U.S. DEPARTMENT OF
ENERGY

PNNL-22594

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

The Raw Transfer Library

DES-0012

Revision 2

Charlie Hubbard

July 2010

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

The Raw Transfer Library

Charlie Hubbard

DES-0012
Revision 2
July 2010

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under
Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Contents

1 Introduction

Our system control projects are all based on a client/server model, in which full control functionality is implemented as a series of concurrently running processes that coordinate actions by passing request and response messsages back and forth.[1]

Overwhelmingly, message-passing is carried out through services provided by the `BaseClient` and `BaseServer` classes (from which all real clients and servers are derived), using a simple text-based encoding called Data Serialization Protocol (DSP).[2]

The DSP encoding scheme works well for client/server messages of up to a few tens of kilobytes in size (this encompasses virtually *all* messages). Beyond that, the task of converting message parameters into and out of a text format starts to be cumbersome. Rarely, there is a need to move very large blocks of binary data between two processes (2D detector histograms, which can be a quarter megabyte or bigger and contain thousands of fields, are a prime example[3]). For this, it is much quicker and more efficient to send the binary data directly rather than using a DSP encoding. The Raw Transfer Library (RTL) provides this capability.

2 Theory of Operation

The RTL provides two functions that, when used together, provide a means for one process to transfer large blocks of contiguous binary data to another process quickly and efficiently.

The library works by opening a TCP connection between the two processes and then transferring the binary data over the TCP connection. TCP is used because it allows the two processes to exist on separate, possibly widely separated, computers (an attribute shared by clients and servers that use the standard DSP-based message passing mechanism as well).

How the library works is probably best described by example. Say there are two processes, A and B. Process A sends a request to Process B for the contents of a very large data structure, and that data structure will be transferred using the RTL. The sequence of events is as follows:

1. Process A sends a message to Process B requesting the large data structure. This initial message goes as a normal, DSP-encoded client request message and does not use the services of the RTL.

2. Process B creates a TCP listen socket and spawns a separate thread to manage it.

[1] The project client/server model is fully described in design document ***DES-0005, The Client/Server Architecture***.

[2] See design document ***DES-0002, Data Serialization Protocol*** for details of the encoding format, and design document ***DES-0005, The Client/Server Architecture*** for a description of the specific format of the DSP-encoded messages used by real clients and servers.

[3] See design document ***DES-0013, The Nuclear Data Acquisition Server*** for details.

3. Process B sends a reply to process A containing at least the IP address and port number of the listening socket. Other pieces of information, like number of bytes to be sent in the upcoming binary transfer, may be included in the reply as well, depending on the specific circumstances. This reply goes out as a normal, DSP-encoded server response message, and does not use the services of the Raw Transfer Library.
4. Process A establishes a new TCP connection to Process B's listening socket.
5. Process B's temporary binary transfer thread writes data to the connection as rapidly as possible (TCP is self-metering), and Process A reads data off the connection as rapidly as possible. This continues until all data has been transferred.
6. Process A and Process B each close their side of the TCP connection that was used for the bulk data transfer.
7. Process B's transfer thread exits. This completes the transfer.

3 Limitations

The RTL has some limitations, the primary one being that the raw data to be moved between processes needs to be contiguous in memory. This makes it difficult to move things like STL lists or maps, or any other data structure in which one or more fields are stored in separate, dynamically allocated blocks of memory. Depending on the situation, this shortcoming can often be worked around by sending the small pieces of a structure using traditional, DSP-encoded messages, and only send the very large field(s) using the RTL. Alternatively, a single transfer could be broken into a few rounds, using the RTL to move a different piece with each round. One can imagine situations for which no amount of RTL trickery will yield a suitable transfer scenario, but, in practice, we have yet to encounter such a situation.

4 Alternatives

One the face of it, the RTL implements an overly complex transfer scheme. For each binary transfer, a new TCP connection is created and torn down, and the sending process spawns a new thread that exists only long enough to complete the transfer. The truth is, the RTL is an ugly hack. It does not fit in well with the project client/server message passing model. Because it does not stand on its own merits without some justification, it's important that we take some time to look at other alternatives that were considered and rejected.

4.1 Existing TCP Connection

The client and server already share a TCP connection, so one possibility is to use the existing connection to transfer the binary data, but there are problems with this as we'll see.

One plausible sequence of events might be:

1. The client sends a normal, DSP-encoded request to the server.[4]
2. The server sends a normal, DSP-encoded response to the client.[5]
3. The client receives the DSP-encoded initial response.
4. Server uses the same connection to send the binary segment of the reply back to the client.
5. Client reads the binary segment off the connection.

