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

Resolution Test Chart Labels:

- 1.0
- 1.1
- 1.25
- 1.4
- 1.6
- 1.8
- 2.0
- 2.2
- 2.5
- 2.8
- 3.2
- 3.6
- 4.0

MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

SAND--94-8646

RECEIVED

MAY 23 1994

OSTI

SandiaXTP

User's Guide

*SandiaXTP—An Object-Oriented
Implementation of XTP 3.7*

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

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED ^{db}

SandiaXTP

User's Guide

Tim Strayer
Sandia National Laboratories
Livermore, California

Introduction

The Xpress Transfer Protocol (XTP) is a transport layer protocol designed to provide a wide range of communication services built on the concept that orthogonal protocol mechanisms can be combined to produce appropriate paradigms within the same basic framework. Rather than using a separate protocol for each type of communication, XTP's protocol options and control of the packet exchange patterns allow the application to create appropriate paradigms such as unreliable datagrams, unreliable arbitrarily long datagrams, reliable datagrams, transactions, unreliable streams, reliable connections, and many others. Error control, flow control, and rate control are each configured to the needs of the communication.

SandiaXTP is an object oriented implementation of XTP 3.7. The core of base classes used in SandiaXTP come from the Meta-Transport Library software. (The MTL 1.1 package must be installed in order to install and use the SandiaXTP software.) The SandiaXTP implementation of XTP 3.7 is a user space daemon. User's application code makes request of the daemon, and the daemon satisfies them.

SandiaXTP is actually a set of classes derived from MTL base classes. 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 bases classes. These base classes provide protocol-non-specific mechanisms upon which specific protocols are built.

The state at one endpoint of an XTP conversation is called a *context*. An XTP implementation applies the user's requests and the incoming packets to move the context through its various states. The term *association* is used to describe

the relationship that two endpoints share. (It also describes the collective relationship shared during a multicast; for now we will use unicast to define the terms.)

The XTP association provides a full-duplex communication path. Each endpoint of the association is both a transmitter and a receiver. The protocol procedures within XTP reflect a transmitter-driven design philosophy. A receiver's primary job is to react to incoming packets; the transmitter describes the characteristics of the communication by which packets it sends, how often it sends them, and the contents of the packets. In particular, the transmitter may set options bits carried in the packet header; the receiver's actions are driven by the options set. This allows the receiver design to be simple and universal.

The XTP specification does not impose a service definition on implementations. This is to allow implementors to tailor the service interface to the needs of its users. Some implementations may expose the full power of XTP to the user, while others may limit its use via the services offered.

XTP decouples error control from the communication paradigm. The degree to which reliability is maintained is under user control. Error reporting can be aggressive, normal, or disabled. Aggressive reporting happens when the transmitter instructs the receiver to send a negative acknowledgement as soon as a lost packet is detected. The transmitter can also instruct the receiver to disable error reporting. Lost data can be reported in the classic go-back-n, style or a list of lost data spans can be given to the transmitter.

XTP also supports multi-party communication. Multicast communication is treated exactly like unicast, with the orthogonal protocol procedures combining to create the service desired. Only those few functions that do not make sense in a multi-way communication are excluded.

This User's Guide describes the installation and use of the SandiaXTP object-oriented implementation of XTP 3.7. The software package includes the full source code for the implementation, as well as man pages and appropriate documents. The fully built SandiaXTP is a user-space daemon that implements XTP 3.7.

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.

Installation

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 SandiaXTP and MTL software packages are available via anonymous FTP from the machine **dancer.ca.sandia.gov**. If you have not already done so, download these packages 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

ftp> get SandiaXTP-1.1.tar.gz

5. Quit FTP:

ftp> quit

Now you are ready to install the SandiaXTP 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.

If you haven't done so already, follow the installation instructions in the MTL Users' Guide and install MTL 1.1 (the version numbers for SandiaXTP and MTL should match). This must be done before installing SandiaXTP.

Once you've decided where to install the MTL software, you have also decided where SandiaXTP will be installed, since MTL and its derived protocols must reside in the same directory.

Concerning installation of SandiaXTP using root privileges, here are the issues. A program sometimes need special permission to use certain underlying data delivery services. For example, raw IP sockets can only be used by a program run by root. UDP sockets, on the other hand, don't require this to use them. There are two ways to give a program root privilege: either the program is run by root itself, or the "set UID" permission is set on the program's root-owned executable file. Either way root privilege is necessary. In the first way, it's always required; in the second way, root is needed only during installation.

