10/4-15-97 JS (1)

SANDIA REPORT

SAND--94-8645 (7/96)

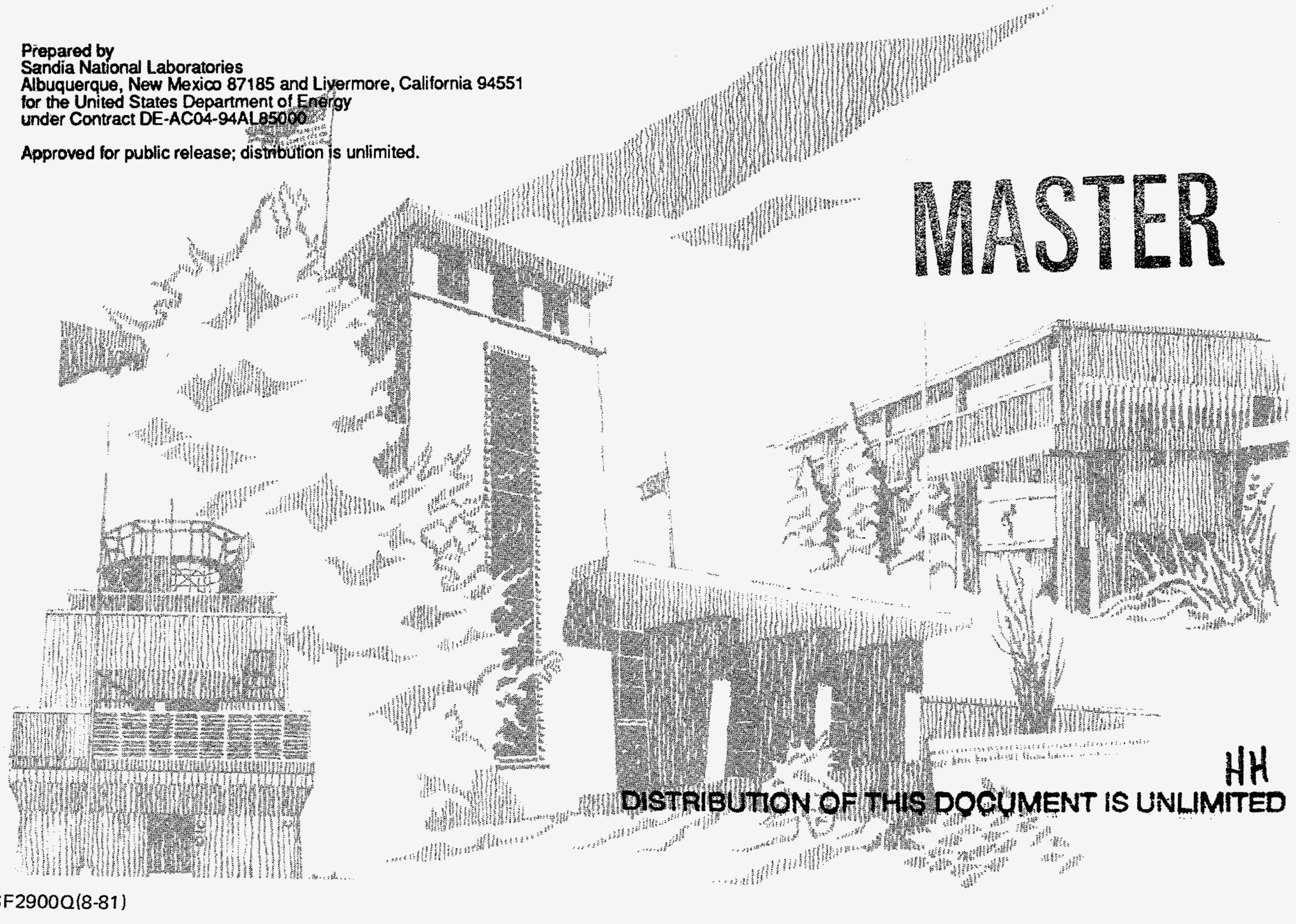
SAND94-8645 • UC-405
Unlimited Release
Printed July 1996

M97052549

META-TRANSPORT LIBRARY USER'S GUIDE

W. Timothy Strayer

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94551
for the United States Department of Energy
under Contract DE-AC04-94AL85000



HH

SF2900Q(8-81)

SAND94-8645
Unlimited Release
Printed July 1996

UC-405

Meta-Transport Library User's Guide

W. Timothy Strayer
Infrastructure and Networking Research Department
Sandia National Laboratories

Abstract

Developing new transport protocols or protocol algorithms suffer from the complexity of the environment in which they are intended to run. Modeling techniques attempt to relieve this by simulating the environment. Our approach to promoting rapid prototyping of protocols and protocol algorithms is to provide a pre-built infrastructure that is common to all transport protocols, so that the focus is placed on the protocol-specific aspects. The Meta-Transport Library is a library of 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 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. In addition to supporting protocol designers, this approach has pedagogical uses.

Table of Contents

Meta-Transport Library—A Protocol Base Class Library

Release 1.5

Meta-Transport Library User's Guide

Infrastructure and Networking Research
Sandia National Laboratories
Livermore, California

http://www.ca.sandia.gov/xtp/mtl

Introduction

A transport protocol is recognized by several common characteristics. Transport protocols 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 can be distilled, the residue would be the necessary component abstractions. There are five: Transport protocol implementations send and receive *packets* via the use the services 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 demultiplexes 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.

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 tool-chest with common transport infrastructure already provided, where the development environment is more natural and accessible to a wider group than kernel programmers. This is a particular advantage when teaching students about protocol concepts.

The *Meta-Transport Library* (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.

An MTL-derived protocol implementation is instantiated as a user-space daemon process. User-space or user-level protocol implementations have been the subject of recent discussion in the literature and at conferences. They are attractive for several reasons, including ease of code maintenance, ease of debugging, and ease of customizing [1], although fears of poor performance pervades. New operating system support [2][3] and new network interface cards [4][5] attempt to circumvent the kernel/user boundary obstacles. Packet filters [6][7] are agents within the kernel that aid in demultiplexing packets by patterns, including by protocol. Library approaches [1] move much of the protocol processing into the user process. Many of these user-level implementation approaches require modification to the operating system, hardware, or both. Our approach uses an agent in the form of a process daemon to embody the 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 [8][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. This makes the implementation process both easier to understand and easier to manipulate.

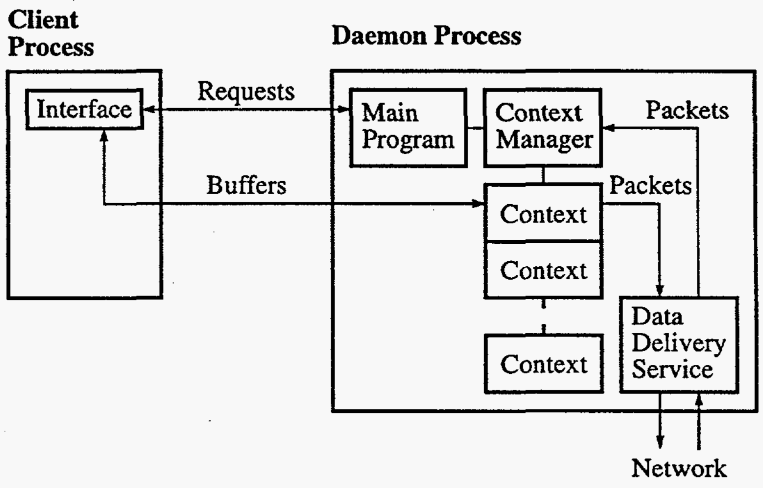


FIGURE 1. MTL Client/Daemon Model

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-orient 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.

This User's Guide describes the installation and use of the Meta-Transport Library protocol base classes. This software package includes the full source code for the implementation, as well as man pages and appropriate documents.

License Agreement

This code and its binary executables are protected by copyright and a licensing agreement; within these bounds, this software is freely available.

BEFORE YOU CONTINUE, please read the license agreement in the distribution file LICENSE. By compiling this code we assume you have read the license and accept its terms.

Here are the highlights: You agree not to sublicense, sell, or sell derivations of this software, and you agree that this software is provided without warranty, and Sandia National Laboratories assumes no responsibilities.

The provision at the end of the License regarding "export control" states that, if this software were export controlled, you must not give it to anyone outside of the United States. Fortunately, this software is *not* export controlled, so these warnings do not apply. The Department of Energy is required to place this warning in all software, even if it doesn't apply.

If you have any questions about this license or any provision within it, please contact Mike Dyer at (510) 294-2678.

Installation

The MTL software package is available via anonymous FTP from the machine **dancer.ca.sandia.gov**. If you have not already done so, download this package by following these steps:

1. FTP to dancer.ca.sandia.gov:

   ```
   $ ftp dancer.ca.sandia.gov
   ```

   or

   ```
   $ ftp 146.246.246.1
   ```

   (note that the IP address is subject to change without notice). Enter "ftp" when asked the user name, and your email address when asked for a password.

2. Change directories:

   ```
   ftp> cd pub/xtp4.0/SandiaXTP
   ```

3. Enter binary mode:

   ```
   ftp> binary
   ```

4. Get the distribution files:

   ```
   ftp> get mtl-1.5.tar.gz
   ```

   or

   ```
   ftp> get mtl-1.5.tar.Z
   ```

5. Quit FTP:

   ```
   ftp> quit
   ```

6. We would like to keep track of the user base during these early releases so we can quickly inform you of updates and bug fixes:

   ```
   $ mail strayer@ca.sandia.gov
   subject: user registration
   ```

   Give the names and Email addresses of all appropriate users.

Now you are ready to install the MTL software. There are a few decisions you should make before getting started, then follow the instructions later in this section for unpacking and compiling the system. It is always a good idea to read the whole set of instructions before starting the installation, especially since you may be doing some things as root.

Decisions

The fundamental question is whether you can use root privileges during installation, since this will determine where you install the software. If you can be root, then you may wish to use the conventional /usr/local or /usr/private directory as the installation directory. These directories are typically where site-specific software is placed. Even if you aren't root during installation, you can install the system anywhere you have privilege to create files.

Make the decision now about which directory to use as the installation directory. In the following instructions, we'll refer to this as compile-dir. When you see install-dir in the instructions, replace that with your installation directory.

Supported Compilers and Platforms

This code is written in C++ and, therefore, must be compiled with a C++ compiler. We do not use templates, so that is not a requisite when deciding which compiler to use. Currently we support the following C++ compilers:

➔ CC (AT&T cfront compiler)
➔ g++ (GNU C++ 2.6.3 or newer)
➔ cxx (DEC C++ compiler)

We support installation of this code using one of the compilers listed above on one of the following platforms:

➔ SGI workstations running IRIX 4.x and 5.x
➔ Sun workstations running SunOS 4.x and SunOS 5.x (Solaris)
➔ DECstation 5000 running Ultrix 4.x
➔ DEC Alpha workstations running OSF/1
➔ HP workstations running HP-UX 9.x (but not with CC compiler)
➔ i386 PCs running BSDI/OS 2.x
➔ i386 PCs running FreeBSD 2.x
➔ IBM RS/6000 workstations running AIX (but not with xlC compiler)

If your platform is not listed here, we have not ported this code to that architecture yet. We will continue to expand this list.

