

# An Object-Oriented Implementation of the Xpress Transfer Protocol<sup>1</sup>

W. Timothy Strayer, Simon Gray, and Raymond E. Cline, Jr.

Sandia National Laboratories, California  
{strayer|sgray|rec}@ca.sandia.gov

**Abstract.** Object-oriented design principles map well onto protocol implementations because protocols essentially manipulate two structures—packets and the states of the endpoints. In this paper we describe an implementation of the Xpress Transfer Protocol as a user-space daemon written in C++. The object-oriented model forces the programmer to properly place functionality and information ownership. The model facilitates porting to various platforms and greatly eases the task of building data delivery services.

## 1. Introduction

Transport protocols have at least two common components: a fundamental unit for information exchange, and a set of structures and procedures for managing the information as it is exchanged. The units of information exchange are called *packets*, and the state structures are called *protocol control blocks* or *contexts*. Protocols are distinguished by the amount of state information maintained, and by how the protocol behaves during its state transitions. For example, a simple unreliable datagram protocol keeps little state and does not react to lost packets, while a robust connection-oriented protocol maintains much more information about the data exchange, and notices and reacts to lost packets. Yet both types of protocols follow the same pattern of processing: a packet arrives, it is parsed, and the data, if present, is delivered to the client. Protocols can therefore be viewed in the abstract, where a specific protocol is an instantiation of this abstraction.

The object-oriented programming paradigm forces ownership of functionality. This is useful for any major software project. The paradigm also forces ownership of information. As a value is needed by an object, the programmer must decide how that value is to be conveyed. If the programmer cannot convey the information gracefully, there is a strong indication that the information is either ill-placed or unnecessary. Object orientation also provides well-defined modules for implementation hiding. This is useful for isolating services from the rest of an implementation, facilitating adaptability to a variety of services that are built with the same interface.

---

<sup>1</sup> . . . . .

This paper discusses the application of the object-oriented programming paradigm to the implementation of a transport layer protocol. Specifically, we have implemented the Xpress Transfer Protocol [1, 2] using C++. The compiled tar get of this protocol implementation is a user-space daemon process. Application processes load a user interface library (also object-oriented) that manages the interprocess communication between the application and the daemon. The daemon accepts user requests and incoming packets, and passes them on to a context container class that manages all of the contexts in the daemon. We describe the class structures and associated methods used to construct XTP, and we make observations that are relevant to object-oriented protocol implementation in general.

## 2. Protocol Implementation

Traditionally, protocols are implemented in the kernel of an operating system. TCP and UDP in Unix [3] are the most common examples. The *x*-kernel [4] and the Mach operating system [5] are examples of attempts to simplify the implementation of protocols in the kernel. Thekkath *et al.* [6] observe that monolithic implementations of protocol stacks in the kernel provide performance and security at the cost of making prototyping, debugging, maintenance, extensibility, and exploitation of application-specific information more difficult. They suggest implementation of protocols as user-level libraries, where an agent in the kernel is responsible only for demultiplexing the packets as they arrive.

Our implementation of XTP runs the protocol as an object-oriented user-level daemon process. This design provides rapid prototyping, portability, adaptability, configurability, and readability.

*Rapid Prototyping.* A user-level implementation of a protocol is typically faster to build than a kernel implementation for two reasons: it is easier to debug, and kernel programming requires an additional learning curve. Also, in theory, object-oriented programming forces the programmer to design the components of the software, and to assign functionality to these components, prior to writing code.

*Portability.* Kernel implementations of protocols are rarely portable. The Distributed Systems Research department at Sandia has a cluster of 50 workstations as a testbed for heterogeneous cluster computing research. This testbed has five vendors and six different operating systems; implementing a protocol for each operating system's kernel would be a difficult and time-consuming task. We coded the implementation in C++ since it is the most widely available object-oriented language.

*Adaptability.* We are interested in protocol characteristics in a variety of LAN and WAN environments. Consequently, we need to be able to rapidly switch between different data delivery services, such as IP, Ethernet, FDDI, ATM, and other solutions like the Scalable Coherent Interface (SCI). The modularity of the design of our implementation supports this.

*Configurability.* In addition to changing the underlying data delivery service, we want to be able to replace the various protocol control algorithms. Again, modular code construction supports this.