If you install SandiaXTP as root, then set the "set UID" permission, you can use XTP over IP; if not, you can only use XTP over UDP.

Make the decision now about whether you can be root during installation, and 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 SandiaXTP-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 SandiaXTP-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 SandiaXTP-1.1.tar

Change directories into SandiaXTP-1.1:

\$ cd SandiaXTP-1.1

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

4. The SandiaXTP software must be made using the same configuration as was used to build MTL. This configuration is reflected in the *install_dir/mtl-1.1/config* directory. If you used *cfront* (CC) to compile MTL, you must use *cfront* to compile SandiaXTP. Likewise with *g++*.
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*. This script is part of the MTL distribution.

\$./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 `lmakefile` 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 you are root, edit the file `/etc/protocols` and add this line to the end:

xtp 36 XTP # Xpress Transfer Protocol

If you are not root, or never plan to use raw IP, then this step is not necessary. The entry here allows the system call `getprotobyname` to look up the protocol number assigned to XTP. The number 36 is the officially assigned protocol number for XTP.

8. If all went well, the compilation will have produced several executables and placed that file into the `bin` directory. Check that this is the case:

\$ ls lib

The files `xtpd`, `xtpdrm`, and `xtpds` should be there. If they aren'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 `lmakefile` 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_dir*/SandiaXTP-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/xtpd bin

\$ mv scr/xtpdrm bin

\$ mv scr/xtpds bin

Remember to recompile and reinstall SandiaXTP after you build a new MTL library.

FIGURE 1. SandiaXTP directory structure

Directory Structure

Figure 1 shows the SandiaXTP directory structure. At the top of this structure is the directory `install_dir` used during installation. Below that are the MTL and SandiaXTP root directories, `mtl-1.1` and `SandiaXTP-1.1`. Under the SandiaXTP root directory are the directories `bin`, `doc`, `examples`, `include`, `man`, and `src`. The `bin` directory is where the daemon and its utility programs are kept. The `doc` directory holds the documents of this distribution. The `examples` directory holds a few example applications. The `include` directory is where the header files are kept. 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 `SandiaXTP-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. The 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 to the pool of packet shells (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 in the user interface routines), and the other in the context associated with this user. In general, one of the managers writes to the buffer and the other manager reads from it, so there must be some way of coordinating the head and tail markers for the buffer. This is done via user request commands; when a send request is issued, the 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 acknowledgement is received. When data is acknowledged, 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.

The SandiaXTP Daemon and Utility Commands

SandiaXTP implements XTP as a user space daemon. User processes send and receive data by employing a set of user interface routines. These routines communicate with the daemon on behalf of the user.

The SandiaXTP daemon executable is in the SandiaXTP-1.1/bin directory, and is called *xtpd*. During the installation instructions you made the decision to build this as root or as a regular user. If you built as root, the daemon *xtpd* is owned by root. In order for the daemon to have root privilege as it runs (this is required for using raw IP), either you must always start the daemon as root, or you must set the "set UID" bit in the permissions for *xtpd*:

\$ chmod u+s xtpd

If you set the "set UID" permission, *xtpd* can be run by anyone who has this directory in the search path.

Along with *xtpd*, there two daemon utility programs, *xtpdrm* and *xtpds*. The program *xtpdrm* causes a running daemon to shut down gracefully. The program *xtpds* prints the status of the daemon.

The XTP Daemon: *xtpd*

Issuing the command *xtpd* starts the SandiaXTP implementation in the form of a daemon user-space process. The daemon listens for incoming packets and incoming user requests, and attempts to satisfy each. User applications communicate with the daemon via a set of user interface routines from the base class *mtlif*.

The *xtpd* command has the following options:

-a format

Indicates which addressing format will be used. (Only the IP addressing format, 0x01, is currently implemented.)

-c num_contexts

Specifies the number of context held in the daemon. Each context is in a quiescent state until it is activated. There are a fixed number of quiescent and active contexts. The default is held in the file SandiaXTP-1.1/include/daemon.h as the #define variable NUM_CONTEXTS. The actual number of contexts maintained by the daemon is always a power of two; if this option is used, the daemon will set the number of contexts to the next higher power of two, if *num_contexts* is not itself a power of two.

-d protocol

Specifies the underlying data delivery protocol being used. Possible values include *ip*, *udp*, *fddi*, *ethernet*, *aal5*, and *xunet*. (Only the IP and UDP protocols are currently allowed.) You must be root, or *xtpd* must have had the set UID privilege allowed, to use IP.