We have found that certain machine types require some modification to their standard configuration in order to compile this distribution. Please see the CAVEATS file for a list of necessary changes.

Installation Instructions

1. If you have a previously installed version, it is a good idea to remove that version before continuing. You may have to become root to do this.

```
$ cd <old mtl directory>
$ make uninstall
```

2. Select where the source files will reside. Copy the compressed tar file to that directory and change to that directory as well.

3. Uncompress and un-tar the file, and change to top-level directory:

   ```
   $ gunzip mtl-1.5.tar.gz
   ```

   or

   ```
   $ uncompress mtl-1.5.tar.Z
   ```

   and

   ```
   $ tar -xmvf mtl-1.5.tar
   $ cd mtl-1.5
   ```

4. Open a Web Browser, then open the file intro.html in this directory. Follow the link called "Installation Instructions."

   If you are unable to use a Web Browser, the information in intro.html is reprinted in text form in the file intro.txt. The installation instructions are in the file install.txt.

   In brief, the steps include:

   ```
   $ ./configure --switches
   $ make
   $ make install
   $ make installman
   $ make examples
   ```

   Once the configuration process has created the Makefiles, the *make* command will visit each subdirectory and make the appropriate object files or the library. When this completes, the *make install* command will install the library and header files in the appropriate directories under install_dir, and the "make installman" command will install the man pages into the appropriate install-dir/man directories.

The Results of Installation

When the MTL distribution file is unpacked, compiled, and installed, there are four major things that happen:

- A base class library is created and installed in a lib directory
- An interface object file is created and installed in a lib directory
- Header files are installed in an include directory
- Manual pages are installed in a man directory structure

The base class library is called libmtl.a, and it is created by combining into an archive the classes that are necessary to derive a protocol. These include the packet and packet manipulators, context and context manager, data delivery services, daemon, and buffer manager classes, as well as some utility classes such as the state machine class and event queue class. The libmtl.a library file must be linked into any derived protocol to create a program that implements that protocol.

In order to communicate with the daemon, the user processes must link in a library as well, but not all of the files are present at MTL's build time. In particular, MTL provides a user interface base class; the user interface library cannot be completed until the derived user interface class or classes are defined. As a consequence, the file interface.o is created. It is an object file containing the user interface base class, the buffer manager class, and any utilities necessary.

During installation, the libmtl.a and interface.o files are placed into the *install-dir*/lib directory.

While building the derived protocol, the derived classes will require access to the header files of the base classes. These are placed in the *install-dir*/include/mtl directory. Some of these header files, like MTLtypes.h, will be included by the user's application code as well.

The manual pages are compressed using *pack(1)* and placed into the *install-dir*/man/cat[3,5] directories as class_name.3.z for base classes, and types_file.5.z for the major types files. There are also hypertext versions of the manual pages written in HTML and viewable via your favorite browser. These are kept in the *install-dir*/man/html[3,5] directories as class_name.3.html and types_file.5.html, respectively.

Recompiling and Reinstalling

Building the Makefiles, as done in the first part of the build script, should only need to be done once as long as no changes are made to the switches to the *configure* script. These changes may include changing the compiler used, or the installation directory, or local compiler flags used. If these change, the whole configuration process must be repeated, starting with cleaning out old things:

```
$ make uninstall
$ make distclean
$ ./configure --new_switches
$ make--new_switches
$ make install
$ make installman
```

If you simply need to recompile and reinstall, just perform the two steps:

```
$ make
$ make install
```

Remember to recompile and reinstall any derived protocols after you build a new MTL library. See the appropriate Users' Guide for details.

Directory Structure

Figure 2 shows the MTL directory structure. At the top of this structure is the directory mtl-1.5. Under this directory are the directories DERIVEDprotocol, aux, doc, examples, include, lib, man, and src. The DERIVEDprotocol directory is an example of how to use MTL to build a derived protocol implementation. The aux directory is where certain configuration and auxiliary files are kept. These files aid in the configuration process. The doc directory holds the documents of this distribution. The examples directory holds a examples of how to use some of the base classes in the library. The include directory is where the header files are kept. The html directory hold hypertext instruction documents. The lib directory is where the MTL library file called libmtl.a and object file interface.o are created. The man directory holds the manual pages for this soft-

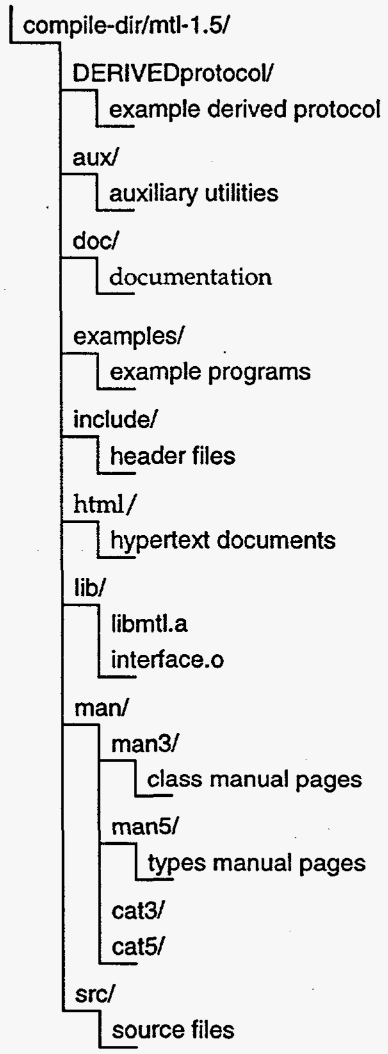


FIGURE 2. MTL Directory Structure

ware. The src directory is where the source files are kept, and where the initial compilation of the library files is done.

Although not shown in Figure 2, the configuration process sets up a symbolic link from a directory called mtl to the physical directory include, so the contents of include are accessible through mtl as well. This is done to maintain consistency with the #include directive when many of the header files from compile-dir/mtl-1.5/include are copied into install-dir/include/mtl. See the section Writing Derived Protocols for more details about using the header files once they have been installed into the install-dir directory.

Also under the directory mtl-1,5 are several text files. The files intro.html and intro.txt are brief introductions to the MTL package. Use a web browser to view the intro.html file. The file intall.txt is the installation description file. The LICENSE file holds the license agreement. It is your responsibility to read this file; compiling the source code and using the MTL library constitutes agreement with its provisions. The Makefile.in file is the template used to create the master Makefile during configuration. The README file refers the reader to the intro.html or intro.txt files. The CAVEATS file lists any known problems with various architectures. The NEW file lists improvements and fixes in this version.

Code Structure

There are six main classes in MTL, the packet class, the context class, the context manager class, the data delivery service classes, the daemon class, and the buffer manager class, and the user interface class. These classes and classes related to them are discussed below. First, however, we'll discuss the contents of the main include file MTLtypes.h.

The MTL Types

The file MTLtypes.h collects all of the standard include files, definitions, and structures required by MTL. The first file that is included is mtlconf.h, where the definitions for machine-specific idiosyncrasies are collected. The mtlconf.h file is created by the configuration process. The inclusion of many other standard header files into MTLtypes.h is dependent on which macros get defined in the mtlconf.h file.

Standard Header Files

Many of the common system header files are included by MTLtypes.h. The header files in the list below are included only if the configuration process finds them in the system:

Description	Header File
general system types	sys/types.h
standard IO	stdio.h

Description	Header File
character type manipulators	ctype.h
string manipulators	string.h, strings.h
memory manipulators	memory.h, bstring.h
standard library routines	stdlib.h
errors	errno.h
symbolic constants	unistd.h
IO control	sys/ioctl.h
file control	fcntl.h
file data structures and	sys/file.h
system entries	sysent.h
time manipulators	sys/time.h, time.h
system statistics types	stat.h
signal manipulators	signal.h
internet types	netinet/in.h
socket routines, especially UNIX domain sockets	sys/socket.h, sys/un.h

A file named prototypes.h is also included by MTLtypes.h to provide prototypes to functions when the system does not provide prototypes. This is especially noticeable when using g++; providing a prototype reduces the warning messages generated.

Basic Types

The MTLtypes.h file defines the common data types used in MTL. In particular, the data types *byte8, short16, word32,* and *word64* are used rather than *unsigned int* and *unsigned short* to reduce confusion about the size of these data types.

Type name	Description
`byte8`	8-bit unsigned integer
`short16`	16-bit unsigned integer
`word32`	32-bit unsigned integer
`word64`	64-bit unsigned integer

The type *word64* may or may not be a native type for a given architecture or compiler. For those that need the type *word64* to be defined, the file word64.h contains the declaration and definition of the type and its operators. If the *word64* type is native to the architecture and recognized or constructed by the compiler, the file MTLtypes.h defines a macro `NATIVE_WORD64` to be true.

The file also defines utility functions associated with these basic types. Since it is normal for some of the data items used in protocols to wrap from 2^n–1 to 0, there must be some way to preserve the inequality relationships as the item goes from "all ones" to "all zeros." The functions *lt32(), lt64(), gt32, gt64, umin32(), umin64(), umax32(),* and *umax64()* preserve inequalities by assuming that the difference between any two numbers being compared will be no more than one half of the range of numbers represented by the data type. For *word32* data items, x is less than y if $(x - y - 1) < 2^{16}$; for *word64*, $(x - y - 1) < 2^{32}$. Here is a list of the utility functions defined as macros:

Macro	Meaning
`lt32(x, y)`	word32 less-than comparison
`lt64(x, y)`	word64 greater-than comparison
`gt32(x, y)`	word32 greater-than comparison
`gt64(x, y)`	word64 less-than comparison
`min(x, y)`	integer minimum
`max(x, y)`	integer maximum
`umin32(x, y)`	word32 minimum
`umin64(x, y)`	word64 minimum
`umax32(x, y)`	word32 maximum
`umax64(x, y)`	word64 maximum

There are also byte manipulation routines for switching the order of bytes for *short16* and *word32* types. These are defined as in-line functions.

Function	Description
`short16 switch16(short16 i)`	returns a short16 value equal to switching the order of *i*'s two bytes.
`word32 switch32(word32 i)`	returns a word32 value equal to reversing the order of *i*'s four bytes.

The *word64* type has several special utility functions to manipulate it since a 64-bit word is not guaranteed to be native. The file MTLtypes.h defines in-line functions for manipulating the most significant bit of the 64-bit word. Printing functions are included to give both native and constructed word64 types a common set of print functions. There are also byte manipulation routines for converting from host to network byte order and visa versa.

