AIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202

Centimeter

Inches

MANUFACTURED TO AIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

Meta-Transport Library User's Guide

*Meta-Transport Library—A
Protocol Base Class Library*

Release 1.1

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

*Sandia National Laboratories
P.O. Box 969
Livermore, California 94551-0969*

MASTER

Meta-Transport Library

User's Guide

Tim Strayer
Sandia National Laboratories
Livermore, California

RECEIVED
MAY 23 1994
OSTI

Introduction

The Meta-Transport Library is a collection of C++ base classes from which specific transport layer protocols can be derived. The object of MTL is to distill the transport protocol commonalities into a set of base classes. These base classes provide protocol-non-specific mechanisms upon which specific protocols are built.

Communication protocols have several common components: a some fundamental unit for information exchange, data structures to hold each endpoint's state, a processing mechanism to parse incoming packets, a mechanism to construct outgoing packets, and a mechanism to handle service requests. The Meta-Transport Library is a collection of C++ base classes that encapsulates these common components into a single class library. Specific transport protocols are derived from these MTL classes.

In general the units of information exchange are called *packets*, and the state structures are called *protocol control blocks* or *contexts*. The mechanisms that parse the incoming packets and service requests and, in doing so, provide the service characteristics of the protocol are called the *protocol procedures*. Protocols themselves are classified according to the service characteristics they provide; in transport protocols, we generally refer to such characteristics as *robustness*.

There are two central data structures, the *packet* and the *context*. The protocol procedures operate on these structures. Packets require construction, data insertion, transmission, reception, and data retrieval. The endpoint state structure is changed according to the parsing of packets and the servicing of user requests. In both cases there is a central repository of data and actors, or methods, on the reposed data. This profile matches the definition of an *object*.

The object-oriented programming paradigm forces ownership of functionality. This is useful for any major software project; in protocol implementation it is particularly useful in deciding where to place the methods. An instance of this can be seen with packets. Packets are sent and received, so it seems natural that “send” and “receive” could be methods on the packet object. However, while data within the packet object is actually sent, a packet object itself does not get received — data in the form of a packet simply arrives from the underlying delivery service. The context parses the packet’s contents. The paradigm also forces ownership of information. As a value is needed by an object, the programmer is compelled to decide how that value is to be conveyed. If the programmer cannot convey the information gracefully, there is a strong indication that the information is either ill-placed or unnecessary.

Object orientation also provides well-defined modules for implementation hiding. This is useful for isolating the underlying delivery service (how the packets are sent and received) from the rest of the protocol. In this way the protocol can be adapted to various delivery services, since the interface to these services remains constant.

MTL is a collection of base classes that implement or require the implementation of the common aspects of transport protocols. A particular protocol is instantiated through derived classes whose protocol-specific member functions implement the protocol procedures.

One important benefit from this approach is in protocol and protocol procedure comparison. Rarely are two different protocols built in exactly the same manner, yet unless this happens, comparison of the protocols reduces to comparison of how skilled the implementor is, and what choices were made. If two different protocols are implemented using MTL, the protocols themselves are exposed for comparison. Secondary artifacts are factored out. This “apples-to-apples” aspect applies to comparison of the protocol procedures within a protocol as well.

Another benefit is rapid prototyping of new or existing protocol specifications. Since a fair amount of the protocol is already implemented in MTL, the programmer can focus on the procedures specific to the protocol at hand. Further, the base class definitions give a good indication of what procedures need to be implemented; failure to define a pure virtual member function strongly suggests that the implementation is incomplete.

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. The fully built MTL is a C++ library file that is linked into derived protocol code.

License Agreement

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

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 these warnings 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.cs.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/mtl
3. Enter binary mode:
ftp> binary
4. Get the distribution file:
ftp> get mtl-1.1.tar.gz
5. Quit FTP:
ftp> quit

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` directories. 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.

Once you've decided where to install the MTL software, you have also decided where the derived protocol software, such as SandiaXTP, will be installed, since MTL and its derived protocols must reside in the same directory. In the case of the derived protocols, however, the decision to install as root is more profound: the software's capability is restricted if the daemon is not owned by root. See the appropriate Users' Guide for the specifics on the issues.

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

Installation Instructions

This code is written in C++ and, therefore, must be compiled with a C++ compiler. The one we used to develop the code is the AT&T C++ compiler (also called *cfront*). We do not use templates, so that is not a requisite when deciding which compiler to use. We have used the GNU g++ compiler. If you plan to use this to make the binaries, please ensure you are using version 2.5.7 or higher. We have found that g++ for IRIX 5.x.x (e.g., what is used in the SGI Indys) doesn't handle global constructors, so g++ should not be used with IRIX 5.x.x.

