP subsystem. The field r_state records the state of the receiver and s_state records the state of the transmitter. A typical state for a transmitter is SENDING, while a typical state for a receiver is RECEIVING. Initially, both fields are set to IDLE.

Two queues are defined in the structure to hold pending control blocks. The process_control_block() function places the control block on the appropriate queue. If data is to be sent by the control block, it posts the control block on the tcbq. The transmitter removes control blocks from this queue to service them. If the control block passes a memory block for receiving data, the function posts it to rcbq, where the receiver removes it. Another queue, called rpbufq, is used to store received pbufs passed up from the LLC layer. These packets must be queued if there are no receive buffers available. The variable rpbufqmax records the maximum number of pbufs allowed to be queued for the context. Any pbufs received after the rpbufq is full are posted back to the list of free pbufs.

Several variables in the structure are assigned to log the sequence numbers used by a transmitter. The fields sseq, nseq, and eseq refer to the sequence numbers used for sending the data described by the current control block. The first sequence number used for the current control block is stored in sseq, the sequence number of the next packet to send is stored in nseq, and the final output sequence number for the control block is stored in eseq.

Management and flow control information are also contained in the XTP_context_record structure. A count of the number of packets received since the connection has been established is maintained in rcount. This field can be used for management functions. Round-robin scheduling of contexts is implemented using the robin field which contains the number of bytes a

context is permitted to transmit until it must let all other contexts try to transmit. The field, named rtrip, is used to store the estimated round-trip time a packet takes to travel from the sender to the receiver on this connection. This time is calculated using the sync/echo protocol defined in the protocol definition.

The XTP_context_record structure also contains several timers: ctimer is used to ensure that a context has had sufficient time to close, wtimer is used to determine if the remote host has taken too long to respond to a status request packet, and rtimer is used as a rate control timer which provides the interpacket gap defined in the protocol definition. The ctimeout_flag is used to indicate that the context timer has expired and the ctimer/wtimer protocol[2] is in process. The wtimeout_count flag keeps track of the number to times wtimer has expired during this process.

The XTP_cr_control structure is defined below:

```
struct  XTP_cr_control  {
    CMD_TYPE  cmd;
    short  event;
    long  key;
    long  seq;
    long  eseq;
    long  rseq;
    long  alloc;
    long  dseq;
    long  echo;
    long  sync;
    long  duration;
    long  separation;
    long  nresend;
    long   resend[NUM_RESEND_PAIRS];
};
```

All of the fields in the structure correspond directly to fields in XTP packets. On transmission, the fields are be filled in by the XTP subsystem and then copied into the header portion of an XTP packet. On reception, the fields are copied from the received packet header to the data structure.

[2]The ctimer/wtimer protocol is defined in Section 3.3.8 of the Revision 3.25 XTP Protocol definition.

4.3.2.2. Context Record Management

Context records, like control blocks, are managed by VRTX. During XTP initialization, cr_init() builds a pool of free context records. The pool is created by posting the integer index of each context record to a VRTX queue called XTP_CRQ. The control record indices correspond to the array indices of the cr array, defined below:

```
struct  XTP_context_record  cr[NUM_CONTEXT_RECORDS];
```

The cr array consists of NUM_CONTEXT_RECORDS - 1 context record structures. References to cr[0] are not permitted as it is considered an illegal context record.

Context records may only be requested by the XTP host. When one is needed, the host invokes sc_qpend() on XTP_CRQ. If no context records are available, the host is put asleep until one is posted to XTP_CRQ. If a context record is available, its index , i, is returned to the host. The context record may then be accessed by the host as the array element cr[i]. When the host is finished with a context record, it is returned to the system by posting it to XTP_CRQ using sc_qpost().

A user application is not permitted to access a context record directly. Only the XTP host and subsystem may do this. The application does, however, use context identifiers as parameters to XTP service primitives. Context identifiers may be thought of as socket identifiers or Transport Service Access Points (TSAPs).

4.3.3. Buffers

XTP defines a buffer as "an arbitrary area of memory designated by an application program" [4]. In the ITA implementation buffers are always allocated within the COMMSTORE address space either by the buffer allocation service defined in DBOS or in an area defined by the application. COMMSTORE is used because it is directly accessible by all boards in an ITA node. DBOS uses the VRTX memory allocation system calls to manage the buffer area.

All buffers allocated by DBOS are of the same size, MAX_BUF_SIZE which is defined at compile time. Care must be taken when selecting the buffer size. If it is too large, a lot of buffer space may be wasted by messages which only partially fill buffers. If it is too small, the application process has to allocate a large number of buffers per message. Since the management of these buffers is the responsibility of the application process, it may spend an unacceptable amount of time managing these buffers. As an example, a disk server controlling a disk with a 512 byte block size might choose a buffer size of 512 bytes for an optimum buffer allocation size. The initial buffer size selected for this implementation is the maximum packet length for Ethernet, 1518 bytes.

The single buffer size memory allocation scheme is chosen over dynamic memory allocation schemes because it is deterministic. Dynamic memory allocation schemes, such as the buddy system [CALINGAERT82], are usually non-deterministic because of their garbage collection algorithms. Non-determinism can make real-time systems harder, if not impossible, to implement.

A DBOS buffer is simply an array of characters. The size of the array is determined by MAX_BUF_SIZE. An array of buffers, called buf_store, is defined to allocate all buffers available to an ITA node. The number of buffers in buf_store, called NUM_BUF, is determined at compile time. This array, along with any other data structures which must be located in COMMSTORE, are stored in a special section[3] so that they can be located in COMMSTORE by the linker/loader. When a control block is passed to the XTP, its msg_ptr field is set to the address of the buffer associated with that control block and its msg_len field is set to MAX_BUF_SIZE.

All buffers are allocated from an array called buf_store which is defined as follows:

```
char   buf_store[NUM_BUF][MAX_BUF_SIZE];
```

As implied from the definition of buf_store, a buffer is a logical data structure, not a physical data structure as it is uniquely defined by its starting address and its length. Three functions are available to operate on buffers: buf_init(), get_buf(), and free_buf().

---

[3] A section is an area of memory where a group of files, specified in the load file, are located in memory.

4.3.3.1. init_buf()

To initialize the VRTX memory partition that manages the allocation of buffers from buf_store, init_buf is called during initialization. This function may only be called once and may only be called by DBOS. If the function call fails, an error message is printed on the system console. Control is passed from DBOS to RTSCOPE. The definition of a scheme to gracefully recover from ITA node errors is beyond the scope of this thesis. This is part of future investigations done by the ITA group.

4.3.3.2. get_buf()

A call to get_buf() returns a pointer to a buffer which may be used by the application process. If no buffers are available, an error message is printed on the system console. The function does not return in this event, however, as control is passed to RTSCOPE.

4.3.3.3. free_buf()

The function, free_buf(), is called to return a buffer to the pool of free buffers managed by DBOS. The user passes a pointer to the buffer it wishes to return. (The pointer must have been returned by a call to get_buf()). If the free_buf() call fails, an error message is printed on the system console and control is passed to RTSCOPE.

4.3.3.4. Extending the Buffer Scheme

If a single buffer size proves to be too limiting, several options are available to an application. First, the it may choose to not use the DBOS memory management services. In this case, the application is responsible for allocating the memory to be used as buffers. As with DBOS, these buffers must be located in COMMSTORE. Second, the buffer management algorithms may be extended to accommodate a fixed number of buffer sizes. The number of different block sizes should be kept small, however, so that they can be managed easily.

To implement this scheme, init_buf(), get_buf(), and free_buf() can be redefined to accept another parameter, blocksize. Suppose three block sizes are needed (i.e. SMALL, MEDIUM, and LARGE). Now, init_buf() is called once with each blocksize parameter. A medium sized block can be allocated by a get_buf(MEDIUM) call. The medium sized block obtained can be freed by a free_buf(MEDIUM,buf_ptr) call.

4.3.4. Messages

A message is generated by an application process. It consists of one or more buffers. If a message can not fit in a single buffer, it is segmented into multiple buffers by the application process. The application marks the last control block of a message by setting its EOM bit.

4.3.5. Packets

XTP defines a packet to be "the data transfer unit defined by the network media."[5] This definition is extended slightly in this implementation. Packets exist at the transport, network, LLC, and MAC layers, not just at the network media. Packets grow as they are passed down through the layers and shrink as they are passed up through the layers. The maximum packet size, as in the XTP definition of packet, is defined by the network media.

Suppose an application process has a message to send. The message is segmented into buffers, if necessary, by the application process. The buffers are individually passed down to the communications subsystem for transmission. The communications subsystem (initially the Transport layer in the ITA node implementation) segments buffers into packets if necessary. (Segmentation is done if a buffer is larger than the data segment of a packet).

Packets are logically divided into three parts: a header, a trailer, and a data segment. The Transport layer fills the data segment with as much data from a buffer as it can. If the data segment of the packet is smaller than the buffer, multiple packets are built. Transport protocol header information is prepended to the beginning of the data segment, while transport protocol trailer information is appended to the end of the data segment. The relationship between messages, buffers, and packets is shown in Figure 22.

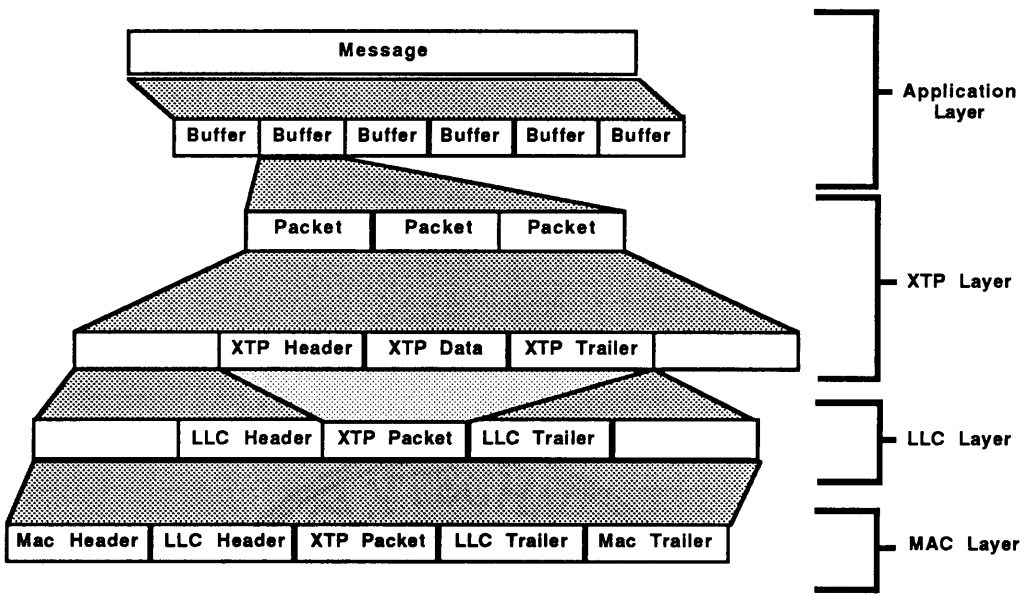


Figure 22. Memory Management Data Structures

In addition to the data to be sent, control information is exchanged between protocol layers. For example, suppose an application process on an ITA node with a SAFENET I protocol stack and an 802.5 media decides to send a message. It determines that this is an important message should be sent as quickly as possible. For this implementation, the application process can relay the importance of the message to the communications subsystem in two ways. First, it may choose to use XTP as the Transport layer by using LSAP 0x74 (the XTP LSAP). Additionally, it must be able specify that the packets which comprise the message are given high priority at the MAC layer. In an 802.5 MAC, priority is specified in priority bits which are part of the MAC frame.

A structure, called a pbuf, is defined which incorporates both the packet structure required by the protocol and the mechanisms needed to pass control information between protocol layers. Each pbuf is a composite of other data structures; one for control information for each protocol layer, one for the storage of packet data, and several data pointers.

The number of pbufs is determined by NUM_PBUF. The size of the packet data segment of a pbuf, MAX_PACKET_SIZE, is set at compile time. The MAX_PACKET_SIZE is the sum of MAX_HEADER_SIZE, the maximum header size, MAX_DATA_SIZE, the size of the largest data segment allowed in a packet, and MAX_TRAILER_SIZE, the maximum trailer size.

MAX_HEADER_SIZE and MAX_TRAILER_SIZE are computed by summing the maximum size of the header and trailer at the Transport layer and each layer below it. For example, if the Transport layer has a maximum header size of 50 bytes, the Network layer 10 bytes, the LLC layer 5 bytes, and the MAC layer 5 bytes, the MAX_HEADER_SIZE for this implementation would be 70 bytes.

A pbuf structure is defined in "C" as follows:

```
#define  MAX_PACKET_SIZE  \
(MAX_HEADER_SIZE+MAX_DATA_SIZE+MAX_TRAILER_SIZE)

struct  pbuf_struct
{     struct     user_struct  user;
      struct     transport_struct  transport;
      struct     network_struct  network;
      struct     llc_struct  llc;
      union      mac_struct  mac;
      char  packet[MAX_PACKET_SIZE];
      char  *hp;  /*  header  pointer  */
      char  *dp;  /*  data  pointer  */
      char  *tp;  /*  trailer  pointer  */
};
```

Each layer member (i.e. network_struct, llc_struct, etc.) of a pbuf structure is the mechanism used to pass control data between the protocol layers. Each of the layer components defines the information used at a given layer. When building a packet for transmission at layer X, its layer component is filled in to be used by a lower layer, Y, where X > Y. Conversely, when disassembling a received packet at layer X, its layer component may be filled in to be used by a higher layer, Y, where X < Y.

For example, the llc_struct, is defined in "C" as follows:

```
struct  llc_struct  {
      U8    ssap;
      U8    dsap;
};
```

It is usually generated by the application process and used by the LLC layer when transmitting a packet to determine which dsap and ssap to use. It may be filled in at any layer above the LLC layer. On message reception, the llc_struct might be filled in at the LLC layer and used to allow layers above it determine the ssap used by the received packet.

The packet segment of a pbuf is a data structure used to pass packets constructed at a protocol layer to the next higher or lower protocol layer. The pbuf pointers (hp, dp, and tp) are used to divide the packet segment into a header segment, data segment, and trailer segment. The initialization and operation of pbufs used to store received packets and those used to store packets to be transmitted is different. This is due to the pbuf pointer manipulation algorithms defined below.

4.3.5.1. Receive pbufs

In order to initialize a receive pbuf, the pbuf pointers are set to the appropriate values in the order shown as follows:

```
hp = &pbuf.packet;
dp = &pbuf.packet  +  sizeof(mac_layer_header);
tp = &pbuf.packet  +  sizeof(received_packet)  -
       sizeof(mac_layer_trailer);
```

The initialization is show graphically in Figure 23.

Figure 23. Receive pbuf Pointer Initialization at the Mac Layer

This initialization is done at the MAC layer since packets are received there first in an ITA node. The references to mac_layer_header and mac_layer_trailer refer the size in bytes of the header and trailer field for the current MAC layer. The initialization of tp requires knowledge of the received packet length. Since this can not be known in advance, tp may only be calculated after a packet is received. At any layer, the packet header is accessed in locations hp to dp - 1, the packet data segment in locations dp to tp - 1, and the packet trailer in locations tp to tp + sizeof(current_layer_trailer).

Before a receive pbuf is passed from layer n to layer n+1, hp, dp, and tp are recalculated. This is done at layer n so that layer n+1 need not have knowledge of the internal workings and packet structures used by layer n. The recalculation must be done in the order shown as follows:

```
hp = dp;
dp = hp  +  sizeof(current_layer_header);
tp = tp  -  sizeof(current_layer_trailer);
```

As implied by the recalculation algorithms, headers shrink from left to right and trailers from right to left as they are passed from layer n to layer n+1 in received pbufs. The recalculation of hp, dp, and tp is shown in Figure 24 as a packet is passed up from the MAC layer to the Application layer.

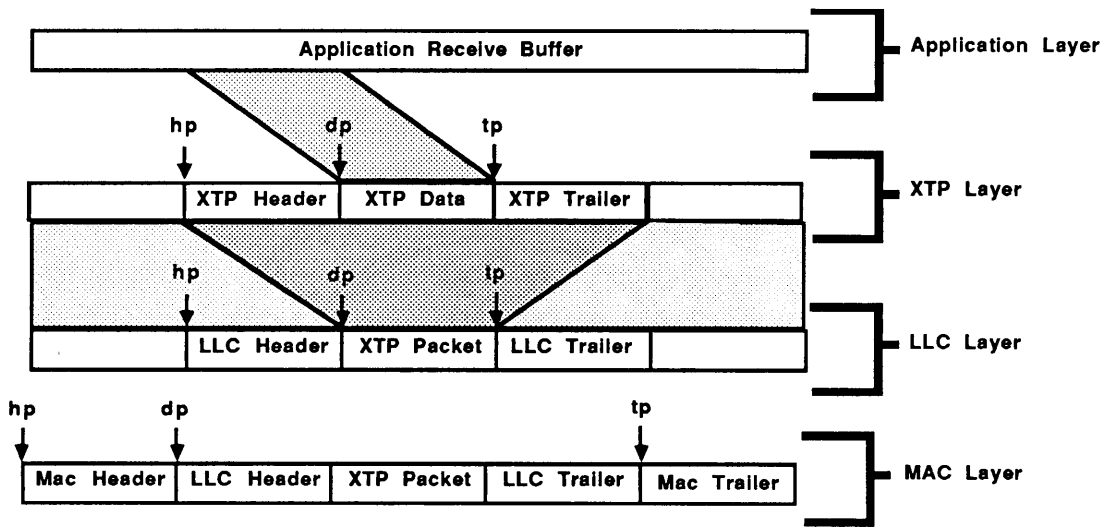


Figure 24. Adjusting Receive pbuf Pointers

4.3.5.2. Transmit pbufs

To initialize a transmit pbuf, the pbuf pointers must be set to the appropriate values in the order shown as follows:

```
hp = &pbuf.packet  +  MAX_HEADER_SIZE  -  1;
dp = hp + 1;
tp = &pbuf.packet[MAX_PACKET_SIZE]  -
     (MAX_TRAILER_SIZE  -  1);
```

This initialization is done at the Transport layer since packets are first generated there in an ITA node. This is shown in Figure 25.

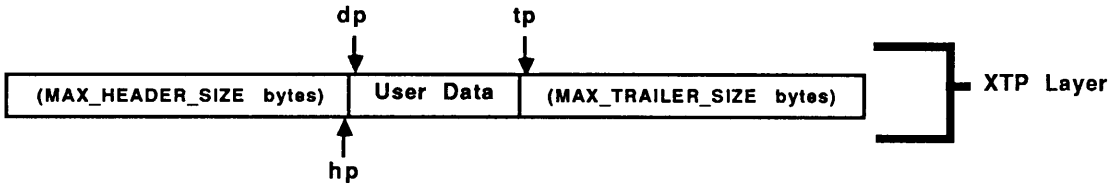


Figure 25. Transmit pbuf Pointer Initialization at the XTP Layer

The hp field is initialized so that the largest possible packet header, MAX_HEADER_SIZE, may be stored in the packet segment of a pbuf. When a layer wishes to prepend its header to the data segment of the packet, it recalculates hp as follows:

```
old_hp = hp;
hp = hp - sizeof(current_layer_header);
```

The sizeof(current_layer_header) refers to the calculated size of the header for the current protocol layer. Memory locations hp through old_hp may be filled with the header information as shown in Figure 26.

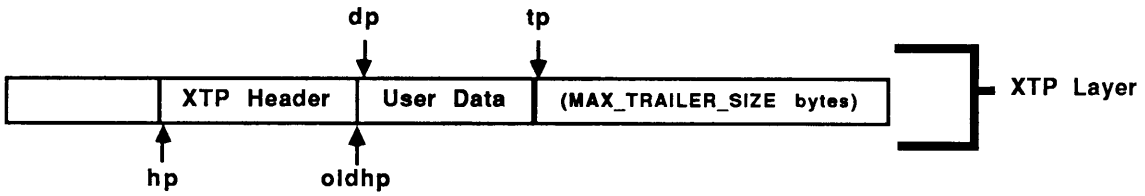


Figure 26. Prepending an XTP Header to a Transmit pbuf

The variable tp is initialized to point one location after the end of the maximum length data segment. It is important to note that this scheme requires the data segment of all packets be the same length, MAX_DATA_SIZE. A suitable number is chosen for this value to accommodate FDDI, 802.5, and Ethernet packets.

When a layer wishes to append its trailer to the data segment of the packet, it first recalculates tp:

```
old_tp = tp;
tp = tp + (sizeof(current_layer_trailer - 1));
```

The sizeof(current_layer_trailer) refers to the calculated size of the trailer for the current protocol layer. Memory locations old_tp through tp may be filled with the trailer information as shown in Figure 27.

