Experience With The Xpress Transfer Protocol

Robert Simoncic,
Alfred C. Weaver,
and
M. Alex Colvin

Computer Science Report No. TR-90-30
October 8, 1990

15th Local Computer Networks Conference
September 30 - October 3, 1990
Minneapolis, Minnesota

EXPERIENCE WITH THE XPRESS TRANSFER PROTOCOL

Robert Simoncic, Alfred C. Weaver, and M. Alex Colvin
Computer Networks Laboratory
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

This paper discusses our experience in implementing the Xpress Transfer Protocol. XTP is a transport and network layer protocol suitable for use as a Safenet Transfer Service in the Navy's emerging SAFENET specification. We describe the background of SAFENET and XTP, the essential features of the protocol, our implementation environment, our communications architecture, the user interface, a description of some working XTP demonstration programs, and finally some measured performance results.

1. SAFENET

The U.S. Navy is currently developing a specification for SAFENET (Survivable Adaptable Fiber Optic Embedded Network) [4,5] which is intended to address the communications requirements of Navy mission-critical computer resources. SAFENET is a communications architecture which uses local area networks and their protocols to provide connectivity among computer systems.

SAFENET provides a protocol architecture as shown in Figure 1. In the upper layers of the OSI Reference Model (layers 5-7), SAFENET User Services may be provided by either a MAP Application Interface (including FTAM and ACSE) in the Application Layer and the standard ISO protocols in the Presentation and Session layers, or by a Lightweight Application Interface and Lightweight Support Services. In the middle layers (layers 3-4), SAFENET Transfer Services may be provided by either the ISO transport (connection-oriented and connectionless) and ISO network (connectionless) protocols, or by the Xpress Transfer Protocol (XTP). In the lower layers (layers 1-2), SAFENET I specifies the IEEE 802.5C token ring, operating at 16 Mbit/s, using a dual counter-rotating ring topology and all fiber optic media. SAFENET II specifies the ANSI Fiber Distributed Data Interface (FDDI) LAN, again using a dual counter-rotating ring topology and all fiber optic media, and operating at a data transmission rate of 100 Mbit/s.

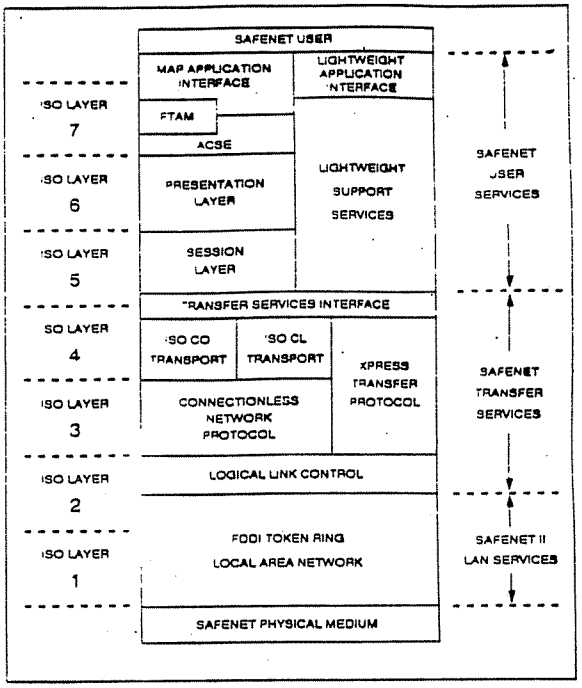


Figure 1.
SAFENET Protocol Profile

Since the lower layer services are provided by hardware and the layer 3-7 ISO protocols are available commercially, the research component of SAFENET is the Xpress Transfer Protocol which spans the OSI transport and network layers. While the ISO protocols are provided to achieve interoperability, XTP is included specifically for real-time applications which typically need only the services of the transport layer and below. The emphasis of XTP is high throughput, low latency, and close coupling between the (presumably real-time) application process and its communications server.