*Readability.* Kernel code is difficult to read since it is so deeply embedded in the operating system. Code written for user-level processes is (perhaps subjectively) easier to read and decipher. We expect our implementation source code to be used as a reference for XTP's protocol control algorithms.

In general, a protocol receives service data units from the service below it and demultiplexes their contexts, or protocol data units, to the various clients of the protocol. In transport protocols, these clients are the session layer (in the OSI view) or some user process (the Internet view). A protocol implemented in the kernel uses its knowledge of which processes have employed its services to do the demultiplexing. Protocols implemented outside of the kernel, in user space, must have some other means for managing multiple users. One approach is to implement the protocol in a library, and let each user process run the protocol. However, there is still a need for some agent, usually in the kernel, to demultiplex the incoming packets. Another drawback for library implementations is that the protocol is distributed, so control algorithms such as rate and congestion control must also be implemented in the common agent.

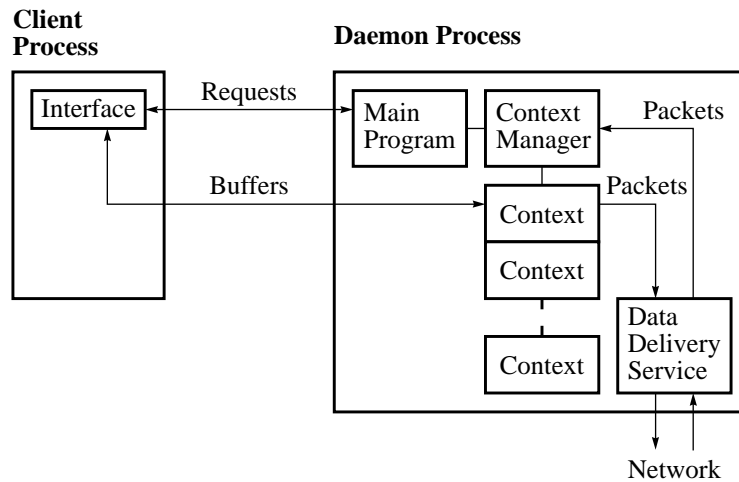
We use a user-level daemon as the single representation of the protocol. The daemon manages each of the client users, and demultiplexes incoming packets. In spite of the considerations listed above, there are drawbacks. This approach adds another process, requiring context switching for interprocess communication. Furthermore, a user-level process must invoke system calls in order to gain access to kernel-level services, such as timers and device drivers.

### 3. Meta-Transport Library

While guided by the above design goals, we added one more: hierarchical design. A useful tool in object-oriented programming is class hierarchies. While designing our implementation of XTP, we built abstract classes that would be applicable to most transport protocols. We call the collection of these abstract bases classes the *Meta-Transport Library* (MTL) because they can be used to build most transport protocols, not just XTP. The objective in designing MTL was to distill as many of the commonalities of transport protocols as possible into a set of classes that are made available as a library. Specific protocols are “derived” from MTL by implementing classes derived from the MTL base classes. A virtual function in the base class suggests that the MTL implementation for that method may not be sufficient, and additional protocol-specific processing may be necessary. A pure virtual function implies that an implementation must be provided by a derived class. These methods are mandated by MTL but require protocol-specific knowledge to implement.

Protocols derived from MTL have essentially the same performance advantages and disadvantages since they share a common foundation. In this way, comparison of derived protocols more accurately exposes the differences in the protocols rather than the differences in the skill of the implementors. Furthermore, since the “skeleton” of the protocol is already in place, an implementor simply needs to flesh out the implementation with the protocol specific-member variable and methods.

Fig. 1 shows the general model for protocols derived from MTL. A client process sends requests to the daemon via an IPC message queue (see the *ipc(1)* Unix manual



page) built into the interface object. The daemon returns the result of the request via the same IPC facility. User data, however, is written to and read from two buffers that are managed by a buffer manager in the interface. This separates the request/response activity from the maintenance of data buffers. Here we use shared memory between the daemon and client process; a buffer manager interface, however, hides these details. The main program in the daemon is a loop that accepts user requests and invokes the appropriate entry point into the actual protocol processing. These entry points are methods in the context manager. The context manager owns all of the contexts in the daemon, and steers incoming packets to the proper context. The contexts themselves implement the protocol-specific procedures, some of which generate packets. The daemon also owns a data delivery service object that manages the use of the network.