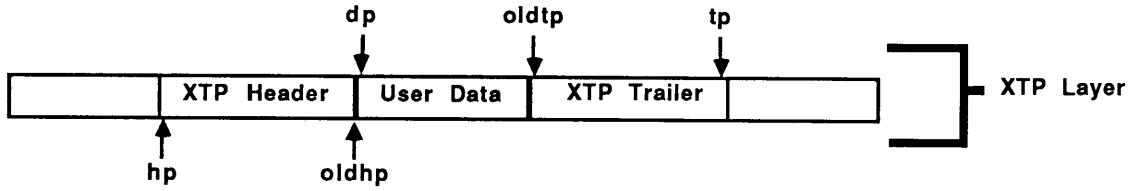


Figure 27. Appending an XTP Trailer to a Transmit pbuf

After hp and tp have been adjusted to reflect the new position of the packet header and trailer, hp, dp, and tp must be recalculated so that the transmit pbuf may passed from layer n to layer n-1. This must be done in the order shown as follows:

```
dp = hp;
hp = dp - 1;
tp = tp + 1;
```

As implied by the recalculation algorithms and illustrated in Figure 28, headers grow from right to left and trailers from left to right as they are passed from layer n to layer n-1 in pbufs to transmit.

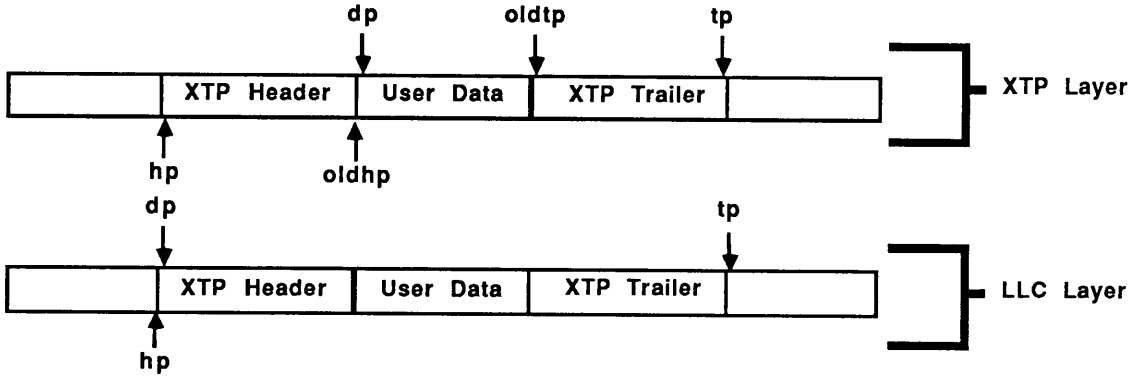


Figure 28. Passing an XTP Transmit pbuf to the LLC Layer

4.3.6. Advantages of the Memory Management Scheme

The memory management scheme presented offers several advantages over other schemes examined. First, the number of memory transfers made by the XTP subsystem is reduced. Second, packet retransmission can take advantage of the memory management scheme to quickly retransmit packets. Finally, the structure of the pbuf_struct is used to pass control messages between protocol layers. This type of message passing is not found in many other methodologies.

A number of other schemes copy all of the packet information at a given layer to a buffer in the next lower layer. Header and trailer information are added to the buffer and it is copied to the next lower layer. This process is repeated until the data reaches the lowest layer, where it is transmitted. The copies between the layers are done to enforce the protocol layer boundaries defined (i.e. the protocol software at a given layer need not rely on the implementation details of any other layer). The memory management scheme presented here accomplishes the same effect by using pbuf_structs and the algorithms for manipulating the header, data , and the trailer pointers without the enormous overhead of copying every byte at each interface.

The pointer manipulation algorithms help provide the insulation between layers described above. Each layer is given pointers to the packet header, its data portion, and its trailer. These pointers have been positioned by the previous layer to hide the details of that layer from the next protocol layer. As long as these pointers are manipulated properly by a protocol layer before a packet is passed on the next protocol layer, this scheme works. To ensure that this happens, trusted macros or functions can be defined to hide the details of the pointer manipulations.

The XTP subsystem use the memory management scheme for fast retransmissions. An XTP implementation may adopt a policy of not returning pbuf_structs to the free memory pool until the transport packet contained in it is acknowledged by the remote XTP system. When a packet is transmitted by the MAC layer, the pbuf_struct describing the packet contains all of the information needed to transmit the packet. Since all layers have a pointer to the pbuf_struct, the XTP layer can simply pass a pointer to the pbuf directly to the MAC layer for retransmission if the packet is found to be lost. In doing so, much of the protocol processing done at the Transport layer,

Network layer, and MAC layer is bypassed. Since this scheme circumvents the layering strategy enforced by the memory management scheme, it must be used with great care. If used properly, it can reduce retransmission times dramatically.

The other fields of the pbuf_struct structures are used for intralayer communication. Control information needed by the MAC layer can be passed down from the Transport layer. Likewise, control information found in the MAC layer passed to the Transport layer. This type of mechanism is not ordinarily provided nor required by other network memory management schemes. XTP, however, requires this type of mechanism to operate.

XTP uses the 48-bit physical address of a node for port filtering and in the translation map. Since this quantity is only available to the MAC layer, it must be passed to the Transport layer where it is used. Similarly, since sort and order determine the priority of an XTP connection, these values must be able to be examined by each layer to ensure that packets are processed in the proper order.

4.3.7. The Translation Map

The XTP translation map is a mechanism which permits a logical association to be made between network addresses and XTP context identifiers. The XTP subsystem consults this database upon receiving a packet to see if the packets should be accepted by the XTP. If a packet is accepted, it is processed by the XTP subsystem and passed on to the application. Otherwise, the packet is discarded. When the XTP subsystem transmits a packet, it first consults the translation map to determine the destination address of the packet.

The translation database is composed of an array of XTP_trans_map_entry structures. These structures have the following format:

```
struct  XTP_trans_map_entry  {
      int  valid_entry;
      int  key;
      char  id[6];
      int  context_id;
      struct  XTP_address  addr;
};
```

If valid_entry is set to TRUE, the translation map entry described by the rest of the structure contains a valid entry. Initially, all entries in the translation map array are set to FALSE. The key field contains the unique 32 bit connection identifier generated by the XTP subsystem. The 48-bit MAC address of the node referenced by the translation map entry is stored in the id field. The context number assigned to this connection is stored in context_id. The addr field is an XTP_address structure used to store the XTP internet address of the referenced node.

Functions calling any of the translation map manipulation functions must first associate the XTP address passed to them in the address parameter to the 48-bit MAC address used as the id parameter of the functions. This is done by searching a table similar to /etc/ethers in 4.2 BSD UNIX.

All translation map manipulation functions are guaranteed to be atomic by a semaphore called trans_map_sem. The first operation in each function is to issue an sc_spend() on the semaphore. The calling function is allowed to proceed only if the semaphore is available, which indicates that a function manipulating the table is currently not in operation.

4.3.7.1. trans_map_insert()

The trans_map_insert() function is used to insert objects into the translation database. This function has three parameters as shown below:

```
trans_map_insert(id,key,context_id);
```

The first parameter, id, identifies the referenced node by its 48-bit MAC address. The second parameter, key, corresponds to the connection identifier for the referenced connection. The pair (id,key) uniquely identifies a connection since id is unique within the internet and key is unique within a node. The final parameter, context_id, specifies the context identifier assigned to the connection. This function is called by both the XTP_context.request() function and the XTP_register.request() functions.

4.3.7.2. trans_map_delete()

The trans_map_delete() function is used to delete objects from the translation database. This function has three parameters as shown below:

```
trans_map_delete(id,key,context_id);
```

The parameters have meanings identical to those for trans_map_insert. Objects are deleted from the database in two ways. First, if id and key are both non-zero, the object specified by that pair is deleted. If context_id is non-zero, the object specified by the context identifier is deleted.

4.3.7.3. trans_map_lookup()

The trans_map_lookup() function is used to look up objects in the translation database. This function has two parameters as shown:

```
int  trans_map_lookup(id,key);
```

An (id,key) pair is passed to this function. If the (id,key) pair is found in the database, the context identifier is returned to the caller. Otherwise, if the object is not found, a zero is returned to the caller.

4.3.7.4 Other Translation Map Functions

Two other translation map functions defined in the Revision 3.25 protocol definition are not implemented. The first of these functions, setmap(), adds an entry to the translation database which links a previously inserted object with a new object. It is unclear how the setmap() function is used in an XTP implementation, so it is not selected for implementation. The other function, match(), determines whether an XTP host is interested in a particular address. The operation of this function appears to be covered by the lookup function. If trans_map_lookup() returns a zero, the address is not of interest. If trans_map_lookup() returns a non-zero value, the address is of interest. The XTP protocol definition must explain in more detail the intended operation of these functions.

4.3.8. XTP Tuning Parameters

The XTP tuning parameters allow communication parameters to be adjusted by the XTP subsystem. The values used for these parameters depend on the application. Their value must be obtained experimentally for each application. This implementation uses default values provided in the XTP default template. The definition of this template is beyond the scope of this thesis and is part of future ITA investigations.

4.4. XTP Processes

Two basic process comprise the XTP implementation: the scheduler and the timer manager. These processes are scheduled by VRTX. The scheduler process contains two sub-processes, the XTP transmitter and XTP receiver, which it manages. These processes interact as shown in Figure 29.

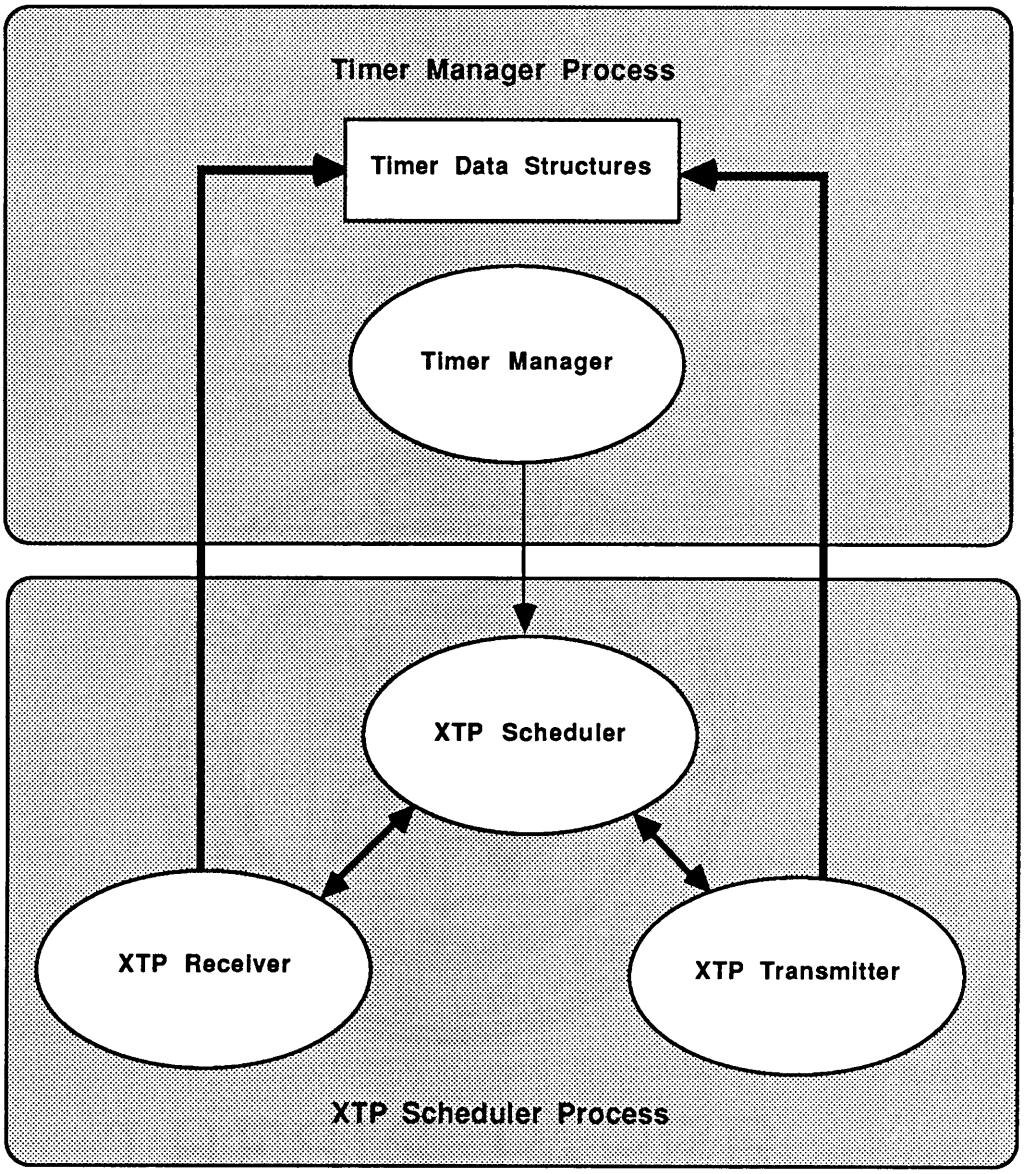


Figure 29. XTP Process Interaction

As shown in Figure 29, all XTP processes and sub-processes interact with the XTP scheduler process. The timer manager communicates with the scheduler by posting messages to a VRTX queue the scheduler pends on. The XTP scheduler controls the XTP receiver and XTP transmitter by calling them directly. The receiver and transmitter communicate with the timer manager by direct manipulation its data structures.

4.4.1. The XTP Scheduler

The XTP scheduler is the main process in the XTP subsystem. All XTP "events" are scheduled through this process. Communication with the scheduler is done through a VRTX queue called XTP_Q. This queue is defined to have a maximum of NUM_XTP_Q_MSGS entries. Only the index of an XTP_q_msg structure, a called scheduler message, may be posted to this queue.

The format of an XTP_q_msg structure is shown below:

```
struct  XTP_q_msg  {
        int  type;
        int  context;
        int  message;
};
```

The type field determines the meaning of the context and message fields. Initially, four types are defined. The first type, CONTROL_BLOCK, is used to inform the scheduler that the message being posted is a control block. In this case, the message field contains the index of the control block being posted and the context field is unused. The second type, TIMEOUT, indicates that an XTP timer has expired. When this type is used, the context field indicates the index of the context which timed out. The message field contains the type of timer which expired: CTIMER, WTIMER, or RTIMER. The third message type, RECEIVE_PACKET, indicates that a packet is being posted from the LLC layer to the XTP layer. In this case, the context field contains the index of the pbuf in which the received data is stored and the message field is unused. The fourth and final packet type, ROBIN_PAUSE, indicates that a context can unblock if it was pausing because of round-robin scheduling. Here, the context field indicates which context can unblock and the message field is unused.

Scheduler messages are requested by both the XTP host and the XTP subsystem. A array of XTP_q_msg structures called xqm is used to allocate storage for these messages. This structure is defined as shown below:

```
struct   XTP_q_msg[NUM_XTP_Q_MSGS];
```

The index of each of these elements is posted to a queue of free scheduler messages called XTP_XQMQ. When a scheduler message is desired, its index is returned by issuing sc_qpend() on this queue. A scheduler message is freed by posting its index back to this queue using an sc_qpost() call.

The scheduler runs a daemon[4] process. Its basic format is outlined below:

```
void  xtp_scheduler()
{
      int  msg_index;

      do  forever
            {
                  msg_index  =  sc_qpend(XTP_Q);
                  switch(xqm[msg_index].type)  {
                        case  CONTROL_BLOCK:
                              process_control_block();
                              break;
                        case  TIMEOUT:
                              process_timeout();
                              break;
                        case  RECEIVE_PACKET:
                              receiver();
                              break;
                        case  ROBIN_PAUSE:
                              process_robin_pause();
                              break;
                        default:
                              error();
                              break;
                  }
            }
}
```

The sc_qpend() call forces the scheduler to sleep until a scheduler message index is posted to it. When the sc_qpend() call returns, the index of the scheduler message is stored in msg_index. Next, the scheduler "switches" on the message type. If an illegal message type is found, an error routine is invoked.

4.4.1.1. Control Block Processing

Control blocks are posted to the XTP scheduler by the XTP host in response to service primitive calls. Control block processing consists of two parts: first the control block is processed

[4]A daemon is a process that generally performs low-level services and never terminates.

and next, if necessary, the context referenced by the control block is processed. The context is usually processed because most control blocks elicit processing on the pertinent context.

The process_control_block() function examines the op field of the control block to determine how the control block is processed. For example, if the op field contains a create context command, a free context record is acquired and the appropriate fields are filled in. Next, the control block is posted to the referenced context. To post a transmit control block to a context, the tcb_post() function is used while rcb_post() is used to post a receive control block. These functions append the control block to the end of either the tcbq or the rcbq queue of control blocks in the XTP_context_record structure of the referenced context.

If the context referenced by the control block needs further processing, a context processing function is called next. In the create context example above, the XTP transmitter is called to transmit any data specified in the control block. The context need not always be processed, however. If a control block requesting an XTP tuning parameter change is posted to the XTP, the tuning parameter is changed completely by the control block processing function. In this case, no further processing is needed on the context.

4.4.1.2. Timer Expiration Processing

If the scheduler invokes the timer expiration processor, the process_timeout() function changes the state of the context timing out. The state changes are defined by the XTP Protocol Definition.

The CTIMER and WTIMER are used by the XTP "to detect host, network, or application software failure"[6]. If a CTIMER expires and the received packet count for the context is zero, the ctimeout flag is set for the context and the WTIMER is started. After four WTIMER expirations, an error condition is signalled by the process_timeout() function.

An RTIMER is used to generate an interpacket gap if one is needed. After the specified number of packets or amount of data is transmitted on a context, the RTIMER for that context is started. The context is moved into the RTIMER_PAUSE state by the XTP transmitter. When the

RTIMER expires and its XTP_q_msg is processed by the process_timeout() function, the context is moved back to the BUSY state.

4.4.1.3. Receive Packet Processing

If the scheduler invokes the receive packet processor, the XTP receiver is called by the process_control_block() function. The process_control_block() function has no further function in this case as the XTP receiver can perform all of the needed processing.

4.4.1.4. Robin Pause Processing

When round-robin scheduling is selected, contexts become blocked after they have transmitted the number of bytes defined by the robin field in their context record. The context must remain blocked until all other XTP contexts have been serviced in round-robin order. To enforce this policy, the XTP transmitter moves the context to the ROBIN_PAUSE state. Next, a scheduler message index is posted to the XTP_Q. In the scheduler message, the type field is set to ROBIN_PAUSE and the context field is set to the context number of the context.

After the scheduler message index is posted, the XTP_Q contains messages for all contexts needing service followed by the robin pause message. Since all of the contexts needing service are processed before the robin pause message is received, the round-robin scheduling policy called out in the XTP Protocol Definition is enforced.

When the robin pause message is removed from the XTP_Q, the referenced context is moved from the ROBIN_PAUSE state to the BUSY state where transmission can continue. Finally, the XTP transmitter is invoked on the context.

4.4.2. Timers

XTP requires three distinct timer classes for each connection. Since there may be hundreds of simultaneous connections, it is important that an efficient timer management strategy is developed. In this implementation, a single task (daemon) is used to implement each timer class. The ctimer, wtimer, and rtimer fields of each context record indicate the current value of the

corresponding timer for that context. Each of the timer fields is set to zero during initialization to indicate that the timer is inactive. A timer is started for a given context by setting the appropriate timer field to the desired timeout value. As long as this value is a positive integer (excluding zero), the timer is started. For example, to the ctimer for context five is to be set to 5 seconds:

```
cr[5].ctimer  =  500;
```

In the above example, it is assumed that the timer clock has a period of 10 ms. The tasks which manage the timers operate according to the following pseudo-code procedure:

```
void  xtimer()
{
      int  i,temp_val;

      sc_delay(XTIMER_TICK);
      do  forever
            {
                  for  (i  =  1;  i  <  NUM_CONTEXT_RECORDS  -  1;  i++)
                        if  (cr[i].xtimer  !=  0)  {
                              temp_val  =  cr[i].xtimer  -
                                                XTIMER_TICK;
                              if  (temp_val  <=  0)  {
                                    cr[i].xtimer  =  0;
                                    timeout(i,XTIMER);
                              }
                              else
                                    cr[i].xtimer  =  temp_val;
                        }
                  sc_delay(XTIMER_TICK);
            }
}
```

In the above pseudo-code procedure, all references to XTIMER or xtimer should be replaced by a reference to either ctimer, wtimer, or rtimer.

The initial sc_delay statement in the procedure ensures that at least XTIMER_TICKs of the VRTX timer have passed before the process turns itself into a daemon. After the delay, the daemon checks the appropriate timer of each context in turn. If the timer is turned on, indicated by a positive time value, one XTIMER_TICK is subtracted from the current value. If the result is less than or equal to zero, the timer expired so its value is set to zero and a timeout is signalled by making a timeout() call. Otherwise, if the timer didn't expire, the old timer value is overwritten with the calculated time. It is important to observe that there are two clocks used in this

implementation: VRTX uses a clock based on clock ticks of the underlying timer hardware while the XTP timer management functions use a clock based on clock ticks of the VRTX clock.