The Computer Networks Laboratory at the University of Virginia, with funding from Sperry Marine Inc. (Charlottesville, Virginia), the Naval Ocean Systems Center (San Diego, California), and the Office of Naval Research (Arlington, Virginia), has developed two implementations of XTP—one for PCs

running MS-DOS and operating with an IEEE 802.5 token ring, and one for Motorola 68020s running the pSOS real-time operating system and using the Martin-Marietta FDDI LAN; see [7]. The remainder of this paper describes our implementation experience with XTP in these environments.

2. Xpress Transfer Protocol

XTP is defined in [8,9] and discussed in [1,2,3]; a 66-page tutorial is provided in [6]. As a transport layer protocol, XTP has the same overall goal as DoD's TCP or ISO's TP4—namely, reliable end-to-end delivery of information. However, XTP represents a significant departure from TCP or TP4 with regard to the mechanisms and services which it provides. These characteristics are briefly summarized here.

(1) XTP uses a combination header/trailer PDU format. Since TCP and TP4 are header-based protocols, the transport checksum can only be located in the header. This forces the protocol processor to make at least two passes over the data, one to compute the checksum and install it in the header, and another to deliver the data to the network interface. By putting the transport checksum in a trailer, XTP can compute the checksum on the fly, thereby requiring only one pass through the data.

(2) For connection-oriented transport services, ISO TP4 requires a six-way packet exchange to set up the connection, send and acknowledge the data, and finally tear down the connection. XTP provides equivalent reliability with only a three-way handshake.

(3) In addition to standard error detection and flow control algorithms, XTP also provides rate and burst control. XTP allows the receiver to specify a rate (bytes/second) and burst (bytes/transmission) which the transmitter may not exceed. Whereas flow control is used to manage the receiver's buffers, rate control provides pacing information about the state of the receiving *system*. Burst control limits the total size of a multi-packet transmission, which helps prevents buffer starvation when the receiver must handle back-to-back packets.

(4) TCP and TP4 recover from out-of-sequence data by using a "go-back-n" protocol; XTP uses selective retransmission. XTP allows the receiver to retain out-of-sequence data while the transmitter resends only lost packets, thereby avoiding the retransmission of data already correctly received.

(5) XTP supports several addressing modes, including a very efficient direct addressing mode, as well as the standard Internet and ISO Network addressing conventions. Only the first packet on a connection need contain the full network address; subsequent packets refer to that address with a four-byte *key* field, thereby reducing overall frame size and address processing time.

(6) XTP supports multicast. If a single message is to be sent to n receivers, it can be sent with one multicast transmission rather than n unicast transmissions. Multicast is valuable for multi-destination file transfer, RPC-style transactions, event synchronization, global time distribution, and management of multiple redundant data objects and devices (e.g., hot spares).

(7) To support real-time communications, XTP provides an internal priority system. XTP defines up to 2^{32} static priorities, with the lowest numerical value having the highest processing priority. At every scheduling opportunity, XTP chooses the highest priority waiting packet to process next. This scheme is operational at both the transmitter and receiver so that high priority data will not only be preferentially transmitted, but will also be expedited upon receipt.

(8) Most importantly, all of XTP's functions can be described as a finite state machine whose complexity is compatible with VLSI implementation. XTP is currently implemented as software, but in 1991 will be available as a VLSI chip set which interfaces directly with FDDI.

3. Environment

Our project for Sperry Marine Inc. was to implement the XTP portion of SAFENET I in the environment of personal computers. Our XTP is operational across a range of low-end and high-end PCs, from the 4.77 MHz Intel 8088 to the 25 MHz Intel 80386. Our performance measurements were made on a pair of ALR FlexCache machines which feature 25 MHz 80386 CPUs, 128K of 25 ns cache, and 4 MB of 80 ns main memory. Our LAN is the Proteon ProNET-4, an IEEE 802.5 compliant token ring operating at 4 Mbit/s. As reported in a later section on performance, the LAN interface proved to be the bottleneck in our system, i.e., our XTP could generate data faster than the ProNET-4 could accept it.