An MTL context is identified by a key value. This is actually a concept from XTP, but it is applicable to any protocol since the MTL model requires some manner of uniquely identifying a context data structure.

#### 4. SandiaXTP

SandiaXTP is the XTP-specific protocol derived from MTL. We implement the latest version of XTP, XTP 3.7. This version changes a few packet formats and splits the control information into three distinct packet types, one for normal flow control information (CTNL), one to report error information (ECNTL), and one to negotiate the traffic shape information (TCNTL).

Below we discuss some of the division of responsibility between MTL and SandiaXTP, and explain some of our design decisions.

---

```
typedef struct {
    char* data;
    int len;
} send_vec;

class packet {
    packet();
    ~packet();
    // As contiguous memory
    byte8* pkt_start();
    short16 TCP_style_xsum(register int len);
    int send(void* dest, int length);
    // As scatter-gather vector
    int add_first_vector(char* p, int len);
    int add_vector(char* p, int len);
    short16 TCP_style_xsumv(int nsv);
    send_vec* get_vectors();
    int get_num_vectors();
    int sendv(void* dest);
    // Virtuals
    virtual void host_to_net() = 0;
    virtual void net_to_host() = 0;
};
```

---

#### 4.1. Packets and Packet Manipulators

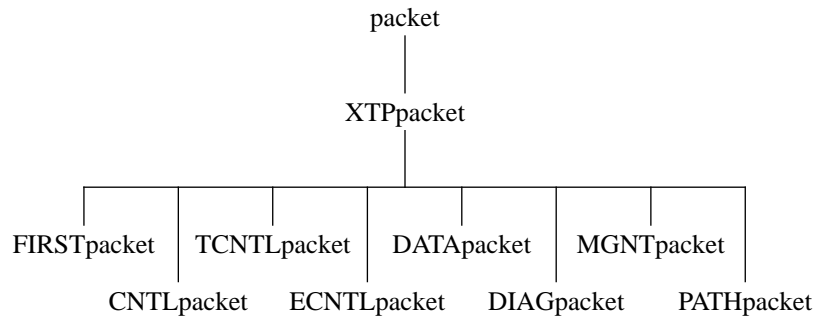
Packets are the vehicle for data and information exchange between endpoints. Packets are sent and received by a data delivery service that treats the contents of the packet as uninterpreted payload. The protocol defines the structure of its packets, and information is placed or extracted only with knowledge of the structures.

The MTL packet base class, shown in Fig. 2, provides two ways to view a packet, as contiguous memory or as a vector of scatter-gather elements.<sup>1</sup> Since MTL's data delivery service assumes that packets are received as a contiguous piece of memory, the first form is generally used for processing a packet within the context. The size of a packet object is at least as big as the maximum protocol data unit length. The packet base class includes a member function that returns a pointer to the beginning of the packet object so that protocol-specific agents can read and write to of fsets within the packet.

The scatter-gather vector is a set of pointer, length pairs. The packet base class provides member functions to set and retrieve the scatter-gather elements. This style is intended for constructing packets with minimal data copying.

---

<sup>1</sup> This is a common technique for representing data in a stream-oriented protocol.



The packet base class provides an overloaded send method for both packet styles. In both cases, the `packet::send()` method calls the data delivery service send method. This ensures that the context, or any other agent constructing a packet, need know nothing of the data delivery service. There is no symmetric receive method, however, since receiving is not actually done *to* a packet in the same way that send is. Data simply arrives at the underlying data delivery service; that data is cast into a packet structure in order to retrieve the contents. For protocols with more than one distinct packet structure, this act of casting is probably done twice, once to retrieve the type, and again when the type is known.

In XTP all packets have a common header that holds the specific packet type. Methods for placing and extracting header information are defined in the class `XTPpacket`, which is derived from the MTL base class `packet`. Specific XTP packet types, such as `FIRST`, `DATA`, `CNTL`, etc., are derived from `XTPpacket`, and provide the type-specific manipulation methods. This hierarchy is shown in Fig. 3.

The packet base class has only two virtual functions, `host_to_net()` and `net_to_host()`. These are pure virtual functions since byte order conversion is only possible with knowledge of the packet structure. Other than these, there are no mandatory methods to be defined by the derived classes. Rather, functions common to all packets classes at one level are defined in the packet class one level above. For example, all packets must send, but sending is not a protocol-specific function since the data delivery service does not care what it is being given to send. So `send()` is defined at the MTL packet class level. All XTP packets must have access to the common header, so `get_header()` is defined in the `XTPpacket` class level.