The function, timeout(), is called with two parameters: i indicates which context timed out and XTIMER indicates which timer of that context timed out. This function builds an XTP_q_msg structure in which the type field is set to XTIMER and the context field is set to i. The message field is not filled in because it is not needed for this message type.

4.5. XTP Operation

In this section, an overview of the packet transmission and packet reception operations are described. Since both the XTP transmitter and XTP receiver are functions called by the XTP scheduler, they do not operate concurrently.

4.5.1. Packet Transmission

All packet transmission is done by the XTP transmitter function, called sender(). The transmitter is always called from within the XTP scheduler process. The transmitter is called by the scheduler in response to response to two events: either a control block requesting packet transmission or by a context enabling signal generated by a robin-pause message, a timer expiration message, or an ALLOC advancement message.

The transmitter maintains a queue of control blocks to transmit in the tcbq field of the context record. When a new transmit control block is posted to the XTP scheduler, it is placed at the end of this queue. The transmitter is then called to process the control blocks in this queue.

When the transmitter is called, the first control block on the tcbq is examined to see if data transmission is permitted. If it is, the length of the remaining data in the control block is calculated by subtracting the nptr and eptr fields of the control block. Next, a pbuf is fetched. If the remaining data fits inside the available area in the pbuf, the data is copied in its entirety to the pbuf, XTP headers and trailers are added to the pbuf, the pbuf pointers are adjusted, and the pbuf is posted to the LLC sublayer. If the data does not fit inside a single pbuf, multiple pbufs are fetched, filled, and posted to the LLC sublayer as described above until either the entire message

is transmitted or the an XTP disabling condition is reached (i.e. ALLOC, Robin, or Rate is reached). If the entire message is transmitted, the completed control blocks are returned to the XTP application which issued them using the cb_post() function and the next control block is processed. Otherwise, the transmitter returns to the XTP scheduler when the disabling condition is reached.

4.5.2. Packet Reception

All received packets are processed by the XTP receiver function, called receiver(). The receiver is only invoked from within the XTP scheduler process when a scheduler message contains a type field of RECEIVE_PACKET. This type of XTP_q_msg is only posted by the LLC sublayer.

The receiver maintains a queue of control blocks to receive data in the rcbq field of the context record. Like the XTP transmitter, when a new receive control block is posted to the XTP scheduler, it is placed at the end of this queue. The receiver is then called to process the control blocks in this queue.

When the receiver is called, the first control block on the rcbq is examined to find a receive buffer area where the data in the pbuf may be copied. If the buffer area is larger than the amount of information in the pbuf, the complete data portion of the pbuf is copied to the buffer area. The pbuf is freed and the receive buffer pointers in the control block are updated. If the entire buffer area is filled before the pbuf is copied, the control block is made available to the XTP host which issued it by calling the cb_post() function, another control block is fetched, and the copy process is repeated. If the EOM bit is set in the packet, indicating that this is the last packet of the message, the control block is posted using the cb_post() function whether the receive buffer is completely filled or not.

If a pbuf is received and no receive control blocks are available on the rcbq, it is added to the rpbufq. Only rpbufqmax entries may be stored in the queue. If this limit is exceeded, the pbuf is discarded by the XTP (i.e. freed). By limiting the amount of pbufs a context may hold for

reception, a context whose application process is not listening can be kept from consuming all of the available pbufs available to the system.

Chapter 5.

5. Results of the Express Transfer Protocol Implementation

In this section, the results of the XTP implementation are presented. Although a complete working implementation of XTP implementation presented in Chapter 4 has not been built yet, a number of valuable observations have been made in the implementation effort.

As the implementation proceeded, it became clear that there are large number of problems in the XTP protocol definition which are not uncovered by detailed reviews alone. It is found that only an actual implementation gets into the level of detailed analysis required to find many of the intricate problems hidden in the specification.

5.1. The XTP Service Definition

The importance of a service definition emerged during the implementation. The design proceeded from the bottom up following the lead of the Protocol Definition. When the implementation reached the point where it interacted with application processes, how processes at the Application layer use XTP became unclear since the Protocol Definition provides no defined services. The interface presented in the XTP Protocol definition implies that the Application layer should have direct knowledge of internal workings and packet formats used by the XTP layer. This contradicts one of the fundamental principals of protocol design, well defined protocol layering.

A new approach, described in Chapter 4, uses a top down approach. First the services provided by the XTP layer are defined, then the interface between the services and the XTP are defined. Using this approach, it is found that if the the interface between the Application layer and the XTP layer is relatively straightforward to implement. Defining the services provided is by XTP is difficult and requires a great deal of thought. While it is felt that the service definition presented in this paper provides a good framework for a standard XTP service definition, a lot more work is needed in this area.

Many of the questions raised during the protocol implementation could have been answered directly through an XTP service definition. Since none is available, "educated" guesses are made about how XTP is to be used and how it should operate.

If "educated" guesses are made during a protocol implementation, chaos can occur when interoperability testing is done. If an implementor isn't "educated" in the same way as you, are your implementations likely to be interoperable? Almost assuredly, they are not. As a recommendation, the SAFENET committee should ensure that a standard XTP service definition is made available to implementors. Either the committee should create the service definition or have one created by Dr. Chesson. Regardless of who generates the document, the committee must ensure that the service definition provides all of the serviced needed by SAFENET. As a suggestion, the services should be tailored to meet the needs of SAFENET applications, not to to the requirements of an XTP implementation.

5.2. Standard Transport Protocol Data Units

A service definition should have standard Transport Protocol Data Units (TPDUs) which correspond to its service primitives. This is done so that a correspondence can easily be drawn between the service primitives invoked and the packets generated by them. TP/4 uses this approach. The SAFENET committee should require that a standard TPDU definition be included with any XTP service definition.

5.3. Connection Collision

Connection collision occurs when two transport entities simultaneously try to establish a connection with one another. This cannot occur using the service primitives presented in Chapter 4 because of the client/server relationship required by them.

A server issues a register request primitive to create a port filter to accept connection requests from clients. When a client wishes to establish a connection with a server, it issues a connect request to that client. This mode of operation corresponds to the sender (client) initialization and receiver (server) initialization described in Section 3.3 of Revision 3.25 of the

protocol definition. A server can not send data to a client until the client has successfully issued a connect request. After the connection is established, full-duplex communications can be done over the channel. Chesson has stated "Node B (the server) !can! send a FIRST packet to Node A (the client)"[7].

This is not explicitly stated in the protocol definition. If this is the case, however, there is no distinction between a client and server. Consequently, symmetric primitives can be used to establish a connection (i.e. both sides can issue a connect request instead of the client issuing a connect request and the server issuing a register request). Is this the intended operation of the protocol? Should one full-duplex connection be created, following TCP, or should two full-duplex connections be created, following TP/4? These questions would be easily answered if a standard service definition is made available.

5.4. Initialization Parameters

As pointed out in the comments on the Revision 3.1 Protocol Definition (Appendix B - comment 1.13.1), a scheme must be defined for initializing context records. As a result of this comment, an initialization template is defined to implement this scheme in Protocol Definitions subsequent to Revision 3.1. The importance and scope of the initialization template is not fully realized until it is implemented, however.

The implementation reveals that not only the names of the to be placed in the template must be defined, but additionally the values the initial values of these parameters must be defined. This may be exemplified using the alloc parameter of the template.

Initially the implementation chose to initialize the entire initialization template to zero. While this didn't cause XTP to malfunction, it did cause some unexpected results. First packets are not sent during an XTP_context_request() because the alloc parameter, which is set to zero in this case, indicates that no space is available on the receiver.

As a second attempt to formulate an appropriate value for alloc, the value negative one is used. Since this value is the largest possible number (using unsigned notation), packets should be continuously transmitted until alloc is changed by the remote station. A problem arises,

however, if an implementation doesn't change alloc unless the new value is greater than the previous value. Alloc is simply never advanced no value is greater than the initial value, negative one.

The algorithm for advancing alloc, described above, is appropriate for an internetworked environment. Since packets may be delayed due to taking different paths through the internet, older values of alloc may arrive after newer values of alloc. If this happens, the old values must be ignored so that possibly committed resources are not taken away from an XTP entity. As shown above, appropriate initial values of initialization template parameters must be defined in the specification.

5.5. Quality of Service Negotiation

Marc Cohn defines a quality of service parameter which is passed as a parameter to his "straw-man" service primitives. The only parameter of this type described is priority. Other parameters might be defined. For example, a selection of selective retry or go-back-n retransmission strategies might be selected. A protocol for negotiating these parameters must be defined.

5.6. Separation of the Wheat from the Chaff

The XTP protocol definitions are a combination of a definition of the protocol and a description of how one might build an implementation of the protocol. This leads to confusion during implementation.

Consider, for example, the title of Chapter 3 of the Revision 3.25 Protocol Definition: "Transport Protocol Description". One might get the impression that this chapter describes the details of the protocol when, in actuality, it describes how an implementation might be built. In Section 3.1 of this chapter it is stated: "the protocol definition is partitioned into four components, or processes: receiver, sender, and reader and writer". Is the protocol definition actually divided into four processes? The answer is "no", but an implementation might use that partitioning.

Further, Chapter 3 presents control blocks and context records. These data structures are in reality implementation aids and not part of the protocol definition.

Chapter 4 uses a "C" pseudo-code implementation of the XTP receiver and XTP transmitter to describe the protocol. This leads to an imprecise definition which is confusing to an implementor. Instead, a precise protocol description language, such as Estelle, should be used.

As shown above, the description of the XTP protocol, the wheat, must be separated from the implementation details, the chaff. The implementation details should be presented in a separate implementors guide, not the protocol definition. The protocol definition should describe what is transmitted "over the wire" and in what order the items should be transmitted. This should be done using a textual description with a complementary formal specification language or state transition table.

5.7. Zero Length Packets

As application processes began to use the XTP implementation, an unexpected behavior of the XTP implementation is found. An application process found that calling XTP service primitives with zero length data fields reduced the coding complexity of some applications considerably. When the applications were tested, they caused the XTP subsystem to fail. The reason for this is that zero length packets are not addressed by the XTP protocol definition. If a zero length packet is received, should the sequence numbers be advanced? If they are, the packet buffering schemes must be redefined. If they are not, several packets can have the same sequence number. This can cause a number of problems in an XTP receiver. How zero length packets should be handled must be specified in the XTP Protocol Definition.

5.8. Evaluation of the Express Transfer Protocol Evaluation Process

In Section 1.8. a new protocol evaluation process is described. The process consists of two phases. First, detailed reviews of the XTP Protocol Definitions are completed. Next, selected aspects of the protocol are implemented. A number of advantages over traditional protocol evaluation techniques are found: faults in the protocol definition are quickly revealed, only

selected parts of the protocol have to be implemented, and a concentrated effort can be made on the interactive behavior of the protocol.

The first advantage, faults in the protocol definition are quickly revealed, results from the detailed reviews. Instead of spending a lot of time planning a system design, as would be done in a complete implementation, or defining a model to be evaluated, as would be done in a simulation, the Protocol Definition is reviewed. Problems are found quickly due to the direct analysis of the Protocol Definitions.

Results of the protocol reviews have a direct impact on subsequent Protocol Definitions because they are promptly delivered to the protocol designers. A large number of problems are pointed out in the detailed reviews presented in Appendices A, B, C, D, and E. Their direct impact on the XTP Protocol is seen in subsequent releases of the Protocol Definition. For example, in Appendix B - comment 1.2.1.it is shown that the extension field of an XTP packet renders the packet unparsable. Because this detailed review is delivered to PEI before the release of Revision 3.2, this problem is corrected in that revision.

The second advantage, only selected parts of the protocol have to be implemented, had a direct impact on the implementation effort. The only parts of the protocol which are implemented are those found worthy by the detailed reviews. Because of the reduced scope of implementation, a concentrated effort can be made in implementing the selected features. By spending a little time up front in the detailed reviews, less time is spend trying to implement protocol features which have specification problems. Again, time is saved and results are quickly delivered to PEI.

Obviously, in a complete implementation, all aspects of the protocol have to be built so the new protocol evaluation process has an obvious advantage. Partial simulation can be done, however, but deciding what to simulate can be a problem. A detailed review of the Protocol Definition should be done to narrow the scope of the simulation.

The final advantage, a concentrated effort can be made on the interactive behavior of the protocol, is realized because of the time saved in the implementation effort. As pointed out above only selected parts of the protocol are implemented. There is a level of confidence that these

parts of the protocol have no "obvious" problems. Once these protocol elements are built, the interaction between them can be studied. Since questionable protocol elements are not even implemented, their interaction with other protocol elements is not studied allowing more time to be spent on the worthy parts of the protocol.

The synergy of combining the protocol evaluations with the implementation of selected parts of the protocol is found to be most appropriate in achieving the goals of this thesis. "Holes" which exist in the XTP Protocol Definition are uncovered early during the detailed protocol review process. The partial implementation serves to uncover more subtle and in depth problems.

The initial evaluation of XTP described in Chapter 3 proved to be invaluable in the implementation effort outlined in Chapter 4. Five major protocol elements are found to be unsuitable for implementation. A substantial amount of time is saved in the XTP implementation by early recognition of unsuitable elements. Full implementation focuses on those XTP features (a subset) deemed most beneficial.

A design is made of an implementation of the protocol subset selected. This work is presented in Chapter 4. The design and its subsequent implementation find a number of problems which are not uncovered in the detailed reviews. The revealed problems, presented in this chapter, are only found because of the rigorous analysis mandated to implement the design.

A comparison can be made between the two phases of analysis. The detailed reviews, performed during the first phase, tend to find problems dealing with a single aspect of the protocol. Take for example, comment 1.2.2. in the 3.1 Protocol Review presented in Appendix B. This comment deals with how the XTP checksum is to be calculated. This comment deals strictly with on aspect of the XTP protocol, stopping further consideration of the checksum algorithm. No analysis is done on how this algorithm might affect or be affected by other parts of the protocol.

During the second phase, the highly complex interactions between parts of the XTP protocol, peer XTP subsystems, and XTP applications become visible. Temporal relationships, which are overlooked in the first phase, stand out in the second phase. For example, in Section 5.4 problems with the initialization of XTP context records are described. The problems indicated only become apparent after several packets are exchanged between peer XTP subsystems.

Another problem missed during the analysis in the first phase deals with zero length packet, presented in Section 5.7. In this case, an application process sends a message of zero length. The possibility of sending such a message is not considered during the first phase because the realm of XTP application needs is inadequately understood until an actual application is designed and tested. The needed analysis is difficult during this phase because of the many complex interactions among parts of the protocol. Often these interactions only become visible in a working implementation. This is where the second phase of analysis, a partial implementation, proves to be a valuable technique in incrementally developing a protocol.

While a simulation of XTP is not done, for the reasons described in Section 1.8, such an analysis would provide useful information not provided by the detailed analysis and partial implementation. Each type of analysis has distinct advantages. Simulation can address performance issues in large systems (i.e. a network with 100 nodes) that is too costly to implement. Implementation may be able to find detailed problems which are perhaps overlooked in a simulation. This is due to the fact that an implementation deals with the protocol directly while simulations tend to deal with protocols in more abstract terms.

5.9. Summary

It is found that the protocol evaluation approach defined in Chapter 1 serves its intended purpose, to find problems in the XTP Protocol Definitions. The XTP implementation process finds a number of problems with the XTP protocol that are not uncovered by the detailed review process alone. These problems arise during the implementation of XTP and as applications begin to use XTP. As a result of the implementation, it is concluded that a number of areas of XTP must be addressed before independent implementations can meaningfully communicate.

Chapter 6.

6. Conclusions and Future Work

In this chapter, some conclusions are drawn about the XTP protocol definitions up to and including Revision 3.25. Future possible work in area of XTP is presented.

6.1. Weaknesses of the Express Transfer Protocol Specification

As shown in Chapter 3 and Chapter 4, there are a number of weaknesses in the XTP protocol definition. In Chapter 3, four components of the XTP protocol, reliable multicast operations, priority mechanisms, the checksum algorithm, and out of band messages, are pointed out as not yet mature enough for implementation. These areas must be addressed in future protocol definitions. In Chapter 4, the implementation process reveals a number of other areas which need to be addressed in future protocol definitions. These areas include: the lack of an XTP service definition, guidelines for advancing alloc, quality of service negotiation, and the fact that implementation details are included in the protocol definition.

The only aspect of the XTP protocol which can not currently be built in a software implementation is the dequeueing needed for making multicast error recovery efficient. This is not required since the protocol works without it. It should be noted that implementations which do not implement this feature generate excess network traffic when retries are done.

6.2. Is the Express Transfer Protocol Definition Implementable?

There is little doubt that a working version of XTP can be built from its protocol definition. A question which would be much harder to answer would be: can different implementations of XTP built from the protocol definition interoperate? The problems pointed out in Chapter 3, Chapter 4, the protocol reviews, and correspondence with Chesson would lead one to conclude that independent implementations are not likely to demonstrate interoperability. The validity of this conclusion remains to be confirmed until interoperability tests are performed.

6.3. What is Next?

This thesis presents the design of an XTP implementation. Many of the concepts presented in the design are implemented and tested on the ITA testbed at NSWC. Future investigations should complete the implementation and testing of the design. As new protocol definitions become available, they are to be reviewed and reported on and the implementation upgraded to incorporate any new protocol features or protocol changes.

Experimentation might be done using simulations to evaluate XTP. A research topic would be to apply the protocol analysis approach presented in this thesis to simulations. A partial simulation might address performance issues early on, thus eliminating them from further consideration during a larger simulation. The protocol analysis approach might also be applied to protocol verification techniques.

6.4. Closing Comments

The protocol definitions reviewed in this thesis are experimental in nature because the protocol is still under development. The reviews of the protocols and the comments and conclusions drawn about the protocol reflect the enthusiastic interest the Navy, SAFENET committee, private industry, and the author have for XTP. Chesson has been both sensitive and responsive to the issues raised by the author and others. This is reflected in the incorporation of many of the authors comments and suggestions in new releases of the protocol definition. Since the development point where the analysis and evaluation is reported herein, Revisions 3.3 and Revision 3.4 have been released. A cursory glance at these documents gives the impression that XTP, like a fine wine, gets better with age.

Appendix A - Listing of All Questions and Comments in the Review of Revision 2.0

Comments on the Revision 2 XTP Specification

(12 January 1988)

Philip M. Irey IV

Naval Surface Weapons Center
Dahlgren, Va 22448

1. Protocol Comments and Questions

1.1. Page 4-3, Section 4.1.1.1.5:

Does the sender need to know that a duplicate packet was recieved if the packet is going to be ignored anyway? If a packet is recieved out of sequence, won't the reciever use the RESEND field to indicate which packets to resend instead of sending a REJ or is the REJ used in conjuction with the RESEND pairs?

1.1. Page 4-4, Section 4.1.1.2.1:

Why would you want to disable address recognition? Is it valid to set the TYPE field to DATAP and the ADDR bit? If so, what does this mean?

1.2. Page 4-6, Section 4.1.2:

Why is the Event field of a packet incremented when one or more of the EMASK bits are set?

1.3. Page 4-6, Section 4.1.3:

How can a unique KEY field be generated by the context initiator that is unique across the internet? Is there a KEY server on the network?

1.4. Page 4-6, Section 4.1.4:

How can you determine how many packets have not been acknowledged when byte offsets are used instead of true sequence numbers if different size packets are used? How do the byte offset sequence numbers help to simplify fragmentation/coalesing of packets?

1.5. Page 4-7, Section 4.1.5:

Does the ESEQ field indicate the sequence number of the next packet to be sent? If this is the case, should the document say "the ESEQ field is used to indicate the sequence number of the first byte of the current event"? Shouldn't it be of the next event?

1.6. Page 4-8, Section 4.2.3:

What does transferred to the host mean?

1.7. Page 4-9, Section 4.2.8:

What are the units used to specify the INTERVAL field? Are they milliseconds like the SPACE field?

1.8. Page 4-9, Section 4.2.9:

Why is NRESEND 32 bits in length? Is this a protocol engine hardware dependency? What is the size of each sequence pair (64 bits??)? If this is the case, does this imply that a maximum of 16 resend pairs will fit in the RESEND field?

1.9. Page 4-11, Sections 4.3.1.2.4 to 4.3.1.2.8:

What are the sizes of these fields?

1.10. Page 4-12, Section 4.3.5:

How do you know where the DLEN field is if you don't know the size of the DATA (or DATAP) field? How do you know what the size of the PAD field is?

1.11. Page 5.4, Section 5.1.1.5:

Is the PRI field used by the host only in managing the order in which commands will be given to the XTP, or is it used by both XTP and the host?

1.12. General Comment:

Is the XTP control block used by both XTP and the host? If this is true, should the size of all of the elements of the control block be defined?