4.1.1 The Threading Problem

For a client application consisting of just a single thread, this technique actually works just fine, but it fails for multi-threaded clients. Clients perform their request/reply transaction using the `SendMessage()` method of the client base class.[6] Because any given client application may have multiple threads using the same client interface at the same time, `SendMessage()` needs to protect the TCP connection from simultaneous access. It does this by mutex-protecting the entire request/reply transaction (steps 1, 2, & 3 above). But steps 4 and 5 are beyond what `SendMessage()` is designed to provide. They would be handled by other class methods. The trouble is, when `SendMessage()` returns, the mutex is no longer locked. If another thread grabs the mutex at that point and sends a request message, it will incorrectly interpret the binary segment being sent by the server to the original thread as the DSP-encoded reply intended for it.

There is no easy work-around for this problem within the current message-passing model. A system could be put in place to send a flag to `SendMessage()` instructing it *not* to release the mutex on exit. Then, whatever method is responsible for receiving the binary segment from the server, as in step 5, would unlock the mutex when it finished. Alternatively, a special version of `SendMessage()` — `SendBinMessage()` perhaps — could be developed to handle all 5 steps outlined above internally. However, both of these solutions are kludgy patches wedged into a design where they really do not fit. Granted, the same can be said for the RTL itself, and at least these proposed schemes do not require a separate TCP connection and a temporary transfer thread. It might be worth choosing the "modified `SendMessage()`" kludge over the "raw transfer library" kludge, except for one problem...

4.1.2 The Unidirectional Problem

Binary data transfer using the existing client/server TCP connection only works from the server to the client. Client-to-server binary transfers cannot be done this way using the

[4] Beginning the sequence with a standard, DSP-encoded request is necessary, because the server expects all incoming messages to be DSP-encoded and parses them as if that were true.

[5] An initial DSP-encoded reply is also necessary, because the requesting client application is expecting at least a success/failure indication from the server in the standard DSP format.

[6] `SendMessage()` is fully described in design document ***DES-0005, The Client/Server Architecture***.

existing message-passing scheme.

When a server sends a reply to a client application under the current model, the client API function that initiated the request receives the reply directly (using the `SendMessage()` method). For that reason, server-to-client binary transfers work. If server *client message handlers* (the server-side counterpart to the client class' API methods) could receive data directly from the client, client-to-server binary transfers would work as well, but servers receive all messages via their *client connection thread*.[7] The thread, as implemented, expects all messages to be DSP-encoded. The thread performs an initial parse of the message, and places the relevant information on the server's message queue to be processed in the order in which the message was received. For this reason, client applications cannot just tag a binary component onto the end of a message and expect the server's relevant message handler to receive it. This is not an issue for the RTL, because the RTL uses a separate TCP connection that is not managed by the server's client connection thread. It can be used for binary transfers in both directions.

Even this problem could be overcome (for example) by modifying the `ReadStream` class' `Read()` method[8] (which the server's client connection thread uses to read data off the TCP connection) to recognize a special character in the data stream that indicated that the following data segment was binary and should not be pre-parsed. Presumably the number of bytes of binary data would have to follow the special flag character in the data stream so the `ReadStream` class would know how many bytes to read to complete the transfer. Then some mechanism would be needed to get the binary portion of the received message back to the proper message handler. If this mechanism involves the standard server message queue, then the queue code would have to be modified to accommodate binary data. As you can see, although possible, this "solution" gets very ugly very fast and, in the end, the result is just bidirectional binary data transfer capability, which the RTL already provides. Significantly complicating the `ReadStream` class, the server's client connection thread and, potentially, the server's message queue in order to provide a capability needed by exactly one application (the XIA nuclear detector's data acquisition server[9]), which is only ever used on a subset of our client/server based projects, seems like a poor idea.

4.2 Pure Binary Message-Passing Model

The fundamental problem that the RTL is trying to solve, is that there are cases in which the DSP-encoded message-passing scheme is too slow to deal with the required amounts of data in a timely fashion. One solution would be to dump the text-based, DSP-encoded message model altogether. The underlying message-handling code could very feasibly be rewritten such that *all* client/server messages are sent as raw binary.

Actually, this is probably the right approach, and it would be worthy of serious consideration

[7] See design document ***DES-0005, The Client/Server Architecture*** for complete details on server client connection threads.

[8] See design document ***DES-0003, The ReadStream Utility Class*** for details.

[9] See design document ***DES-0013, The Nuclear Data Acquisition Server*** for details.

if large data transfers were common, but they are not. Overwhelmingly, client/server request and reply messages are small. To date, the DSP-encoded messages have worked exceptionally well for all cases but one (the XIA nuclear detector data acquisition server mentioned previously).

Switching to a pure binary solution, although feasible, would throw away all the advantages of a text-based protocol (human readability, generic parsing, compatible between different compilers/CPU architectures, and so on), but it would provide a great increase in speed and efficiency. This is a trade-off that may well be worth making at some point in the future if the need to move large data streams between clients and servers becomes more common. However, we feel we have not yet reached that point. Switching to a pure binary messaging scheme would involve a great deal of effort. Keep in mind that client class API methods and server class message handlers build and parse messages directly. Switching to a binary protocol means major changes to every client and server in the code base. Surely this is not an impossible task — similar sweeping changes have been made to the code base in the past — but now is not the time for that. So long as large block binary transfers remain a requirement of just a single server, we are better off retaining the advantages of DSP-encoded messages and using the RTL for the rare instance in which DSP-encoding fails.

