

# A Class-Chest for Deriving Transport Protocols

W. Timothy Strayer

Sandia National Laboratories, California  
*strayer@ca.sandia.gov*

## Abstract

*Development of new transport protocols or protocol algorithms suffers from the complexity of the environment in which they are intended to run. Modeling techniques attempt to avoid this by simulating the environment. Another approach to promoting rapid prototyping of protocols and protocol algorithms is to provide a pre-built infrastructure that is common to transport protocols, so that the focus is placed on the protocol-specific aspects. The Meta-Transport Library is a library of C++ base classes that implement or abstract out the mundane functions of a protocol; new protocol implementations are derived from the base classes. The result is a fully viable user-level transport protocol implementation, with emphasis on modularity. The collection of base classes form a “class-chest” of tools from which protocols can be developed and studied with as little change to a normal Unix environment as possible.*

## 1. Introduction

A networking protocol specification is an abstraction of a process whose inputs are packets and user requests and whose purpose is the delivery of user data. An implementation of the protocol is the instantiation of its abstraction onto a set of control and data structures so that the process described in the specification operates on real inputs. Since experimentation with alternative protocol designs can be expensive in production systems, simulations or mathematical models are often used to predict the nature of the protocol before the investment in a full-scale implementation. Another tact is the development of an environment that facilitates the implementation and deployment of experimental protocols, one where software engineering techniques defray the investment of time and effort. We describe such an approach here.

Traditionally, communication software has been implemented in the kernel for performance and security reasons. In recent years, there has emerged a variety of reasons for departing from the classical monolithic kernel-based implementation approach. Among those often cited are ease of debugging, ease of code maintenance, flexibility in the face of widely varying application requirements, and

configurability through the composition of basic protocol functions. There are two points of departure from a classical kernel-based architecture: designing a new operating system abstractions, as with [1][2][3], and moving the communication services out of the kernel, as with [4][5][6][7].

The x-kernel [1] and ADAPTIVE [2] approaches overcome inflexibility in protocol implementation and deployment through composing protocol functions via *mini-protocols* or *protocol machines*. The Conduit [3] model uses object-oriented language constructs and design methods such as inheritance, dynamic binding, and delegation to implement the protocol state machine in a highly modular fashion. The Jetstream and Afterburner [5][6] experience prove the viability of circumventing the kernel. Library approaches [4][7] move much of the protocol processing into the user process, enhancing code maintenance, debugging, and experimentation.

Our approach combines the software engineering attributes of object-oriented development environments without the ambition of an entirely new operating system or communication subsystem structure. An agent in the form of a user-level daemon process embodies the protocol implementation, yet provides the flexibility to develop and deploy new protocols and protocol algorithms easily.

The Meta-Transport Library (MTL) [8] provides a toolchest in the form of protocol-inspecific base classes from which specific protocols can be built through derivation from these classes. This “class-chest” assembles in one package all of the common components of transport protocols, implemented once and easily used. The resulting set of classes, along with the member variables and functions within these classes, provide insight into the definition of transport protocols. A specific transport protocol built in this way is a fully viable user-level protocol implementation.

Our goals with MTL are to allow an implementor to rapidly prototype a protocol implementation without any kernel modifications or special hardware support, and as little use of root privilege as possible. Indeed, few assumptions about the programming environment — save that it is some flavor of Unix — are made. Toward this end, MTL has

these design characteristics: portability, adaptability, configurability, and readability [9]. It is not a design goal for derived protocols to compete with kernel implementations with respect to performance, although the internals of MTL were built with efficiency in mind.

**Portability** MTL has been ported to most major Unix varieties, including SGI Irix, Sun SunOS and Solaris, HP HP-UX, DEC Ultrix and OSF/1, IBM AIX, FreeBSD and BSDI BSD/OS. The code compiles using the GNU g++ compiler for all platforms supported, and the native C++ compilers on those platforms where the native compilers were sufficiently up to date.

**Adaptability** The modularity of the MTL design allows for easy replacement of the underlying data delivery service. In this way, the derived transport protocol can be run over a variety of networking technologies. MTL has modules for IP (requiring root privilege) and UDP (when root privilege is not available). Other modules, including connection-oriented network services such as AAL5, are under construction.

**Configurability.** In addition to changing the underlying data delivery service, replacement of various protocol control algorithms is easily done when the protocol implementation is modular.

**Readability.** This approach is designed to enhance the understanding of how protocol components interoperate, and to allow a designer to replace components without undo effort. C++ separates interfaces from implementations, and encapsulates concepts into modules.

## 2. The MTL model

Transport protocols are recognized by these common characteristics: they provide data delivery from a transport user to one or more transport users, using the services of the network layer, with some degree of completeness and order. Some transport protocols, such as UDP, add only per-user addressing to the network layer service. Others, like TCP, provide fully reliable data streams, while others still fit somewhere along the continuum of services.