1.13. Page 5-14, Section 5.2.3, paragraph 3:

Is the WTIMER a count-down and a count-up timer? It seems that when an ENQ packet is sent, WTIMER counts down. If WTIMER reaches zero before a STATUS is recieved, another ENQ will be sent and WTIMER will count up until a status is recieved. Is this true? If a STATUS is recieved and its ECHO field does not correspond to the SYNC field of the last ENQ sent out, what happens? Are all subsequent STATUS packets ignored? When a STATUS is recieved, will WTIMER be stopped and its value used as the new WTIMER count-down value? How often are ENQs sent if WTIMER is serving as a count-up timer?

1.14. Page 5-14, Section 5.2.3, paragraph 4:

How are the SYNC and ECHO fields used to distinguish STATUS packets which do now maintain sequence numbers?

2. Documentation Comments and Questions

2.1. General Comment:

For the command bits, you should specify whether a binary one or binary zero causes the action or condition that the bit signifies.

2.2. Page 4-2, Figure 4-4:

Bit 10 is labelled as MULT and it appears that it should be MULTI according to Section 4.1.1.2.7 on page 4-5.

2.3. Page 4-2, Section 4.1.1.1:

In the table both MREPLY and PATH have the same type field, 0011. Is this correct?

2.4. Page 4-4, Section 4.1.1.1.8:

In the last sentence of this section, should you use the word "recover" where you use the word "recovery"?

2.5. Page 4-4, Section 4.1.1.2.1:

Is the DATAP type used to describe a packet which has a DATAP or a DATA field or both? Is there any field called DATA? It is referenced on page 4-12 in Section 4.3.2. According to figure 4-6, the DATAP field comes before the extension field and section 4.3.2 seems to imply that the DATA field comes before it.

2.6. Page 4-4, Section 4.1.1.2.3:

Having a DATAP packet type and a DATAP field and a DATA bit and a DATA field (if a DATA field really exists) is confusing. Could the bit and the field names be different so that this is not so confusing?

2.7. Page 4-6, Section 4.1.4:

Should "bits are sent" be "bits are set"?

2.8. Page 4-7, Section 4.1.5:

Does the ESEQ field indicate the sequence number of the next packet to be sent? If this is the case, should the document say "the ESEQ field is used to indicate the sequence number of the first byte of the current event"? Should it say the "next event"?

2.9. Page 4-12, Section 4.3.2:

Is the data field really called DATA, or is it called the DATAP field as shown in figure 4-6, page 4-9?

2.10. Page 5-4, Section 5.1.1.3:

The status parameters must be defined if an XTP implementation is to be made.

2.11. Page 5-5, Section 5.1.1.9:

What does "that XTP has processed" mean in context to the TR_LEN? Does XTP update this field? If so, how does it know where this field is?

2.13. General Comment:

Is the XTP control block used by both XTP and the host? If this is true, should the size of all of the elements of the control block be defined? I have the same questions for the control block.

2.14. Page 5-6, Section 5.1.2.1:

Will the XTP update the STATE field? If so, the values associated with each of the states should be defined.

2.15. Page 5-6, Section 5.1.2.4:

Does the NSEQ element of the context record correspond to the ESEQ field of the XTP common header as described on page 4-7 in section 4.1.5?

2.16. Page 5-7, Section 5.1.2.5:

Does the XSEQ element of the context record correspond to the SEQ field of the XTP common header as described on page 4-6 in section 4.1.4?

2.17. Page 5-7, Section 5.1.2.6:

Is there a conflict with the ESEQ element of the context record and the ESEQ field of the common header as described on page 4-7 in section 4.1.5? If these are different things, they should probably be labelled differently.

2.18. Page 5-7, Section 5.1.2.7:

What does the OQMAX parameter designate? Is the output buffer really a sequential array of memory locations whose last element is designated by OQMAX?

Appendix B - Listing of All Questions and Comments in the Review of Revision 3.1

Comments on the XTP Protocol Definition (Revision 3.1)

Philip M. Irey IV

Naval Surface Weapons Center
Dahlgren, Va 22448

1. Protocol Comments and Questions

1.1. XPD3.1[1] Page 9: Burst mode

The BURST parameter should be defined more clearly. For example, does the size of BURST include the size of packet headers, packet trailers, control packets, packet data, etc?

1.2. Information Packets

1.2.1. XPD3.1 Page 10: Extension field

The contents and format of the extension field must be defined. Received packets can not be properly parsed until this is done for the reason outlined below.

When receiving XTP information packets, an ambiguity arises concerning the extension field. An XTP processing a Received packet examines the header field and determines there is an extension field present. It "links" the data in the data field to the message pointer in the control block for that context and attempts to update the message length parameter in the control block. This fails, however, because an extension field is present in the packet which prevents the message length from being calculated.

The length of the data field can be determined by computing the following formula:

$L = ML-(LT+LA+LH+EL+PL)$

where:

L = length of data field
ML = length of packet passed up from the MAC layer
LT = length of trailer field
LA = length of optional address field
LH = length of common header
EL = length of optional extension field
PL = length of the pad field

L is easily computed when no extension field is present as *EL* is zero. When an extension field is present, however, problems occur because the value of *EL* is unknown. The end of the data field and thus its length can not be determined. Likewise, the beginning of the extension field and its length can not be determined. A simple "fix" to this problem might be to put an extension field length field in the trailer.

---

[1] XPD3.2 refers to the XTP Protocol Definition, Revision 3.1

1.2.2. XPD3.1 Page 10: Checksum field

Why isn't the information packet header included in the checksum computation? A corrupted header could cause the data field to be interpreted wrong and thus in essence cause the data to be corrupted. TCP includes a "pseudo header" (source address, destination address, protocol identifier, and length of the TCP segment) in it's checksum calculation to protect against misdelivery from the network layer.

Why isn't a checksum field included in control packets? They can also be corrupted.

1.2.3. XPD3.1 Page 11: Port filters

The term "port filter" seems to be used to describe two different things; a part of the translation map, and a type of address used in an address segment. The difference between these two references must be clarified in the XTP document.

When are port filters (in address segments) used in a packet? There appear to be no references in the protocol to using port filters in packets. When, where, why, and how are these address types used? Port filters as part of the translation map seem to be used extensively. Are the port filters (in address segments) used to help initialize the translation map?

The port filter syntax and address field must be explicitly defined in XPD3.1. Independent hardware and software implementations may not have the luxury of using the P-engine or its address translation circuitry.

In XR3.1[2], page 3, the port filter syntax is explained but several points need to be further explained. Does the *filter_type* correspond to the address field format. XR3.1 states that it corresponds to the address format *type*. The values for the SAME, NEXT, and ACTION codes must be defined. A more detailed explanation of the operation of these commands is needed.

1.3. Command Field Tables

The following tables are an interpretation of legal and illegal XTP command field bit patterns which may or may not be correct. A correct version of these or similar tables would be very useful in the XTP Specification.

---

[2] XR3.1 refers to the XTP Review paper dated May 12, 1988. This paper was written by Dr. Greg Chesson in response to comments on XPD3.1 by Dave Marlow of NSWC.

Table 1: Bits which can be set in any XTP control packet[3]

noerr	end	eom	eob	sreq	multi	res	sup	ext	data	esc	addr	type
•	•			•		•						

Table 2: Bits which can not be set in any XTP control packet

noerr	end	eom	eob	sreq	multi	res	sup	ext	data	esc	addr	type
							•	•	•	•	•	

Table 3: Bits which can be set in an XTP control packet generated by a receiver

noerr	end	eom	eob	sreq	multi	res	sup	ext	data	esc	addr	type
•	•			•		•						

Table 4: Bits which can not be set in an XTP control packet generated by a transmitter

noerr	end	eom	eob	sreq	multi	res	sup	ext	data	esc	addr	type
•	•	•	•	•	•	•						

[3]A • indicates the corresponding bit conforms to the conditions stated in the table description

Table 5: Bit permissions for all XTP packet types[4]

noerr	end	eom	eob	sreq	multi	res	sup	ext	data	esc	addr	type
•	•	•	•	•	•	•	•	•	•	•	•	FIRST
•	•	•	•	•	•	•	•	•	•	•	•	DATAP
•				•			•	•	•	•	•	MREPLY
??	??	??	??	??	??	??	??	??	??	??	??	MAINT
•	•					??						REJ
•	•	•	•	•	??	•						ENQ
•	•	•	•	•	??	•						STATUS
??	??	??	??	??	??	??	??	??	??	??	??	PATH

Table 6: Bits ownership within the XTP command field[5]

noerr	end	eom	eob	sreq	multi	res	sup	ext	data	esc	addr	type
A	X	A	X	X	A	A	A	A	A	A	X	A

1.4. MAINT and PATH packets

As can be seen in table 5, MAINT and PATH packets must be more clearly defined.

1.5. FIRST and DATAP packets

There appears to be no difference between the FIRST and DATAP packets (assuming that the ADDR bit is not eliminated from XPD3.1 for the reasons described below). Can either the FIRST or the DATAP packet type be eliminated?

1.6. XPD3.1 Page 13: XTP Implementation procedures

XPD3.1 states, on page 13, "XTP should be derived from one of the references". XTP should be derived from the XTP Protocol Definition. The references may be used as an implementation example, but not as a definition of the protocol itself.

---

[4] A • denotes a bit which can be set. A ?? denotes that I could not decide whether it was legal to set the bit or not. A blank indicates that it is illegal to set this bit.

[5] Bits which are set by the application only are denoted by an "A". Bits which are set by the XTP only are denoted by an "X". Bits which can be set by both the application and the XTP are denoted by a "B".

1.7. Control block operations

1.7.1. XPD3.1 Page 14: Control block ownership

An ownership exchange protocol for ownership of control blocks must be defined. For example, host software wants to post a command to the XTP. It must first gain ownership of a control block then fill it in with all needed information and parameters. Next, ownership of the control block will be turned over to the XTP which will perform the desired operation. When the operation completes, the XTP will fill in the appropriate fields in the control block (i.e. status) and return ownership of the control block to the host software. How can one tell who owns the control block at any given point in time? How is ownership of the control block passed from the host software to the XTP and vice-versa?

The definition of an ownership exchange protocol would help to answer two related questions: 1) how does the XTP know that the host software has posted a command to it?; and 2) how does the host software know XTP has finished with a command that was posted to it?

1.7.2. XPD3.1 Page 14: Posting multiple commands to a control block

On page 14 of XPD3.1, it is stated that "the host can create new active control blocks by copying the *xx* field from an active control block to another control block". Does this imply that the host software should maintain a queue of control blocks for each context? The queue holds unfinished commands which were posted to the XTP. Should a queue of control blocks containing status replies from the XTP also be built to post command responses to the host software? These queues might serve as the mechanisms necessary to define the ownership exchange protocol explained above.

1.8. Translation Map

The translation map used by XTP must be more completely defined. In XPD3.1, pages 15-16, there is not enough detailed description to build a software implementation of a translation map. Specific areas which need further explanation are described below.

1.8.1. XPD3.1 Page 15: Map objects

XPD3.1, page 15, states "they (map objects) are typically address strings in the format given in section 2.3.1". Is this to say that map objects will consist of a length field, format field, key field, destination network field, destination host field, source net field, source host field, destination port field, and source port field? Or, would the source net, source host, and source port be the only fields needed in the map?

1.8.2. Map Initialization

1.8.2.1. Insertion of objects into the translation table

When are objects, specifically port filters, inserted into the translation table?

1.8.2.2. XPD3.1 Page 15: Map management

XPD3.1, page 15, states "agents external to XTP are assumed responsible for the contents of the map". Who are the agents which will manage this map?

1.8.2.3. XPD3.1 Page 15: Insert primitive

It is stated on page 15 of XPD3.1 that "*Insert(obj, val)* stores an object and an associated value in the data base". Is the value inserted the context identification? This must be clarified in the document.

1.8.2.4. XPD3.1 Page 15: Setmap primitive

The use of the setmap primitive should be documented in XPD3.1. Would setmap be used to multiplex multiple streams on a single context?

1.8.2.5. XPD 3.1 Page 15: Port filters

The use of port filters with the translation map must be thoroughly documented in XPD3.1. Would the type field of an address be used to distinguish an address string object from a port filter object in the translation map when doing a match function?

1.9. XPD3.1 Page 17: Port filter operation

An address filter is referenced in this section. Is an address filter the same as a port filter?

An address filter for (*,KA) is established to recognize return packets from host B. (ID(B),KA) would be sufficient to match return packets from host B. Note: ID(B) refers to the ID field of host B. This filter may be viewed as a subset of (*,KA). Why do you want to match any ID field? Matching any ID field would imply that host A, the transmitter, can establish a multiplexed connection with multiple receivers, say host B and host C, using a common context. (It is assumed that host B and host C reside on the same internet). If this a legal operation in XTP, the protocol used to do this must be defined in the XTP protocol definition document.

Opening a single context with multiple receivers, as described above, might be used in a data collection application where host A, a processor for sensor data, wishes to collect data from redundant sensors, host B and host C. Host A generates a key, KA, creates a context, CA, establishes a port filter (*,KA) linked to CA, and sends a FIRST packet to host B using the key KA. Host A now has a full duplex connection established with host B. Host A then wishes to simultaneously receive data over the same connection as host B from the redundant sensor, host C. Is it sufficient, or valid in XTP, for host A to send a FIRST packet to host C using the key KA to establish such a connection?

Packets to host A from host B and host C should be routed to the context CA. Would the setmap translation map function be used to perform this routing? Would this function aid in doing a "broadcast" to all hosts which a connection has been established using the common key?

1.10. Multicast Operations

1.10.1. XPD3.1 Page 22: Full Duplex Multicast

The multicast capability of XTP must be more clearly defined. XPD3.1 states that an MREPLY packet is syntactically identical to other data packets which would imply that full duplex communications can occur between the multicast transmitter and multicast receiver. The protocol used for full duplex multicast operations must be given.

1.10.2. XPD3.1 Page 22: MREPLY command

XPD3.1 states on page 22 that the use of MREPLY is restricted to single packet transmissions. Does this mean that the receiving application can only give one MREPLY response to each multicast packet Received by the transmitter? The protocol used here must be explained.

1.10.2.1. MREPLY command syntax

Although MREPLY packets are syntactically identical to other data packets, several bits may not be settable in this packet type. Which bits are settable in an MREPLY and which are not should be documented in the XTP protocol definition.

For example, since MREPLYs are restricted to single packet transmissions, setting the EOM or EOB bits in the MREPLY command field would be unnecessary. Similarly, setting the MULTI bit in an MREPLY packet does not appear to be a legitimate operation.

1.11. Eliminating the DATA bit and the ADDR bit

1.11.1. XR3.1 Page 2: The DATA bit

XR3.1, page 2, states that "the DATA bit can be eliminated since it is subsumed by the DATAP type". Wouldn't the DATA bit be needed to send (optional) data with a first packet?

1.11.2. MREPLY and the DATA bit

A multicast receiver may wish to send either a "status" MREPLY packet (an MREPLY packet containing no information segment) or an "information" MREPLY packet (an MREPLY packet containing an information segment). The multicast transmitter would need the DATA bit to determine which packet contained data and which did not contain data. Again the multicast protocol needs to be explained in much greater detail.

1.11.3. Eliminating the ADDR bit

Eliminating the ADDR bit from the specification because an address segment is assumed to be joined to every FIRST packet prevents the user from specifying an address when using a DATAP, MREPLY, or MAINT packet. Since MAINT packets have not been defined in XPD3.1, is it to early to say that they will not need to use the ADDR bit?

1.12. Command bit ownership

Table 6 implies that there are no bits settable by both the application (host) and the XTP. Is this true? On page 7 of XPD3.1, it is stated that "only the ADDR, SREQ, EOB, and END" bits are generated by the XTP output state machine" and "the SUP bit is for supervisory functions." On page 18 of XPD3.1, SREQ is declared to be settable "(1)when either the packet is the last one permitted by the flow control allocation, or (2) the host set the SREQ bit in an output control block command. Should the SUP bit be settable by the XTP output state machine, the application, or both? It may be desirable for the application to request a status packet by setting the SREQ bit. Should SREQ be settable by both the XTP output state machine and the application?

1.13. Context Record Initialization

Although context records are not part of the XTP protocol, their initialization will determine the way the protocol operates. Guidelines must be given for the initialization of the context records.

1.13.1. XPD3.1 Page 18: Context Record Initialization, the ALLOC field, and the DSEQ field

It is stated on page 18 of XPD3.1 that "the output packet sequence number will always lie between x.r.dseq and x.r.alloc". What value should x.r.dseq and x.r.alloc have if you are transmitting to a receiver which has not returned a packet containing an alloc and dseq field? Should x.r.alloc be initialized to an arbitrarily large number? If they are both initialized to zero, the XTP transmitter will not transmit more than the first packet and will have to wait for the receiver to send such a packet. This would not be much more efficient (if it is more efficient) than current transport implementations.

2. Documentation comments and questions

2.1. XR3.1 Page 5: When the ADDR and DATA bits may be set

XR3.1, page 5, declares "ADDR may be set anytime" and "DATA may be set anytime". "ADDR may be set in any information packet" and "DATA may be set in any information packet"? Table 5 shows that ADDR and DATA may be set only in information packets and not in control packets so the bits may not be set "anytime".

2.2. XPD3.1 Page 5: Information segments

XPD3.1 defines an information packet to consist of a common header followed by an information segment. An information segment is defined on page 5 and page 10 of XPD3.1 to consist of a common header followed by an address field, etc. Is the common header included twice in an information packet? If so, why?

2.3. XPD3.1 Page 8: Decoding information segments

XPD3.1 states, on page 8, "ADDR, ESC, and EXT are needed to decode the contents of information segments". Aren't DATA and SUP also needed to do this?

2.4. XPD3.1 Page 8: Keyspace definition

XPD3.1 states, on page 8, "Keyspace is 31 bits. This high-order bit is set by the originator...". Is the high-order bit the 31st bit or the 32nd bit? It seems as though the keyspace is limited to 31 bits so that the 32nd bit may be reserved to designate whether the key was generated locally or not.

2.5. XPD3.1 Page 10: Network addresses

A more complete definition of network addresses is needed. Although this definition may be viewed as being handled by layers above XTP, independent implementations (i.e. the NSWC XTP implementation and the PEI reference implementation) will not be able to communicate without some sort of guideline. For example, what values should used for the destination/source network and destination/source host? Should standard internet values be used? Should host and network tables be defined?

2.6. XPD3.1 Page 16: Sequence number rules reference

In section 3.1.5 of XPD3.1, section 2.1.5 is referenced as giving sequence number ruses. Should this be section 2.1.6?

2.7. XPD3.1 Page 17: Connection refusal

The packets and protocols used for connection refusal (XPD3.1, page 17) must be defined in the specification. The "prepared error message" and the contents of the SUP segment must also be defined.

2.8. XPD3.1 Page 18: Figure 1

Where is Figure 1?

2.9. XPD3.1 Page 20: DLEN field

In SXTP2.0[6] in figure 4-6, a DLEN field is defined to follow the PAD field. In XPD3.1, page 10, the trailer field is defined to follow the PAD field. The DLEN field appears to not exist. In paragraph one on page 20 of XPD3.1, it is stated that "...if present, but excluding*dlen* or any pad bytes". Is this the same DLEN parameter defined in SXTP2.0? Does this field exist? If so, it must be defined.

---

[6] SXTP2.0 refers to the Safenet Express Transfer Layer Protocol Functional Specification Revision 2, 12 January 1988. This document was written by Marc D. Cohn.

Appendix C - Listing of All Questions and Comments in the Review of Revision 3.2

Comments on the XTP Protocol Definition (Revision 3.2)

Philip M. Irey IV

Naval Surface Weapons Center
Dahlgren, Va 22448

1. Protocol Comments and Questions

1.1. XPD3.2[1] Page 2.1: Extension field size

The *extension* field is declared to be a "fixed" size. What is the "fixed" size?? The size should be a multiple of 8 because the trailer must be on a 0 mod 8 boundary.

1.2. XPD3.2 Page 2.1: Pad field boundaries

Pad bytes are defined to align certain fields on 0 mod 8 boundaries. Are these boundaries physical address boundaries, or offsets from the beginning of a packet?

1.3. XPD3.2 Page 2.6: Data segment padding

XPD3.2 states "The data segment is padded with zeroes to be a multiple of 8 bytes". Is the actual data segment padded with zeroes or is the pad field used to make the data+pad field length a multiple of 8 bytes?

1.4. XPD3.2 Page 2.6: Address field and FIRST packets

An address field is declared to be optional. Is an address field optional on a FIRST packet? Is an address only allowed in a FIRST packet? This should be pointed out in the document.

1.5. XPD3.2 Page 2.8: Checksum algorithm

The checksum algorithm and protocol must be better defined in the XTP specification. Questions which must be answered are: 1) when is the checksum reset?; 2) is the checksum kept running on the entire message (i.e. over multiple packets excluding packet headers and trailers) or is it only on a per packet basis? 3) how are REJected packets handled and how is their checksum computed? 4) If packets are received out of order, how is the checksum computed? 5) If packets are received out of order, and a checksum is found to be bad, what should be resent, the entire message, the packet with the bad checksum, or all packets following the packet with the bad checksum?