Tools you'll need:

- A C++ compiler
- The GNU compression program *gunzip*
- The X11 making facility *imake*
- One of the install programs: *bsdinst*, *bsdinst.sh*, or *install*.

Please do a *which* on each of these to be sure you have access to them. Also, read these instructions once through carefully before starting.

1. Select the target directory, `install_dir`, and change to that directory. Copy the zipped tar file to that directory, then change directories there as well.

```
$ cp mtl-1.1.tar.gz install_dir
$ cd install_dir
```

2. Uncompress the file using GNU's *gunzip*. If you don't have this, you can obtain it from one of the GNU mirror sites.

\$ gunzip mtl-1.1.tar.gz

3. Unfold the tar file. This will create the directory tree containing the source files, document and man pages.

\$ tar -xovf mtl-1.1.tar

Change directories into mtl-1.1:

\$ cd mtl-1.1

List the directory. You should have a directory called mtl-1.1. List that directory. You should see several files and directories, including: config, doc, include, lib, man, and src.

4. If you are compiling this code for a Sun under SunOS 4.x or an SGI under IRIX 4.x or 5.x, and you are using the native AT&T C++ (CC) compiler, you will not need to change any of the configuration files.

If, however, you plan to use GNU g++, change the configuration files in the following way.

Change directories to config.

\$ cd config

Edit the <arch>.cf file, where <arch> is the name of your architecture. For example, if you are on a Sun SparcStation, edit sun.cf. Find the line where the variable HasGcc is defined. It looks like this:

#define HasGcc NO

If there is no such define in the file <arch>.cf, then you can't use g++ at this time. If the line is there, change the NO to YES, and close the file.

Go back to the install_dir/mtl-1.1 directory.

\$ cd ..

5. Ensure you have the X11 tool *imake*.

\$ which imake

It is important that this and the related imake tools are accessible, since the making of this software uses the imake facility.

Also ensure you have an installation program, either *bsdinst*, *bsdinst.sh*, or *install*, according to which architecture you have.

\$ which bsdinst

\$ which bsdinst.sh

\$ which install

If you are using a machine with a SystemV operating system (like SGI), you'll use either *bsdinst* or *bsdinst.sh*, otherwise you'll use *install*. You'll find that you have only one of these programs; if you have *bsdinst.sh*, you must edit the file *Imake.tmpl* in the *install_dir/mtl-1.1/config* directory, find the line with "bsdinst" in it (there's only one) and add the ".sh" at the end of "bsdinst".

6. Compile the software. This is done with the script *build*.

\$./build

This invokes *imake* to make the Makefiles in the appropriate directories with the appropriate settings for your architecture. It then makes everything, so you'll see lots of files being compiled. When the compilation is done, build may ask a few simple questions, especially if you are installing without being root. If this fails at any point, record the circumstances and mail to Tim Strayer at the address given below.

One possible problem is that an included file may not be found by the *imake* utility *makedepend*. The *makedepend* utility tries to satisfy the `#include` directives by following the include search path. If *makedepend* fails, then the compilation will probably fail as well, since *makedepend* sticks the file not found in the dependency list anyway, and *make* tries to make this file. Try editing the Makefile in the `src` directory and removing the offending file(s) from the dependency list at the bottom, then refer to the next section on how to recompile the software. This should not be considered a fix; there is something wrong with the configuration files if you have to remove a dependency. In any case, mail the details to Tim Strayer.

As long as you don't change any *Imakefile* or the files in the `config` directory, you won't have to use build again. See the next section for instructions on how to build the software without rebuilding the Makefiles.

7. If all went well, the compilation will have produced a library file, called `libmtl.a`, and placed that file into the `lib` directory. Check that this is the case:

\$ ls lib

The file `libmtl.a` should be there. If it isn't, something failed during the building.

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 an *Imakefile* or to any of the configuration files in the `config` directory. If you need to recompile and reinstall but do not need to rebuild, follow these instructions.

1. Change directories to `install_dir`:

\$ cd install_dirmti-1.1

2. If you have changed any `#include` directives, remake the dependency lists:

\$ make depend

3. Issue the commands to clean the directories and then make everything:

\$ make clean

\$ make all

This will make everything necessary in all of the subdirectories.

4. You will install the software differently if you are root or if you are a normal user. If you are root, issue:

\$ make install

If you are not root, issue:

\$ mv src/libmtl.a lib

Directory Structure

FIGURE 1. MTL directory structure

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 1 shows the MTL directory structure. At the top of this structure is the directory `install_dir` used during installation. Below that is the MTL root directory, `mtl-1.1`. Under this directory are the directories `config`, `doc`, `include`, `lib`, `man`, and `src`. The `config` directory is where the imake configuration files are kept. These files are modified from the X11 distribution files. The `doc` directory holds the documents of this distribution. The `include` directory is where the header files are kept. The `lib` directory holds the MTL library file called `libmtl.a`. The `man` directory holds the manual pages for this software. The `src` directory is where the source files are kept, and where the initial compilation of the library is done.