If the essence of a transport protocol were extracted, the result would be the necessary component abstractions. There are five: Transport protocol implementations send and receive *packets* via the use of some *underlying data delivery service*. Typically this is the network layer, but with ATM and other switched media, this is not necessarily always the case. The packets carry data and control information, the latter of which helps change the state of the communication. This state is maintained by some *context*. The *context manager* is the agent that demul-

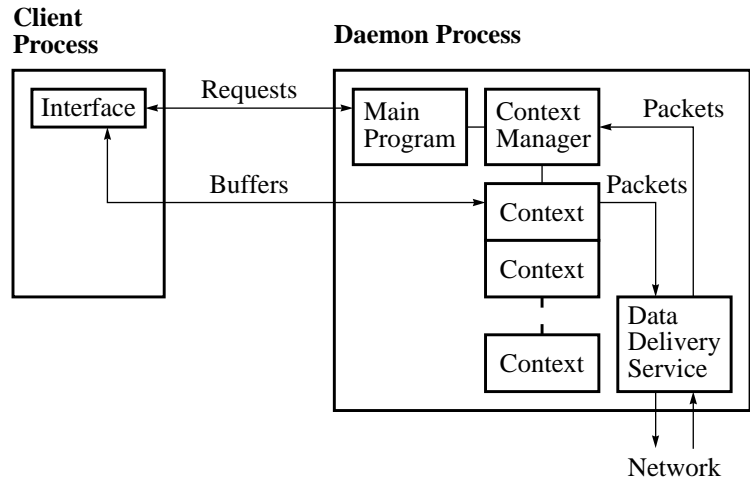


Figure 1 — The MTL Model

tiplexes incoming packets and incoming user requests, delivering these to the proper contexts. Finally, the *user interface* is the abstraction through which access to the functionality of the protocol is granted to the user.

In Unix sockets implementations of transport protocols, constructed packets are given to the underlying data delivery service via entry points into IP; protocol control blocks maintain the state information; the socket data structure maintains a list of active protocol control blocks; the socket entry points provide the interface.

MTL is a set of C++ base classes designed to present an infrastructure for building transport protocols. The classes represent the necessary protocol components, and the member variables and functions of these classes represent the state each component must keep and the work each must do. A particular protocol is derived from MTL by extending the base classes with protocol-specific algorithms.

Figure 1 shows the general MTL model. A client process uses the user interface object to send requests to the daemon process (whose interface is the daemon object) via an IPC facility agreed upon and built into these two objects. The daemon object returns the result of the request via this IPC facility as well. User data, orthogonally, are written to and read from two buffers that are also kept by the user interface object. The main loop of the daemon object accepts the user requests and uses the context manager to direct the requests to the proper contexts. Some of these requests may cause the contexts to generate packets; these are constructed and sent through the data delivery service object. The daemon object also listens for incoming packets, and uses the context manager to steer them to the proper contexts for processing.

The object-oriented programming technique generally provides some correlation between the constructs and the purpose of the information. Some of the C++ constructs

map nicely onto implementation guidelines. This is, of course, a product of the software engineering characteristics of object-oriented languages; MTL exploits these characteristics to show which aspects of protocols are common and which require protocol-specific knowledge to implement. A virtual function within one of these classes suggests that the MTL implementation may not be sufficient, and additional protocol-specific processing may be necessary. A pure virtual function implies that an implementation must be provided by the derived class; these methods are mandated by MTL but require protocol-specific knowledge to implement.

In general, there are three questions that help determine the division of labor in MTL and, consequently, how the class structures are populated. The first question — what are the major concepts — leads to the five major classes described above. For each piece of functionality, the answer to the second question — who owns this functionality — helps place the function into the class structure. Likewise for the state kept within the protocol, the third question — who owns this data — helps determine where the state should be kept. These questions also apply to the ownership of the major classes themselves; the daemon object owns a context manager object, which in turn owns the context objects. Exceptional means required to access certain functionality or data is good indication that the functionality or data are ill-placed.

These questions, and how they are used to develop the class structure, are not unique to object-oriented protocol implementations, but protocols have a fairly well-defined set of functionality and associated data, so the mapping process is rather straightforward.

There are six main classes within the MTL library package, five of which correspond to the main abstractions named above: a data delivery service class, a packet class, a context class, a context manager class, and a user interface class. The sixth class is a daemon class that wraps everything into an entity that can be handled by the operating system. Each of these classes except for the data delivery service class is designed to be a base class for a protocol-specific class. While these classes, and the particular protocol's functionality, cannot be known until derive-time, MTL ties together the protocol infrastructure through the dynamic binding of the virtual functions.

In the next several sections, we will describe the six major MTL classes and their methods, and how these classes fit within the MTL model of a transport protocol implementation.

## 2.1. Packets

Central to any communications protocols are the mechanisms used to deliver data and state information. These

mechanisms are, of course, packets. Transport layer packets are sent and received by the network layer delivery service, where the contents are simply payload. The same is almost true for MTL: the actual structures of a specific protocol's packets can only be known at derive time. Consequently, MTL must provide two things: a way to get to the packet's interior, and a way to send the packet.