MTL also includes two classes that manipulate abstract packet objects. For efficiency, a packet pool class is provided that pre-allocates and then manages packet shells. The packet FIFO class holds packets, then emits them in first in, first out order. If a key is given, the packets destined for the context with that key are emitted in FIFO order.

## 4.2. Context Objects

A context is the collection of all state information for an endpoint of an association. Certain state information is common to all transport protocols. Accordingly, the MTL context base class holds:

- a key value to identify the context
- the context's priority information
- information to identify the context's client
- addressing information
- send and receive buffers
- the maximum protocol and service data unit sizes
- a place to get fresh packet shells
- a place to store outgoing packets that are held by flow or rate control
- a place where incoming packets are held until processed

The functions declared virtual in the MTL context base class are the functions whose inclusion is mandated by the class. These include methods to:

- determine if the context is active or quiescent
- cause the context to become quiescent
- get and change the context's priority
- initialize the context's state variables
- bind an address to the context
- process a packet
- put data in the client's receive buffer
- send data from the client's send buffer

The derived context class must redefine these virtual functions so that they represent the functionality specified by the protocol. The abstract class cannot mandate that error, flow, rate, or other control algorithms be present. In SandiaXTP, the send method checks flow and rate control parameters to decide whether a newly constructed data packet can be sent.

The derived protocol may need to implement some form of timer control to guard against lost packets or inactive connections. The XTPcontext class implements the XTP specification of these timers. Also, the protocol state machine is not included in the methods mandated by the context base class, although every protocol moves through a series of states over its lifetime. Since the states and their meanings are protocol-specific, the context base class mandates only that a test for quiescence be present.

## 4.3. The Context Manager Objects

The context manager is the container class for all of the contexts in a protocol implementation. The main purpose of the context manager is to match user requests and incoming packets to the appropriate context, so that the contexts can do the necessary protocol processing. To this end, the context manager base class provides functions that:

- allocate a new context and key value
- find an active context
- initialize a specified context

---

```
class context_manager {
    context_manager(int nctxts, int npkts, int csize);
    virtual ~context_manager();

    // allocate a new context and key value
    virtual context* get_next_context(word32 prio);

    // find an active context
    context* get_context(word64 key);

    // initialize a specified context
    virtual int init_context(user_request* request) = 0;

    // bind an address to a context
    virtual int bind_context(user_request* request) = 0;

    // order active contexts by priority (head of priority list)
    context* get_head();

    // match incoming packet with context
    virtual void handle_new_packet(packet* pkt, void* from) = 0;

    // visit each context to do work pending
    virtual void satisfy() = 0;

    // release the context
    virtual int release(word64 key);

    // plus other utility functions...
};
```

---

- bind an address to a context
- order active contexts by priority
- match an incoming packet to the appropriate context
- visit each active context to satisfy pending work
- release the context

Since these functions are common to all protocols that would be implemented using the MTL model, they are mandated by the MTL context manager base class (shown in Fig. 4) through virtual functions. The derived class, `XTPcontext_manager`, redefines the virtual functions in terms of an XTP context rather than an abstract context. There are a few additional functions in `XTPcontext_manager` that reflect the way XTP handles incoming packets. Specifically, there is a function that finds the listening XTP context given the address segment of a `FIRST` packet. The fields and methods are not general enough to put this function in the base class, although most protocols would have some similar capability. Another function specific to XTP is the full context

---

```
class del_srv {
    del_srv();
    virtual ~del_srv();

    // Pure virtual functions to be filled in by derived classes

    virtual int install(address_segment* addr, void* dest) = 0;
    virtual void* alloc_addr_struct() = 0;
    virtual void free_addr_struct(void* asptr) = 0;
    virtual int sizeof_addr_struct() = 0;
    virtual int recv(char* data, int length, void* from) = 0;
    // send contiguous payload
    virtual int send(char* data, int dlen, void* dest) = 0;
    // send scatter-gather payload
    virtual int sendv(send_vec* sv, int nsv, void* dest) = 0;
    virtual int get_pdu_size() = 0;
};
```

---

lookup. For packets carrying the key value defined at the destination host, the XTPcontext\_manager can match the packet directly to the appropriate context. For others, a table lookup is required. The function of matching a packet to a context is common to all protocols, but how it matches is protocol specific, so this function is not included in the base class.