Also under the directory `mtl-1.1` are several text files. The file `Imakefile` is the imake target description file. It is used to make a machine-dependent Makefile. The `Makefile` file will be rewritten when you configure the system for compilation. 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 `README` file explains how to unpack and install these directories. The script `build` configures and compiles the system.

Code Structure

There are four main classes in MTL, the packet class, the context class, the buffer manager class, and the user interface class.

Packets and Derived Classes

Packets are structured bins of information that are sent by a delivery service that treats the contents of the packet as uninterpreted payload. A protocol defines these packet structures, and extracts the information from the packets only with knowledge of the structure. As a consequence, the raw payload that is received is cast into some structure so the data can be meaningful.

In MTL, the packet base class is simply a byte array the size of the maximum protocol data unit. There are two base methods, one for getting the address of the start of the packet, and one for sending the packet. There is no receive method, since receiving is not actually done to a packet in the same way that sending is. A protocol-specific packet derived class would specify such methods as value insertion and extraction, and possibly a checksum function.

Two supplemental classes are used to manipulate packets, a packet pool class for memory management, and a packet FIFO class for parking packets until they can be processed.

Context and Context Management

The context class holds all endpoint state information. Methods include state transitions, packet parsing, and user request satisfaction. Each context also has two buffer managers, one for the send buffer and one for the receive buffer. The context manager class contains and manipulates all of the contexts, and is responsible for associating user requests and newly arrived packets with the proper context.

When a packet arrives, the context manager examines the packet only enough to figure out which context should get the packet. The packet is placed on the receive packet fifo (an object owned by the target context) so that the context, when it is allowed to satisfy any outstanding work, can retrieve the packet and process it.

Within the protocol-specific context derived class should be methods for parsing each type of incoming packet, and responding to the directives contained within that packet. Once the packet's contents have been digested, the context should return the packet shell to the packet pool, where it can be reused.

Buffer Management

The buffer manager provides both the user and the protocol access to data buffers. Methods exist for writing to and reading from the buffer. The buffer is also where data is kept until it can be released, so methods exist for moving the various markers in the buffer.

The buffer manager class manages a data buffer. A context contains two buffers, a send buffer and a receive buffer. The manager hides the implementation of the data buffer, but treats the structure like a circular buffer that is shared between the user application and the daemon. There are two managers per buffer — one at the user application side (and hidden by the user interface routines), and the other in the daemon in the context associated with this user. In general, one of the managers writes to the buffer and the other manager reads from it, so there must be some way of coordinating the head and tail markers for the buffer. This is done via user request commands; when a send request is issued, the request carries the new marker values so that the context's buffer manager will know what data to send (a similar exchange happens for receiving data through the receive buffer).

The buffers are also used to hold data until the data can be released. When data is released, the context moves a low watermark marker. This marker must be given, via a response to a user request command, to the user application's buffer manager so that the buffer space can be freed.

User Interface

The user interface class provides a standard set of methods for users to request service. Protocol- and service-specific access methods can be derived from this class. Hidden within this class is the manner in which a user application process makes its requests known to the protocol implementation, which is a daemon.

The user's application code instantiates an interface object to provide the user's application a means of issuing commands to the daemon and of managing the data from or to the application.

The user interface uses two buffer managers, one for the send buffer and one for the receive buffer. The user writes data into these buffers and issues the send command. The presence of the data is made known to the context, and the context's buffer manager pulls the data into the protocol. As packets with data arrive, the data is written into the receive buffer. As the application asks for data, the context informs the interface's receive buffer manager about the size and location of the received data.

Different interfaces can be constructed using the user interface library by simply deriving a specific interface from the user interface provided. In this way standard APIs, like sockets or TLI, can be emulated, and code written to use those APIs would not have to be modified to use XTP.

Help

There are manual pages in the `mtl-1.1/man` directory. (You may move these to another manual directory, or change your `MANPATH` variable to look here for the man pages.)

Suggested Reading

The manufacturer provided literature on IPC, makefiles, and software installation may answer some of your questions. The book "Software Portability with imake" by Paul DuBois (O'Reilly) is good for customizing your configuration files in mtl-1.1/config. I don't suggest this unless you are daring or knowledgeable or both.

Write to me 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.

Tim Strayer
strayer@ca.sandia.gov

Suggested Reading

The internals of this software are reported in a paper in preparation.

**DATE
FILMED**

6/15/94

END