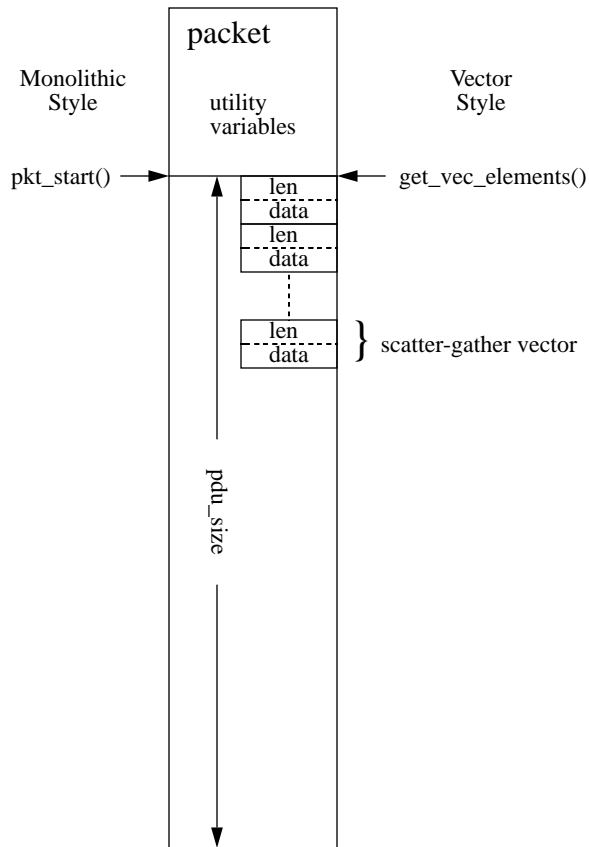
As an abstraction, the packet is a shell with enough memory to hold the largest packet, and functions to access that memory and cause it to be sent (note that a receive function is not necessary; we speak to this later). In MTL, there are two styles for packet objects, a monolithic piece of memory, as the abstraction suggests, and a scatter-gather vector of pointers and lengths. This is shown in Figure 2. The second style is included as an efficiency to reduce the amount of copying, and is especially useful if the underlying data delivery service can operate on such a vector.

The methods in the packet class reflect the two main things that happen to packets: some action is taken upon them, and some access to their contents is required. The *send()* methods are the most obvious actions upon a packet. The packet class has an overloaded *send*, one for the monolithic and one for the vector style. Within these methods, the appropriate data delivery service member functions are invoked to actually send the packet. In this respect, the packet class hides the use of the raw data delivery service from the sender of the packet (usually a context object, who constructs and sends packets on behalf of the user). This hiding ensures that neither the context nor any other entity working on behalf of the user need know anything about the actual data delivery service being employed.

Two other functions that act on a packet are the *host\_to\_net()* and *net\_to\_host()* byte order representation converters. These are virtual functions since the size of a data item determines how to convert, and the data items are only known when the packet structure is defined at derive time. These are not pure virtual functions because the protocol's correct processing is not dependent on them; conversions are not part of the essence of a transport protocol. These converter functions are intended to do in-situ conversions of the packet.

Access to the data within a packet is fundamentally different for monolithic and vector styles. For the monolithic packet, a method *pkt\_start()* returns the point from which offsets can be added to access data items deeper into the packet. Vector style packets are more difficult. First, it must be known which vector element holds the data item desired, then the vector of these elements (gotten via a method *get\_vec\_elements()*) is indexed to give the proper vector element. The pointer to this element is then used as the basis from which an offset is added to find the data item.

Constructing packets in the monolithic style is an access-and-set operation. The data item is found, then the



```

// As monolithic contiguous memory
void init_as_mono();
int is_mono();
byte8* pkt_start();
short16 xsum(register int len);
int send(dds_address* dest, int length);

// As scatter-gather vector
void init_as_vector();
int is_vector();
int add_vec_element(char* p, int len);
vec_element* get_vec_elements();
int get_num_vec_elements();
short16 xsumv(int nsv);
int send(dds_address* dest);

// DDS address information
void put_from(dds_address* from);
dds_address* get_from();

// Usage counts
void init_use_count();
void increment_use();
void decrement_use();
int is_unused();

// Virtuals
virtual void host_to_net() { }
virtual void net_to_host() { }

```

**Figure 2 — Packet Class**

new value is placed there. For vector style packets, the vector elements are added to the packet in front-to-back order using the *add\_vec\_element()* method. The total number currently included is returned from the method *get\_num\_vec\_elements()*.

Two checksum functions, one for monolithic and one for vector, are also provided. These require access to the packet, from its start, and then sum over some number of bytes (provided by the parameter to the calls). This allows partial checksumming, say of just the header. The checksum provided here is the 16-bit IP checksum; others may be provided by the derived classes. (These methods are not overloaded because their signatures are the same.)