#### 4.4. The Data Delivery Service

The data delivery service abstract class presents a common interface to the underlying packet transport, as shown in Fig. 5. Specific delivery services are derived from this abstract class. In particular, the derived classes must define receive and send methods, as well as a method for translating between the MTL address structures and the service-specific address structure. A method for determining the maximum protocol data unit size for the service are also included.

#### 4.5. The Daemon

In SandiaXTP, the daemon, called xtpd, instantiates a context manager object and a data delivery service object, then blocks waiting for incoming user requests. The daemon unblocks when the client's request arrives, does a switch on the request type, awaits the result of the request, then sends the result back to the client. The user can specify that the request is a blocking request, where the daemon does not return the results until the request is satisfied. Either way, the daemon loops back and blocks waiting for another request.

Incoming packets interrupt this block. The packet is retrieved from the data delivery service, the destination context is determined by the key and other addressing infor-

mation, and the packet is placed on a packet FIFO where it awaits processing by the context. When all of the newly arrived packets have been received from the data delivery service, the daemon invokes the context manager's satisfy routine, directing each context to do any pending work. Once all the active contexts have been satisfied, the daemon blocks again.

XTP_REG	register the client with the daemon, and allocate a context
XTP_BIND	bind the client's address structure with the context
XTP_LISTEN	listen for a FIRST packet to establish the association
XTP_REJECT	reject the association
XTP_GETADDR	get the client's address structure
XTP_GETSTATE	get the state of the context
XTP_SEND	send data in the send buffer
XTP_RECEIVE	get data from the receive buffer
XTP_UPDATE	synchronize the client's state with the daemon's
XTP_SENDCNTL	send a control packet
XTP_RELEASE	return any resources, release the context

**Table 1** XTP Request Types

There are eleven types of XTP requests accepted by xtpd, as shown in Table 1. These requests correspond closely to the services offered by the XTP interface to the client. The intent is to offer request types that expose the flexibility of XTP without being overly complex. The rationale for the interface is given later.

Registering with the daemon causes a context, and the key that identifies the context, to be allocated for this client. The key will uniquely identify this context in all future requests (as well as future packet exchanges). The "bind" request sets the addressing structure. This must be complete before sending a FIRST packet, or, if listening, this address specifies the filter that discriminates which FIRST packets to accept. The "listen" request causes XTP to look for FIRST packets that match the address specified with the bind request. The "get address" request retrieves the fully specified address structure at any time after the association establishment. The "reject" request allows the user to reject an association if its parameters do not meet the user's needs.

Data placed in the send buffer is sent by issuing the "send" request, and data is retrieved from the receive buffer by issuing the "receive" request. Data is not actually moved by these commands; rather, the client, through its interface, and the context, through its buffer managers, read and write the data to the appropriate buffers. The "update" request synchronizes the client's and the context's buffer managers.

Since XTP is a transmitter-driven protocol, the control packets are not automatically generated by the protocol. The user has several mechanisms for causing control packets to be generated, such as using the SREQ and DREQ options field of a send request. The “send control” request is another such mechanism, giving the client direct access to generating a control packet.

The “get state” request returns the internal state of the client’s context. The “release” request causes all of the resources allocated to this client to be returned, and the key associated with the context to become invalid.

#### **4.6. Buffer Management**

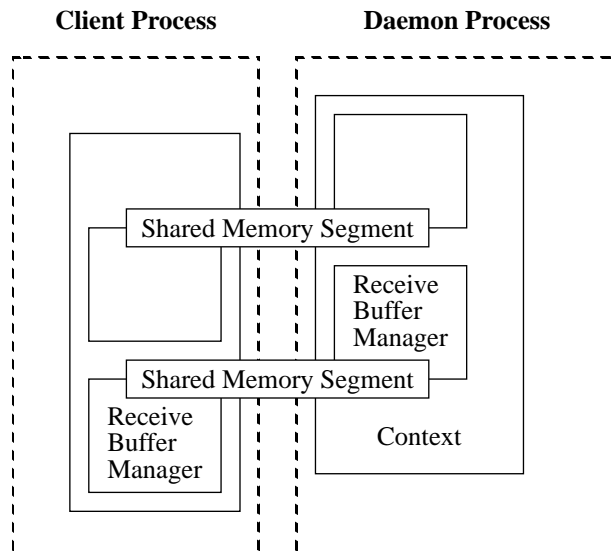
A client has two data buffers, one for sending and one for receiving. Each of the client’s buffers is controlled by a buffer manager. When the client registers these buffers with the daemon, the context assigned to the client attaches its two buffer managers to the client’s buffers. In this way, each buffer is “controlled” by two buffer managers—one at the user application side (and hidden in the user interface routines), and the other in the context associated with this user. In general, one of the managers writes to the buffer and the other manager reads from it, so there must be some way of coordinating the head and tail markers for the buffer. This is done via user request commands; when a send request is issued, the interface object places the new marker values into the request so that the context’s buffer manager will know what data to send. A similar exchange happens for receiving data through the receive buffer.