1.6. XPD3.2 Page 2.9: SNAP header

The LLC encapsulation section (2.4.3) should be expanded. A better explanation of the rationale for choosing the SNAP and how it will be used to do protocol multiplexing should be given. Several questions arise concerning the SNAP packet format.

Should the pad field be placed at the end of the packet to accommodate existing LLC implementations? If this is done, the SNAP header may not line up on an 8 byte boundary.

---

[1] XPD3.2 refers to the XTP Protocol Definition, Revision 3.2.

The type field is declared to be equivalent to the ethernet type field. The ethernet specification states (in appendix B) that types are to be administered by the Xerox Corporation. Is this who would administer address to us?

1.7. XPD3.2 Page 3.2: Control block fields

The fields of a control block should be further explained (i.e. what is transfer, *nptr, *eptr, *dptr, etc).

1.8. XPD3.2 Page 3.3: Control block status

Control block statuses should be more clearly defined. For example, if BUSY means the control block is owned by the controller, how is it signaled that the control block is owned by the host?

The operation of the states can be ambiguous. For example, suppose the XTP owns a control block whose state is BUSY and the host requests a MODIFY operation. When the MODIFY is complete, the XTP will change the status state to ADONE. How does the host know who currently owns the control block if the state is ADONE. Should the host have to do a status operation to see if the control block is still owned by the XTP? This should be explained in the specification.

1.9. XPD3.2 Page 3.4: Context record states

Shouldn't there be a state in a context record for both sending and receiving? State is currently defined for the entire context record.

1.10. XPD3.2 Page 3.4: Control block command field

The function of the cmd field within the control structure of a context record should be explained for both a transmitter and a receiver.

1.11. XPD3.2 Page 3.6: Packet types

In [5], would (DATA,SREQ+END) be a legitimate packet type? If so, this should be added to the documentation.

1.12. XPD3.2 Page 3.7: Sequence number algorithm

XPD3.2 states "Each subsequent sequence number is the value, mod 2**32, of the previous sequence number plus the size in bytes of the previous information segment". In section 2.3 an information segment/packet (is there a difference?) is defined to contain a header, an address field, a data field, a pad field, an extension field, and a trailer field. Should all of these bytes be included in the sequence number?

If one assumes that only the bytes in the data field are included in this count, a question arises on a FIRST packet. Should the bytes from an address be included in the count?

1.13 XPD3.2 Page 3.7: *Oqmax* size

"... the output queue will contain at most *oqmax* bytes". Does the number of bytes include bytes of data, packet headers, packet trailers, addresses, etc?

1.14. XPD3.2 Page 3.7: Event scheduling

The policy used to schedule events after a sender enters the PAUSE state should be documented in the XTP specification. For example, if sender X enters the PAUSE state, while context Y and Z need service. Should context Y and context Z be serviced completely before

context X receives service again? If context Q is established between the servicing of context Y and context Z, should it be serviced before context X is serviced again?

1.15. XPD3.2 Page 3.8: Reciever initialization

If a connection is established with a remote XTP, how, when, and to what state should the receiver portion of an XTP's context record be initialized. Should it be the ACTIVE state?

1.16. XPD3.2 Page 3.8: SYNC protocol

When the sender enters the SYNC state, sends out some ENQ packets, and receives a CONTROL packet with x.r.echo "older" than x.s.sync, should it be discarded? Should the host or XTP be notified? This could be useful management information, i.e. it could show that WTIMER was set too low.

1.17. XPD3.2 Page 3.9 (paragraph 6): ADDRESS delivery

If XTP is truly a transport and network layer protocol, why does it deliver an ADDRESS to a host? Should the host and application program not be concerned with the ADDRESS field?

1.18. XPD3.2 Page 3.9: Command bit overhead

Delivering the command bits to a host for interpretation may require a large amount of host processing for packet decoding. XTP packet formats may also incur a large DMA penalty on the XTP host. These factors may seriously degrade the performance of the XTP "system".

For example, a data structure is being passed between two XTP entities. Packets comprising the data structure are delivered to the host. The host can not simply pass a pointer to the beginning of the XTP buffer area to the application. The host must first calculate the offset from the address segment and if and ESC or SUP segment exists add its length to the offset. Next, the host has to calculate the length of the packet so that the proper amount of data may be DMAed to the application data buffer. The length calculation consists of subtracting the header+trailer length (a constant), the extension field length (if one exists), the pad field length, and the address field length (if one exists) from the length of the packet passed up from the LLC layer. The data from the calculated offset to the calculated offset plus the calculated data length is DMAed from the XTP buffer area to the application data area. This process must be repeated for each packet received.

1.19. XPD3.2 Page 3.9: Context closing conditions

In paragraph 9, three conditions are given for closing a context. Shouldn't a context remain in the closing state for $2\Delta t$. (The Δt referred to in this comment was originally pointed out by Dick Watson of Lawrence Livermore Laboratories. This was discussed at the June 29, 1988 TAB meeting).

1.20. XPD3.2 Page 3.11: MREPLY

Why does an MREPLY advance its event field? The event field is ignored by the MREPLY receiver.

1.21. XPD3.2 Page 3.13: Reciever timers

XPD3.2 states "There are no timers defined for the XTP receiver". Isn't CTIMER enabled for the receiver?

2. Documentation Comments and Questions

2.1. XPD3.2 Page 2.1: Header field

The header field should be removed from the information segment figure (table) on page 2.1 and page 2.6. Studying the figures on page 2.1, one might get the impression that there are two header segments on an information packet.

2.2. XPD3.2 Page 2.2: SREQ bit

It should be pointed out that SREQ can be set by both the host and the XTP (see page 3.8, paragraph 2). The fact that the TYPE bits may only be set the the XTP should also be pointed out.

2.3. XPD3.2 Page 2.2: ENQ packet type

In the section 2.1.2 where the type codes are being defined, ENQ seems to be left out. Has it been left out or has it been dropped as a valid packet type? If it exists, what is its type value?

2.4. XPD3.2 Page 2.3: ESC and EXT bits

It should be pointed out that the ESC bit may only be set in a FIRST packet (if this is true) and that the EXT bit may only be set in an END or EOM packet (if this is true).

If the ESC may only be set in a first packet, this may cause problems to the user who wants to establish a context with another XTP without sending any data in the first packet. The ESC may not be sent in a DATA packet, so the context may not be established until the ESC field is ready for transmission.

2.5. XPD3.2 Page 2.5: Field swapping

The space and interval fields defined in Revision 3.1 of the XTP Specification have been renamed so that their names are indicative of their function. This is good. The fields appear to be swapped, however. Duration which corresponds to interval in Revision 3.1 comes first in a control segment. Swapping fields like this can easily be missed by the implementor. Is there a reason the fields were swapped?

2.6. XPD3.2 Page 2.6: *Burst* size

It should be made clear in the definition of *burst* what is included in the burst size. For example, should the number of bytes specified by burst include the bytes in the header, address field, data segment, trailer, etc?

2.7. XPD3.2 Page 2.6: Boundary locations

In the last paragraph it is stated that the address segment length is rounded up to a multiple of 4, if necessary. Should this be a multiple of 8 to accommodate the 0 mod 8 alignment required on page 2.1?

2.8. XPD3.2 Page 3.6: MAC address description

"B" is used to describe "B"'s MAC address and "A"'s output buffer in the first paragraph of 3.3.1. This can easily confuse the reader.

2.9. XPD3.2 Page 3.7: Incorrect section reference

The reference to 3.2.4 in paragraph 3 seems to be wrong.

2.10. XPD3.2 Page 3.8: Incorrect section reference

The reference to 3.2.2 in paragraph 8 seems to be wrong.

2.11. XPD3.2 Page 3.9: Incorrect section reference

The reference to 3.2.6 in paragraph 4 seems to be wrong.

2.12 XPD3.2 Page 3.9: *dlen.*

Remove all references to *dlen.*

2.13. XPD3.2 Page 3.10: Incorrect section references

The reference to 3.2.3 in paragraph 3 and 3.2.3 in paragraph 4 seems to be wrong.

2.14. XPD3.2 Page 3.11: Incorrect section reference

The reference to section 3.2.3 in paragraph 2 seems to be wrong.

2.15 XPD3.2 Page 3.13: Incorrect section reference

The reference to section 3.2.3 seems to be wrong.

3. General comments

3.1. Packet Length

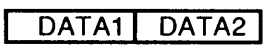
It should be pointed out in the XTP specification, that the length of a packet must be passed up from the physical layer, through the MAC and LLC, to the XTP. XTP needs the length to determine if there is data associated with a first packet.

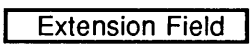
3.2. Buffer Management

The needed capabilities for data movement should be defined in the XTP document. It is not clear how messages are to be assembled by the XTP, delivered to the XTP host, and finally delivered to the application process. What is the division of labor in this process?

For example, suppose the following two packets were delivered from the XTP from the LLC layer.

What XTP would deliver to the host must be defined in the specification. Also, what the XTP host would deliver to the application process must be defined. The application might want to receive pointers to the following data structures:





Appendix D - Listing of All Questions and Comments in the Review of Revision 3.25

Comments on the XTP Protocol Definition (Revision 3.25)

Philip M. Irey IV

Naval Surface Warfare Center
Dahlgren, Va 22448

1. General Comments

1.1. Full Duplex Operations

The method used to perform full-duplex communications must be defined in the XTP Protocol definition. On page 3-6 of XPD3.25[1], section 3.3.1, the process for creating a context is explained. The explanation does not cover all of the steps needed for sender initialization.

A port filter, to recognize control packets from the receiver, must be established as a part of this process. The port filter should be created before any packets are sent to the receiver to ensure optimum performance.

If full-duplex data communications (i.e. both control packets and data packets) is desired, a master/slave relationship exists between the sender and receiver. The sender may be considered a master and the receiver a slave. This should be pointed out in the specification because it defines a specific ordering of events which is not readily apparent.

For example, a conversation is started between node A and node B. The XTP at node A was given a control block with a command to create a new context. A new context is created by the XTP at node A, a new key is generated, and a port filter is set up to recognize packets returned from node B. A FIRST packet is sent to node B. Node B, the slave, should have set up a port filter to recognize packets from node A prior to the sending of the FIRST packet. Node B can not send packets to node A until it has received the FIRST packet from node A, as it doesn't yet know what key to use.

Node B can not send a FIRST packet to Node A. If this were done, a key conflict would occur. Each node could generate a key, yielding two keys, when only one key can be used.

1.2. ESC fields

In our discussion of Revision 3.2 of the Protocol Definition in Seattle, we agreed that an ESC field could only occur in the first packet of a message and an EXT field could only occur in the last packet of a message. I assume this applies to IESC and TESC, also. If it does, it should be presented in the Protocol Definition. If it doesn't, how, when, and why ESC fields are used must be presented.

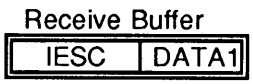
For example, suppose the following 3 packets are received:

P1	IESC	DATA1

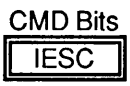
[1] XPD3.25 refers to the XTP Protocol Definition, Revision 3.25.

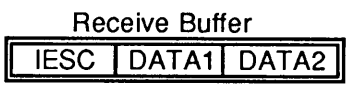
The CMD field delivered to the host for each completed control block and the host's receive buffer memory would appear as follows:

After P1 is received:

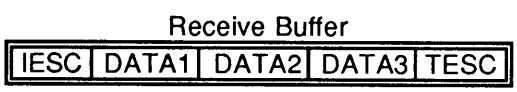


After P2 is received:





After P3 is received:



In this way, the ESC fields can be meaningfully interpreted by the host. If they can occur in any packet, chaos could develop. If the ESC fields have the SUP bit set, should they be delivered to the host buffer area?

1.3. CMD Bit Formats

It would be nice if the header CMD bits and trailer CMD bits occupied disjoint bit positions. In this way, they could be "or'd" together to be delivered to the host. Host interpretation would be easier using this technique.

1.4. Sequence Numbering Scheme and Address Field

Page 3.9 of XPD3.25 states "x.s.rseq is updated by the number of bytes in the information segment including the address field and extension fields if present." Sequence numbers should only cover the data field of the information segment. Including the address field could cause buffering problems for the host. In this case, the address field would have to be delivered to the host in its receive buffer (dseq must be advanced to meet rseq) in front of the message. Why would the host want the address there?

Page 2.8 of XPD3.25 gives a possible reason why this was done: "The address field is present in the data segment of the initial packet of a connection/datagram. This permits the address field to be variable in length and included within the error-corrected information segment." Since the address field may change (i.e. through a gateway or router), it can not really be included in the checksum anyway.

1.5 Control Blocks

The algorithm used to process control blocks at the receiver should be explained in greater detail. It is not clear how the control blocks will be used by the receiver.

Using the example on page 2.6 of XPD3.25, suppose the receiver has 4 outstanding receive control blocks, RCB1, RCB2, RCB3 and RCB4. RCB1 and RCB2 each have buffers of 1024 bytes. RCB3 and RCB4 have buffers of 512 bytes. The data from the packets with sequence numbers 0, 1024, and 2048 will be stored in RCB1, RCB2, and RCB3 respectively. It is assumed that the EOM bit is set in packet 2048. Should the packet with sequence number 2059 (from another message) be stored after the data from packet 2048, or should RCB3 be marked complete and passed to the host and the data be stored in RCB4?

1.6. Sort Field

The XTP scheduling algorithm is based on servicing contexts. A context is serviced until it becomes blocked for allocation. The current context enters the WAIT state and the next context with work to do is selected for service by XTP. If ROBIN is enabled, it determines how much data can be transmitted by a context before all other contexts must be given a chance to transmit. (XPD3.25 does not define how robin can be turned on and off.).

The sort field should allow XTP to service the "most important" object (i.e. the object with the highest sort value) at any given point in time. The service will continue until the object becomes blocked for allocation. If round-robin scheduling is enabled, objects with the same sort value should be serviced in the round-robin order defined by XTP.

In the round-robin scheduling scheme defined by XTP, the objects serviced in round-robin order are contexts, not control blocks. If sort is performed on control blocks instead of contexts, round robin scheduling will no longer work because contexts will be serviced in the pseudo-random order determined by the sort values of their control blocks, not in the round-robin order defined by the contexts themselves. (This is explained in detail in a short paper I wrote for Dave Marlow. If you want a copy, let me know.)

2. Protocol Comments and Questions

2.1. XPD3.25 Page 2.3: Permanent Virtual Circuits

The operation of Permanent Virtual Circuits (packets with the DADDR bit set) must be explained in more detail. Two questions which need to be answered are: how are port filters set up for permanent virtual circuits?; and are these connections only valid within an internet?

2.2. XPD3.25 Page 2.10: EOCB bit

What is the EOCB bit used for? Is this set by a receiver, transmitter, or both? How would a receiver or transmitter respond to receiving a packet with this bit set?

2.3. XPD3.25 Page 2.11: Checksum Algorithm

A number of questions arise about the specification of the XTP checksum algorithm.

Is the checksum computed on a per packet basis or does each packet contribute to a checksum for the entire message? (A per packet checksum will not detect internal gateway errors.) This should be clearly stated in XPD3.25.

When is the checksum reset?

The definition of which bits and fields are included in the checksum calculation must be defined in the Protocol Definition. Care must be taken if more than the data field is included in the checksum. Gateways could change a number of fields so that the checksum would no longer be valid for an entire message. The gateway could calculate a new checksum, but internal gateway errors would no longer be detected. Fields which could change because of gateway fragmentation or coalescing are: SEQ, CMD, KEY, and ROUTE. If these change, why cover them

by the checksum? If these are not included in the checksum, why cover more than the data field of the information segment?

A bit should be defined in the CMD field of each packet to tell whether checksumming was enabled (bit = 1) or disabled (bit = 0) for that packet. In XPD3.25, a zero in the checksum field signifies that checksumming was disabled for that packet. If the checksum changes from a non-zero value to a zero value due to an error, the receiver will assume that checksumming was disabled for that packet and may assume that it was received correctly. If the proposed bit were included in the CMD field, it could be compared with the checksum. If both are zero, the receiver could assume checksumming is disabled more confidently than without the bit. If both are non-zero and the checksum is correct for the packet, the receiver could assume that checksumming was enabled for that packet and that the checksum is correct.

Is the SEQ field included in the checksum? Why is the IESC bit toggled (XORed with 1) in the checksum algorithm?

If there is sufficient concern to cover the header, trailer, and data bits by the checksum, why is there no checksum defined for control packets? Control packets (TYPE=CNTL, not TYPE=CDATA) have headers and data fields (the XTP status information), too.

3. Documentation Comments and Questions

3.1 XPD3.25 Page 2.3: ENQ field

Since the ENQ field has the CNTL bit set, it should be included in the list of packet types with a control segment.

3.2. XPD3.25 Page 2.3: ADDR bit

On page 2.3 it is stated "any packet with the ADDR bit set contains an address segment". The ADDR bit is not defined in the table of #defines on that page. (We know that an address field will always be included in a FIRST packet, but this should be explained in the documentation.)

3.3. XPD3.25 Page 2.5: RREQ bit

A bit called RREQ is referenced on this page. Is this supposed to be SREQ?

3.4. XPD3.25 Page 2.5: Seq field

It should be pointed out in the section on the seq field, that the sequence number is incremented modulo-64 if the XFMT bit is set and the MASK bit is not set.

3.5. XPD3.25 Page 2.10: EOCB and TESC bits

The #define table on this page defines EOCB to be 8 whereas the diagram on this page shows TESC occupying this bit position.

Appendix E - Comments on "Sort"

An End-to-End Priority Mechanism for XTP

Philip M. Irey IV

Naval Surface Warfare Center
Dahlgren, Va 22448

1. Introduction

The XTP Protocol Definition - Revision 3.2 provides several mechanisms: robin, separation, and duration, which affect the scheduling order of the control blocks of XTP contexts. While these mechanisms determine the scheduling order of packets to be transmitted, they do not ensure that the next packet transmitted by the sender is the "most important" packet. These mechanisms also fail to provide the receiver with a measure of the relative importance of a received packet.

The XTP Protocol Definition - Revision 3.25 introduces ordering mechanisms which try to overcome the shortcomings outlined above. *Order* is intended to provide an end-to-end priority between systems. The order value is passed to the destination node during connection establishment. *Sort* is intended to provide a scheduling mechanism which ensures that the next packet sent comes from the "most important" context.

The purpose of this paper is to show the shortcomings of the mechanisms presented in Revision 3.25 and to provide a detailed definition of a modified version of these mechanisms which do not suffer from these shortcomings.

2. Problems the Current Specification of Sort

One of the main differences between the algorithm outlined and the one specified in the XTP Protocol Definition - Revision 3.25 is that contexts are sorted, not control blocks. Sorting control blocks does not achieve the goal of sort, to ensure that the next packet transmitted is the "most important" packet.

Another difference is that in Revision 3.25 sorting is done by creating multiple output queues. Each output queue corresponds to sort level. A packet is placed in the queue corresponding to the sort level of the control block which created the packet. This technique will be shown to fail because the receiver will not receive packets in the proper "sort" order.

2.1 The XTP Scheduling Algorithm

The XTP scheduling algorithm is based on servicing contexts. A context is serviced until it becomes blocked for allocation. The current context enters the WAIT state and the next context with work to do is selected for service by XTP. If ROBIN is enabled, it determines how much data can be transmitted by a context before all other contexts must be given a chance to transmit. (It should be noted that the current XTP Protcol does not define how robin can be turned on and off).

The following notation will be used in analyzing the scheduling algorithms:

c(x) = context number x
cb(x,y,z) = control block number y, from context x, with sort value z
pk(w,x,y,z) = packet number w, from context x, control block y, with sort value z

The output queue will consist of a sequence of packets as follows:

(pk(1),pk(2),...,pk(n))

The output queue is really a composite of sort queues. There exists a queue for each sort value. These queues will be concatenated together to form one output queue so that packets from the queue with the lowest sort value will be placed at the head of the queue with the rest of the packet following in ascending sort order.

Suppose c(1) is currently selected for service. c(1) has control blocks cb(1,1,5), cb(1,2,6), and cb(1,3,3). cb(1,1,5) will be selected first for service under the XTP scheduling algorithm. cb(1,3,3) should have been selected for service, however, because it has the lowest sort value.

Suppose each control block generates one packet. It could be argued that the packets would be ordered properly in the output queue, O. cb(1,1,5) generates pk(1,1,1,5). O = (pk(1,1,1,5)). Next, cb(1,2,6) generates pk(1,1,2,6). O = (pk(1,1,1,5),pk(1,1,2,6)). Next, cb(1,3,3) generates pk(1,1,3,3). O = (pk(1,1,3,3),pk(1,1,1,5),pk(1,1,2,6)). Note that pk(1,1,3,3) has moved to the head of the output queue because it has the lowest sort value. The operation sorts the packets in the proper order for transmission.