There is no receive method symmetric to the send, since receiving is not an act upon a packet. Network layer data simply arrive at the data delivery service, without structure or meaning; this data must be “cast” into the proper packet structure so that the contents may be retrieved.

Packets in the monolithic style are typically used when network layer data are being received, since there is no way to know how to “scatter” the network layer data. The vector style, however, is quite useful for assembling a packet, since headers and control structures and even the payload itself come from different sources.

## 2.2. Contexts

A context object is the collection of all state information for an endpoint of a communication, and is the representative of the user for that communication. In MTL, the context class provides the basic tools for accessing identification and addressing information, managing user buffers, moving through basic states, blocking on user requests, and processing incoming packets. All contexts are created at once at protocol start-up time, and all are quiescent (unused but ready). As contexts are needed, they become active (anything but quiescent), then return to quiescence when they are no longer needed. When a user process requires the services of the protocol, the context manager (described below) assigns a context to the user. From that point on the context is no longer quiescent; it has state, it has an identification key, and it has a knowledge of who the user is.

Figure 3 shows some of the information kept within a context object. In some way or another these pieces of information are necessary for a transport protocol implementation: the identification of the context (the key), the identification of the user (the user’s process ID and the user’s IPC address), the transport service access point (the

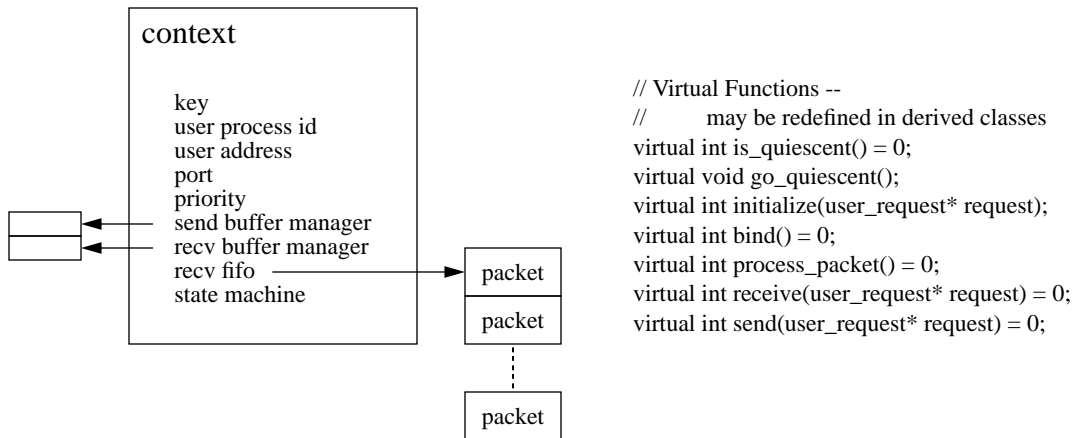


Figure 3 — Context Class

port), the priority of the context, access to the data buffers (through the send and receive buffer managers), access to incoming packets (the receive packet FIFO), and a state machine.

A context is initialized — and caused to become not quiescent — when the *initialize()* method is called on the context object. MTL defines this method, but leaves it virtual for redefinition; the derived class's *initialize()* method must call the base *initialize()* since some internal non-transport-essential things are done there. Among the structures that are initialized are the send and receive data buffers. The actual buffers are shared between the user interface object in the user process and the context in the daemon process; buffer manager objects at each process manipulate the data within.

The next thing that happens is an address is bound to the context with the *bind()* call. This address is either the destination address or it is a filter for receiving. The *send()* and *receive()* calls are pure virtual functions for sending and receiving user data as described in the user request structure.

The pure virtual function *process\_packet()* is called for each packet that has been put on the context's receive packet queue. This function is pure virtual because the processing of the packet only makes sense when the packet's internals are known, and that is only after derive time. The function may act as a clearinghouse, first determining the packet type, then calling the appropriate helper function to parse that type of packet.

### 2.3. The context manager

The context manager object manages all of the contexts in the system. These contexts are all instantiated at daemon start-up time and are kept in an array so that the context manager can easily access them. Since the size of the context is not known until derive time, the derived context