The buffers themselves are implemented as segments of shared memory. The user application, through the user interface library, creates two shared memory segments. The identifiers for these segments are relayed to the daemon, where the newly initialized context for this user attaches the daemon process to the shared memory segment. This detail could, of course, have been accomplished with any number of IPC facilities, and replacement of the underlying mechanism for this object is easily done as long as its replacement uses the same class interface. Fig. 6 shows how the buffer managers, the shared memory segments, and the client and daemon are related.

#### **4.7. The User Interface**

The user interface is also implemented as a base class and a protocol-specific derived class, but it is not compiled into the daemon. Rather, it is targeted as a library to be compiled into the user’s application code. The interface is instantiated as an object, although this is not entirely necessary. The important part of the interface is that it provides the user’s application a means of issuing requests to the daemon and of managing the data in the user’s send and receive buffers.

The user interface controls the user’s end of the two buffer managers mentioned above. The user writes data into these buffers and issues the send command. The presence of the data is made known to the context, and the context’s buffer manager pulls the data into the protocol. As packets with data arrive, the data is written into the receive buffer. As the user application asks for data through the receive request, the context



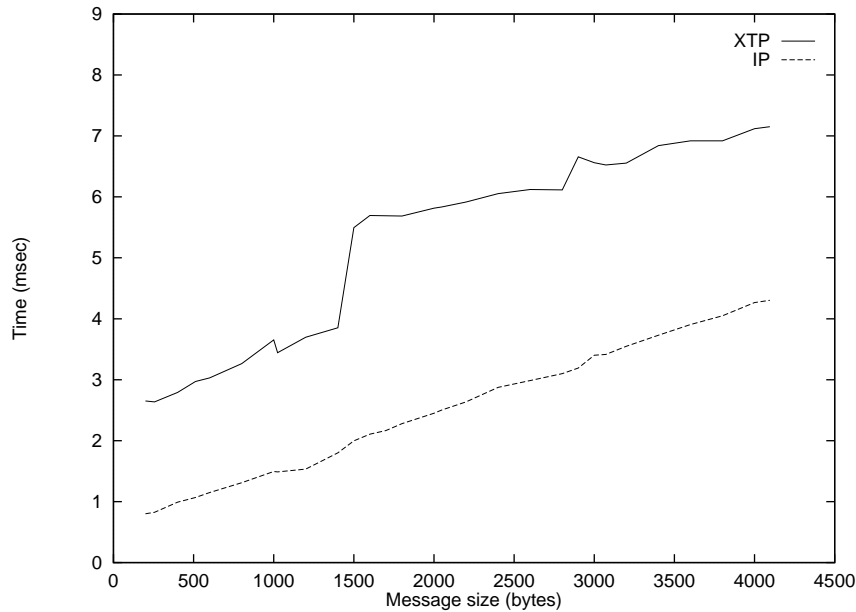
informs the interface's receive buffer manager about the size and location of the received data.

The MTL interface base class, `mtlif`, has six member functions:

```
int install_buffers(int snd_buf_size, int rcv_buf_size,
                  char* sndaddr, char* rcvaddr);
int issue(reqbuf* req);
int inform(reqbuf* req);
buffer_manager* get_s_bm(); // send buffer
buffer_manager* get_r_bm(); // receive buffer
virtual void perror(int res, char* usr_msg);
```

The `install_buffers()` method creates the buffer managers. They are accessed via the `get_r_bm()` and `get_s_bm()` methods. User requests are sent to the daemon via the `issue()` method, which waits for the results, and the `inform()` method, which does not wait. The `perror()` method prints the error messages corresponding to return codes.

The SandiaXTP user interface derived from `mtlif` is called `xtpif`. This class uses the methods from `mtlif` to generate and issue the requests given in Table 1. These requests reflect the several basic things a user would need from XTP and, as such, they are fairly raw. Different interfaces can be constructed by deriving a class from the `xtpif` class. In this way, standard APIs, like sockets or TLI, can be emulated, and code written to use those APIs would not have to be modified to use XTP.




---

## 5. Discussion

It is difficult for user-level protocol implementations to compete with kernel implementations, mostly because user-level processes have to use system calls to gain access to services which reside in the kernel. Crossing domain boundaries usually requires buffer copies. Without direct access to the network device, efficiencies such as flow-through checksumming are not possible. Another problem is that a context switch is required to change from the client process to the daemon process. On the other hand, userlevel processes are much easier to build, debug, and configure. Further, the trend is to move functionality out of the kernel, provide better access to the functionality left in the kernel, and reduce the cost of context switching.

One of the data delivery services we have implemented is IP via raw sockets. In order to evaluate how expensive SandiaXTP is, we conducted a few latency tests for SandiaXTP and raw IP. Fig. 7 shows these two curves. Latency was measured from user process to user process on two SGI Indies over Ethernet. One process sent data to the other; that data was completely received, then the receiving process sent the same amount of data back to the original sender. The one-way latency is this roundtrip time divided by 2. To get statistically viable results, we took five samples of 50 iterations for each message size. Checksumming was enabled across the whole XTP packet.

Since SandiaXTP is running over the raw IP service, the difference between the curves is roughly the amount of time SandiaXTP is processing packets. From the graph

we can see that SandiaXTP adds about 2 msec processing overhead for packets smaller than 1448 bytes. The size 1448 is the maximum PDU size used because we wanted an XTP packet to fit fully within one Ethernet frame. The characteristic jump in the graph is due to SandiaXTP segmenting and reassembling messages into two (at 1449 bytes) and three (at 2986 bytes) XTP packets. (Performance measurements of a kernel implementation of XTP versus IP, TCP, and UDP can be found in [7].)

There are several areas where improvements can be made to the SandiaXTP implementation. Checksumming is as costly as a data copy; inlining efficient assembly code could help reduce this cost. Similarly, the cost of conversion from host to network byte order can be reduced with more efficient methods. Currently, asynchronous packet arrivals and timer expirations are caught by trapping signals, yet we know that processing interrupts is fairly costly. Also, reducing the cost of crossing the user/kernel domain boundary will improve performance. Some of these improvements are simple, others, such as making the boundary less of a barrier, are more difficult. Nonetheless, our initial experience with implementing a protocol in user space has been satisfactory.

## References

- 1 W.T. Strayer, B.J. Dempsey, A.C. Weaver: XTP: The Xpress Transfer Protocol. Reading, Mass.: Addison-Wesley 1993.
- 2 XTP Protocol Definition, Revision 3.6. PEI-92-10, Protocol Engines, Inc., January 1992.
- 3 S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman: The Design and Implementation of the 4.3BSD UNIX Operating System. Reading, Mass.: Addison-Wesley 1989.
- 4 N.C. Hutchinson, L.L. Peterson: The *x*-Kernel: An Architecture for Implementing Network Protocols. IEEE Transactions on Software Engineering 17(1), 64-76 (1991).
- 5 J. Boykin, D. Kirschen, A. Langerman, S. LoV erso: Programming Under Mach. Reading, Mass.: Addison-Wesley 1993.
- 6 C.A. Thekkath, T.D. Nguyen, E. Moy, E.D. Lazowska: Implementing Network Protocols at User Level. Proceedings of SIGCOMM '93, San Francisco, Ca., September 13-17, 1993, pp. 64-73.
- 7 W.T. Strayer, M.J. Lewis, R.E. Cline, Jr.: XTP as a Transport Protocol for Distributed Parallel Processing. To appear in Proceedings of the USENIX Symposium on High-Speed Networking, Oakland, Ca., August 1-3, 1994.

## Acknowledgements

The authors wish to thank Paul S. Wang for his expert assistance in developing the object-oriented model for MTL, and the members of the XTP Forum for testing and suggesting refinements for MTL and SandiaXTP.