Suppose that cb(1,1,5) has a large amount of data to transmit (i.e. one control block generates many packets). cb(1,1,5) generates pk(1,1,1,5) to pk(n,1,1,5). O = (pk(1,1,1,5),...,pk(n,1,1,5)). If n is large, cb(1,2,6) may not be serviced for a long time. If cb(1,1,5) reaches an allocation constraint, say outqp, before completing, packets for this context will be transmitted until pk(n,1,1,5) is placed in O, and there is enough room in O to fit a packet from cb(1,3,3). This violates the goal of sort, to transmit the "most important" packet next. This problem can also be shown to occur with multiple contexts. If c(2) is introduced to this scenario, it will not be serviced until c(1) becomes blocked for allocation even if c(2) has a control block with a sort value less than any in c(1).

XTP packets can be delivered to the wrong buffer area under the Revision 3.25 scheme. Suppose that c(1) has cb(1,1,5) which requires n packets to transmit. i packets have already been transmitted and n-i remain in O, such that O = (pk(i+1,1,1,5),...,pk(n,1,1,5)). pk(n,1,1,5) has the EOM bit set in its CMD field. If cb(1,2,1) is now posted to c(1), and pk(1,1,2,1) is generated, O = (pk(1,1,2,1),pk(i+1,1,1,5),...,pk(n,1,1,5)). The receiver of these packets will place pk(1,1,2,1) at the end of the buffer where pk(1,1,1,5) through pk(i,1,1,5) were placed because it will be sent with sequence number which follows pk(i,1,1,5). pk(i+1,1,1,5) will be placed after pk(1,1,2,1). pk(1,1,2,1) should have been placed after pk(n,1,1,1,5), however.

The receiver can not determine in advance that pk(n,1,1,5) has the EOM bit set to tell XTP to place packets for cb(1,2,1) in a different buffer area. Furthermore, since pk(1,1,2,1) was sent "early" its sequence number indicates that it should be placed after pk(i,1,1,5).

It can be seen that if control blocks are allowed to have different sort values for a given context, the XTP buffering scheme will have to be changed. A simpler solution is to require all control blocks for a given context to use the same sort value. If this is done, control block packetization can not be interrupted by another control block in the same context. Since all control blocks for a given context have the same value, the value could instead be assigned to the contexts and sorting done on the contexts. Xtp contexts can be created and destroyed quickly. If a user wishes to change the sort value for a context, the user can close it down and restart it with a new sort value. If contexts are not flexible enough to easily open and close, can they really be called lightweight and are they significantly better than connections?

3. A Proposal for Sort

During an XTP_connect.request, the user application passes a parameter to the XTP indicating the sort value to be used for that context. This parameter, called sort, will be passed to the XTP in a field of the initial control block for that context. The value will be stored in a sort field in the context record created for the connection. The XTP will place the context record in a queue of context records of the same sort value.

The XTP transmitter will schedule contexts for service by their sort value. The contexts with the lowest sort value will be serviced first. If the lowest sort priority is x, XTP will service all contexts with sort value x. If robin is enabled, the contexts at sort value x will be serviced as a group using the rules governing robin. Otherwise, they will be serviced in a pseudo-random order.

If all contexts with sort value x become blocked (i.e. waiting for allocation in the PAUSE, WAIT or SYNC state), contexts with the next highest sort value will be selected next for service. These contexts will be processed using the scheduling policies described above. This process continues until all contexts at all sort levels have been serviced or until a context with a smaller sort value than the current value becomes unblocked.

If all contexts have been serviced, XTP has no packets to send and can go to sleep.

If contexts with sort value y are being serviced and any context with a sort value smaller than y becomes ready for service, XTP will discontinue service to contexts at sort level y at a clean point and service those with the lowest sort value. Clean points occur after each packet operation is completed. XTP will continue servicing contexts using the scheduling policies described above until all contexts at all sort levels have been serviced.

The scheduling policies for sort will ensure that the contexts with the lowest sort values will always be serviced first.

4. A Proposal for Order

The order field of XTP allows a transmitter to establish the sort value for a context in the receiving node. The transmitter copies the order value passed in the initial control block for the context to the order field in the address segment of the packet sent to establish the connection with the receiver. The receiver uses the order field to determine the sort value it should use for processing the context. The order field is copied into the sort field of the context record and interpreted in exactly the same way as in the transmitter.

Receive context records are processed in sort order using the same algorithms as the transmitter.

Appendix F - XTP Mail Exchanges with Dr. Chesson

****

From @cypress.sgi.CS.NET:greg@sgi.com Thu Jul 7 15:28:15 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa14569; 7 Jul 88 15:07 EDT
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.vjs)
(for pirey@nswc-g.arpa) id AA02773; Thu, 7 Jul 88 12:07:31 PDT
Received: by mojo.sgi.com (5.52/871217.vjs)
(for pirey@nswc-g.ARPA) id AA08375; Thu, 7 Jul 88 12:04:34 PDT
Date: Thu, 7 Jul 88 12:04:34 PDT
From: Greg Chesson <greg@sgi.com>
Message-Id: <8807071904.AA08375@mojo.sgi.com>
Apparently-To: pirey@nswc-g.ARPA
Status: R

Phil,

got your msg this morning.

yes, we have troff -ms - that's how the xtp documents are produced.
hope you receive this.

greg

****

The message below was sent to Dr. Chesson on July 7, 1988 at 15:46.

Greg,

I recieved your message loud and clear. I didn't realize that it would be this easy to exchange mail...it usually isn't. I have enclosed the tables from my review document in case you wanted them. A 3.2 review should follow soon.

Phil (pirey@nswc-g.arpa)

----------cut here----------

```
.KS
.TS
box,center;
c|c|c|c|c|c|c|c|c|c|c|c|c.
n       e       e       e       s       m       r       s       e       d       e       a
        type
o       n       o       o       r       u       e       u       x       a       s       d
e       d       m       b       e       l       s       p       t       t       c       d
r                               q       t                               a               r
r                                       i

_
\(bu    \(bu                    \(bu            \(bu
.TE
.sp
.ce
Table 1:  Bits which can be set in any XTP control packet\u\s-23\s+2\d
.sp
.KE
.KS
.TS
box,center;
c|c|c|c|c|c|c|c|c|c|c|c|c|c.
```

```
n       e       e       e       s       m       r       s       e       d       e       a
        type
o       n       o       o       r       u       e       u       x       a       s       d
e       d       m       b       e       l       s       p       t       t       c       d
r                               q       t                               a               r
r                               i
_                                                       \(bu    \(bu    \(bu    \(bu    \(bu
```

.TE
.sp
.ce
Table 2: Bits which can not be set in any XTP control packet \u\s-23\s+2\d
.sp
.KE
.KS
.TS
box,center;
c|c|c|c|c|c|c|c|c|c|c|c.

```
n       e       e       e       s       m       r       s       e       d       e       a
        type
o       n       o       o       r       u       e       u       x       a       s       d
e       d       m       b       e       l       s       p       t       t       c       d
r                               q       t                               a               r
r                               i
_
\(bu    \(bu                    \(bu            \(bu
```

.TE
.sp
.ce
Table 3: Bits which can be set in an XTP control packet generated by a receiver\u\s-23\s+2\d
.sp
.KE
.KS
.TS
box,center;
c|c|c|c|c|c|c|c|c|c|c|c.

```
n       e       e       e       s       m       r       s       e       d       e       a
        type
o       n       o       o       r       u       e       u       x       a       s       d
e       d       m       b       e       l       s       p       t       t       c       d
r                               q       t                               a               r
r                               i
_
\(bu    \(bu    \(bu    \(bu    \(bu    \(bu    \(bu
```

.TE
.sp
.ce
Table 4: Bits which can not be set in an XTP control packet generated by a transmitter\u\s-23\s+2\d
.sp
.KE
.FS
\u\s-24\s+2\d A \(bu denotes a bit which can be set. A ?? denotes that I could not decide whether it was legal to set the bit or not. A blank indicates that it is illegal

to set this bit.
.FE
.KS
.TS
box,center;
c|c|c|c|c|c|c|c|c|c|c|c|c.

n	e type	e	e	s	m	r	s	e	d	e	a
o	n	o	o	r	u	e	u	x	a	s	d
e	d	m	b	e	l	s	p	t	t	c	d
r				q	t				a		r
r					i						
\(bu	\(bu FIRST	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu
\(bu	\(bu DATAP	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu	\(bu
\(bu	 MREPLY			\(bu			\(bu	\(bu	\(bu	\(bu	\(bu
??	?? MAINT	??	??	??	??	??	??	??	??	??	??
\(bu	\(bu REJ					??					
\(bu	\(bu ENQ	\(bu	\(bu	\(bu	??	\(bu					
\(bu	\(bu STATUS	\(bu	\(bu	\(bu	??	\(bu					
??	?? PATH	??	??	??	??	??	??	??	??	??	??

.TE
.sp
.ce
Table 5: Bit permissions for all XTP packet types \u\s-24\s+2\d
.sp
.KE
.KS
.TS
box,center;
c|c|c|c|c|c|c|c|c|c|c|c|c.

n	e type	e	e	s	m	r	s	e	d	e	a
o	n	o	o	r	u	e	u	x	a	s	d
e	d	m	b	e	l	s	p	t	t	c	d
r				q	t				a		r
r					i						
A	X A	A	X	X	A	A	A	A	A	A	X

.TE
.sp
.ce
Table 6: Bit ownership within the XTP command field \u\s-25\s+2\d
.KE
.FS
\u\s-25\s+2\d Bits which are set by the application only are denoted by an "A".
Bits which are set by the XTP only are denoted by an "X". Bits which can be set by both the application and the XTP are denoted by a "B".
.FE

****

From @cypress.sgi.CS.NET:greg@sgi.com Thu Jul 7 17:18:30 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa15659; 7 Jul 88 17:12 EDT
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.vjs)
(for pirey@nswc-g.arpa) id AA04130; Thu, 7 Jul 88 14:12:21 PDT
Received: by mojo.sgi.com (5.52/871217.vjs)
(for pirey@nswc-g.ARPA) id AA08477; Thu, 7 Jul 88 14:10:33 PDT
Date: Thu, 7 Jul 88 14:10:33 PDT
From: Greg Chesson <greg@sgi.com>
Message-Id: <8807072110.AA08477@mojo.sgi.com>
Apparently-To: pirey@nswc-g.ARPA
Status: R

we're lucky with the mail.

i've been typing in all the action/discussion items i wrote down
at the TAB meeting. i'll send the draft to you guys for comment.

greg

****

The message below was sent to Dr. Chesson on July 15, 1988 at 16:43.

Greg,

Enclosed is a copy of my review of the 3.2 spec. I will mail you (the old fashioned way) a printed copy for the document register in case something screws up in the electronic copy.

I am just putting the finishing touches on an interactive packet builder for XTP packets. The program ask the user to enter each of the bits and fields of an XTP packet and then builds a contiguous XTP packet suitable for transmission. The program will also disassemble incoming packets and display each of the fields to the user. This program was written to test the data structures and subroutines I will use in the XTP implementation.

In writing the "packet builder", I have found that packets require a lot of processing to build and a lot of processing to take apart. This could seriously affect the performance of an implementation. Have you found this to be the case in your software implementation? Is this just part of the normal transport overhead?

Disassembling recieved packets is strange. The packet is processed from left to right up to the data field, and then right to left from the trailer. Calculating the data length is messy. A lot of processing would be saved if a data length field was added in the packet. What do you think?

phil (pirey@nswc-g.arpa)

---------- cut here ----------

```
.TL
Comments on the XTP Protocol Definition (Revision 3.2)
.AU
Philip M. Irey IV
.AI
Naval Surface Weapons Center
Dahlgren, Va 22448
.EQ
delim $$
.EN
.NH 1
Protocol Comments and Questions
.FS
\u\s-21\s+2\d XPD3.2 refers to the XTP Protocol Definition, Revision 3.2
.
.
.
.
...the rest of the 3.2 review document was appended here.  It was deleted from
this section as it appears in the appendicies.
```

****

```
From @cypress.sgi.CS.NET:greg@sgi.com Fri Jul 15 17:08:27 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa20048; 15 Jul 88 17:01 EDT
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.vjs)
        (for pirey@nswc-g.arpa) id AA01803; Fri, 15 Jul 88 14:01:05 PDT
Received: by mojo.sgi.com (5.52/871217.vjs)
        (for pirey@nswc-g.ARPA) id AA16004; Fri, 15 Jul 88 14:00:18 PDT
Date: Fri, 15 Jul 88 14:00:18 PDT
From: Greg Chesson <greg@sgi.com>
Message-Id: <8807152100.AA16004@mojo.sgi.com>
Apparently-To: pirey@nswc-g.ARPA
Status: R
```

Phil,

Thank you for the 3.2 review. I'm printing it out now and will be
studying it over the weekend.

Regarding your experience in building/parsing packets, I haven't
felt that the overhead was too much, but am more than willing to
do things that reduce packet processing. You are right in suggesting
that the trailer design should be improved, but my reasons are
different from yours. The existing trailer should not cause you
any processing problems at all. Here's why: you know that
the lenght of the XTP frame will be a multiple of 8 bytes.
So set a pointer to the trailer which you find by subtracting
8 from the end of the packet. At this point it makes no difference
to a software implementation what order the contents of the trailer
are in. It makes a bit of difference to a hardware implementation,
but you're probably not thinking along those lines. Anyway now

that you know the trailer, you also know the padlength.
Data packet length is just total XTP frame size minus
(16+8+padlen) - (sizeof extension field if present).
I don't think this is too much processing. An earlier version of XTP
had a packet length field where we now have the padlen.
The difference between having a padlen field or a length field
is as follows:

(if we have a padlen field)
then packets can be arbitrarily large
even though we only have a few bits for the padlen.
actually 8 bits might be enough for a padlen.

there is an extra couple of subtracts on the receive
side to figure out the data length.

(if we have a length field)
it has to be large enough to handle future packet sizes,
however large they might be.
it wants to be in the trailer, so we can build packet
headers before knowing the packet length.
it causes some extra arithmetic in the sender,
and removes some arithmetic in the receiver.

I'm interested in knowing what your packet processing overhead is.
For example, I can count assembly language instructions for either
a 68020 or a MIPS cpu for my code to parse or generate a packet
and we can compare numbers.

****

From @cypress.sgi.CS.NET:greg@sgi.com Thu Jul 28 02:56:24 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa25768; 28 Jul 88 2:55 EDT
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.vjs)
(for pirey@nswc-g.arpa) id AA00111; Wed, 27 Jul 88 23:55:20 PDT
Received: by mojo.sgi.com (5.52/871217.vjs)
(for pirey@nswc-g.ARPA) id AA08178; Wed, 27 Jul 88 23:54:04 PDT
Date: Wed, 27 Jul 88 23:54:04 PDT
From: Greg Chesson <greg@sgi.com>
Message-Id: <8807280654.AA08178@mojo.sgi.com>
Apparently-To: pirey@nswc-g.ARPA
Status: R

considering some changes in XTP header and trailer.
do you have a moment for comment?

g

****

From @cypress.sgi.CS.NET:greg@sgi.com Thu Jul 28 03:08:52 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa25815; 28 Jul 88 3:02 EDT
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.vjs)
(for pirey@nswc-g.arpa) id AA00208; Thu, 28 Jul 88 00:02:45 PDT
Received: by mojo.sgi.com (5.52/871217.vjs)
(for pirey@nswc-g.ARPA) id AA08185; Wed, 27 Jul 88 23:54:54 PDT

Date: Wed, 27 Jul 88 23:54:54 PDT
From: Greg Chesson <greg@sgi.com>
Message-Id: <8807280654.AA08185@mojo.sgi.com>
Apparently-To: pirey@nswc-g.ARPA
Status: R

changes would go as follows:

eseq is out (we knew that)
event(2) is out and replace by life(2)
which is packet lifetime counter
there would be two key fields: xkey and pkey.
xkey is end-to-end
pkey is internal to gateways and is important
for reducing table space requirements and keeping
track of aggregate rate control.

The RES bit is eliminated from the cmd word.
It is a connection "mode" and is moved to a new MODE field
that appears in the address field of a FIRST packet.
The MULTI bit is also eliminated from the cmd word
and is moved to the new MODE field along with RES.

There's a new bit defined in the cmd field for end-of-control-block.

question:
do you still think that having a packet length field is easier
or better than having the pad length? I've been experimenting
with my routines, trying it both ways.

greg

****

The message below was sent to Dr. Chesson on July 28, 1988 at 18:02.

Greg,

Sorry that I didn't respond sooner but things here are a mess. I have included a portion of the code that I use to generate pointers to fields in a packet. I have implemented all of this code as macros. This was done to be fast and flexible. The source code generated for this is also included.

Normally, very little code is generated. As shown below, CMD_OFFSET, CMD_PTR, EVENT_OFFSET..., generate very little code. Generally just a few instructions.

***** Macro Definitions *****

```
#define CMD_OFFSET 0
#define CMD_PTR ((CMD_TYPE *)(buffer + CMD_OFFSET))

#define EVENT_OFFSET (CMD_OFFSET + sizeof(CMD_TYPE))
#define EVENT_PTR ((EVENT_TYPE *)(buffer + EVENT_OFFSET))

#define KEY_OFFSET (EVENT_OFFSET+sizeof(EVENT_TYPE))
#define KEY_PTR ((KEY_TYPE *)(buffer + KEY_OFFSET))
```

***** Generated code *****

```
* 16        i = CMD_OFFSET;
          CLR.L  (A4)
* 17        j = CMD_PTR;
          MOVE.L        A2,A0
          MOVE.L        A0,((.j).L)
* 18        i = EVENT_OFFSET;
          MOVEQ         #2,D0
          MOVE.L        D0,(A4)
* 19        j = EVENT_PTR;
          MOVE.L        A0,D2
          ADDQ.L        #2,D2
          MOVE.L        D2,((.j).L)
* 20        i = KEY_OFFSET;
          MOVEQ         #4,D0
          MOVE.L        D0,(A4)
* 21        j = KEY_PTR;
          MOVE.L        A2,A0
          MOVE.L        A0,D2
          ADDQ.L        #4,D2  <---------- Note the constant folding done by the compiler.
          MOVE.L        D2,((.j).L)       (Without this, I could have never made my
                                          implementation very flexible. This is done
                                          automatically by the compiler.  All of the
                                          generated code in this letter was not compiled
                                          using full optimization.)
```

A lot of code is generated for the DATA and ESC macros as shown below:

***** Macro Definitions *****

```
/*
  these macros assume that the global variable buffer points to the beginning of the
  buffer of interest.
*/

#define DATA_OFFSET (ADDR_OFFSET + (IS_first_packet ? (*ADDR_LENGTH_PTR) : 0) +
(ESC_BIT_SET ? sizeof(struct XTP_esc_field) : 0))
#define DATA_PTR ((DATA_TYPE *)(buffer + DATA_OFFSET))

#define ESC_OFFSET ((IS_first_packet) ? (ADDR_OFFSET + *ADDR_LENGTH_PTR) :
ADDR_OFFSET)
#define ESC_PTR ((struct XTP_esc_field *)(buffer + ESC_OFFSET))
#define ESC_FOR_SUP(esc_ptr) (esc_ptr->high & (1 << ((8*sizeof(XTP_esc_field_htype)) - 1)))
```

***** Generated Code *****

```
* 68        i = DATA_OFFSET;
          MOVEQ         #0,D1
          MOVE.W        (A2),D1         *fr
          MOVE.L        D1,D0
          MOVEQ         #15,D3
          AND.L  D3,D0
          MOVEQ         #1,D1
          CMP.L  D0,D1
```

```
        BNE.S _L145
        MOVEQ       #0,D0
        MOVE.W      16(A2),D0       *fr
        MOVE.L      D0,D3
        BRA.S _L143

_L145:
        MOVEQ       #0,D3
_L143:
        MOVEQ       #0,D1
        MOVE.W      (A2),D1         *fr
        MOVE.L      D1,D0
        ANDI.L      #256,D0
        BEQ.S       _L142
        MOVEQ       #8,D2
        BRA.S _L140

_L142:
        MOVEQ       #0,D2
_L140:
        MOVE.L      D3,D0
        MOVEQ       #16,D1
        ADD.L  D1,D0
        ADD.L  D2,D0
        MOVE.L      D0,(A4)
* 69       j = DATA_PTR;
        MOVEQ       #0,D1
        MOVE.W      (A2),D1         *fr
        MOVE.L      D1,D0
        MOVEQ       #15,D3
        AND.L  D3,D0
        MOVEQ       #1,D1
        CMP.L  D0,D1
        BNE.S _L139
        MOVEQ       #0,D0
        MOVE.W      16(A2),D0       *fr
        MOVE.L      D0,D2
        BRA.S _L137

_L139:
        MOVEQ       #0,D2
_L137:
        MOVEQ       #0,D1
        MOVE.W      (A2),D1         *fr
        MOVE.L      D1,D0
        ANDI.L      #256,D0
        BEQ.S       _L136
        MOVEQ       #8,D0
        BRA.S _L134

_L136:
        MOVEQ       #0,D0
_L134:
        MOVEQ       #16,D1
        ADD.L  D1,D2
```

```
        ADD.L  D0,D2
        MOVE.L      A2,A0
        MOVE.L      A0,D4
        ADD.L  D4,D2
        MOVE.L      D2,((.j).L)
* 70       i = ESC_OFFSET;
        MOVEQ       #0,D1
        MOVE.W      (A2),D1         *fr
        MOVE.L      D1,D0
        MOVEQ       #15,D3
        AND.L  D3,D0
        MOVEQ       #1,D1
        CMP.L  D0,D1
        BNE.S  _L133
        MOVEQ       #0,D0
        MOVE.W      16(A2),D0       *fr
        MOVEQ       #16,D1
        ADD.L  D1,D0
        BRA.S  _L131

_L133:
        MOVEQ       #16,D0
_L131:
        MOVE.L      D0,(A4)
* 71       j = ESC_PTR;
        MOVEQ       #0,D1
        MOVE.W      (A2),D1         *fr
        MOVE.L      D1,D0
        MOVEQ       #15,D3
        AND.L  D3,D0
        MOVEQ       #1,D1
        CMP.L  D0,D1
        BNE.S  _L130
        MOVEQ       #0,D0
        MOVE.W      16(A2),D0       *fr
        MOVEQ       #16,D1
        ADD.L  D1,D0
        BRA.S  _L128

_L130:
        MOVEQ       #16,D0
_L128:
        MOVE.L      A2,A0
        MOVE.L      A0,D2
        ADD.L  D0,D2
        MOVE.L      D2,((.j).L)
```

This seems to be a lot of code. If you want to see more, I can send you the rest of my macros.