Our project for NOSC/ONR was to implement the XTP portion of SAFENET II using Motorola 68020s and FDDI. We acquired four identical systems which duplicated hardware already acquired at NOSC; these systems included a 9-slot Force VME chassis and power supply, Motorola 133XT board with 25 MHz 68020 CPU and 4 MB memory, the pSOS real-time operating system and pROBE real-time debugger from Software Components Group, and the Martin Marietta FDDI LAN.

4. Software Architecture

4.1 General

The general architecture of the XTP implementation consists of three layers as shown in Figure 2:

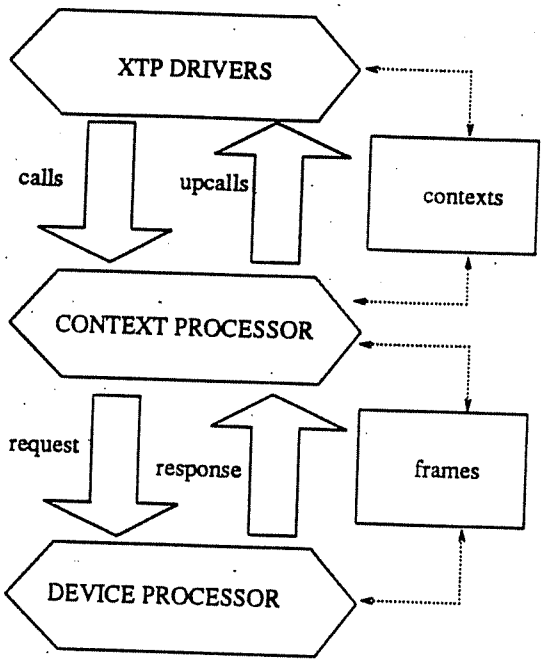


Figure 2.
Software Architecture

The lowest layer is the Device Processor which is implemented in hardware and drives the Proteon ProNET-4 or FDDI token ring. The middle layer is the Context Processor which implements the XTP protocol itself. There is a request/response type of communication between the Context Processor and the Device Processor and they both operate on shared data structures called *frames*. The highest layer is represented by our XTP Drivers; Drivers provide various interfaces for XTP users. The communication between XTP Drivers and the Context Processor is based on *calls* and *upcalls* and they both share context structures.

Our XTP implementation provides two different interfaces for the XTP user: (a) a low level XTP interface and (b) a higher-level XTP Driver. The low level XTP interface provides access to all the complex services that the XTP protocol offers. The user accesses the Context Processor directly and shares context structures with it. In this case the user communicates with XTP as an XTP Driver would. Since this interface reveals all the XTP functions, it is necessarily complex and difficult to use; this interface is primarily intended for knowledgeable programmers and is used to write XTP Drivers.

Most applications will obtain their communications services through use of one or more XTP Drivers. XTP Drivers are implemented using the low level interface and provide a simpler interface for the application writer. Example XTP Drivers include file transfer, memory-to-memory transfer, STDIO (standard input/output), and timer services. These Drivers are easy to use and are intended for application writers who have no need to know the internal details of XTP.

The whole XTP system consists of two software libraries which represent the two main components of the system:

Xtpl.lib This library consists of modules implementing the XTP protocol and the MAC layer. It also provides the low level XTP interface as C subroutine calls.

Drvl.lib This library consists of modules implementing the XTP Drivers. The services of the XTP Drivers are also implemented as C subroutine calls.

Applications using the services of the low level interface must be linked with the *Xtpl.lib* software library, while application using XTP Drivers must be linked with both the *Xtpl.lib* and *Drvl.lib* software libraries.