Function	Description
`int is_hi_bit_set(word64 x)`	returns true of the high bit of *x* is set
`word64 hi_bit()`	return a *word64* with high bit set
`word64 set_hi_bit(word64 x)`	return a *word64* equal to *x* with the high bit set
`word64 clear_hi_bit(word64 x)`	return a *word64* equal to *x* with the high bit cleared
`void xprint(word64 x)`	prints *x* in hexadecimal
`void xprint(word64 x, const char* str)`	prints *x* in hexadecimal, appending *str*

Function	Description
`void uprint(word64 x)`	prints *x* in decimal
`void uprint(word64 x, const char* str)`	prints *x* in decimal, appending *str*
`void fxprint(FILE* fd, word64 x)`	prints *x* in hexadecimal to file *fd*
`void fxprint(FILE* fd, word64 x, const char* str)`	prints *x* in hexadecimal, appending *str*, to file *fd*
`void fuprint(FILE* fd, word64 x)`	prints *x* in decimal to file *fd*
`void fuprint(FILE* fd, word64 x, const char* str)`	prints *x* in decimal, appending *str*, to file *fd*
`word64 net2host(word64 x)`	returns a *word64* value resulting in converting *x* from network byte order to host byte order
`word64 host2net(word64 x)`	returns a *word64* value resulting in converting *x* from host byte order to network byte order

Defined Structures

A scatter-gather type is also defined in MTLtypes.h. This type, called *vec_element,* is used to manipulate data via pairs of pointer and lengths, especially in the packet class. It is defined as:

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

A type is also defined to represent the address structure used by the user and daemon for communicating requests. This type is called *req_addr_struct,* and it is the address structure used in the client/server relationship between the user and the daemon.

Command Codes

The *user_request* type is the header for all user requests sent to the daemon by the user, and all responses returned to the user by the daemon. The specific request will be tagged by a value in the cmd field and may have additional fields appended to the end. Several cmd field values are defined in MTL:

Command	Description
`REGISTER`	register user with the daemon
`RELEASE`	release user from the daemon
`DAEMON_STOP`	cause the daemon to stop
`DAEMON_STATUS`	get the daemon's status
`DAEMON_RESET`	reset the daemon
`DAEMON_PING`	ping the daemon

Extra Modes

The extra_modes field of the *user_request* type is used to cause the context to act in certain ways. Several extra_modes values are defined in MTL:

Extra Mode	Description
`BLOCK`	this is a blocking request
`ASYNC`	this is an asynchronous request
`NOANSWER`	this request requires no response

Error Codes

The file MTLtypes.h defines error codes that are not protocol-specific.

Error Code	Meaning
`EXOK`	okay
`EXCOMM`	could not establish communications with daemon
`EXSEND`	could not complete send command
`EXRECV`	could not complete receive command
`EXMEM`	could not allocation memory
`EXSYSC`	system call error
`EXINVA`	invalid argument
`EXPOOL`	no more packets in packet pool
`EXQUEUE`	failure in the event queue
`EXBUF`	could not create send or receive buffer
`EXNCTXT`	exceeded the maximum number of contexts in daemon
`EXSTATE`	invalid state for this request
`EXKEY`	key does not exist
`EXUSER`	invalid user
`EXPORT`	port already in use
`EXRJCT`	rejected service
`EXTMOUT`	connection timeout
`EXNETW`	could not access underlying network

MTL Parameter Macro Definitions

Certain aspects of MTL are given initial default values obtained from the definition of macros.

Parameter Macro	Description
`DAEMON_REQ_ADDR`	UNIX domain socket file name for daemon
`USER_REQ_ADDR`	UNIX domain socket file name for users
`IPC_PATH`	a well-known path name for use by *ftok(3)*
`NUM_CONTEXTS`	default number of contexts (64)

Parameter Macro	Description
`NUM_PACKETS`	default number of packet shells in packet pool (200)
`DAEMON_PORT`	well-known port number for daemon, if needed
`PDU_SIZE`	default protocol data unit (packet) size (8192)
`POLL_FREQ`	default timeout for daemon during sleep (10000 ms)
`DDS_RCV_BUF`	data delivery service receive buffer size, if able to be set (32768)
`DDS_SND_BUF`	data delivery service send buffer size, if able to be set (32768)
`MCAST_DIAMETER`	multicast transmission diameter, if able to be set (10)

The `DAEMON_REQ_ADDR` is the rendezvous point for the UNIX domain socket used to make requests to the daemon. This path is defined as /tmp/s.unixdg. The `USER_REQ_ADDR` is the UNIX domain socket address for the client (user) process. The path for `USER_REQ_ADDR` is /tmp/dg.XXXXXX. The details are covered in the User Interface section, but the idea is that a user opens a socket to the daemon on this daemon's well-known address (`DAEMON_REQ_ADDR`), and sends a request. The user will have included the return address (`USER_REQ_ADDR`) for the daemon to use in sending back the results. The `USER_REQ_ADDR` path name is actually converted to a unique file name by *mktemp(3)*, which is why the "XXXXX" suffix is included in the definition.

The `IPC_PATH` macro, defined as /etc/protocols, is a well-known file name for use by the *ftok(3)* function used by the buffer manager to get shared memory segments. Details are given in the Buffer Manager section.

The `NUM_CONTEXTS` and `NUM_PACKETS` are the default number of contexts and packets allocated by the daemon. The context manager instantiates the contexts, and they remain instantiated throughout the lifetime of the daemon. The the packet pool object instantiates the packet shells (which are of size `PDU_SIZE`) for use by the contexts and context manager as packets are received or created and sent.

The `DAEMON_PORT` is for use by a data delivery system that requires the daemon to have a well-known port. An example of such a data delivery service is UDP; the *udp_del_srv* class requires the port number for packets sent to the daemon. The class *ip_del_srv* does not require a port, so this value is ignored when the IP data delivery service is used.

The `POLL_FREQ` is a default wakeup time for the daemon. The daemon typically suspends itself until the arrival of a new packet or a new user request, or a timeout occurs. The `POLL_FREQ` is the longest the daemon will sleep. This value is chosen at 10 seconds since UNIX processes are typically paged out of core memory if they remain suspended for more than 10 seconds [10] (this may be more or less on a particular architecture; set this value to avoid the daemon being paged out as this will affect performance).

The `DDS_RCV_BUF` and `DDS_SND_BUF` are default buffer sizes for the data delivery service, if the buffer sizes can be changed. For UDP and IP, there are system calls that can reset the internal buffer sizes; the larger the send and receive buffer sizes, the lower the likelihood that a packet would be lost due to lack of data delivery service resources. These values are ignored if the data delivery service has no mechanism for changing the buffer sizes.

The MCAST_DIAMETER value is used by the data delivery service to set the distance into the network that a multicast packet will be propagated. The use of this value depends on the presence of mechanisms to set the time a multicast packet will live.

MTL also includes a debugging macro for use in both MTL classes and in classes derived from MTL. The macro,

```
DEBUG(fd, level, string)
```

prints *string* to the file *fd* given that debugging level *level* has been set using the --with-debug=LEVEL switch to the configuration script.

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 derived protocol defines the structure of its packets, and information is placed or extracted only with knowledge of the structures. Therefore, the *packet* class provided by MTL does not impose a structure on the packet, but rather provides a packet shell; packet shells are then manipulated by both MTL and the derived protocol as is appropriate.

There are two packet manipulator classes in MTL: the *packet_pool* class and the *packet_fifo* class. Packet objects are managed by a packet pool object. The packet pool is a general repository for packet objects between uses. The packet pool is responsible for allocating all of the packets in the system (NUM_PACKETS is the default number of packets allocated), and deallocating them when the daemon terminates.

Contexts and the context manager, discussed later, get packets from the packet pool to hold either incoming or outgoing packet information for the derived protocol. Each context has two packet FIFO objects, one for holding unprocessed received packets and one for holding outgoing packets that have not been sent.

```
class packet {
public:

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

        // DDS address information
        void put_from(void* ffrom);
        void* 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();
    };
```

This definition provides two ways to view a packet, as *monolithic contiguous memory* or as a *vector of scatter-gather elements*. Methods that operate on a monolithic packet use a function to get a pointer to the start of the contiguous data. Protocol-specific agents then use this pointer to read and write to offsets within the packet. The scatter-gather vector is a set of pointer, length pairs. The packet class provides member functions to set and retrieve the scatter-gather elements. This style is intended for constructing packets with minimal data copying.

Viewing a Packet as Monolithic

Since the DDS in MTL assumes that packets are received as a contiguous piece of memory, the monolithic style is generally used for processing a packet within MTL. When a packet object is gotten from the packet pool, it is initialized as a monolith (*init_as_mono()*) and its usage count is cleared (*init_use_count()*). Then a pointer to the beginning of the packet is given to the DDS. The function used for this is *pkt_start()* since a packet object actually holds more information than just the data within the packet. This code fragment illustrates this.

```
// Get a packet shell
packet* pkt = mtldaemon::pool->get();
if (!pkt) return(-1);

// Get a pointer to the packet's "from" structure
dds_address* from = pkt->get_from();

// Initialize the packet as mono, init the count
pkt->init_as_mono();
pkt->init_use_count();

// Receive the packet from the data delivery service
res = dds->recv(pkt->pkt_start(), PDU_SIZE, from);
```

The other functions are used a various times during the packet's tour of duty. The function *xsum()*, which takes *len* as an argument, computes a checksum over *len* bytes starting from *pkt_start()*. This can be used to set and check error detecting mechanisms within the protocol's packet structures. The function *send()* (which is overloaded, as will become evident below) invokes the DDS to send a monolithic packet. At any point during the processing or manipulation of a packet, but after it is initialized as monolithic (or a vector), the function *is_mono()* will return true if this packet is in monolithic representation.

There is no receive method in the packet class since receiving is not actually done *to* a packet in the same way that send is. Data simply arrives at the underlying DDS; 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.

Viewing a Packet as a Scatter-Gather Vector

Similarly, the packet object can use the scatter-gather representation. Fundamental to this style is the data pointer, length pair that constitutes one element of the vector (recall this structure from MTLtypes.h):

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

When a packet object is initialized as a vector (*init_as_vector()*), an internal counter sets the number of vector elements to zero. (Calling *is_vector()* will return true if this packet object has been initialized as a scatter-gather vector.) The method *add_vec_element()* adds a element to the vector and increments the counter. The order in which elements are added to the vector is important; if a vector style packet were laid out as a monolithic style packet, the element added first would be first in the monolithic packet. The function *get_vec_elements()* returns the vector of elements, and the function *get_num_vec_elements()* returns the number of elements in the vector. The function *xsumv()*, given *nsv* as the number of send vector elements, returns the checksum over *nsv* elements.

As mentioned above, the packet class provides overloaded send methods, one for each packet style. In both cases, the *packet::send()* method calls the DDS *send()* method. This ensures that the context, or any other agent constructing a packet, need know nothing of the data delivery service.

Usage Counts

A packet object is designed to be handled by reference to avoid excessive data copies. Several member functions are used to keep tract of the use of a packet object. When a packet is gotten from the packet pool, its use count should be initialized; this is done by calling *init_use_count()*. Each time a copy of the reference to this packet is given out, a call to the function *increment_use_count()* increases the use count. As the entities complete their use of the packet, the function *decrement_use_count()* reduces the use count until the function *is_unused()* returns true, at which point the packet can be returned to the packet pool.