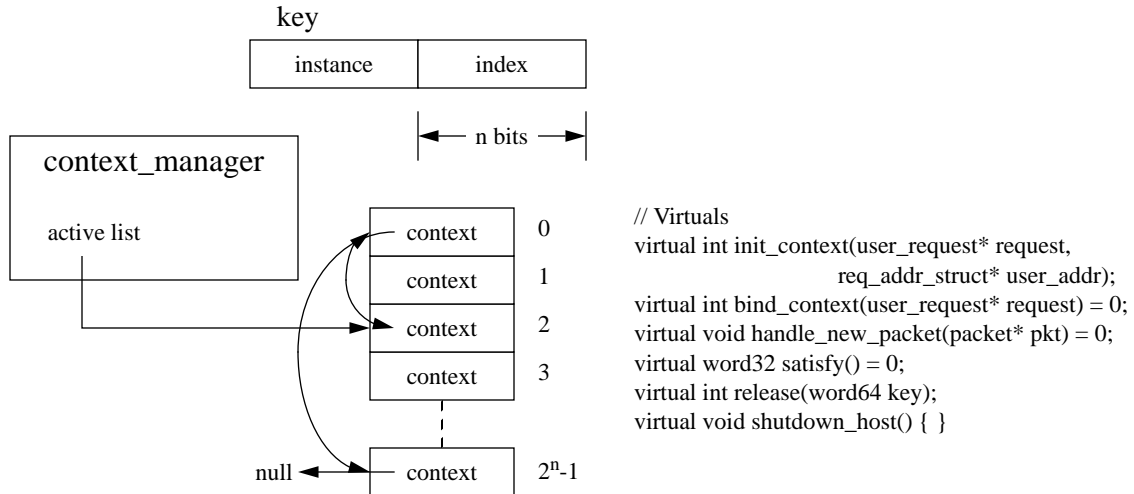
manager must allocate the array of derived contexts. The context manager base class in MTL assumes that a pointer to contexts will be set by the derived context manager object.

Each context is identified by a *key* value. The key is a 64-bit value divided into two parts, one for the *instance* of use, and one for the *index* into a conceptual array of contexts, as shown in Figure 4. In MTL, the index for a particular instantiation of a context never changes, but the instance is incremented after each use. This gives the key value the property that it is unique over the uses of the context (since the key is 64 bits wide, the time to wrap also ensures that the key value is unique for a very, very long time (see [10] for details on why the uniqueness of connection identifiers is important).

When a user registers with the daemon, the context manager method *init\_context()* is called to prepare a context for the user. Within *init\_context()*, the context manager chooses a quiescent context, calls that context object's *initialize()* method, and assigns to that context a key value so that all other user requests carrying that key value will be matched to this context.

When the user submits a bind request, the context manager method *bind\_context()* completes the address, if necessary, and then calls the context object's *bind()* method. Once the user is registered and an address has been bound to the context, the user's request are all forwarded directly to the context by the daemon. This is possible because each request carries the context's key value.

Incoming packets must also be matched to the appropriate context. The context manager provides the method *handle\_new\_packet()* for determining the context and passing the packet on to that context's receive queue. Periodically, the daemon calls the method *satisfy()*, which passes over each active context and gives the context a chance to do outstanding work, including handling the packets on the



**Figure 4 — Context Manager Class**

receive queue with the context's *process\_packet()* call.

When the daemon is about to cease, it calls the context manager's *shutdown()* method to return all of the resources. This function has an implementation but is virtual so that additional, protocol-specific activities can happen, such as aborting open associations.

## 2.4. The user interface

The user interface class is the base class for code that links into a user program in the form of a library and provides access to the functionality of the protocol. MTL's user interface class is called *mtlif*; this compiled code, and compiled code for auxiliary classes, are gathered into a file called *interface.o*. When a specific protocol's interface is defined, and a user interface class is derived from *mtlif*, that compiled class code and the *interface.o* object code are formed into a *protocol interface library* to be linked into the user application programs.

A user process can have more than one user interface object instantiated at the same time, for multiple open associations. The *lib\_manager* object is an auxiliary object that manages the several user interface objects, and multiplexes user requests to the daemon through a single IPC channel to the daemon process.

The user interface's methods encapsulate the raw functionality of the protocol. Precisely what methods are offered, and the functionality provided, cannot be determined until a protocol-specific user interface class has been derived. The methods in the *mtlif* class, however, provide rudimentary communication of user requests to the daemon, and access to the data buffers through buffer manager objects.

User requests are sent to the daemon via IPC facilities hidden in the interface methods. User data are made avail-

able to the daemon through buffer managers at both the user interface and the context objects. This is shown in Figure 5.

There are three methods that send user requests and accept responses. The *issue()* method sends a user request and waits until a response is returned. By withholding the response, the daemon can cause the user to block; there are mechanisms within the user request structure for indicating to the daemon if the user is willing to block waiting for the response. The *inform()* method sends the user request but does not wait on a request. This is useful for situations where the user process is ceasing or for synchronizing the buffer manager states. The *accept()* method is used when there are more than one responses from the *issue()* call; *accept()* blocks waiting on a message from the daemon.

While the protocol-specific interface class will provide the methods appropriate for that protocol, there are two that must be provided: the register command and the release command. Implementations for these virtual methods are provided, since these are basic to the working of MTL but are not protocol specific, but more complex and specific register and release commands can also be defined. When the *reg()* method is called, the user request message received by the daemon causes the context manager to issue the *init\_context()* method; the response from the daemon has the key value filled into the user request structure so that all subsequent requests are directed to the newly activated context. The *release()* method is required to tell the daemon to free the context's resources.

## 2.5. The data delivery service

The data delivery service class *del\_srv* is an abstract class specifying the interface to a data delivery service system for sending and receiving packets. Classes derived

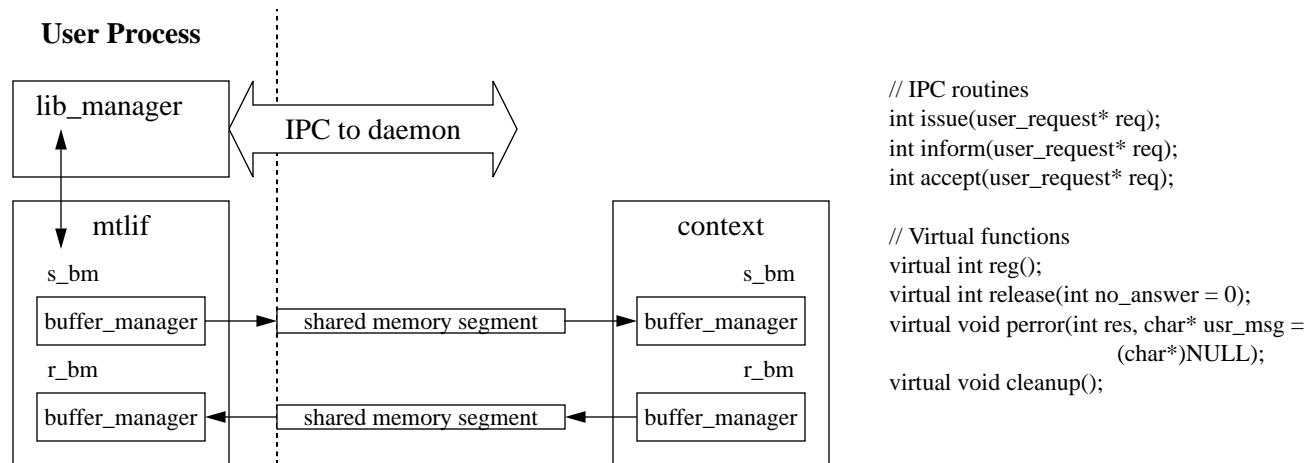


Figure 5 — User Interface

from `del_srv` implement this abstract interface by employing a particular data delivery service. An instantiated derived data delivery service object is the daemon's access point to the network.

A particular data delivery service class, such as for IP and UDP, is derived from the `del_srv` class. In MTL, the IP data delivery service object is called `ip_del_srv`, and the UDP data delivery service object is called `udp_del_srv`.

## 2.6. The daemon

The functionality necessary for the protocol implementation program to become a user-level daemon process is provided by the `mtldaemon` class. Besides launching the process as a daemon, the `mtldaemon` class provides methods for running the protocol, as shown in Figure 6. Since the daemon must communicate with the user processes, the IPC functions `recv_request()` and `send_request()` encapsulate that detail.

Two static variables are presented as public to allow any other object in the system access to these variables. The first variable, `pool`, is the object where new packet shells are gotten and returned. The second is the pointer to the data delivery service. Both of these objects are used by many of the components of the system; by making them public and static, they do not require gratuitous interface functions which would provide no more protection than they receive as public variables.

The `is_another_daemon_running()` method checks to see if there is another MTL-derived daemon process currently running. This is important to know since things like the shared memory or the IPC channels could possibly conflict. It is possible to have two or more MTL-derived daemon processes running, but care must be taken to avoid these conflicts.

The method `parse_args()` customizes the daemon and

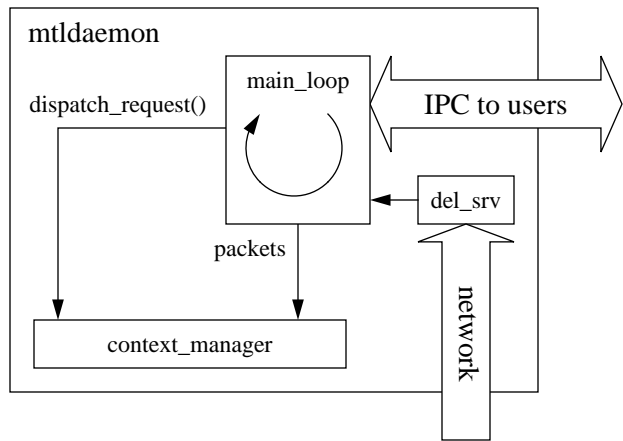
the way the protocol will act via implementor-defined arguments. When the program is ready to convert itself into a daemon, the function `init_daemon()` is called with a protocol name or number. This protocol identifier is to be used if the data delivery service also handles other types of protocols (for example, IP demultiplexes the IP packets according to the `ipproto` field in the IP packet header). When the `init_daemon()` call returns, the program has been turned into a Unix daemon process. During that process, interrupt handlers are installed with `install_handlers()`. This helps the daemon to trap certain signals and handle them properly, like invoking `shutdown()` to cause the daemon to cease.

The method `main_loop()` awaits some form of input — either from the user request channel, or from the data delivery service — or it will time out. If a user request has arrived, the request is given to the `dispatch_request()` method, and the request is parsed and the context manager is invoked. If a packet arrived from the data delivery service, the context manager's `handle_new_packet()` method is called to start processing the packet.

## 3. Observations on performance

We have used MTL to build an implementation of the Xpress Transport Protocol for prototyping revision 4.0 [11] before it became official, and for studying new multicast strategies within XTP, described in [12]. It has been our intention all along to also implement the internet protocol suite, but our overall research plans have not compelled to do so yet. Consequently, it is very dangerous to present performance comparisons of our MTL-based implementation of XTP (called SandiaXTP [13][9]) and a kernel-based implementation of TCP, but we will attempt to draw some salient observations nonetheless.

Our experiments were run between two SGI Indy work-



```

// Static variables
static packet_pool* pool;
static del_srv* dds;

// IPC to user processes
int recv_request(user_request* reqmsg, req_addr_struct* user_addr);
int send_reply(user_request* reqmsg, req_addr_struct* user_addr);

// Main loop functions
int is_another_daemon_running();
void main_loop();
void shutdown();

// Virtuals
virtual int parse_args(int argc, char** argv);
virtual int init_daemon(char* protocol);
virtual int init_daemon(int protocol_number);
virtual void install_handlers();
virtual req_action dispatch_request(user_request* request,
                                   req_addr_struct* user_addr);
  
```

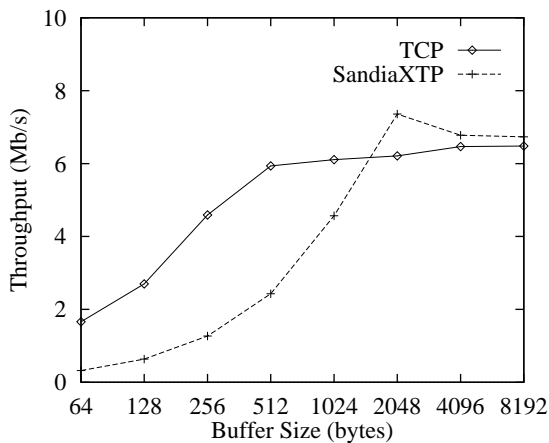
**Figure 6 — The Daemon Class**

stations over a single segment of an otherwise production Ethernet network, during the middle of a normal work day at a national laboratory. The loads on the workstations were reduced but little else special was done. The results were consistent and repeatable.

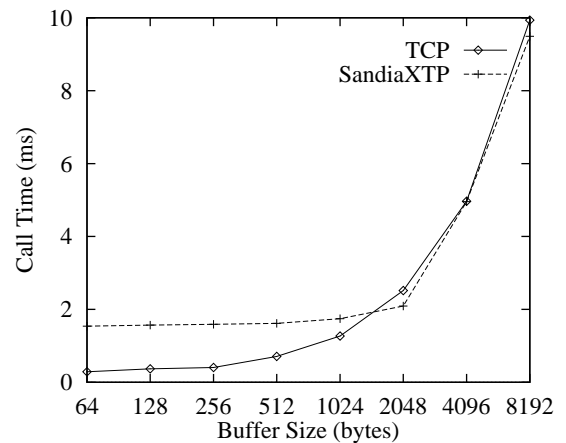
Comparing throughput, as in Figure 7, is particularly tricky. Both TCP and XTP do essentially the same protocol processing during bulk data transfers, and both use IP as their network layer delivery service, so on a gross level the curves should be similar. Indeed, the noteworthy thing is that the throughput curves are so close for protocols implemented in such desperate manners. For these results we used the popular program *ttcp* for the TCP numbers, and a modified version which uses the SandiaXTP user interface for the XTP numbers. SandiaXTP sends data as soon as the protocol has it, so the NODELAY option was used for TCP

to attempt to mimic that behavior.

Perhaps more interesting than throughput comparison are the performance results of the send call times and rates. Figure 8 shows call times for logarithmically increasing buffer size. Since SandiaXTP eventually will make at least one socket system call for every send service call, we expect that TCP should be faster, especially for small buffer sizes, where the cost of protocol processing is more evident. For larger buffer sizes, the cost of moving the data, making packets of the data, sending the packets, and doing the protocol processing, were essentially the same, whether out of the kernel or not. Figure 9 shows this a different way: the rate of service calls is much higher for TCP socket calls than for MTL-based interface calls until the buffer size reaches about 2048 bytes, when the protocol processing costs dominate.



**Figure 7 — Performance of MTL vs. Kernel-based**



**Figure 8 — Send Service Call Time**

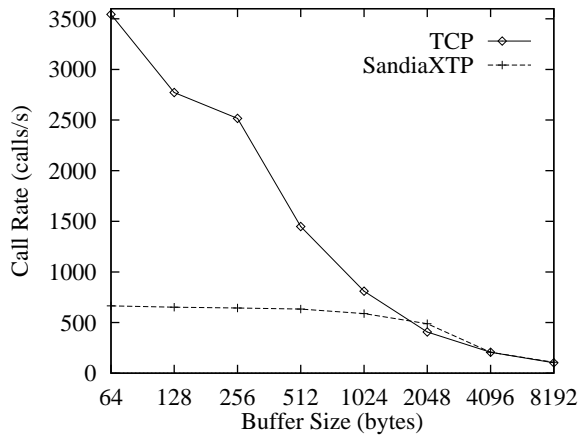


Figure 9 — Send Service Call Rate

While these numbers should not be used in protocol religious warfare, the shapes of the curves do suggest that, for similar amounts of protocol processing, user-level implementations are not *significantly* worse, and are viable means for constructing experimental implementations for study under real conditions.

#### 4. Conclusions

While in-kernel sockets-based implementations of protocols have provided good performance and a common structure and interface, they are not well-suited for modeling alternative protocol procedures or whole new protocols. It is simply more difficult to develop protocol implementations from within the kernel: the compile-and-test cycle is long and complex, debugging requires special effort, and crashes are non-trivial. Our approach is to provide an implementation toolchest with common transport infrastructure already provided, where the development environment is more natural and accessible to a wider group than kernel programmers.

Having the mundane infrastructure built and in place reduces the tedium and allows the implementor to focus on the protocol. The Meta-Transport Library provides such an infrastructure for a user-level protocol implementation. This approach support rapid prototyping of new protocols or protocol procedures, and de-emphasizes the environment within which the protocol is to run. Once the protocol has been prototyped and studied, the investment in placing it into more performance-oriented environments is less onerous.

This approach has been used to prototype XTP 4.0 [13] and aspects of the new multicast functionality of XTP, described in [12].

In addition to aiding in the development cycle of a protocol definition and implementation, the MTL approach is well suited to use in a classroom, where protocol concepts

can be taught with little or no advanced Unix programming knowledge. Almost every aspect of an MTL-derived protocol is modular, supporting replacement of algorithms. The skeletal approach also illustrates the essential components of transport protocols.

More information about MTL and — as an example protocol — SandiaXTP is found at these pages:

- <http://www.ca.sandia.gov/xtp/mtl/>
- <http://www.ca.sandia.gov/xtp/SandiaXTP/>

#### 5. References

- 1 Hutchinson, N. C., Peterson, L. L., "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, Vol 17, No 1, pp. 64-78, January 1991.
- 2 Schmidt, D. C., Box, D. F., and Suda, T., "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, Vol. 5, No. 4, pp. 269-286, June 1993.
- 3 Zweig, J. W., "The Conduit: a Communication Abstraction in C++," *Proceedings of the 2nd USENIX C++ Conference*, pp. 191-203, April 1990.
- 4 Thekkath, C. A., Nguyen, T. D., Moy, E., and Lazowska, E. D., "Implementing Network Protocols at User Level," *Proceedings of SIGCOMM '93*, San Francisco, Ca., September 13-17, 1993, pp. 64-73.
- 5 Edwards, A., Watson, G., Lumley, J., Banks, D., Calamvokis, C., and Dalton, C., "User-space protocols deliver high performance to applications on a low-cost Gb/s LAN," *Proceedings of SIGCOMM '94*, London, England, August 31-September 2, 1994, pp. 14-22.
- 6 Edwards, A., and Muir, S., "Experiences implementing a high performance TCP in user space," *Proceedings of SIGCOMM '95*, Cambridge, Mass., August 28-September 1, 1995, pp. 196-205.
- 7 Braun, T., Diot, C., Hoglander, A., Roca, V., "An Experimental User Level Implementation of TCP," *INRIA Rapport de recherche No. 2650*, September 1995.
- 8 *Meta-Transport User's Guide*, Release 1.5, SAND94-8645, Sandia National Laboratories, California, July 1996.
- 9 Strayer, W. T., Gray, S., and Cline, R. E. Jr., "An Object-Oriented Implementation of the Xpress Transfer Protocol," *Proceedings of 2nd IWACA*, Heidelberg, Germany, September 26-28, 1994.
- 10 Watson, R. W., "Timer-Based Mechanisms in Reliable Transport Protocol Connection Management," in **Computer Networks 5**, North-Holland, Amsterdam, The Netherlands, 1981.
- 11 *Xpress Transport Protocol Specification*, XTP Revision 4.0, XTP 95-20, XTP Forum, March 1, 1995.

- 12 Atwood, J. W., Catrina, O., Fenton, J., and Strayer, W. T., "Reliable Multicasting in the Xpress Transport Protocol," *Proceedings of the 21st Local Computer Networks Conference*, Minneapolis, Mn., October 13-16, 1996.
- 13 *SandiaXTP User's Guide*, Release 1.5, Sandia National Laboratories, California.

### **Attribution**

This work is supported by Sandia Corporation under its Contract No. DE-AC04-94AL85000 with the United States Department of Energy