4.2 Low Level Interface

In the low level interface the XTP user (an XTP driver) communicates with the XTP system through two loosely coupled channels: the *data channel* and the *event channel*. The data channel handles the user's data which is being transferred through the network. Data is sequenced by bytes, so a unique sequence number is associated with each byte in the data stream. The event channel handles various events associated with the data. Each event refers to a certain sequence number in the data byte stream.

As a transmitter, a driver will send data bytes through the data channel in sequence; at the same time it will send events through the event channel which describe how to represent the information that is being transferred on the data channel. As a receiver, a driver will receive data bytes in sequence from the data channel and at the same time will receive events from the event channel which determine how to interpret the received data.

The data and event communication between the XTP driver (CLIENT) and the low level interface (SERVER) is implemented using shared ring buffers as shown in Figure 3:

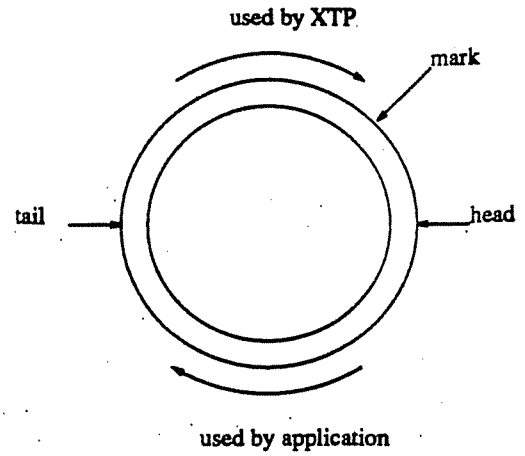


Figure 3.
Data and Event Communication

There are two separate rings, one for data and one for events. Both rings are shared between the CLIENT and the SERVER. If the CLIENT is a transmitter it submits data and events to the SERVER through data and event rings; if the CLIENT is a receiver it reads data and events from both the data and event rings.

All rings have the same structure and each of them is associated with three pointers:

head Points to the last byte of data or last event that was written into the ring. On the transmitting side, CLIENT writes to the ring and SERVER reads from the ring; on the receiving side, SERVER reads from the ring and CLIENT writes into the ring. In the case of the transmitter, the head pointer is moved by the CLIENT, whereas in the case of receiver it is moved by the SERVER.

tail Points one byte beyond the last byte of data or event information that was read from the ring. In the case of the receiver the tail pointer is moved by the CLIENT, whereas on the transmitting side it is moved by the SERVER.

mark Mark is set in both cases by the CLIENT. On the transmitting side, if data or events read from the ring by the SERVER pass the mark pointer, CLIENT is signalled at a defined location. On the receiving side, the CLIENT is signalled when data or events written to the ring by the SERVER pass the mark pointer.

A ring is always divided into two parts by the head and tail pointers; one part belongs to the CLIENT while the other belongs to the SERVER. The location of these two parts varies as the head and tail pointers are moved. The mark pointer is added for synchronization.

The low level interface provides subroutine calls for allocating ring buffers for data and event communication. Each ring buffer has these attributes:

address is the location of the ring buffer structure.

size is the size of the ring. For the data ring it is in units of bytes; for the event ring it is in units of events.

pointers are *head, tail, and mark* as shown in Figure 4.

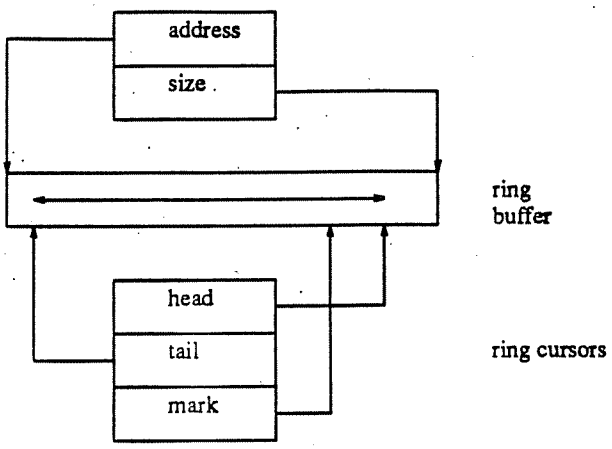
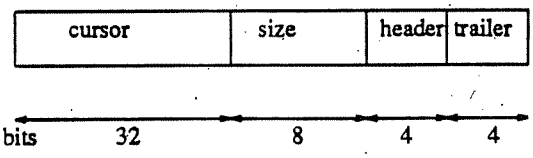