Address Information

Packets are, of course, used to hold data coming from or going to the data delivery service (DDS). When a packet arrives at the DDS, the DDS must have some knowledge — directly or indirectly acquired — concerning where the packet came from. This information can be attached to the packet via the *put_from()* function call, and retrieved by the *get_from()* function call. The packet object has the memory for this address allocated by the packet pool when the packet object is instantiated, so *put_from()* copies the address into this memory, and *get_from()* returns a pointer to the memory.

Virtuals

There are two virtual functions: *host_to_net()* and *net_to_host()*. These are defined virtual because there is no way for the packet base class to perform byte-ordering conversions without knowledge of the contents of the packets; this is distinctly protocol-specific. The two functions are defined as empty virtuals in case no conversion is ever required.

The Packet Pool and Packet FIFO Classes

MTL also includes two classes, *packet_pool* and *packet_fifo*, that manipulate packet objects. The packet pool class pre-allocates and then manages the dispensation and recollection of packet objects. The packet FIFO class holds packets in a queue, then emits them in first in, first out order.

Both of these classes are friends to the packet class because they each require some special knowledge about the private workings of the packet class. Part of the extraneous information that is kept by the packet includes pointers to other packets, so the packet objects can be put into linked lists. Two methods, *get_next*() and *put_next*(), are private but accessible to packet_pool and packet_fifo.

When the packet_pool object is instantiated, the number given as an argument to the constructor is the number of packet objects that this pool should create and initially hold. The constructor then calls the *new()* operator on packet, and links each of these new packet objects together in a list. In addition to instantiating each of the packet objects, the packet pool constructor also allocates enough memory for holding the DDS addressing information referred to above.

```
class packet_pool {
public:
    packet_pool(int size);
    ~packet_pool();
    packet* get();
    packet* get(packet_tags tag);
    void put_back(packet* pkt);
}
```

When a packet object is required, a call to *get()* returns a packet taken from the pool list; when that packet is no longer needed, the *put_back()* method returns the packet object to the list. The *get()* functions are overloaded. If a packet tag (one of *Monolithic* or *Vector*) is given, the packet returned will have been initialized as that style of packet.

The packet_fifo class has the following definition:

```
class packet_fifo {
public:
   packet_fifo();
   ~packet_fifo();
   void put(packet* pkt);
   packet* get();
   packet* peek_head();
   packet* peek_tail();
   int empty();
};
```

The packet_fifo class uses the packet's *get_next()* and *put_next()* methods to build a FIFO list of packets. There is no restriction on the size of this FIFO except for the number of packets in the system. The *put()* method puts a packet at the end of the queue, where *peek_tail()* allows last-minute access to it. The method *peek_head()* allows access to the head of the FIFO without removing the packet, while *get()* does indeed remove the packet from the queue. When all packets are gone, *get()*, *peek_head()*, and *peek_tail()* will return *(packet*)NULL*, and *empty()* will return true.

The code below is in the examples directory in the MTL distribution, in the file pkttest.h.

```
#include <mtl/MTLtypes.h>
#include <mtl/packet.h>
#include <mtl/udp_del_srv.h>
#include <mtl/packet_pool.h>
#include <mtl/packet_fifo.h>

// These must be defined before using packet, packet_fifo,
// and packet_pool from the MTL library
mtldaemon* DAEMON = (mtldaemon*)NULL;

void main(int argc, char** argv) {
   if (argc != 2) {
      printf("%s <number of packets>\n", argv[0]);
      exit(1);
   }

   int ans;
   int num_pool = atoi(argv[1]);
   int num_fifo = 0;
   packet_pool* pool = new packet_pool(num_pool);
   packet_fifo* fifo = new packet_fifo;
   packet* pkt;

   printf("\n");
```

```
    do {
        printf("Pool: %d, Fifo: %d\n", num_pool, num_fifo);
        printf("1. pool -> fifo\n");
        printf("2. fifo -> pool\n");
        printf("3. quit\n");
        printf("\n  => ");
        scanf("%d", &ans);
        printf("\n");
        switch (ans) {
            case 1:
                pkt = pool->get();
                if (!pkt) {
                    printf("no more packets in the POOL\n");
                    break;
                }
                num_pool--;
                fifo->put(pkt);
                num_fifo++;
                break;
            case 2:
                if (fifo->empty()) {
                    printf("no more packets in the FIFO\n");
                    break;
                }
                pkt = fifo->get();
                num_fifo--;
                if (!pkt) {
                    printf("error getting packet from FIFO\n");
                    break;
                }
                pool->put_back(pkt);
                num_pool++;
                break;
            case 3:
            default:
                break;
        }
    } while (ans != 3);
    delete(pool);
    delete(fifo);
}
```

This example is designed to be compiled by linking in the MTL library file libmtl.a:

> **$ CC -c -DHAVE_CONFIG_H -I*install-dir*/include -O pkttest.C**
> **$ CC pkttest.o -o pkttest -L*install-dir*/lib -lmtl**

or type

> **$ make pkttest**

in the examples directory.

Contexts and the Context Manager

A context is the collection of all state information for an endpoint of an association. Certain state information is common to all transport protocols; MTL reflects this in the context base class. Much of the information is just clerical, such as identification and priority values. The *context* class provides access to these variables and references through get, put, and set functions. The context also holds the state of the communication. Most of this information is protocol-specific.

The *context_manager* class manages contexts. The context object has a relationship to the context manager object similar to the relationship between the packet and packet pool objects: the context manager is a container class for all of the contexts in the system. Similarly to the packets, the context class also has pointer variables for linking contexts together in lists that are easily manipulated by the context manager.

This is the public interface to the context class:

```
class context {
public:
   virtual ~context();

   // Context linked-list manipulators
   context* get_next();
   context* get_prev();

   // Access to things by outsiders
   word64 get_key();
   pid_t get_upid();
   void put_upid(pid_t mypid);
   int is_client_alive();
   reg_addr_struct* get_user_addr();
   void put_user_addr(reg_addr_struct* ua);
   word32 get_port();
   int get_priority();
   void put_priority(int prio);

   // Buffers
   int are_buffers_installed();
   void release_buffers();

   // Blocking methods
   void set_blocked(block_state val);
   int is_blocked();
   int is_satisfied();
   void set_blocked_msg(user_request* req, int len);
   int get_blocked_type();
   user_request* get_blocked_msg();

   // Virtual Functions--may be redefined in derived classes
   virtual int is_quiescent() = 0;
   virtual void go_quiescent();
```

```
        virtual int initialize(user_request* request);
        virtual int bind() = 0;
        virtual int process_packet() = 0;
        virtual int receive(user_request* request) = 0;
        virtual int send(user_request* request) = 0;
    };
```

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 class definition includes the following functions:

```
    class context_manager {
    public:
        virtual ~context_manager() { }

        // Validate a user
        int validate_key(word64 key, pid_t pid);

        // Adding/deleting from active list
        context* get_context(word64 key);
        context* get_active_head();
        int get_num_active();
        void add_active_context(context* c, int prio);
        void delete_active_context(context* c);
        void reorder_active_context(context* c, int prio);

        // Port assignments
        short16 get_next_port();
        int check_port(short16 port);

        // Virtuals
        virtual int init_context(user_request* request,
                                 req_addr_struct* user_addr) = 0;
        virtual int bind_context(user_request* request) = 0;
        virtual void handle_new_packet(packet* pkt) = 0;
        virtual word32 satisfy() = 0;
        virtual int release(word64 key);
        virtual void shutdown_host() { }
    };
```

To aid in manipulating the contexts, the context manager is a friend to the context class.

Key Values

A context is identified by its key value. The key value is a 64-bit handle by which the context manager can find the context among all contexts in the system. When a context is quiescent, its key value does not matter, but when it becomes active (whatever that may mean in the protocol-specific sense), a key value will be assigned to the context, and put there by the context manager.

Internally, a key is divided into to parts, the *index* and the *instance*, as shown in Figure 3. The number of contexts maintained by the context manager is always

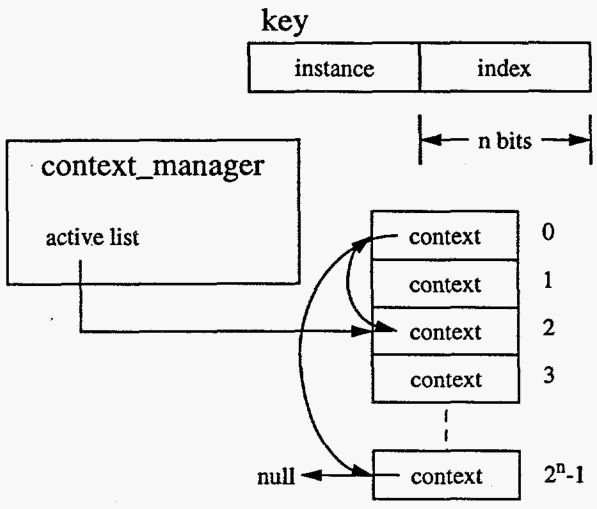


FIGURE 3. **How the Context Manager finds the Contexts**

a power of two, so the lower *n* bits, where *n* is the number of contexts log 2, are used as the index. This number will never change for a given context object. The instance is the remaining higher order bits; the instance value is incremented once per use of the context, assuring that the context object will not be reused for quite a while (there is math for this, but suffice it to say that the context will not be reused in the lifetime of the company that made the computer).

How keys are assigned to a context is really hidden by the context manager, but knowing that key are unique over uses of the context object, and are used by the context manager to find the context, are important pieces of information. This is central in validating a key (*context_manager::validate_key()*) and getting a context given a key (*context_manager::get_context()*).

User Information and User Requests

Half of the job of the context manager is dispatching user requests; the other half is giving incoming packets to their contexts. Both jobs require matching, and the key value is used — directly or indirectly — in finding the proper context. For user requests, this is straightforward: MTL dictates that all user requests have the same header format.

```
class user_request {
public:
    int cmd;
    int len;
    int result_code;
    int extra_modes;
    pid_t upid;
    word64 key;
```

```
        int snd_shmid;
        int rcv_shmid;
        word32 snd_buf_size;
        word32 rcv_buf_size;
    };
```

The command value in the *cmd* field is protocol-specific, and identifies what the request is. Some predefined commands are given in the MTLtypes.h file. The *len* field holds the total length of this structure; this is important when specialized user requests are derived from this class. The *result_code* is either one of the common error codes defined in MTLtypes.h or an addition error code defined in the protocol-specific types file. The *pid* value is the process identifier for the user process. The *key* field is used to match this request to the appropriate context; all request from the user, once the user has registered with the daemon, must contain the same key and upid values so the request can be properly delivered. The *snd_shmid* and *rcv_shmid* values are used by the buffer manager for setting up shared send and receive buffers between the daemon and the user. The *snd_buf_size* and *rcv_buf_size* are the sizes of the buffers. The section on the User Interface will discuss how actual user requests are constructed from this user request type.