And now on to your new stuff...

I have the following questions about your proposals:

If the event field is replaced by a life field, how are events counted? Has the

event field moved or is it no longer needed? If it is no longer needed, why not (i.e. how will one determine if an event packet is missing?)?

Will xkey replace the current key field? If not, what will it be used for? What are the sizes of xkey and pkey? Will the common header expand to 32 bytes? What is the function of the pkey and how will it be used? Is pkey reserved for gateways and routers to play with and its use not really part of the XTP spec? Will the xkey and pkey fields be duplicated in the address field?

Why is the key field duplicated in the address field of a first packet? Is it to help the P-Engine hardware?

Since the address field is now 52 bytes long will you stuff the mode field and , xkey and pkey fields (if they go in the address field) in the remaining 4 bytes (0 mod 8 boundary)? Having the MODE field in the address field forces one to establish the "MODE" of communications when establishing a connection. The "MODE" can not be changed on the fly. Is this too restrictive? One might want to establish a connection in non-reservation mode and then at some future time, say when the system becomes heavily loaded, enable reservation mode.

What is an end of control block???? What is it used for? When you say it will be in the CMD field, do you mean the CMD field of a packet or of a control block?

As to your question concerning packet length versus pad length, let's look at the how I calculate the size of the data field of a received packet. Currently, the length of the received packet must be passed up from the LLC layer. The data field ends where the pad field (if one exists) begins. So, the length of the data field may be calculated by subtracting the offset of the pad field from the offset of the data field (note: offsets are from the beginning of the packet).

To calculate where the pad field begins, one must subtract the pad field length of the received packet from the extension field offset, if there is an extension field, or the trailer offset if there is not an extension field. The trailer offset is the end of the received packet - the sizeof(trailer_field). The extension offset is the trailer offset - the sizeof(extension_field). I think where the most code is generated is in compensating for the optional fields and the 0 mod 8 boundary requirements. If a data_size field was present, a simple memory reference would take place to calculate the size of the data field.

I have noticed that received packets are disassembled from right to left and transmitted packets are built from left to right (at least from the data field to the trialer). This forced me to build two sets of macros, those beginning with an R are for received packets, and those beginning with an S are for sending packets. Having a data length field would allow packets to be disassembled from left to right so one set of macros could be used.

I agree in your July 15 note that a length field has to be long enough to accomadate future packet sized. This could be a tough number to calculate. You also said that it wants to live in the trailer. Why is this?

Dave Marlow heard a rumor from someone at NSA that the 3.3 Revision of the XTP Spec is out. Is this true? If so, where can we get one?

Is it possible to get an electronic copy of the XTP spec/specs? I think that it would make the review process easier for me in that I could grep for what I was looking for

instead of leafing back and forth in the document.

phil (pirey@nswc-g.arpa)

```
***** Macro Definitions *****

#define RTRAILER_OFFSET (buffer_cb.packet_len - sizeof(struct XTP_trailer))
#define RTRAILER_PTR ((struct XTP_trailer *)(buffer + RTRAILER_OFFSET))

#define RPLEN_OFFSET (RTRAILER_CMD_OFFSET + sizeof(TRAILER_CMD_TYPE))
#define RPLEN_PTR ((PLEN_TYPE *)(buffer + RPLEN_OFFSET))

#define REXTENSION_OFFSET (RTRAILER_OFFSET - sizeof(struct XTP_extension))
#define REXTENSION_PTR ((struct XTP_extension *)(buffer + REXTENSION_OFFSET))

/*
  these macros assume that the global variable buffer points to the beginning of the
  buffer of interest.
*/

#define RPAD_OFFSET ((EXT_BIT_SET ? REXTENSION_OFFSET : RTRAILER_OFFSET) -
*(PLEN_TYPE *)(buffer + RPLEN_OFFSET))
#define RPAD_PTR ((PAD_TYPE *)(buffer + RPAD_OFFSET))

#define RPAD_LENGTH (*(PLEN_TYPE *)(buffer + RPLEN_OFFSET))
#define RDATA_LENGTH (RPAD_OFFSET - DATA_OFFSET)

***** Generated Code *****
* 101       i = RDATA_LENGTH;
        MOVEQ       #0,D1
        MOVE.W      (A2),D1         *fr
        MOVE.L      D1,D0
        ANDI.L      #512,D0
        BEQ.S       _L16
        MOVE.L      44(A3),D4
        MOVEQ       #16,D0
        SUB.L  D0,D4
        BRA.S _L14

_L16:
        MOVE.L      44(A3),D4
        SUBQ.L      #8,D4
_L14:
        MOVEQ       #0,D1
        MOVE.W      (A2),D1         *fr
        MOVE.L      D1,D0
        MOVEQ       #15,D3
        AND.L  D3,D0
        MOVEQ       #1,D1
        CMP.L  D0,D1
        BNE.S _L13
        MOVEQ       #0,D0
        MOVE.W      16(A2),D0       *fr
        MOVE.L      D0,D2
```

```
        BRA.S _L11

_L13:
        MOVEQ       #0,D2
_L11:
        MOVEQ       #0,D1
        MOVE.W      (A2),D1         *fr
        MOVE.L      D1,D0
        ANDI.L      #256,D0
        BEQ.S       _L10
        MOVEQ       #8,D5
        BRA.S _L8

_L10:
        MOVEQ       #0,D5
_L8:
        MOVE.L      D4,D0
        MOVE.L      A2,A0
        MOVE.L      44(A3),D1
        MOVEQ       #-2,D3
        ADD.L  D3,D1
        MOVE.L      D1,A5
        MOVEQ       #0,D1
        MOVE.W      0(A5,A0.L),D1  *fr
        SUB.L  D1,D0
        MOVEQ       #16,D1
        ADD.L  D1,D2
        ADD.L  D5,D2
        SUB.L  D2,D0
        MOVE.L      D0,(A4)
```

****

From @cypress.sgi.CS.NET:greg@sgi.com Fri Jul 29 21:28:54 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa18397; 29 Jul 88 21:28 EDT
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.vjs)
        (for pirey@nswc-g.arpa) id AA17039; Fri, 29 Jul 88 18:28:17 PDT
Received: by mojo.sgi.com (5.52/871217.vjs)
        (for pirey@nswc-g.ARPA) id AA10162; Fri, 29 Jul 88 18:27:24 PDT
Date: Fri, 29 Jul 88 18:27:24 PDT
From: Greg Chesson <greg@sgi.com>
Message-Id: <8807300127.AA10162@mojo.sgi.com>
Apparently-To: pirey@nswc-g.ARPA
Status: R

phil,

here is a quick response to yesterday's email.

What language are you programming in?

I don't think you need the DATA and ESC macro's in a real
implementation. I agree with you that the code produced seems
excessive.

re your questions.
About events. I'm considering using only the sequence number to detect missing data. When we got rid of eseq, it was understood that we were giving up the ability to accept out of order data among multiple events. Since that is true, we no longer need to know that we've skipped an event - it is sufficient to know that we skipped some data and to have it retransmitted. This can be done with just the sequence number.

XKEY and PKEY are both 32 bits. The header is still 16 bytes.
The next header size, if we have to expand, is 24 bytes.
Both XKEY and PKEY are part of the spec. However, for communications between nodes on the same LAN only XKEY is needed and it has the same meaning as KEY presently has. In order to go through gateways, the sender must also generate PKEY. Gateways and routers use PKEY the way they used to use XKEY, although there are differences in the algorithm that reduce memory usage by a lot.

The MODE field does indeed go in the address field. And you're right - if we don't do anything else it would not be possible to switch into or out of reservation mode on a dynamic basis. It probably should be made possible to change MODE on the fly. However, this usually causes horrible problems. We're talking about a mode (RES) that is easy to change in and out. Suppose we add a mode that is not so easy to change. Then would be have to discriminate between the modes that can change and the ones that can't, and have the code check for it?

End-of-control-block (EOCB) is a bit in the CMD field of an XTP header. It is filled in when we transmit the last packet for a control block. Among other things this bit would tell us when to reset the checksum. Remember the conversation about checksums we had in Seattle?

I'm still experimenting with length fields and trailers.
The length or padlength field want to be in the trailer for several reasons - all of them related to hardware. It is important (for low latency and real-time reasons) to be able to start shifting data out to the network on short notice.
Right now all we have to do is fill in the sequence number and command field and let it go. This gives us enough time to start the data going right behind the header. But it doesn't give enough time to figure out exactly how much data is going to go and to fill in that amount in the header. So we append the amount to the trailer as it goes out. In software it doesn't seem to make much difference if the information is at the beginning or end of the packet because we typically have to build the entire packet image before sending and we receive the entire packet image before processing it.

It should be clear that the checksum goes at the trailing end of the packet.

The rumor about 3.3 is disinformation. Somebody probably heard that I'm creating 3.3 and it will be distributed at the next TAB.

You can certainly have my troff source for the XTP document.
You will need a PostScript printer to print it out exactly the

way I have it. I'd like to send it via email. It will have
to go in pieces.

g

****

From @cypress.sgi.CS.NET:greg@sgi.com Wed Aug 3 17:53:27 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa23646; 3 Aug 88 15:05 EDT
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.vjs)
(for pirey@nswc-g.arpa) id AA02267; Wed, 3 Aug 88 11:10:07 PDT
Received: by mojo.sgi.com (5.52/871217.vjs)
(for pirey@nswc-g.ARPA) id AA14058; Wed, 3 Aug 88 11:09:20 PDT
Date: Wed, 3 Aug 88 11:09:20 PDT
From: Greg Chesson <greg@sgi.com>
Message-Id: <8808031809.AA14058@mojo.sgi.com>
Apparently-To: pirey@nswc-g.ARPA
Status: R

Phil,

Would you mind considering the following experiment?
See what happens to your macro's if you replace the padlen field
by a length field (32 bits). Parsing rules remain the same:
trailer is aligned on a mod-8 boundary. The length field
would tell the amount of user data.

I've included some routines that I use for printing out packets.
Note where "dlen" is set. The subtract operations would be replaced
by a single assignment if we switch to a length field.
Nothing else would change. I claim that software is not affected
very much by whether we have a length field or a padlength field.
However, the code produced by macro's like yours would be affected
because the length adjustment arithment is included in almost
every macro whereas software routines like the ones below are
careful to do the arithmetic just once. Of course it is possible
to make trickier macro's that do the length calculations once
and save the results in temporaries.

Then there is the issue of what the output side looks like.
Were you guys involved in the AEGIS evaluation, or do you work
pretty much on Safenet?

greg

```
char *_typenames[] ={
        "DATA",                 /* 0 */
        "FIRST",        /* 1 */
        "MREPLY",       /* 2 */
        "MAINT",        /* 3 */
        "PATH",                 /* 4 */
        "bad", "bad", "bad",    /* 5, 6, 7 */
        "REJ",          /* 8 */
        "bad",
```

```
        "CNTL",                 /* a */
        "bad",          /* b */
        "bad",          /* c */
        "bad",          /* d */
        "bad",          /* e */
        "bad",          /* f */
};

mpcmdbits(m, bits)
char *m;
{
        printf("%s", m);
        pcmdbits(bits);
        printf("\n");
}

pcmdbits(bits)
{
register t;

        t = 0;
        if (bits&SREQ) {
                printf("SREQ");
                t++;
        }
        if (bits&EOB) {
                if (t) printf("|");
                printf("EOB");
                t++;
        }
        if (bits&EOM) {
                if (t) printf("|");
                printf("EOM");
                t++;
        }
        if (bits&END){
                if (t) printf("|");
                printf("END");
                t++;
        }
        if (bits&EOCB) {
                if (t) printf("|");
                printf("EOCB");
                t++;
        }
        if (bits&ESC) {
                if (t) printf("|");
                printf("ESC");
        }
}


pheader(h)
struct header *h;
{
```

```
register bits, type;

        bits = h->cmd & ~0xf;
        type = h->cmd & 0xf;


        printf("%s ", _typenames[type]);
        if (bits) {
                pcmdbits(bits);
        }
        printf(" S(%d)", h->seq);
        printf(" K(%x)", h->key);
}

mpheader(m, h)
char *m;
{
        printf("%s ", m);
        pheader(h);
        printf("\n");
}

ppacket(h, plen)
struct header *h;
int plen;                       /* packet length */
{
struct trailer *tp;
struct ext *ex;
int i, j, dlen;

        switch(h->cmd&0xf) {
        case FIRST:
        case DATA:
        case MREPLY:
        case MAINT:
        case PATH:
                i = j = (int)h;
                i += plen;
                i -= TRAILERSIZE;
                if (i&7) {
                        printf("non-aligned trailer\n");
                        return;
                }
                tp = (struct trailer *)i;
                h->cmd |= tp->cmd;
                pheader(h);
                j += HEADERSIZE;
                dlen = i - j - tp->padlen;   /* or, dlen = tp->length */
                if (h->cmd & EXT) {
                        i -= EXTSIZE;
                        ex = (struct ext *)i;
                        /* pext(ex) */
                }
                printf(" pad %d ", tp->padlen);
                printf(" dlen %d ", dlen);
```

```
                break;
        case REJ:
        case CONTROL:
                pheader(h);
                break;
        }
}
```

****

The message below was sent to Dr. Chesson on October 20, 1988 at 15:18.

Greg,

How are things going? I haven't talked to you in some time, so I decided to drop you a line. I have heard that there might be a 3.2 update paper. I would like to get a copy if this document becomes available. I have an XTP question for you...

Robin, alloc, burst, duration, and separation are parameters used to control a transmitter. Robin is a parameter set locally by the transmitter. Alloc, burst, duration, and separation are different, however. Initially they are set by the transmitter in the prototype context passed to XTP at system initialization time. They can then be filled in with the values returned by the receiver in a control packet.

If a receiver would like to use alloc as the only parameter to restrict the transmitter, it must have the capability to turn off the burst, duration, and separation restrictions.

If a receiver would like to use burst and duration to limit the transmitter, but wants to put no restriction on the separation between packets, it must have the capability to turn off separation.

Values should be defined in the specification which indicate that these parameters are turned off. For instance, a -1 value for burst, duration, or separation could indicate that these mechanisms are not being used.

An example where this feature might be useful is in a receiver which does not know at what rate it can receive data. Initially, it turns off burst, duration, and separation and times how long it takes to process a packet. When the packet processing time is known, the receiver may be able to determine the rate at which it can receive data and can throttle the transmitter appropriately.

Is there a way to turn the rate control mechanisms off currently defined in the 3.2 spec? If not, do you think that this would be a useful feature?

phil (pirey@nswc-g.arpa)

****

The message below was sent to Dr. Chesson on November 8, 1988 at 20:15.

Greg,

The note below is a copy of some email that I sent you on October 20. I am resending it in case you didn't get the first copy. I have the 3.25 spec and am reviewing it now. WOW!!! I guess the packet structure has changed slightly. I am writing a paper on the sort and order fields. I will send you a copy when I get it finished.

I am stuck on a point right now. Can Robin be turned on and off? If so, how? I go into this question a little more below.

Were your workstations virusproof?

thanks,
phil (pirey@nswc-g.arpa)

p.s. Congratulations on your engagement!!!!

----------other note below----------

Greg,
How are things going? I haven't talked to you in some time, so I decided to drop you a line. I have heard that there might be a 3.2 update paper. I would like to get a copy if this document becomes available. I have an XTP question for you...

Robin, alloc, burst, duration, and separation are parameters used to control a transmitter. Robin is a parameter set locally by the transmitter. Alloc, burst, duration, and separation are different, however. Initially they are set by the transmitter in the prototype context passed to XTP at system initialization time. They can then be filled in with the values returned by the receiver in a control packet.

If a receiver would like to use alloc as the only parameter to restrict the transmitter, it must have the capability to turn off the burst, duration, and separation restrictions.

If a receiver would like to use burst and duration to limit the transmitter, but wants to put no restriction on the separation between packets, it must have the capability to turn off separation.

Values should be defined in the specification which indicate that these parameters are turned off. For instance, a -1 value for burst, duration, or separation could indicate that these mechanisms are not being used.

An example where this feature might be useful is in a receiver which does not know at what rate it can receive data. Initially, it turns off burst, duration, and separation and times how long it takes to process a packet. When the packet processing time is known, the receiver may be able to determine the rate at which it can receive data and can throttle the transmitter appropriately.

Is there a way to turn the rate control mechanisms off currently defined in the 3.2 spec? If not, do you think that this would be a useful feature?

phil (pirey@nswc-g.arpa)

****

From @cypress.sgi.CS.NET:greg@sgi.com Tue Nov 8 21:19:15 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa20002; 8 Nov 88 21:17 EST
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.SGI)
(for pirey@nswc-g.arpa) id AA05469; Tue, 8 Nov 88 18:17:19 PST
Received: by mojo.sgi.com (5.52/871217.vjs)
(for pirey@nswc-g.ARPA) id AA10096; Tue, 8 Nov 88 18:10:57 PST
Date: Tue, 8 Nov 88 18:10:57 PST
From: Greg Chesson <greg@sgi.com>
Message-Id: <8811090210.AA10096@mojo.sgi.com>
Apparently-To: pirey@nswc-g.ARPA
Status: R

The robin field should be capable of being turned off, although
it hasn't been discussed. One way to do it is to say that it is
unsigned. So "-1" is the biggest possible number. If we assume
that we would never have that much to transmit in one burst,
then robin would not restrict output in any way.

The 3.25 packet format changes are not quite final.
You can pretty much believe the header, although the exact
encoding of the cmd word is being finalized. The trailer
and trailing escape field are being studied right now.
We are planning on having a final design for the 3.3 spec
by early next week.

I will email to you the C structures and defines that specify
the header and trailer formats that we use for your use in preparing
for our testing.

Yes, our workstations were virusproof because the virus could
only run on Suns and Vaxen. We've considered fixing the bugs
in the virus and porting it to our machines. haha.

g

****

The message below was sent to Dr. Chesson on November 30, 1988 at 17:03.

Greg,

Here is a copy of my review of the 3.25 Protocol Definition.
If you would respond to me directly about comment 1.5 (control blocks),
1.6 (sort field), and 2.2 (EOCB bit), I would be able to proceed more
confidently in my implemenation. I will send a hardcopy to PEI.

When is 3.3 due to be released?

I just received your designers notebook. I'm printing it out now.
I briefly glanced over it. I still don't understand how XTP scheduling
will be done with multiple output queues. Will ROBIN be eliminated?
My comment 1.6 touches on this.

thanks,
phil

---------- 3.25 Review Below ----------

.TL
Comments on the XTP Protocol Definition (Revision 3.25)
.AU
Philip M. Irey IV
.AI
Naval Surface Warfare Center
Dahlgren, Va 22448
.NH 1
General Comments
.FS
\u\s-21\s+2\d XPD3.25 refers to the XTP Protocol Definition, Revision 3.25
.FE
.NH 2
Full Duplex Operations
.PP
The method used to perform full-duplex communications must be
defined in the XTP Protocol definition. On page 3-6 of
XPD3.25\s-2\u1\d\s+2, section 3.3.1,
.
.
.
.
.
...the rest of the 3.25 review document was appended here. It was deleted from
this section as it appears in the appendicies.