Figure 4.
Ring Buffer Descriptors

Items in an event ring describe events occurring at a certain byte in the sequenced data. Each event is described as shown:



where

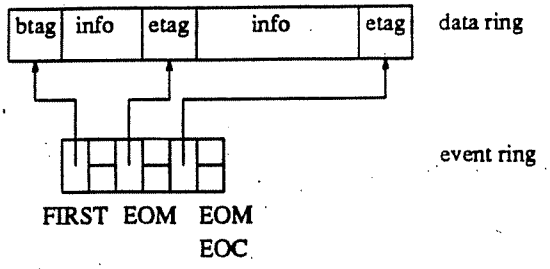
cursor is the sequence number of the data byte where this event occurs.

size is the size of this event. Some events have a known size.

header is the event bits that will be carried in the XTP packet header.

trailer is the trailer bits that will be carried in the XTP trailer.

Some examples of events are *beginning of connection, end of connection, end of message,* and *user-tagged data.* Each event ring is associated with some data ring by a data structure as shown below.



where

btag is user-tagged data that may be interpreted by some higher protocol. It refers to out-of-band data at the beginning of the XTP packet.

etag is user-tagged data that may be interpreted by some higher protocol. It refers to out-of-band data at the end of the XTP packet.

info is the user data being transmitted.

FIRST is the event representing opening of a new connection.

EOM is the event representing end of message.

EOC is the event representing end of connection.

Communication through XTP is represented by an active connection between two endpoints. At each end of the connection there is a *context* associated with it. A context is represented as a shared structure accessed both by the CLIENT and the SERVER. Each new connection will have two new contexts, one at each end. Each transmitting and receiving CLIENT is associated with a unique context. One SERVER serves many contexts. An XTP driver may in fact establish many connections with other XTP drivers on the network; for each connection a new context has to be reserved and two rings on each side have to be allocated. One XTP driver may represent more than one CLIENT.

4.3 Xtpl Library

The XTP services of the low level XTP interface are provided by the software library *Xtpl.lib.* This must be linked in with any program using these services. All services are implemented as C subroutine calls and are logically divided into several groups. Services of the Xtpl library are used for writing XTP drivers.

4.3.1 General XTP Operations

These calls provide general services for XTP users to initialize, open, start, stop, and close the XTP application.

XTP_init()

This call will initialize the XTP application. It should be called before any other XTP service is called.

XTP_start()

The XTP protocol engine is implemented as several processes runing at different priority levels. XTP_start() will initialize and start all the internal XTP processes. An XTP driver using this call is a CLIENT and will have its own process with the same priority level. Before calling XTP_start(), the programmer must define a C subroutine which will represent the client process (the main body of the XTP driver itself), and the name of this subroutine is passed as a parameter to the XTP_start() function. The user may also define more processes with priority levels; however, these have to be forked from the client() procedure. XTP_start() should be the second XTP call after XTP_init().

XTP_open(locadr,fadr, gadr, devn)

Initialize and start the MAC interface process at the device with node, group, and functional address given by the caller. This subroutine should be called only after XTP_init() and XTP_start(), since the XTP processes have to be already started.

XTP_close(devn)

Closes the MAC interface. XTP_close() should always be called before an XTP driver exits. If this is not called and the XTP driver exits, all the interrupt handlers will still be installed and the operating system may not function correctly.

XTP_stop(devn)

Release XTP buffers. This should be called only after XTP_close() because the buffers released are still used by XTP_close().

4.3.2 Context Services

All the context calls provide access to the internal structures of the XTP engine and those parts of the context that the XTP driver needs to access.

C_get_ctx(next, db, eb, dl, el, typ, devn)

Tries to allocate the context *next*. If successful, it allocates data and event ring buffers and returns those in data and event ring structures provided by a caller. It also allocates buffers for the receiving and transmitting frames and returns the context number. If the context *next* is already open or the allocation was not successful, it returns CTXON or CTXALC as an error code. C_get_ctx() is called once for each new context before the connection is established.

C_set_net(nadr,next)

Sets the network address on the side of the receiver. The format is described in section 2.3 of the *XTP Protocol Definition* [8]. The address is set for a particular context, which will then listen to all the FIRST packets destined to this network address. The network address is transmitted as a part of the first data packet and is used only for establishing a connection.

C_set_dadr(dest,src,next)

Set the direct network address of the transmitting or receiving context. The direct addressing scheme can be used for systems with a fixed communication topology where the *key* field is interpreted as a direct address. In the direct addressing scheme the address is included in each packet.

C_put_ctx(next)

Returns the context *next* to the free context list. Buffers associated with this context will be deallocated.

C_set_len(len, devn)

Sets the frame length for the device (the network interface). The frame size includes all the headers and trailers (MAC, LLC, and XTP). The maximum value is 2025 for ProNET-4, which is used as a default if C_set_len() is not called.

C_set_rate(burst, rate, next)

Burst determines the maximum number of bytes in one burst transmission, while *rate* determines the maximum rate in bytes per second. C_set_rate() sets burst and rate for the context *next*. On the transmit size it will set the initial value and is then changed according to the value received in the control information from the receiver.

C_set_delay(delay)

Delays the transmit engine for a certain time. It may be used instead of C_set_rate() or as a supplement to it.

C_act_ctx(next)

Denotes the context as active. The receiving context is always active, while the transmitting context is active only if it has more work to do.

C_dec_ctx(next)

Denotes the context as not active, so the protocol engine will not serve it until it becomes active again.

C_get_ack(next)

The ack indicator is set on the arrival of new control information on the transmitting context. C_get_ack() returns the ack value and clears it.

C_set_options(opt,next)

Sets the options for the context.

C_set_flags(flag,next)

Sets SREQ and DREQ flags, and these will go out with the next XTP packet prepared for transmission.

4.3.3 Scheduler ZEK

ZEK is a simple and fast scheduler used for scheduling XTP protocol engine processes. The user writing XTP drivers may want to use ZEK to schedule the processes of the XTP driver itself. ZEK handles 16 processes, each on a different priority level. Our protocol engine uses four different priority levels, leaving twelve others free for use by the XTP Driver designer.

levon(new)

This call will activate the process running on level *new*; if a process with higher priority (lower number) is currently running, the *new* process will be set pending, otherwise, if it is the highest priority ready to run, it will start running.

levoff()

Passivate the process which calls levoff(). If the process was set pending by levon() while it was active, it will drop its pending bit and stay running.

levfor(new, body, stack, depth)

Creates a new process on the priority level *lev*. The body of the new process is the procedure body().

levto(new)

Changes the running level of the existing process. If some other process was previously active on this level, it is discarded.

5. XTP Drivers

We provide several XTP drivers for those users who do not want to write their own applications using the low level interface. Our drivers include:

(1) STDIO

Contains three standard I/O subroutines:

X_putc() is equivalent to putc() in STDIO.
X_getc() is equivalent to getc() in STDIO.
X_print() is equivalent to printf() in STDIO.

(2) FILETRANS

This driver is for file transfer applications and uses block read/write. It has two subroutines:

X_putf() is for block write.
X_getf() is for block read.