The *REGISTER* cmd value instructs the user request dispatcher to initialize a new context for use by a user. The user sends the request via the user interface methods discussed later; the protocol-specific request dispatcher must call the virtual function *init_context()* with the user request and the user's address for sending results and information back to the user.

Although the function *init_context()* is virtual, it has a default implementation:

```
    // Get the next available context
    context* c = get_next_free_context();
    if (!c) return(EXNCTXT);

    // Add the context to the active list
    // (priority -1: without priority)
    add_context(c, -1);
    c->port = 0;

    // Get the key and place it in the request structure
    // so it gets returned.
    request->key = c->get_key();

    // Initialize the context with the user's request values
    int res = c->initialize(request);
    if (res != EXOK) {
        c->go_quiescent();
    }

    c->put_user_addr(user_addr);
    return(request->result_code);
```

If the derived protocol's context manager does not use the default definition of *init_context()*, the implementation of that derived context manager should at least include the functionality shown above, including getting a free context (there's a protected function *get_next_free_context()* for this), adding the context to the active list (see Active Contexts below), initializing the port number, calling the context class's virtual function *initialize()* with the user request structure, and putting the user's return address into the context structure (*put_user_addr()*).

A pure virtual function is included to bind an address to this user's context. This function, *context_manager::bind_context()*, must be defined by the derived protocol since the addressing format is only known at that time. This function should call the context's *bind()* pure virtual function.

When the user is finished with the context, the user issues a *RELEASE* command, and the context manager's *release()* function handles removing the context from the active list and returning any resources. The context *go_quiescent()* function causes the context to go directly the quiescent state.

Active Contexts

Placing a context on the context manager's active list puts the context into the processing stream. The context manager cycles through each of the active contexts allowing them to do work. In MTL, an active context is just one that is on the active list, the derived protocol's definitions of states notwithstanding.

The context manager adds a context object to the active list by calling the method *add_active_context()* with the context's address and a priority value as the two arguments. The active list is arranged in order of priority, with the highest priority contexts at the beginning of the list. A context can be moved to a different part of the active list by calling *reorder_active_context()* with the context's address and the new priority as the arguments. Direct access to the context's priority is gotten using the methods *context::get_priority()* and *context::put_priority()*, although putting a new priority value into the context with *put_priority()* does not change the context's position in the active list. When a context is no longer active, the method *delete_active_context()* removes the context from the active list.

The active list is returned by the context manager method *get_active_head()*. The list can then be traversed using the methods *context::get_next()* and *context::get_prev()*. The function *context_manager::get_num_active()* will return the number of contexts in the active list.

User Related Information

The context becomes the user's representative during communication, so it must hold information about the user process. Among this information is the user's process ID and the user's return address. When a user requests that a context be initialized, the *put_upid()* method installs the user's PID (from the user request structure) into the context. The return address is gotten from the IPC facility, and is installed into the context using the *put_user_addr()* method. Methods *get_upid()* and *get_user_addr()* allow access to these values.

Knowing the user's PID allows the derived protocol to send the process a signal when some event happens, if the daemon has sufficient permission. The

PID is also useful in the internal implementation of the method *is_client_alive()*, where the context can ask if its user has perished.

While the user makes requests via an IPC mechanism hidden by the user interface, the second way information is exchanged between the context and the user is via data buffers. Both the user and the context hold two buffer managers each, one for the send buffer and one for the receive buffer. When the user sends the request to initialize a context, as described above, the context establishes two buffers according to the size given in the user request. The status of installing the send and receive buffers can be checked with the *are_buffers_installed()* method. The buffers are released by calling the *release_buffers()* method.

According to the semantics of the user's request, the context may not respond with a result or other information immediately, but rather may hold the user's request until some condition is met. This is blocking, and the context class provides methods for effecting blocking requests. The methods is_blocked() and is_satisfied() return the answer to their questions; the results of a satisfied request can be returned to the user, and the context will become unblocked. To change the blocking status, *set_blocked()* sets the blocking state to the argument given (*NOT_BLOCKED*, *BLOCKED*, or *SATISFIED*). The method *get_blocked_type()* returns the cmd value of the request that is blocked waiting. The actual user request being blocked is recorded using the *set_blocked_msg()* call, and retrieved using the *get_blocked_msg()* call.

If an incoming user request allows blocking and the request can not yet be completed, the agent handling dispatching the user requests (probably derived from the *mtldaemon* class) can cause the user to block by setting the block state to *BLOCKED* and holding the user request message by calling *set_blocked_msg()*. When the conditions are right to unblock the request, the user request is retrieved using *get_blocked_msg()*, the request is satisfied, and the blocked state is changed to *SATISFIED*. When the context manager notices that the context *is_blocked()* and *is_satisfied()*, the context manager returns the results to the user, then sets the context blocking state to *NOT_BLOCKED*.

Handling Packets

The pure virtual function *context_manager::handle_new_packet()* is an entry point to a specific protocol's packet handling code. The simplest thing to do is to put the packet on a packet FIFO and deliver it to the proper context later. If the packet's destination context is easily discovered, this function may put the packet on the destination context's received packet FIFO.

Shutting Down

The context manager has a virtual function to be used when the daemon is shutting down. This call should return any resources the context manager gathered, and possibly notify each user that the daemon is shutting down.

The Data Delivery Service

The data delivery service class *del_srv* is an abstract class specifying the interface to a data delivery service system. Classes derived 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.

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

    // Indicates if the delivery service got initialized
    // properly
    int is_ready();

    void free_dds_address(dds_address* addr);

    // Pure virtual functions to be filled in by
    // derived classes
    virtual dds_address* alloc_addr_struct() = 0;
    virtual int recv(char* data, int length,
                     dds_address* from) = 0;
    virtual int send(char* data, int dlen,
                     dds_address* dest) = 0;
    virtual int send(vec_element* sv, int nsv,
                     dds_address* dest) = 0;

    // Delivery service characteristics
    virtual word32 get_maxpdu();
    virtual word32 get_rate() = 0;
    virtual word32 get_burst() = 0;

    // Multicast support
    virtual int add_mcast_membership(dds_address* addr);
    virtual int drop_mcast_membership(dds_address* addr);
};
```

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*. The file del_srv.h contains an enumerated type for many of the possible data delivery services:

```
enum dds_types {
    IP_type,          //  Raw IP
    UDP_type,         //  UDP
    TCP_type,         //  TCP
    FDDI_type,        //  FDDI
    ETHERNET_type,    //  Ethernet
    AAL5_type,        //  ATM AAL5
    XUNET_type,       //  XUNET
    Unknown_type      //  Unknown
};
```

These *dds_types* are used by the mtldaemon class to keep track of the data delivery service to instantiate.

The data delivery service addressing structure *dds_address* is also an abstract class. It hides the internal structure of the address, and provides instead a common set of functions on the address.

```
class dds_address {
public:
    virtual ~dds_address() { }
    virtual int size() = 0;
    virtual void* get_addr() = 0;
    virtual void put_hostid(void* hostid) = 0;
    virtual void* get_hostid() = 0;
    virtual int sizeof_hostid() = 0;
    virtual void print_hostid(FILE* fd) = 0;
    virtual void copy(dds_address* from) = 0;
};
```

Specific data delivery services derived an address class from dds_address, providing an implementation for each of the pure virtual functions. For example, the *ip_dds_address* class is based on the sockaddr_in addressing structure.:

```
class ip_dds_address : public dds_address {
private:
    struct sockaddr_in addr;
public:
    ip_dds_address() : dds_address() {
        memset((char*)&addr, (char)0, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_port = htons(DAEMON_PORT);
    }
    ~ip_dds_address() { }

    void* get_addr() { return((void*)&addr); }
    int size() { return(sizeof(addr)); }
    void put_hostid(void* hostid) {
        memcpy((char*)&(addr.sin_addr), (char*)hostid, 4);
    }
    void* get_hostid() { return((void*)&(addr.sin_addr)); }
    int sizeof_hostid() { return(4); }
    void print_hostid(FILE* fd) {
        fprintf(fd, "%s",
            inet_ntoa(*(struct in_addr*)&(addr.sin_addr)));
    }
    void copy(dds_address* from) {
        addr.sin_family = AF_INET;
        addr.sin_port = htons(DAEMON_PORT);
        put_hostid((void*)from->get_hostid());
    }
};
```

The class derived from dds_address must be the addressing structure used in the corresponding class derived from del_srv.

The del_srv class function *is_ready()* indicates if the constructor was able to initialize the data delivery service. This is necessary because the constructor does not have a return value. Appropriate address structures are obtained via the pure virtual function *alloc_dds_address()*. The *recv()* function only operates on contiguous data (e.g. a monolithic style packet), but the *send()* functions are overloaded for either contiguous or scatter/gather output.

There are three functions that return the data delivery service characteristics. The *get_maxpdu()* function returns the maximum protocol data unit size possible on this delivery service. The *get_rate()* and *get_burst()* functions return the number of bytes per millisecond, and number of bytes in a single burst, respectively. These values are useful for rate control.

The following program, called *dds*, illustrates how to use the dds_address and del_srv derived classes. To compile the program from the examples directory, type

```
$ CC -c -DHAVE_CONFIG_H -Iinstall-dir/include  -O dds.C
$ CC dds.o -o dds -Linstall-dir/lib -lmtl
```

or

```
$ make dds
```

The text of this program is found in the examples directory in the file dds.C.

```
#include <mtl/MTLtypes.h>
#include <mtl/del_srv.h>
#include <mtl/ip_del_srv.h>
#include <mtl/udp_del_srv.h>
#include <arpa/inet.h>
#include <netdb.h>
#ifdef irix4
#   include <getopt.h>
#endif /* irix4 */

#define ARGS \
"-r | -t <dest> [-d <dds_type>]"
#define HELP \
"           -r : receiver\n\
    -t <dest> : transmitter and destination\n\
-d <dds_type> : type of data delivery service to use\n\
           -? : print this message\n"

void main(int argc, char** argv) {
   if (argc < 2) {
      fprintf(stderr, "Usage: %s %s\n", argv[0], ARGS);
      exit(1);
   }

   extern int optind;
   extern char *optarg;
   char* hst = (char*)NULL;
   int recv = 1;
   int c;
```

```
    fd_set in_fds;
    del_srv* dds = (del_srv*)NULL;
    char buf[1024];

    while ((c = getopt(argc, argv, "?rt:d:")) != -1) {
        switch (c) {
        case 'r':
            recv = 1;
            break;
        case 't':
            recv = 0;
            hst = optarg;
            break;
        case 'd':
            if (strcmp(optarg, "ip") == 0)
                dds = new ip_del_srv(&in_fds, 36);
            else
                dds = new udp_del_srv(&in_fds, 36);
            break;
        case '?':
        default:
            fprintf(stderr, "\nUsage: %s %s\n", argv[0],ARGS);
            fprintf(stderr, "\n%s\n", HELP);
            exit(0);
        }
    }

    if (!dds)
        dds = new udp_del_srv(&in_fds, 36);

    printf("maxpdu = %d\n", dds->get_maxpdu());
    printf("burst = %d\n", dds->get_burst());
    printf("rate = %d\n", dds->get_rate());

    if (recv) {
        int done = 0;
        dds_address* from = dds->alloc_dds_address();
        fd_set test_fds = in_fds;
        printf("Ready to receive...\n");
        while (!done) {
#ifndef hpux
            int n = select(32, &test_fds, 0, 0, 0);
#else
            int n = select(32,(int*)&(test_fds.fds_bits[0]),
                                0,0,0);
#endif
            int res = dds->recv(buf, 1024, from);
            if (res > 0) {
                buf[res] = 0;
                printf("%s\n", buf);
            } else
                done = 1;
            done = (strcmp("quit", buf) == 0);
        }
```

```
            dds->free_dds_address(from);
        } else {
            dds_address* dest = dds->alloc_dds_address();
            word32 dsthost;
            struct hostent *host;
            if ((host = gethostbyname(hst)) == NULL) {
                if ((int)(dsthost = (word32)inet_addr(hst)) < 0) {
                    fprintf(stderr, "%s: unknown host\n", hst);
                    exit(1);
                }
                if ((host = gethostbyaddr((char*)&dsthost,
                            sizeof(dsthost), AF_INET)) == 0) {
                    fprintf(stderr,
                        "%s: Could not locate hostentry\n", hst);
                    exit(1);
                }
            } else
                dsthost = *(word32*)(host->h_addr);
            char hostname[32];
            gethostname(hostname, 32);
            dest->put_hostid(&dsthost);
            dest->print_hostid(stdout);
            printf("\n");
            vec_element psv[1];
            int len;
            printf("Send \"quit\" to end...\n");
            do {
                printf("=> ");
                gets(buf);
                printf("%s\n", buf);
                psv[0].data = buf;
                psv[0].len = strlen(buf);
                len = dds->send(psv, 1, dest);
                printf("  - sent %d\n", len);
            } while (strcmp(buf, "quit") != 0);
            dds->free_dds_address(dest);
        }
        delete(dds);
    }
```

The Daemon

The base class for the daemon is *mtldaemon;* the main program of the protocol implementation instantiates a global daemon object through which the protocol processing is conducted. The base class mtldaemon provides some fundamental functionality and access to several objects used throughout the system.

```
class mtldaemon {
public:
    mtldaemon();
    virtual ~mtldaemon() { }

    // Static functions and variables
```

```
        static packet_pool* pool;
        static del_srv* out_dds;
        static del_srv* in_dds;
        static word32 timestamp();

        // 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);

        // Context information
        int get_num_contexts();
        word64 get_key_mask();

        // 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_num);
        virtual void install_handlers();
        virtual req_action
            dispatch_request(user_request* request,
                             req_addr_struct* user_addr);
    };
```

Statics

There are three static member variables and one static function available through the daemon object: the packet pool, the data delivery service for outgoing packets, the data delivery service for incoming packets, and a timestamp function. Being static implies that these variables and function are owned by the daemon but are essentially global. This preserves the sense of ownership without causing undue contortions to gain access.

The packet pool object supplies the packet shells for incoming packets and for constructed outgoing packets. The two data delivery services allow the derived protocol implementation to specific separate channels for input and output. The timestamp function is useful for timing events, like roundtrip time.

IPC to User Processes

A daemon must be able to communicate with its users, so the mtldaemon class provides functions to facilitate this. The *recv_request()* and *send_reply()* functions hide the underlying IPC mechanisms by with the daemon gets user requests and sends back replies.

Context Information

A daemon is started with some constant number of contexts available for use. It is the context manager's responsibility to instantiate these contexts and provide indexing into them. To facilitate this, the context manager must learn from the daemon the number of contexts to instantiate, and a mask used to convert keys into indices for the contexts. These are gotten from the daemon via the *get_num_contexts()* and *get_key_mask()* functions.

Main Loop and Virtual Functions

A daemon must (1) determine if another daemon is running in order to avoid colliding with some of its IPC mechanisms, (2) parse the arguments given to the daemon program, (3) initialize the daemon, (4) start the main loop of getting packets and user request and processing them, and (5) shutting down when the daemon is told to do so. The three "main loop" functions and the five virtual functions correspond directly to these activities.

As is the case with other classes with virtual functions, the virtuals provided in mtldaemon offer a default implementation but allow, in fact, encourage protocol-specific implementations to defined. The *parse_args()* function recognizes a fundamental set of switches that can be passed to the daemon program; the protocol-specific *parse_args()* may add, remove, or redefine the set of switches it recognizes.

The overloaded functions *init_daemon()* start the daemon, which converts the program into a Unix daemon with no process group, parents, children, or siblings. The routine that starts the daemon also calls *install_handlers()* to install the interrupt handling routines. The *init_daemon()* functions then instantiate the data delivery service objects. The first *init_daemon()* function takes a protocol name in the form of a character string; this string is given to a system call to determine the protocol number for that protocol name. These numbers are assigned by the IETC in periodic RFCs. The second flavor of this call takes the protocol number itself. The data delivery service objects are instantiated using this number. These are the activities that may have to be done with root permissions, to they are done first. Next, the daemon's effective user id is returned to the user that started the program, so any activities from then on are done without root privilege, including opening the log file. (You will agree that a program that can open and close arbitrary files as root poses a hugh security hole; any protocol-specific replacement for *init_daemon()* should consider this.) Finally, the IPC channels are opened for user requests, and the packet pool is initiated.

The *init_daemon()* function is really intended to be redefined, and then called by the derived protocol's *init_daemon()* function, since the context manager must be instantiated from within the *init_daemon()* call. Since the context manager object must be derived, the mtldaemon function *init_daemon()* cannot make the call to *new()*; it must be done with the derived context manager is known.

The function *main_loop()* blocks waiting for incoming user requests or incoming packets. This is shown in Figure 4. The daemon unblocks when the client's

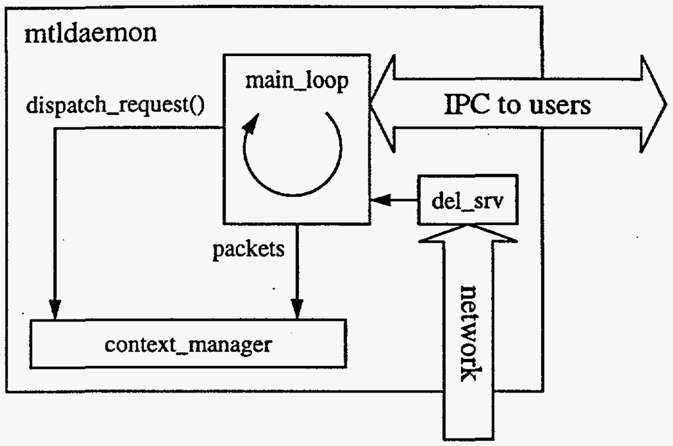


FIGURE 4. Mail Loop of mtldaemon

request arrives, does a switch on the request type, calls *dispatch_request()*, 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. The daemon will also unblock upon receipt of a packet, at which time it invokes the context manager's *handle_new_packet()* function. Either way, the daemon loops back and blocks waiting for another request or packet.

There are four friends of the mtldaemon class:

```
friend void log_print(const char* fmt ... );
friend void log_event();
friend void log_buffer(buffer_manager* bm);
friend void log_state(state_machine* st);
```

These functions print information to the log file, and thus require some intimate knowledge of the mtldaemon. The *log_print()* function takes the same format as does *printf()*, except that it can print 64-bit numbers by using the substitution string %ld for decimal, and %lx for hexadecimal.

The User Interface

All of the classes discussed so far are designed to be compiled directly into the program that implements the protocol. The user interface, however, is targeted to be part of a library that will be compiled into the user's application code. The user instantiates the interface object; 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.

```
class mtlif {
protected:
   // Information accessible by derived classes
```

```
        pid_t upid;
        buffer_manager s_bm;
        buffer_manager r_bm;
        user_request* request;
        int registered;
        int released;

        // Buffers
        int install_buffers(user_request* req);
        void release_buffers();

    public:
        mtlif();
        virtual ~mtlif();

        // IPC routines
        int issue(user_request* req);
        int inform(user_request* req);
        int accept(user_request* req);

        // Virtual functions
        virtual int reg();
        virtual int release(int no_answer = 0);
        virtual void perror(int res, char* usr_msg = NULL);
        virtual void cleanup();
    };
```

The *install_buffers()* method creates the buffer managers. They are accessed via the *r_bm* and *s_bm* member variables. 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 context writes the packet's data 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.

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 *user request* class, shown below, is used as a base class for protocol-specific user requests.

```
class user_request {
public:
    int cmd;           // requested command
    int len;           // length of the request
    int result_code;   // result of the action taken at daemon
    int extra_modes;   // modes of operation
    pid_t upid;        // user's pid, for verification
    word64 key;        // key used for identifying context
    int snd_shmid;     // shared memory id for send buffer
    int rcv_shmid;     // shared memory id for recv buffer
    word32 snd_buf_size; // send buffer size
    word32 rcv_buf_size; // recv buffer size
```

```
        user_request(const user_request* req
                                            = (user_request*)NULL);
        virtual ~user_request();
    };
```

Two user requests are essential: registering with the daemon, and releasing from the daemon. The *reg()* function is sends a *REGISTER* message to the daemon. This is virtual, so it can be replaced by a derived implementation. The register command causes the daemon to allocate a context for the user and attach to the send and receive buffers. The *release()* function causes the daemon to return any resources and detaches the daemon from the buffers by sending a *RELEASE* message to the daemon. (*REGISTER* and *RELEASE* are defined in MTLtypes).

The constructor for the user request class can take another user request and make a copy of it. If the argument is null, it instantiates a clean request. This helps if the protocol has a prototype user request that it copies common information from for other types of requests.

Buffer Management

Each client has two data buffers, one for sending and one for receiving, as shown in Figure 5. Each of the client's buffers is controlled by a buffer man-

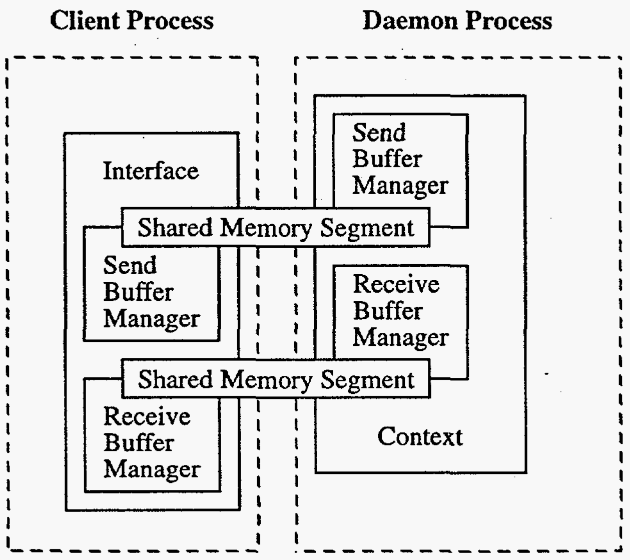


FIGURE 5. **Buffer Management**

ager object. Data are written into and read from the buffers through the buffer manager interface routines. A shared memory segment is used to reduce the amount of copying required to send or receive data.

The buffer manager class *buffer_manager* controls access to a data buffer. Internally, this data buffer is a piece of shared memory that two or more process gain access to using the buffer manager methods. Consequently, there can be multiple buffer manager objects granting access to a single data buffer.

```
class buffer_manager {
public:
   buffer_manager();
   ~buffer_manager() { }

   // lifecycle routines
   int create(word32 b_size, word64 beg_seq,
                 int index = -1);
   int attach(int shmid);
   void release();

   // Sets and Gets
   void set_dseq(word64 dseq);
   void set_tail(word64 tail);
   void set_beg_seq(word64 seq);
   word64 get_head();
   word64 get_tail();
   word64 get_dseq();
   word64 get_hseq();
   word32 get_room_left();
   word32 get_contig_len();
   word64 get_alloc();

   // Writes and Reads
   int write(const void* b, word32 len, int overwrite = 0);
   int write(const void* b, word32 len, word64 where,
            int overwrite);
   int read(void* b, word32 len);
   int read(void* b, word64 from, word64 to);
   int read(word32 len);
   int read(vec_element* sv, int* n, word32 len);
   int read(vec_element* sv, int* n,
            word64 from, word64 to);

   // Moves
   void advance_dseq(word32 amount);
   void acknowledged(word64 dseq);

   // Misc
   void print(FILE* fd);
   buffer_manager& operator=(const buffer_manager& sb);
};
```

Lifecycle

One process must create the physical data buffer using the *create()* method; a size *b_size* and a beginning sequence number *beg_seq* are passed as arguments. If an integer *index* is also passed to *create()*, the buffer manager will use this value as the "key" to the shared memory allocation system call (this should be used only if the *shmget(1)* system call is broken, as is the case with some i386 operating systems). The *create()* call returns a shared memory id, or *EXBUF* if failure occurs.

Once a data buffer has been created, another process can attach to that buffer so data written into the buffer by one process can be read by the other process. To do this, the second process must instantiate a buffer manager object, then get the shared memory id from the first process (communicated by some IPC facility). The second process calls the *attach()* method on its copy of the buffer manager object, with the shared memory id *shmid*, to attach that process to the data buffer. Now both processes are attached to the shared memory, and any activity on the buffer from one buffer manager will be immediately seen by the other buffer manager.

When a buffer manager is finished with a buffer, the *release()* method detaches the manager from the buffer. When all processes release their buffer managers, the memory will be deallocated back to the system.

In MTL, the two parties interested in the data buffer are the user and the appropriate context in the protocol daemon. The context object creates the data buffer and the user interface object attaches to it. This ensures that the user's demise will not prevent the removal of the shared memory buffer.

Also in MTL, buffers are created and attached to in pairs, one for sending, and one for receiving. If the buffer is a send buffer, the user writes to the data buffer and the context object reads the data from the buffer to send it. The context also keeps track of what data has been acknowledged, so that portion of the data buffer can be reused. If the buffer is a receive buffer, the context writes received data to the buffer and the user reads that data. The buffer manager class provides the methods for reading, writing, and otherwise manipulating the data buffers.

Sets and Gets

The data buffers keep track of data by sequencing each byte. There are four sequence numbers that serve as markers in the stream of data as it passes through the data buffer. The *tail* is the sequence number of the next byte to write, the *head* is the sequence number of the next byte to read, the *dseq* is the sequence number indicating what has been delivered to the user, and the *hseq* is the highest sequence number yet seen. The head and tail markers start at the beginning sequence number (gotten from the *create()* method or by setting it later using the *set_beg_seq()* call). As data are written to the buffer, the tail and hseq markers are advanced. It is possible for the hseq marker to be further into the sequence than the tail; the tail marks the last contiguously written byte. The tail value can never exceed the hseq value. As data are read from the buffer, the head marker is advanced. As data are delivered to the user, the dseq marker is advanced. Therefore, the dseq value can trail behind the head value, but will never exceed it. The tail and dseq values can be set manually, if necessary, using the two methods *set_tail()* and *set_dseq()*.

Several values are of interest as data are written to and read from the buffers. The number of contiguous bytes the buffer can currently hold is given by *get_room_left()* (the size of the buffer minus the difference between hseq and dseq), the number of contiguous bytes currently in the buffer is given by *get_contig_len()* (the difference between the tail and the head), and the highest sequence number acceptable without overrunning the buffer, given by *get_alloc()*.

The values of each of the marker sequence numbers can be gotten via the methods *get_head()*, *get_tail()*, *get_dseq()*, and *get_hseq()*.

Writes and Reads

The several overloaded *write()* and *read()* methods allow the owner of the buffer manager object to insert data into the buffer (at the tail) and remove data from the buffer (from the head).

The overloaded *write()* methods take a parameter *overwrite* that indicates that the buffer should not preserve data if new data will overwrite it. The *write()* methods will write all or none of the data to the buffer; if the *overwrite* parameter is not set to true (or is left to the default value), the *write()* methods will fail and write no data to the buffer if there is not enough room to write all of the data specified. All of the *read()* and *write()* methods return the number of bytes written or read.

Moves

As data is acknowledged, the last unacknowledged byte is recorded internally in a marker called dseq. The receive buffer moves the dseq marker using the method *advance_dseq()*, providing as a parameter the *amount* of data that has been given to the receiving user. The method used to indicate that data has been acknowledged is *acknowledged()*, where the parameter dseq is the sequence number up to which receipt of delivered data has been acknowledged.

Miscellany

The contents of a buffer manager, but not the contents of the data buffer itself, can be printed to a file using the *print()* method. This is useful in debugging and following the progress of data as it is sent and acknowledged. The assignment operator copies the sequence markers from one buffer manager object into another.

Below is an example of how to create, attach, write to and read from a data buffer. In this case there is only one process, but a shared buffer is created nonetheless.

```
#include <mtl/MTLtypes.h>
#include <mtl/buffer_manager.h>

void main() {
   char a[200];
   char b[200];
   char ans[10];
   int len;
```

```
    // Fill in the source user buffer
    int i;
    for (i = 0; i<200; i++)
        a[i] = 'a' + (i%26);

    // Instantiate the buffer manager
    buffer_manager bm;

    // Attach the buffer manager to a piece of memory,
    // starting at seq number 0.
    if (bm.create(200, word64(0)) == EXBUF) {
        fprintf(stderr, "Error in creating the buffer\n");
        exit(1);
    }

    // Dump the initial state of the buffer_manager
    printf("START:\n");
    bm.print(stdout);
    printf("alloc = "); uprint(bm.get_alloc(), "\n\n");

    do {
        //  Write 75 characters to the buffer from "a"
        printf("WRITE: writing 75 characters to ");
        printf("the buffer...\n");
        len = bm.write(a, 75, 0);
        printf("==> returned len = %d\n", len);
        fflush(stdout);
        bm.print(stdout);
        printf("alloc = "); uprint(bm.get_alloc(), "\n\n");

        // Read 75 characters from the buffer_manager into "b"
        printf("READ: reading 75 characters from ");
        printf("the buffer...\n");
        len = bm.read(b, bm.get_head(), bm.get_head() + 75);
        printf("==> returned len = %d\n", len);
        fflush(stdout);
        bm.print(stdout);
        printf("alloc = "); uprint(bm.get_alloc(), "\n\n");

        // Advance the dseq 75 characters. This is like
        // acknowledging that the data have been used and are
        // no longer needed.
        printf("ADVANCE: advancing the dseq ");
        printf("75 characters...\n");
        bm.advance_dseq(75);
        bm.print(stdout);
        printf("alloc = "); uprint(bm.get_alloc(), "\n");
        printf("\n\ncontinue: [y|n]");
        scanf("%s", ans);
    } while (ans[0] != 'n');

    // Release the buffer
    bm.release();
}
```

The text of this program is found in the examples directory in the file bmtest.C. To compile the program, issue

```
$ CC -c -DHAVE_CONFIG_H -Iinstall-dir/include  -O bmtest.C
$ CC bmtest.o -o bmtest -Linstall-dir/lib -lmtl
```

in the examples directory, or type

```
$ make bmtest
```

Miscellaneous Classes and Utilities

There are a few classes and utility functions that are not part of the major architecture of MTL, but which are necessary nonetheless. Most protocols maintain a state machine to guide the protocol through its processing; the state machine class is an abstract class designed to be a place holder until a protocol-specific class is defined. The event queue class is a utility class for keeping track of event/sequence number pairs. The signal handling utility smooths out the inconsistencies between signals across architectures and operating systems.

State Machine

The state machine class *state_machine* is a base class from which protocol-specific state machines are derived. This class services as a pure abstract class. The state machine is designed to be used within a context to keep track of the states of the communication. These states are entirely protocol specific, so this class is just a place holder.

```
class state_machine {
public:
    state_machine() { }
    virtual ~state_machine() { }
    virtual void print(FILE* fd) = 0;
};
```

If the function *log_state()* (see the discussion on the mtldaemon class) is ever called, there must be a derived state machine class defined, and the function void *print()* must be defined within that class.

Event Queues

The event queue class *event_queue* implements a first in, first out queue of events. An event is a 32-bit protocol-specific value. Events and their associated sequence numbers are kept in the queue until retrieved.

```
class event_queue {
public:
    event_queue();
    ~event_queue();
    int put(word32 event, word64 seq);
    word32 pull(word64& seq);
    word32 peek(word64& seq);
};
```

Once an event queue object is instantiated, the *put()* method places the *event* in the queue, and associates the sequence number *seq* with the event. The *pull()* method returns the event and passes the sequence number through the reference variable *seq*. The event is removed from the queue. The *peek()* method returns the same values as the *pull()* call, but does not remove the event from the queue. If no events are in the queue when *pull()* or *peek()* are called, a 0 is returned.

Signal Handling

This utility function replaces the *signal(2)* system call with a generic, machine-independent version:

```
Sigfunc* set_signal(int signo, Sigfunc* func);
```

When MTL is configured for compilation on a specific machine type, the idiosyncracies of that machine type are taken into account, and a uniform function *set_signal()* is the result. The function *set_signal()* is then used to set the signal handling routine for a signal number.

When a signal handler is declared, it should be of type *RETSIGTYPE*:

```
RETSIGTYPE _intr_handler(int sig);
RETSIGTYPE _child_handler(int sig);
```

Writing Derived Protocols

There are two parts to implementing a protocol using MTL: building the daemon process that will implement the protocol, and creating a user interface that exposes the appropriate functionality to the user. The configuration and installation of the MTL package produces two files, the libmtl.a library file, and the interface.o object file. Suppose we were implementing a protocol call Derived Transport Protocol, or DPT. The daemon process that will run the derived protocol (we'll call this process *dtpd*, for Derived Transport Protocol Daemon) links in the libmtl.a library to use the base classes provided. The user interface and a few other utilities, like the buffer manager, are placed into the interface.o object file that, along with the derived user interface class, are combined into a new, protocol-specific, library that will then be linked into the user's code during application compilation. We'll call this new library libdtp.a. These two parts of MTL then allow the two processes — the user's application and dtpd (the protocol implementation daemon process) — to communicate requests, responses, and user data.

The *mtlif* class is designed to offer basic functionality to a derived class. As Figure 6 shows, the intended approach to providing user interfaces is to first derive a protocol-specific interface class (*dtpif*) that exposes as much detail of the protocol's functionality as possible. This "wizard's" interface class would emphasis completeness rather than ease of use, and is targeted toward the truly knowledgeable. Then, for each type of application programmatic interface required, derive from the full exposure class a set of API-specific classes (*API_Specific1_if, etc.*) that limit the functionality to what is appropriate for that

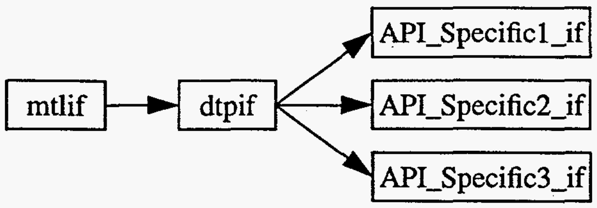


FIGURE 6. — **Derivation of User Interfaces**

API. The file interface.o and the object files of the derived interfaces are then collected into an archive file that becomes the user's library, libdtp.a.

Building the actual protocol implementation using MTL requires creating a main routine that anchors the protocol implementation program as it becomes a daemon process, and deriving protocol-specific classes from the MTL base classes. Here is the main routine for the MTL-based protocol DTP:

```
#include <mtl/MTLtypes.h>
#include <mtl/mtlif.h>

#include "include/DTPdaemon.h"

DTPdaemon __tmp;
DTPdaemon* DAEMON = &__tmp;

void main(int argc, char **argv) {

    // Check first to see if another dtpd daemon is running.
    if (DAEMON->is_another_daemon_running()) {
        fprintf(stderr, "Another XTP daemon is running\n");
        exit(1);
    }

    char my_name[128];
    gethostname(my_name, 128);
    time_t t = time(0);

    if (DAEMON->parse_args(argc, argv) == EXOK) {
        fprintf(stderr, "Derived Transport Protocol Daemon");
        fprintf(stderr, " (%d) started on %s %s",
                    getpid(), my_name, ctime(&t));
        DAEMON->init_daemon(99);
        log_event();
        log_print("Derived Transport Protocol Daemon");
        log_print(" (%d) started on %s %s",
                    getpid(), my_name, ctime(&t));
        DAEMON->main_loop();
        DAEMON->shutdown();
    }
}
```

There must be two global variables: a static instantiation of the derived daemon object, and a pointer to the derived daemon object. The daemon object is known universally throughout MTL as *DAEMON*, and all of its member functions and variables are accessed through this name.

The main routine first checks that no other daemons derived from MTL are currently running. Next the main routine calls *parse_args()* with the arguments. This allows the user who starts the daemon to control some initial properties of the daemon. The program is then initialized via the *init_daemon()* method, and becomes a full Unix daemon. The daemon process then invokes the *main_loop()* method, and runs there until told to stop, at which time some shut down procedures are done (*shutdown()*), and the daemon process exits, destroying the statically instantiated DTPdaemon object DAEMON. As the *main_loop()* method runs, the user requests and incoming packet provide the inputs for the protocol implementation. Through dynamic binding, the virtual functions from each base class are replaced with the derived class's version, so the MTL infrastructure always invokes the proper method.

A Small Example

The directory structure under compile-dir/DERIVEDprotocol, shown in Figure 7, is essentially what a protocol derived from MTL would may use. There are a

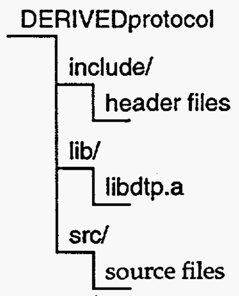


FIGURE 7. Derived Transport Protocol Directory Structure

number of approaches; this is just one of them.

The include directory contains the definition files for each of the derived classes, and the src directory contains the implementation files for those classes. As an example, the file DTPcontext.h contains the derived context class, called *DTPcontext*:

```
#include <mtl/MTLtypes.h>
#include <mtl/context.h>

class DTPcontext : public context {
friend class DTPcontext_manager;
private:
protected:
public:
    DTPcontext();
    ~DTPcontext();
```

```
        int is_quiescent();
        void go_quiescent();
        int initialize(user_request* request);
        int bind();
        int process_packet();
        int receive(user_request* request);
        int send(user_request* request);
    };
```

Since DTPcontext derives from context, the context.h header file is included. Note that this file assumes that `-Iinstall-dir/include` is used as a compiler switch.

The functions defined in the DTPcontext class are redefinitions of virtual or pure virtual functions from the context class. A real derived context class would have protocol-specific functionality, as well as many helper functions left as private or protected.

The protocol DTP can be build from the DERIVEDprotocol directory by typing

$ make

The Makefile will compile all of the derived class files in the src directory, then link these together with the installed libmtl.a file to produce an executable program called *dtpd,* left in the src directory. This is the DTP daemon program. Additionally, the executables *dtpds, dtpdrm,* and *dtpdreset* are created. The *dtpds* command gets the current status of the daemon, however that's defined by the daemon's implementation. The *dtpdrm* command removes a running daemon, causing it to clean up and shut down. The *dtpdreset* command clears all of the contexts in the daemon, and returns the daemon to a state similar to when it first was started.

The make continues into the lib directory, when the library file libdtp.a containing the user's interface routines is created. The libdtp.a file is created by combining the MTL interface.o (from the *install-dir*/lib directory) with the derived interface class or classes (here it is dtpif) into a library archive file.

By typing

$ make install

the DTP executables and the library file, and well as the include files, are installed into the same *install-dir* directory structure as was MTL. More specifically, dtpd and its cousins are placed into *install-dir*/bin; libdtp.a is placed into *install-dir*/lib; and the include files are placed into *install-dir*/include/dtp. Note that, if MTL required root privilege to run "make install", then so will DTP.

To write a user application program that includes DTP's interface routines, the program source file must include any header files to the DTP interface (here it would be

```
#include <dtp/dtpif.h>
```

To compile the application program, use

```
$ CC -c -Iinstall-dir/include  -O application.C
$ CC application.o -o application -Linstall-dir/lib -ldtp
```

A Full-Blown Example

Sandia National Laboratories has developed a full implementation of the Xpress Transport Protocol using MTL as the foundation. The implementation is called *SandiaXTP* [11][12], and is available from the Web page

http://www.ca.sandia.gov/xtp/SandiaXTP/

Help

There are at least nine mechanisms in place for receiving help:

- This document
- README file
- The MTL Reference Manual
- On-line manual pages
- Hypertext manual pages
- Papers included with the distribution
- Users Group mailing list
- Your Unix documentation
- Personal assistance

The first one is obvious. The MTL Reference Manual is in the compile-dir/doc directory under the name of mtl_reference_manual.ps. It is a postscript document form of all of the manual pages.

The README file is in the compile-dir directory. It explains where to find the introductory material and installation instructions through regular text or through a hypertext browser. The information in these files is a shortcut version of the information in this document, and is good for quick reference.

The on-line and hypertext manual pages are available in two places: before installation, they are kept in the compile-dir/man directories; after installation, they are also in the install-dir/man directories (note that the manual pages are installed using "make installman"). Read the manual page for *man(1)* to determine how to make the man command find these manual pages. This may involve setting the *MANPATH* environment variable.

The hypertext versions of the manual pages are best found by opening the file intro.html and following the link "MTL Man Pages".

When white papers exist to give more detail on a subject, they are put in the compile-dir/doc directory. The paper "A Class-Chest for Transport Protocols" is in the file class_chest.ps.

There is a mailing list for users of MTL and SandiaXTP. It is

SandiaXTP@lucky.ca.sandia.gov

If you have any issues you would like to discuss with the other users of MTL, mail to this list. News of updates and bug fixes will come through this list. If you are not already a member, mail to the address below with the subject line "user registration" and include your Email address.

The manufacturer provided literature on IPC, makefiles, and software installation may also answer some of your questions.

For more personal assistance, write to Tim Strayer at

strayer@ca.sandia.gov

with as much information as you can, including your machine type, compiler, directory structure, the methods you used to get yourself in the jam, and a good description of the jam itself.

Bibliography

1 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.

2 Hutchinson, N. C., Peterson, L. L., "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, Vol 17, No 1, January 1991, pp. 64-78.

3 Boykin, J., Kirschen, D., Langerman, A., and LoVerso, S., **Programming Under Mach**, Addison-Wesley, Reading, Mass., 1993.

4 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.

5 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.

6 Mogul, J. C., Rashid, R. F., and Accetta, M. J., "The Packet Filter: An efficient mechanism for user-level network code," *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987, pp. 39-51.

7 McCanne, S., and Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *Proceedings of the 1993 Winter USENIX Conference*, San Diego, Ca., January 25-29, 1993.

8 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.

9 Strayer, W. T., "A Class-Chest for Transport Protocols," Submitted to *Proceedings of 21st Local Computer Networks Conference*, Minneapolis, Minnesota, October 13-16, 1996.

10 Leffler, S.J., McKusick, M.K., Karels, M.J., Quarterman, J.S., **The Design and Implementation of the 4.3BSD UNIX Operating System**. Reading, Mass, Addison-Wesley 1989.

11 Strayer, W.T., Gray, S., Cline, R.E., Jr., An Object-Oriented Implementation of the Xpress Transfer Protocol. *2nd IWACA*, Heidelberg, Germany, September 26-28, 1994.

12 *SandiaXTP User's Guide*, Sandia National Laboratories, California, http:// www.ca.sandia.gov/xtp/SandiaXTP/.

UNLIMITED RELEASE

INITIAL DISTRIBUTION:

9003 D. L. Crawford, 8900
9011 R. E. Palmer, 8901
9011 P. W. Dean, 8910
9012 W. T. Strayer, 8910 (10)
9012 J. E. Costa, 8920
9012 S. C. Gray, 8930
9019 B. A. Maxwell, 8940
9011 D. B. Hall, 8970
9001 T. O. Hunter, 8000
Attn: J. B. Wright, 2200
E. E. Ives, 5200
M. E. John, 8100
L. A. West, 8200
W. J. McLean, 8300
R. C. Wayne, 8400
P. N. Smith, 8500
T. M. Dyer, 8700
P. E. Brewer, 8800
9021 Technical Communications Department, 8815, for OSTI (10)
9021 Technical Communications Department, 8815/Technical Library, MS 0899, 13414
0899 Technical Library, 13414 (4)
9018 Central Technical Files, 8940 (3)