-e error_type error_parm

Causes the daemon to drop packets according to *error_type* and *error_parm*. This artificially induces errors in the packet stream. If *error_type* is 0, then no errors are induced. If the value is 1, packets are lost with uniform distribution over the interval [0..*error_parm*].

The default is for no errors to be induced. The *xtpd* code must be compiled with the `-DINDUCE_ERRORS` definition included for this option to be valid.

-l logfile

Specifies the name of the log file. The default is `./xtp_log`.

-n Causes the daemon to remain in the foreground, rather than becoming a true daemon process in the background.

-p num_packets

Specifies the total number of packet shells the packet pool will hold. The default value is held in the file `SandiaXTP-1.1/include/daemon.h` as the `#define` variable `NUM_PACKETS`.

-r Causes the daemon to report whenever a new roundtrip time is calculated.

-t Causes the daemon to print tracing statements to the log file. Those statements show context state changes and packet arrivals and departures. Tracing severely degrades performance.

Removing the Daemon: *xtpdrm*

Issuing the command *xtpdrm* causes the SandiaXTP daemon *xtpd* to send closure DIAG packets to the other endpoints of all active contexts, return to the system any resources it has, and exit. A user application will have an EXCOMM error returned to any *mtlif* or derived method once *xtpd* has been shut down.

If the daemon becomes corrupted, and cannot be removed by the *xtpdrm* command, use the system kill command to remove this daemon process. This method may leave some IPC facilities unredeemed. Issue the command *ipcs* (see *ipcs*(1)) to see the current facilities in use. Use the command *ipcrm* (see *ipcrm*(1)) with a `“-m shmid”` option to remove the shared memory segments, and a `“-q msqid”` to remove the message queue segments (see “Cleaning Up If Bad Things Happen” below).

Although improper, the *xtpdrm* command will indicate success even if the daemon is corrupted and cannot be shut down this way. This is targeted for correction in future versions.

Showing the Daemon's Status: *xtpds*

Issuing the command *xtpds* causes the SandiaXTP daemon *xtpd* to print its current status to the window in which the daemon was started. This status includes the state and state variables of each of the active contexts in the daemon.

The *xtpds* only prints if the daemon was started with the `“-n”` option (no background).

Programmatic Interface

User application programs can use SandiaXTP for passing message by linking in the MTL library file and including the interface include file. The details are given below. Also below is a description of some example applications and how to compile and run them.

The XTP Interface Library

The *xtpif* class implements the programmatic interface to XTP. An instantiated *xtpif* object is the user's access point to the functionality of XTP. Each object represents a potential endpoint of an association, and serves as the unique association identifier (the unique *key* value is actually stored in the object and used to associated this object with the context within the XTP daemon process). All operations on this association must be made via one of the provided methods.

```
class xtpif : public mtlif {
public:
    xtpif(\ );
    ~xtpif(\ );
    int config(int cmd, xtp_config* xcf);
    int reg(int snd_buf_size, int rcv_buf_size,
        void* sndaddr = (void*)NULL, void* rcvaddr = (void*)NULL);
    int bind(address_segment* addr);
    int listen(short16 block, short16* options);
    int send(void* p, int length, short16 block, short16* options);
    int send(short16 block, short16* options);
    int receive(void* p, int* length, short16 block, short16* options);
    int receive(int* length, short16 block, short16* options);
    int sendctl(short16 block, short16* options);
    int release(\ );
    int write(void* p, int len, short16 options = 0);
    int read(void* p, int len);
    void perror(int res);
};
```

Users include the header file *xtpif.h* in their application code (written in C++) and link the library *libmtl.a* when compiling this application code in order to use the *xtpif* class. The constructor *xtpif()* takes no arguments. The destructor *~xtpif()* (called automatically) releases the user's resources kept for the association.

The member function *config()* allows the user to configure certain parameters for this association. This call must be made prior to registration. The member function *reg()* registers the user with the XTP daemon; this establishes the object as a legitimate endpoint for the association. This call must be made prior to any other service call except *config()*. The member function *bind()* binds an address to this association. When this endpoint is the listener, this address is the filter for incoming FIRST packets. When this endpoint is the initiator of the association, this address specifies the destination.