(3) MEMTRANS

This driver is for memory-to-memory transfer applications. It contains two subroutines for byte transfer and two for block transfer:

X_putb() reads one byte from memory and passes it to the network.
X_getb() reads one byte from the network and stores it in memory.
X_putm() reads one block from memory and passes it to the network.
X_getm() reads one block from the network and stores it in memory.

(4) TIMER

This driver is for applications which need timeout services; it can be used to implement the WTIMER and CTIMER within XTP drivers. It has two subroutines:

XTwtime will set up the timer for a context.
XTwoff will turn off the timer for a context.

6. XTP Demonstrations

XTP is most impressive when seen in action. To convey a sense of its power and speed, we have developed a set of demonstration programs.

(1) *Integrity test.* Data is generated by a random process with a known seed, then framed and transmitted. The screen displays data content and the state of the protocol state machine. The receiver checks the expected data byte-by-byte. This test has run for weeks without error.

(2) *Memory-to-memory transfer.* Joystick data is collected on one machine, translated into navigation instructions for a simulated ship, and transmitted as XTP packets using STDIO. The receiver moves the ship on the screen in accordance with the remote joystick commands.

(3) *Performance test.* Using memory-to-memory transfers on the Proteon ProNET-4 token ring (4 Mbit/s capacity), XTP sustains 1.8 Mbit/s in NOERR mode. Using Western Digital WD8003E Ethernet interfaces, throughput rises to over 4 Mbit/s. See section 7 below for more details.

(4) *Pipes.* Data input on one machine's keyboard is piped to the screen of another.

(5) *File transfer.* A megabyte of data can be transferred disk-to-disk in approximately 10 seconds, for a throughput rate of 800 Kbit/s (including disk I/O time).

(6) *Multicast file transfer.* The same megabyte of data can be transferred to any number of receivers (in our demo, five) simultaneously using multicast. After our multicast negotiates and enumerates the group membership, the megabyte of data moves to all receivers in approximately 20 seconds. Recall that this is a transport multicast, not a link level multicast, and so it is entirely reliable. Any loss of data (or the loss of a receiver) can be detected and corrected transparently.

(7) *Image transfer.* Video frames, captured by a video camera and stored on disk, are transferred to a remote screen where gray-scale thresholding is performed. The overhead of the XTP transmission is negligible; the transfer operates at the speed of the disk access and the display computations.

(8) *Real-time image transfer.* As above, except the video image is captured in real-time by the video camera. Visitors to the lab often find themselves being multicast on their next visit!

(9) *Multicast polygons.* Randomly placed and colored polygons are distributed via multicast. Again, the time needed to draw and color the polygons overshadows XTP transmission time. The multicast screens update in unison about 10 times per second.

(10) *Multicast images.* Multiple video images are multicast from an "image server" to any number of receivers. Each receiver displays an image in one of five areas of the screen. Figure 5 is a photograph of one of our multicast receiver screens.

(11) *XTP over FDDI.* We use our own real-time network monitors (*WireTap* for Ethernet and token ring; *FiberTap* for FDDI) to capture performance data and packet traces from the network. Using the Motorola 68020s and the Martin Marietta FDDI interface, our XTP sustains a transmission rate (in NOERR mode) of about 20 Mbit/s; the FDDI MAC alone sustains approximately 30 Mbit/s.

7. XTP Performance on PCs

The following data are measured on the ALR FlexCache (25 MHz Intel 386 processor) using the Proteon ProNET-4 token ring.

7.1 Datalink Layer Limits

No transport protocol will operate any faster than its underlying datalink layer. While our ProNET-4 LAN operates with a signalling speed of 4 Mbit/s, no single station can generate that much data; interfacing to the board, handling interrupts, and copying frames to and from the token ring interface board all consume time. The best performance of the ProNET-4 is obtained using 2000-byte packets, and the highest sustained throughput we observed was 1.80 Mbit/s. Thus, in this particular implementation, XTP throughput is constrained by the LAN interface to be less than or equal to 1.80 Mbit/s. Other interfaces (e.g., Ethernet, FDDI), of course, would impose different limits on single-station throughput at the datalink layer.