****

From @cypress.sgi.cs.net:greg@sgi.com Mon Dec 5 20:37:08 1988
Received: from cypress.sgi.cs.net by SH.CS.NET id aa13388; 5 Dec 88 17:59 EST
Received: from mojo.sgi.com by sgi.sgi.com (5.52/880418.SGI)
(for pirey@nswc-g.arpa) id AA05577; Mon, 5 Dec 88 14:48:17 PST
Received: by mojo.sgi.com (5.52/871217.vjs)
(for dmarlow@nswc-g.ARPA) id AA07385; Mon, 5 Dec 88 14:48:18 PST
Date: Mon, 5 Dec 88 14:48:18 PST
From: Greg Chesson <greg@sgi.com>
Message-Id: <8812052248.AA07385@mojo.sgi.com>
Apparently-To: dmarlow@nswc-g.ARPA
Apparently-To: pirey@nswc-g.ARPA
Status: R

phil,

Thanks you for your comments on 3.25 and the time you took
to prepare them. Here are some answers.

1.1 Full Duplex
I'll try to explain it better. However Node B !can! send a FIRST

packet to Node A in your example -- and it can even choose the same
key that Node A used. That's because the initiator uses the key, K,
and the responder uses K' (i.e. K with the top bit on).

1.2 ESC fields
People have been confused about escape fields, thinking they were
some kind of way to extend the protocol. As you know, they are
for the user and are opaque to XTP. I renamed them TAG fields.
There is the BTAG field at the beginning of an information
segment and the ETAG field which is at the end. We got rid of
the SUP field. So now we have the information segment and two
bits, one in the header and one in the trailer, that indicate
if the data is "tagged" by the user. The BTAG bit goes in the header.
The ETAG bit goes in the trailer. I think this is a simplification
of the previous situation.

1.3 Cmd bit formats
What we do is concatenate the two 16 bit fields to make a 32-bit quantity
instead of "oring" them together. Whenever XTP needs to deliver some bitflags
to the user, it's pretty easy.

1.4 Sequence Numbering and Address Field
Including the address field in the information shouldn't cause
problems for the host. Lots of times the host application wants to know
the address of the remote host. Actually copying the address field
to the host application should probably be an option.
Delivering it across the network to the receiving XTP buffer
seems to be ok.

I don't expect the address field would be changed by a router.
However, if we had a router that wanted to do that,
it would calculate the effect of the change on the check function
and change the check value.

It may not be in the protocol definition yet, but we allow the initial
sequence number to be anything. What I've been doing is have the
sequence number of the data begin at zero. The sequence number of
the first address byte is (zero-sizeof(address)). So the sequence
number in the FIRST packet is "negative". If a router needed
to diddle the address segment, it could do it by growing or shrinking
the bytes before byte zero.

1.5 Control Blocks
CB's and how they fit in the definition are still not helping
people understand the protocol. Do you have a suggestion?
Suppose we delete control blocks from the document and replace
by something that resembles an ISO Service Definition?

1.6 Sort Field and Robin
The "robin" parameter must not be confused with round-robin scheduling.
I should rename it to something else like "burstsize".
The idea is that whenever a context is being served by the output queue
mechanism, it is limited by a number of things: flow control (alloc),
rate control (credit), and channel anti-hog control (robin/burstsize).
The actual amount of data transferable to the network at any instant

is the minimum of these three quantities (assuming that data is available).
One conclusion from this is that the robin/burstsize parameter does
not directly affect scheduling.

As you point out, the output scheduling algorithm
might work better on lists of control blocks, or i/o requests, and not
necessarily contexts. Actually it does work on contexts if we say that
all of the cb's associated with a context have the same sort value.
In fact if we do not have all the output cb's on a context at the
same sort value, then there are serious problems with keeping sequence
numbers straight. Suppose a later control block has a higher-"priority"
than an earlier one. If the later control block is transmitted first,
then we get into a deadlock trying to transmit the earlier control block
which hasn't been sent yet.

I don't consider this issue to be a serious problem for a number of
reasons. But one proposal that I like a lot is to reduce the role
of the context, shifting the sequence number tracking functions
to the control block: same algorithms, but more copies of the state
variables.

2.1 DADDR bit
If DADDR is set, then you get to skip using a port filter
or other pattern matching algorithm. The idea is that in software
you would say

```
if (cmd&DADDR)
        context = context_tab[KEY&0x3fffffff];
else
        context = match(mac, KEY);
```

This is basically the gospel according to GAM-T-103, and I think
it's a useful thing to have in a software-based XTP implementation
for real-time applications.

There is no reason why DADDR packets would have to be restricted to
a LAN. Of course the routers would have to have knowledge of address
assignments. Since I believe that DADDR packets are most useful
for embedded systems, it follows that routers in embedded systems
would probably contain the necessary information.

2.2 EOCB bit
EOCB gone. It was an advisory bit and turned out to be superfluous.

2.3 Checksums
The check function has not received as much attention as other parts
of the protocol. Instead of saving something easy for the last,
it has turned out to be strikingly complex. Anyway, I believe
that we have settled most of the holes and open questions.

First, a data check will be calculated of the information segment
of every outgoing packet by the sender. Another check is calculated
over everything else: header, trailer, and possibly control.
Realize that we managed to eliminate the need to have both
an information and control segment in the same packet.
The data check is reset on a FIRST packet. This helps in catenating

packets in a router. The control check is reset for every packet.

Let f(A) denote the value of the check function over a block of bits, A. Let f(A||B) denote the value of the function over the catenation of block A followed by B. For our check function f(A||B) = c(f(A),f(B)) where c is a simple function that calculates the value of the check function of the catenated blocks from the values of the check for each block. There is another function that computes the effect of a change in a block of data on the check function of that block.

Fields in the control segment have been moved around so that everything in a packet that could possibly be changed by a router are grouped together in a contiguous block. The update operation is thus simplified.

You are right about the need for a checksum control bit.
There are two bits: NOCHECK in the header means check functions are not being generated, DCHECK in the trailer means the trailer contains a the data check value.

3.1 ENQ
It's gone again.
Every packet has a trailer and SREQ does the job.
Packet formats are simplified to either H|D|T or H|C|T
where H is header, D is data, and T is trailer.

3.2 ADDR
Taken care of.

3.3 RREQ
was a typo. fixed.

3.4 Seq field
64 bit increment.
good suggestion.

3.5 Bit Conflicts
Fixed now. I've got a couple guys reviewing this time.

Appendix G - XTP Service Primitives

G.1. Service Primitive Function Parameters

A number of parameters may be passed to the XTP service primitives. Because the service primitives are implemented as C functions, all parameters must be specified on the command line. In the case of optional parameters, dummy values are specified.

Both src_addr and dest_addr are pointers to XTP_address structures. The variable src_addr specifies the address of the source of the data transfer and dest_addr specifies the address of the destination of the data transfer. An XTP_address structure is defined below:

```
struct  XTP_address  {
        int  dst_host;
        int  src_host;
        short  dst_port;
        short  src_port;
        int  pad;
};
```

This structure corresponds to the DARPA format address defined in the Revision 3.25 Protocol Definition. Other formats are not supported in this implementation.

A pointer, named data, identifies the buffer used to store the data to be transmitted or received. When a primitive is invoked to transmit data, data_len indicates the length of the data in the buffer for transmitting. When the primitive is invoked to receive data, data_len indicates the amount of data in the receive buffer.

Flags is a 32-bit bit field used to indicate an exceptional condition to the remote application. Each bit represents a different flag. Currently, the only flag settable is EOM that indicates if data is being transmitted or received (i.e. complete control block operation) the current message is terminated. The address of flags is passed as the parameter to the service primitives. Because its address is passed, flags may be modified by the service primitive call to return status information.

The variable context_id denotes the context identifier associated with the context being manipulated by the service primitive call. A context identifier is returned by the XTP_context.request() and XTP_register.request() primitives. When either of these calls is made, the address of context_id is passed to the XTP host as a parameter to the call. The host copies

the context identifier created for this connection to this address, where it is used by the application. All other service primitive calls, except XTP_datagram.request, must pass the context_id as a parameter to uniquely identify to which context the service primitive applies.

G.2. Service Primitive Function Return Values

All service primitive functions return an integer value which indicates the control block used to service the call. The application process is expected to record the control block number returned, monitor the progress of the control block via sc_pend() and sc_vpend() calls, and return the control block back to the system when the operation is complete. If the control block identifier returned is negative, an error condition exists. The returned value indicates which error occurred.

G.3. Context Primitives

To understand the operation of the context primitives, an understanding of the client/server model used by XTP is needed. In general, a client makes a request to a server for a particular service and the server responds to the client with a reply. This request/reply paradigm is used by the XTP. To establish a connection with a remote server application process, a client application process issues an XTP_context.request() primitive to send its connection request to the server. When the XTP subsystem on the server receives the clients connection request, it issues an XTP_context.indication() to inform the server application process of the request. The server application process may either accept or reject the request. If the request is accepted, it then issues an XTP_context.response() to send its positive response back to the client. The XTP subsystem on the client receives the response and issues an XTP_context.confirm() to deliver the response to the client application process. If, however, the request is not accepted, the server issues an XTP_disconnect.request(), defined below.

In order for the client's request to be received by the server, a port filter must first be established by an invocation of the XTP_register.request() primitive. The operation of this primitive and port filters is explained below.

The context primitives and their parameters are shown below:

```
XTP_context.request(address,&context,&data,datalen,&flags);
XTP_context.indication(context);
XTP_context.response(context);
XTP_context.confirm(context);
```

G.3.1. XTP_context.request()

The XTP_context.request() primitive is used by a client application to request connection with a remote server. This primitive creates a new context for the connection and sends a FIRST packet to the server and does not return until the context record for the requested connection is completely initialized. After the primitive returns, the context may be referenced in other service primitive calls. For example, an XTP_send.request() may be issued to send data on the context immediately after the XTP_context.request() call returns. Allowing the context to be referenced before receipt of an XTP_context.confirm allows data to be transferred to a remote entity before a connection is explicitly established. This is an important feature of XTP.

Five parameters are passed to the XTP_context.request() primitive. An XTP_address structure, address, is used as a parameter to specify the address of both the client and the server. The address of a variable called context is passed as the second parameter to the call. The context identifier for this connection is copied into this address by the XTP host before the call returns. The parameter, data, specifies the address of any information to be transmitted during connection establishment. If no information is to be transmitted during this period, a null pointer is used. The parameter, datalen, specifies the length of the information pointed to by data. Finally, the address of flags is given. Any flags to be transmitted to the server should be set before the system call is made. When the call returns, flags contains any exceptional conditions set by the server process.

G.3.2. XTP_context.indication()

The XTP_context.indication() primitive is invoked by the server's XTP subsystem in response to a client's XTP_context.request(). The function of this primitive is to create a mapping in the translation map for the connection which associates a port filter with a previously created waiting context so that future packets are automatically routed directly to that context. A positive

acknowledgement is then returned to the client by issuing an XTP_context.response() primitive. If the request primitive finds that a context is already established on this port or if no port filter matches the incoming request, a refusal packet is sent back to the client using the XTP_disconnect.request() primitive defined below. The XTP_context.indication() primitive has only one parameter, address, which indicates the address of the remote client.

G.3.3. XTP_context.response()

The XTP_context.response() primitive is issued by the XTP subsystem on a server in response to an XTP_context.indication() primitive. The response is issued to indicate to the client that a connection is successfully established. This response assumes that if an incoming context request matches a previously established port filter, permission is automatically given to complete the connection establishment. This primitive has one parameter, context, which indicates the context identifier used by the server for this connection.

G.3.4. XTP_context.confirm()

The XTP_context_comfirm() primitive is issued by the XTP host of a client to indicate that a connection is successfully established with a server. The single parameter to this primitive, context, references the context which is being confirmed. This primitive is not defined in Cohn's service definition.

Although XTP allows data to be transmitted before an explicit connection establishment confirmation is received, implementations may choose to wait until the confirmation is made before transmitting packets. If packets are indiscriminately transmitted by a client before a server is ready to receive them, network bandwidth is wasted by packets which are not received at their intended destination. These packets are "lost" in the network and must be retransmitted. Suppose, for example, that a server is down. Every packet sent to that server constitutes wasted network bandwidth.

Many transmission strategies can be used by a client to take advantage of this primitive. Consider the following example. First, the server issues an XTP_register.request() primitive

followed by a XTP_receive.request() which allocates X bytes of receive buffer storage. Next, the client issues an XTP_connect.request() which transmits up to X bytes of data. The client has taken advantage of the fact that data can be transmitted during connection establishment. Since it is unsure if any more receive buffers exist on the server or if the server is going to accept the connection, it waits for an XTP_context.confirm(). Using this strategy takes advantage of the gains which can be made by transmitting data during connection establishment, but limits the amount of wasted bandwidth if the connection can not be made.

G.4. Registration Primitives

There is only one registration primitives defined, XTP_register.request(). The function of this primitive is to prepare a server to accept connection requests from clients. A call to XTP_register.request() creates a port filter and a "waiting" context to accept incoming connections. The port filter specifies from which clients the server is accepting connection requests. This primitive has three parameters. An XTP_address structure, address, is passed as the parameter used to specify the destination address from which packets are accepted (the client) and the source address of the node issuing the primitive (the server). The destination address is inserted as a port filter in the translation map, defined below. The address of a variable called context is passed as the second parameter to the call. The context identifier for this connection is copied into this address by the XTP host before the call returns. The final parameter, flags, is used to return any exceptional conditions back to the application process.

The registration primitive is shown below:

```
XTP_register.request(address,&context,&flags);
```

G.4. Send Primitives

Two primitive types are defined for sending data: request and confirm. The request primitive invokes procedures which cause data to be transmitted. The confirm primitive is executed when a send request is completed.

The send primitives are shown below:

```
XTP_send.request(context,&data,datalen,&flags);
XTP_send.confirm(context,&status);
```

G.4.1. XTP_send.request()

The XTP_send.request() primitive is called when data is to be transmitted. Each send request call is encapsulated in an XTP control block. The send request primitive has four parameters. The variable, context, defines the connection on which to send the data. The address of the information to be transmitted is stored in data. The length of the information is denoted by datalen. The flags variable is used to return status information back to the application process.

G.4.2. XTP_send.confirm()

The XTP_send.confirm() is called by the XTP subsystem to inform the application process that a send has successfully completed. This primitive moves the control block which defines the corresponding send request to the queue of completed control blocks as described in section 4.3.1.1. This primitive has two parameters. The parameter, context, references the connection on which the send operation has completed.and status denotes the success or failure of the send operation.

G.5. Datagram Primitives

Datagrams are usually thought of as a connectionless transmission mechanism. In the XTP paradigm, datagrams are really short lived connections. Since XTP claims to be able to quickly establish and close a connection, little overhead is incurred for datagram service. Datagrams are not acknowledged, which makes them different from XTP's connection oriented services.

A single datagram primitive exists for transmitting datagrams. The XTP_datagram.request() primitive has four parameters. These parameters are identical to the XTP_send.request() primitive. An XTP_address structure, address, is used to specify the address of both the client and the server. The context identifier for this connection is copied into

the variable called context by the XTP host before the call returns. The variable, data, specifies the address of the information to be transmitted in the datagram.and datalen specifies the length of the information pointed to by data. Finally, the address of flags is given. Any flags to be transmitted to the server are set in this variable before the system call is made. When the call returns, flags contains any exceptional conditions. The datagram request primitive is shown below:

```
XTP_datagram.request(address,&data,datalen,&flags);
```

G.6. Receive Primitives

There are two types of primitives defined for receiving data, request and confirm. The XTP_receive.request() primitive specifies a buffer area where information is to be received. The XTP_receive.confirm() primitive is used to complete the receive request primitive.

The receive primitives are shown below:

```
XTP_receive.request(context,&data,datalen,&flags);
XTP_receive.confirm(context,&flags);
```

G.6.1. XTP_receive.request()

The XTP_receive.request() primitive creates a control block which describes a receive operation and posts the control block to the XTP subsystem. It has four parameters. The first, context, identifies the connection on which the information should be received. The address of the buffer to receive the information is specified in a variable called data. The size of this buffer is specified by datalen. The variable, flags, is used to return exceptional conditions which occurred during the primitive call.

G.6.2. XTP_receive.confirm()

The XTP_receive.confirm() primitive is used to complete a receive request operation. This primitive is called by the XTP subsystem to move the control block created during the receive request primitive to the queue of completed control blocks. This primitive has two parameters.

The connection on which the receive is completed is defined by the context parameter. The parameter, flags, returns any exceptional conditions which occurred during the primitive call.

G.7. Disconnect Primitives

The disconnect primitives are used to close a connection. These primitives may be invoked for several reasons. First, if a conversation is completed on a connection either the client or the server may close the connection by issuing an XTP_disconnect.request() primitive. Second, if the XTP subsystem on a server decides that a connection request by a client should be denied, it may also issue a disconnect primitive.

It should be noted that XTP provides both a graceful connection termination procedure using the (CNTL,SREQ) protocol and an ungraceful one which abruptly aborts a connection disregarding any undelivered data. The United States National Bureau of Standards (NBS) rejected ISO's selection of an abrupt disconnection procedure for TP/4. It instead added a graceful termination procedure to its version of TP/4 which insures that all data is delivered before the connection is terminated. Further study must be done to determine whether the abrupt disconnection defined in XTP should be used in NTDS. If both types of disconnection are allowed, a flag might have to be defined in the disconnect request primitive to indicate which type of termination should be done.

There are two types of disconnect primitives, request and response, which are shown below:

```
XTP_disconnect.request(context,&data);
XTP_disconnect.confirm(context,&flags);
```

G.7.1. XTP_disconnect.request()

The XTP_disconnect.request() primitive is called by a client to release a connection between a client a server. This primitive has two parameters; context defines which connection is being closed, and data points to any information to be transmitted over the connection before it is closed.

G.7.2. XTP_disconnect.indication()

G.7.3. XTP_disconnect.confirm()

The XTP_disconnect.confirm() primitive is called by the XTP subsystem on the node wishing to terminate the connection. This primitive is invoked after the graceful context termination procedure is completed. The confirm primitive moves the control block associated with the disconnect request to the queue of completed control blocks. This primitive has two parameters: context identifies the connection being closed, and flags is used to indicate any error conditions.

References

[1] VRTX32/68020, Versatile Real-Time Executive for the MC68020 Microprocessor, Users Guide, April 1987, page 1-1.

[2] XTP Protocol Definition - Revision 3.25, Section 3.3.6., Page 3.10.

[3] MIL-STD-1778, 12 August 1983, Section 9.2.6 - Checksum, Page 78.

[4] XTP Protocol Definition - Revision 3.25, Section 3.2.3, Page 3.5.

[5] XTP Protocol Definition - Revision 3.25, Section 3.2.3, Page 3.5.

[6] XTP Protocol Definition - Revision 3.25, Section 3.3.8, Page 3.12.

[7] Correspondence from Dr. Chesson, 12/05/88 (20:37:08), Section 1.1

Bibliography

[ANSI86] American National Standards Institute. FDDI Token Ring Media Access Control (MAC). Draft Proposed American National Standard, ASC X3T9.5, Revision 10, February 28, 1986.

[CALINGAERT82] Calengaert, Peter: "Operating Systems Elements - A User Perspective," Page 45, Prentice-Hall, 1982.

[CHESSON87] Chesson, Greg: "The Protocol Engine Project," Unix Review, September 1987, pp. 70-77.

[COHEN81] Cohen, D.: "On Holy Wars and a Plea for Peace," IEEE Computer Magazine, vol. 14, pp. 48-54, Oct. 1981

[COHN88] Cohn, Marc: "SAFENET Express Transfer Layer Protocol (XTP), Functional Specification, Revision 2, 12 January 1988", Northrop Corporation, Advanced Systems Division.

[DIX82] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, "The Ethernet: A local Area Network: Data Link Layer and Physical Layer Specifications. Version 2.0, November 1982.

[DBOS84] Powers, Mark: "Data Bus Controller Firmware Design.", NSWC working Papers, Dahlgren, Virginia, March 1984.

[IEEE85] The Institute of Electrical and Electronics Engineers. Token Ring Access Method and Physical Layer Specifications. American National Standard ANSI/IEEE Std 802.5, 1985.

[ISO84] International Organization for Standardization, "Open Systems Interconnection - Basic Reference model," ISO 7498, 1984.

[ISO87] International Organization for Standardization. Logical Link Control, DIS 8802/2, 1987.

[LEFFLER89] Leffler, S., McKusick, M., Karels, M., and Quarterman, J., "The Design and Implementation of the 4.3 BSD UNIX Operating System", Addison-Wesley Publishing Company, pp. 181-184, 1989.

[MAP87] General Motors Manufacturing Automation Protocol, A Communications Network Protocol for Open Systems Interconnect, Version 3.0 Implementation Release, 22 July 1987.

[PEIX.Y] Protocol Engines, Incorporated: "XTP Protocol Definition - Revision X.Y"

[RFC1025] TCP/IP Bakeoffs, Request for comments 1025, Network Information Center.

[SAFENET87] SAFENET I - Revision 1A, 23 September 1987.

[SUN86] Sun Microsystems Corporation, "Networking on the Sun Workstation - External Data Representation Protocol Specification", February 1986.

[TCP83] MIL-STD-1778, 12 August 1983, Section 9.2.6 - Checksum.

[X.210] CCITT, "X.210 - OSI Layer Service Definition Conventions", 1984.

The vita has been removed from the scanned document