The member function *listen()* causes this endpoint to listen for incoming FIRST packets. This method can only be called one, and cannot be called before the *reg()* and *bind()* calls, nor after a *send()* or *receive()*. The member function *send()* is overloaded to allow the user to send data provided (in variable *p*) or to send the data that has been previously written to the send buffer. If this call is made without a prior *listen()* call, the packet sent is a FIRST packet.

The member function *receive()* is overloaded to allow the user to receive data into a place provided (pointer variable *p*) or to receive the data into the receive buffer without reading the receive buffer. If this call is made without a prior *listen()* or *send()* call, the context is placed into a listening state. In this case, the *receive()* call supplants the need for a *listen()*. The member function *sendcntl()* causes an unsolicited control packet to be sent.

The member function *release()* causes the XTP daemon to release all resources for this association, and places the context into the quiescent state. The object can be reused for another association.

The member functions *write()* and *read()* write to and read from the send and receive buffers, respectively. These methods do not cause data to be sent or received; they decouple the buffer access from the action of the protocol.

The member function *perror()* prints out the error condition associated with a negative return value from one of the other member functions.

Example Applications

Now you are ready to compile and run a simple test application. Open at least three windows. Copy the contents of the examples directory to your own private directory. Edit the *Makefile* so that the variable *CC* is the same C++ compiler you used to make the library. Also edit the *INSTALL_DIR* variable to reflect where the *MTL* and *SandiaXTP* files can be found. Now, make the files:

```
$ make clean  
$ make depend  
$ make all
```

This makes the example applications. We're interested in *xping*, and the pair *fserver* and *fclient* right now.

In one window start the XTP daemon:

```
$ install_dir/SandiaXTP-1.1/bin/xtpd -n -t -r
```

(the *-n* flag start the daemon in "non-daemon" mode — it does not go into the background; the *-t* flag starts the trace, and the *-r* flag makes the daemon report any changes to the round trip time estimate). If you are using UDP, start the daemon with the *-d udp* flag as well; otherwise, the daemon will use raw IP.

In one of the windows invoke the *xping* program:

```
$ xping hostname
```

where *hostname* is the name of this host. It should respond with:

hostname is alive

If *xping* doesn't respond in a very few moments, there is something wrong with the daemon. You may need to recompile and reinstall.

To run the file client/server example, invoke the *fserver* program in the same window you used for *xping*:

\$ fserver

and in the other window start the file client:

\$ fclient

The file client will ask which machine is the server. Type in this machine's name (this uses the system call *gethostbyname*, so the name must be recognizable). Next the file client asks you which file to get. Enter "fclient.C". This file should be printed out. The file was retrieved from the file server and sent via XTP to the file client. If something went wrong, recheck the steps, record the problem, then seek help if you cannot find the reason. Usually failure will be because *gethostbyname()* didn't find the host name, or the daemon is not running.

Stop the daemon by issuing the *xtpdrm* command:

\$ install_dir/SandiaXTP-1.1/bin/xtpdrm

If this doesn't stop the daemon, issue *kill -9*. In this case you may have some resources that need to be cleaned up. These are discussed next.

Writing your own applications

First step: read the man pages in section 3 (SandiaXTP-1.1/man3). Step two: study *fclient.C* and *fserver.C*.

SandiaXTP provides a highly parameterized interface. You can control virtually every aspect of the protocol through this interface, so make sure you are familiar with what XTP provides so that you can use the methods in the *xtpif* class appropriately.

Cleaning Up If Bad Things Happen

The daemon uses IPC facilities to talk with its client users. Normally these will be reclaimed when the user process or the daemon dies. Unfortunately, there are some cases where it is impossible to reclaim these resources, name, if the daemon is killed using *kill -9*. In these cases, issue the command

\$ ipcs

to see the status of the IPC facilities. If there are any queues (q) or shared memories (m) then notice the ID number and issue

\$ ipcrm -q msqid -m shmid

(you'll need a -q for each queue and a -m for each shared memory).

Help

The daemon will not start if there are unclaimed IPC facilities left. This implies that only one daemon can be running per host.

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.)

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

This software reflects the XTP version 3.7 specification. The XTP specification for version 3.6 still explains the general philosophy. The book **XTP: The Xpress Transfer Protocol** by Strayer, Dempsey, and Weaver (Addison-Wesley) gives a more complete overview of XTP for lay-readers, and explains where XTP fits into the taxonomy of transport protocols. This book, too, is based on the version 3.6, but still accurately reflects the major issues.

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

DATE

FILMED

6/15/94

END