7.2 Memory-to-Memory Transfer

Our measured performance for XTP, using NOERROR mode and 2000-byte packets, is 1.74 Mbit/s. This 4% degradation in single-station throughput (compared to performance of the raw datalink layer) results from the transmission of the 24-byte headers and 16-byte trailers which XTP attaches to each data frame.

7.3 Ultimate Limit

To determine the ultimate throughput of our software implementation, we preserve all the system overhead (including interrupt processing) *except* for the actual transmission of the data itself. This removes the ProNET-4 interface from the code path and thus simulates an infinitely fast transmitter. In this mode, using 2000-byte packets, our XTP can transmit or receive about 950 packets/sec which translates to 15 Mbit/s. Clearly, the bottleneck in our system is the LAN interface.

7.4 File Transfer

Moving a large file from one system to another measures much more than just XTP; file transfer results depend upon disk access time, file layout, disk controller efficiency, and other system-dependent parameters. Still, it is an interesting measure since file transfer is such a common application. Our file transfer rate between two ALR FlexCache machines runs at 800 Kbit/s.

7.5 Confirmed Response

Finally, we measured the elapsed time to prepare and send a small message and then receive its confirming acknowledgement. This measurement includes: preparing an 8-byte message, processing of the message by the Context Processor, transmitting the message with SREQ set (forces an acknowledgement), receiving the message at the destination, protocol processing at the destination, preparing a CNTL (control) packet which includes the acknowledgement, transmitting and receiving the CNTL packet, and advising the original transmitting context that the message has been correctly received at its destination. Our measurement for this confirmed response operation varies from 4 to 7 ms.

8. Conclusions

Our experience with XTP has been very positive. Our implementation is stable; it is operational in both PC and VME bus environments; we are transmitting reliably over Ethernet, token ring, and FDDI; and our performance data indicate that our software implementation is primarily limited by the underlying hardware MAC interface. It is our opinion that XTP represents a major improvement in transport protocol design.

Our next task is to compare our implementation of XTP with commercially available implementations of TCP/IP and ISO TP4. Functionality and performance measurements, made using identical hardware platforms, operating systems, LAN interfaces, test programs, and timing routines, should be especially illuminating.

9. Acknowledgements

The Computer Networks Laboratory gratefully acknowledges the financial support and technical assistance provided by Mr. Ross Bennett of Sperry Marine Inc. and Mr. Will Gex of the Naval Ocean Systems Center.

10. References

[1] Greg Chesson, "Protocol Engine Design," *Usenix Conference Proceedings*, June 1987.

[2] Greg Chesson, "The Protocol Engine Project," *Unix Review*, September 1987.

[3] Greg Chesson, "XTP/PE Overview," *Proc. 13th Local Computer Networks Conference*, October 1988.

[4] Space and Naval Warfare Systems Command, "Survivable Adaptable Fiber Optic Embedded Network I -- SAFENET I," MIL-HDBK-0034 (Draft), 31 January 1990.

[5] Space and Naval Warfare Systems Command, "Survivable Adaptable Fiber Optic Embedded Network II -- SAFENET II," MIL-HDBK-0036 (Draft), 1 March 1990.

[6] Robert Sanders, "The Xpress Transfer Protocol (XTP)— A Tutorial," Department of Computer Science Report Number TR-89-10, January 1990 (available by e-mail request to "weaver@cs.virginia.edu").

[7] Alfred C. Weaver, "UVa XTP Status Report," *Transfer*, Vol. 3, No. 2, March/April 1990.

[8] Protocol Engines Inc., "XTP Protocol Definition," version 3.4, 17 July 1989.

[9] Protocol Engines Inc, "XTP Protocol Definition," version 3.4.5, 27 July 1990.

Figure 5.
Multicast Video Images