5 Implementation

The RTL provides two functions — `SendRaw()` and `ReceiveRaw()` — which are defined in *rawTransferLib.h* and implemented in *rawTransferLib.cpp*. These two functions are the focus of this section.

5.1 Connection Handle

Both of the library functions take a *connection handle* as one of their parameters. A connection handle is a bit of information that tells the receiving process how to create the network connection necessary to complete the data transfer. Connection handles are instances of a small data structure, called `RawHandle`, defined in *rawTransferLib.h*. A slightly abbreviated version of the definition appars below.

```
struct RawHandle {
   string ipAddr;
   WORD   port;
};
```

- `ipAddr` — The IP address the receiving process should connect to in order to receive the data to be transferred.
- `port` — The TCP port the receiving process should connect to in order to receive the data.

These fields are bundled in a connection handle structure rather than made available directly because the underlying details of the data transfer are supposed to be hidden from the user. Currently the library uses TCP to do its data transfer, so IP address and TCP port are appropriate. However, we might decide to change this to some other connection type in the future, in which case some different set of fields would be appropriate. By bundling the specific connection details into a generic *connection handle* concept, the underlying details can easily be changed in the future without affecting existing user code.

5.2 The Send-Side Interface

5.2.1 The `SendRaw()` Function

```
void SendRaw(void *data,
             int size,
             RawHandle &handle,
             const string hostname = "localhost");
```

- `data` — This is a pointer to the buffer that needs to be transmitted to the receiving process.
- `size` — This is the number of bytes that should be transferred from the data buffer to the receiving process (it is assumed that the buffer pointed to by `data` is large enough to hold at least this many bytes).
- `handle` — On successful return, this field will contain the details the receive-side process will need in order to connect to the sending process.
- `hostname` — This field contains the host name of the local computer (i.e., the computer hosting the sending process). If there is a simple, reliable way to get the IP address of a local socket *before* someone has connected to it, this author does not know what it is. Unfortunately, we need to send that IP address to the receiving process so it can establish a transfer connection. The cheesy method we have come up with, is to take a host name (this field) and then look it up using the `gethostbyname()` system call to find a corresponding IP address. The field defaults to "localhost," so for those cases in which both the sender and receiver process are running on the same computer (we anticipate this to be the most common case by far), the user should never need to actually fill in this field. However, if the receiver is on a different computer, then a real host name will need to be given.

The `SendRaw()` function creates and binds a TCP listen socket for the receiving process to connect to, and then it starts a separate thread to handle accepting the receiving process' connection. On return, the data in the `handle` field will be valid and the caller can then get this information to the receiving process in any convenient way (typically the standard, DSP-encoded message-passing mechanism is used to transfer the handle to the receiver).

5.2.2 The `RawTransmissionThread()` Function

```
static void* RawTransmissionThread(void *data);
```

This is a *pthread* thread start function for the thread that gets spawned by the `SendRaw()` function. Like all *pthread* start functions, it takes a single, void pointer parameter. In this specific application, `SendRaw()` spawns the thread with this pointer pointing to an instance of a `DataDefinition` structure (defined near the top of *rawTransferLib.cpp*). This data structure is filled in by `SendRaw()`, but used by `RawTransmissionThread()` to complete the TCP connection and perform the data transfer.

```
struct DataDefinition {
   int   sock;  // The socket the thread will do the accept on
   int   size;  // The size of the data block to be transferred
   void *buff;  // A pointer to the data buffer to be transferred
};
```

The thread waits for a TCP connection request from the receiving process. It then completes the connection and begins transferring data. Once all data has been transferred, the thread closes the sending side of the connection and exits.

5.3 The Receive-Side Interface: The `ReceiveRaw()` Function

```
void ReceiveRaw(const RawHandle &handle, void* buff, const int size);
```

- `handle` — This field contains the connection handle supplied by the sending process. The sending and receiving processes are free to exchange this handle in any way that is convenient, but it is typically transferred from the sender to the receiver using the standard, DSP-encoded message-passing mechanism.
- `buff` — A pointer to the buffer into which the receiving process wants the received data to be copied (it is assumed this buffer is big enough to hold all the data).
- `size` — The size of the buffer in bytes.

In a typical implementation of the receiving process, the receiver will begin by using the normal client/server message-passing mechanism to send a message to the sending process requesting data that will be returned via the RTL mechanism. The sending process will initially reply, also using the normal client/server message-passing mechanism. As part of this initial reply, the sending process will include the connection handle information the receiver needs. Finally, the receiver will call `ReceiveRaw()`. When the call returns, the receiver's buffer will contain the data transferred from the sender using the Raw Transfer Library.

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

U.S. DEPARTMENT OF
ENERGY

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov