U.S. DEPARTMENT OF ENERGY

PNNL-22587

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

The Client/Server Architecture

DES-0005

Revision 4

Charlie Hubbard

February 2012

Pacific Northwest
NATIONAL LABORATORY

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

The Client/Server Architecture

Charlie Hubbard

DES-0005
Revision 4
February 2012

Contents

1 Introduction

This document describes the client/server architecture used to implement the system control software. It looks at the history of this architecture as it pertains to our control systems, discusses why this architecture is superior to traditional monolithic designs, and examines some of the requirements for a client/server architecture and the design challenges those requirements present. Finally, an in-depth examination is made of the actual implementation of the client/server framework that has been implemented for use under Linux.

2 History

Originally all of our system control software was written to run under the QNX operating system. QNX is a *microkernel* architecture operating system, meaning that internally, the operating system is implemented as a number of small, concurrently running processes, that each perform a specific task (process management, file system management, network driver, etc.). These processes communicate with one another by passing messages back and forth via the operating system kernel.

This paradigm (a large system comprised of many independent, narrow-scope processes communicating via message passing) also proved to be excellent for system control applications, so it was adopted in the form of our *client/server* model architecture. Like the QNX OS itself, our early system control software was also comprised of a number of concurrently running applications, each of which was responsible for one piece of the over all system control problem. Individual applications coordinated with each other by passing messages back and forth. Under QNX, this was easy to do. Because QNX itself relied heavily on message-passing, it provided a coherent and easy-to-use set of system library calls specifically designed for sending, queueing, acting on, and responding to messages sent between two processes. For our own system control applications, all we had to do was take advantage of these pre-existing system libraries.

Today however, our system control software is implemented on the Linux operating system. This change from QNX to Linux has partly to do with cost (QNX is expensive, Linux is free), partly to do with customer demand, and partly to do with the rich development environments and toolsets available under Linux that are not available under QNX.

Unfortunately, Linux is not a microkernel, message-passing operating system, so it lacks the extensive, message-based framework for inter-process communication (IPC) that QNX had. Instead, a similar framework for Linux had to be developed in-house, so we could continue to develop systems using the client/server model that had proven so effective for system control applications over the years. This document describes the underlying architecture we developed to support client/server applications under Linux.

3 Client/Server Overview

In this section, the client/server architecture is discussed in general, higher-level terms. Here we explain the client/server model, and discuss some of the design requirements necessary to use the model successfully. What this section does not do is discuss actual implementation details. Those will be covered beginning with section 4.

Before going too much further, it would be worthwhile to explain exactly what we mean by "client/server architecture." First some definitions.

- ***server*** — A server is any independent process (program) running on the system that can accept requests from other processes, execute those requests, and then return the result (if any) to the requester.
- ***client*** — A client process is any independent process (program) that requests services of another process.

Although it is possible to have pure clients and pure servers, in practice, on our real-world control systems, most servers are also clients. That is to say, although they do expect to do work on behalf of client requests, they also send requests to other servers to do a portion of their work. In real system control applications, nearly all the servers are also clients to at least one other server. Pure clients are somewhat more common than pure servers.

3.1 Why Use a Client/Server Model?

Implementing a software design for an instrument control system (or most anything else) is typically done by building a single, monolithic executable that implements all aspects of the design. Implementing a design as a series of independent, concurrently running, executables that cooperate by passing messages back and forth is unusual. So why bother? Why spend the additional development effort needed to support a client/server solution if the project requirements can be met by a single, monolithic application?

Years of experience using the client/server model to implement 20+ instrument control systems has shown that the client/server model has numerous advantages over a monolithic solution. Here are some of them:

- In the client/server model, individual applications (servers) are typically focused on one, narrow task (providing the software interface to a turbo pump, for example). By keeping the servers tightly focused, they stay small and more easily maintained.
- Client and server applications all run as independent processes. Independent processes running under a modern multi-tasking operating system like Linux, enjoy hardware-enforced memory protection. That is, one process can not corrupt the memory of another process even if that is the author's intention. When each process runs in its own, isolated memory island this way, common memory corruption bugs like accessing an array past its boundaries or writing through an uninitialized pointer, can crash the application with the bug, but they cannot crash other applications on the system. This

is enormously helpful, because this class of bug often presents itself highly intermittently, and can be very difficult to track down and fix. In a monolithic implementation, such bugs can literally affect every part of the system. With a client/server implementation, only the process containing the bug is affected. Because client/server processes are, by their nature, small, that greatly reduces the volume of suspect code that needs to be searched for the problem.

- Clients and servers can be started, stopped, and restarted while the system is running. This is enormously useful during development and operation. New servers can be started, tested, stopped, fixed, and restarted without having to take down the entire system. Old versions can be replaced with new ones, on the fly, as the system runs. Client/server relationships that existed before a server was stopped will automatically be re-established when the server is restarted.
- Utility applications, unanticipated during the initial design and implementation, can easily be added to a system without having to recompile any of the existing system code, and even without restarting or suspending a currently operating system. This ability has been enormously useful in the past, allowing new system monitoring utilities or sensor data plotters (for example) to been added to (or removed from) operating systems without having to stop or otherwise perturb them.
- There is generally no requirement that clients and servers run on the same computer (although they generally do). In fact, clients and servers have no idea whether they're on the same hardware or not. Switching from one mode to the other requires absolutely no code changes. This attribute makes the client/server model particularly applicable to distributed control systems.
- Because servers are narrowly focused in scope, they are also naturally highly modular. This means, once the initial effort has been expended to develop a new server, odds are that that same server can be used again and again on new projects without any code changes at all. This is not just theoretical. On the real-world control systems we have implemented under the client/server model, typically 80% of the servers used on a new project are completely unaltered versions taken from our server code base. This has obvious reliability ramifications, since using pre-existing servers means using tested and proven servers. That they require no modifications also means there is no opportunity to introduce new bugs.

3.2 Message-Passing Model

The clients and servers that make up the system control software use a send-receive-reply message-passing model. It works like this. Servers that are not currently busy satisfying client request messages are *receive blocked*. That is to say, they are doing nothing but waiting for a new request to arrive from a client. While in this state, they are suspended by the operating system, so that they are not using any CPU cycles.

When clients send a request to a server, the server wakes up and begins to process the request. While waiting for the reply, the client is *reply blocked*. That is to say, the client is suspended by the operating system until the server returns the results of the client's request.

When the server has finished with its processing, it sends the reply message back to the requesting client and goes back to sleep. The operating system releases the client, the client uses the result returned from the server and goes on its merry way. The send-receive-reply mechanism is shown diagrammatically in figure 1.

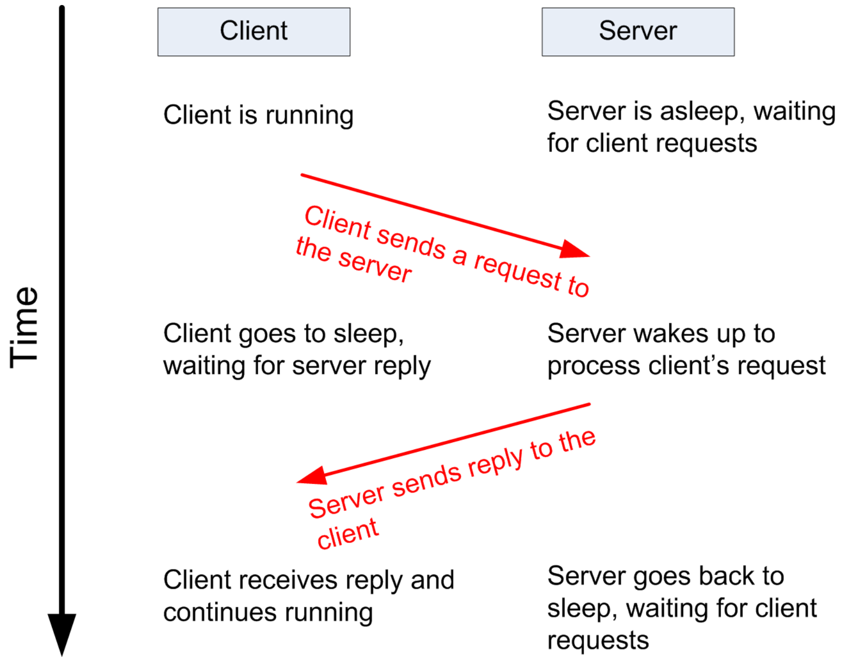
Client
Server
Time
Client is running
Server is asleep, waiting for client requests
Client sends a request to the server
Client goes to sleep, waiting for server reply
Server wakes up to process client's request
Server sends reply to the client
Client receives reply and continues running
Server goes back to sleep, waiting for client requests

Figure 1: The send-receive-reply mechanism used during client/server communication.

3.3 Deadlock

As mentioned previously, it is common in the system control software for servers to also send request messages to other servers (in which case the sending server is, for that instant, a client). When servers can send requests to other servers, there is a danger of a condition called *deadlock* occurring. Figure 2 illustrates the problem.

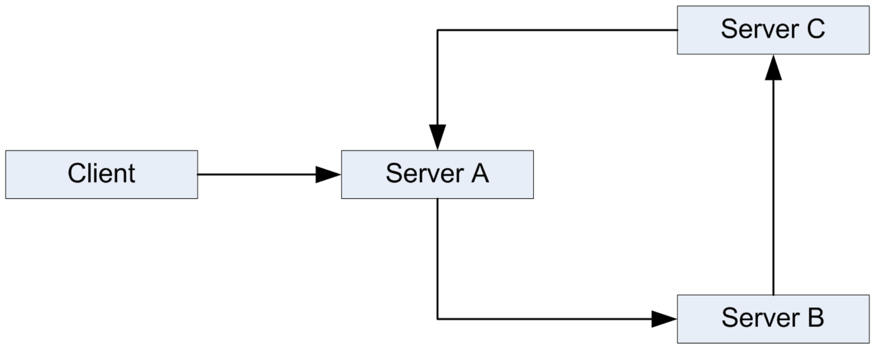

Figure 2: Deadlock occurs when request chains form a loop.

In this example, a client makes a request of *Server A* and becomes suspended waiting for a reply. In the course of processing the request, *Server A* sends a request to *Server B*, at which time *Server A* becomes suspended, waiting for a reply. Further, *Server B* sends a request to *Server C* and goes suspended, and *Server C* sends a request to *Server A* and goes suspended. And there is the problem. *Server A* is already suspended, so it is unable to respond to *Server C's* request. As a result, *Server C* never wakes up to reply to *Server B*, and *Server B* never wakes up to reply to *Server A*. In short, *Server A* is now suspended for all time and the client is as well. This is *deadlock*.

With our send-receive-reply message-passing model (figure 1), deadlock can occur anytime request message chains form a loop.

3.4 Connection Timeouts

In our actual implementation, the deadlock situation described in the previous section is not quite as dire as described. Client connections have a timeout value associated with them, so if a server doesn't respond within the timeout interval, the client request will be aborted and return an error.

In the current implementation, the timeout interval is set to three seconds for all client/server message transactions. That is, servers have one second to respond to a client request before the client gives up. In truth, when a message transaction fails, all clients will re-try the request exactly one additional time[1], so, the total delay before a failure is reported will be twice the timeout delay (so two seconds for the current implementation).

Placing a limit on how long servers have to respond is no solution to the deadlock problem. It simply provides a way for clients/servers in deadlock to break free so that they do not stay unresponsive for all time. Even so, the original client request that created the deadlock will never complete successfully. Even with connection timeouts, request message chains cannot be allowed to form a loop.

[1]It is this automatic reconnect and re-request mechanism that makes it possible to shutdown and restart servers, on the fly, without having to restart any of their associated clients.

3.5 Server Hierarchy

As mentioned in both of the previous two sections, the solution to deadlock is simple. *Do not* allow request message chains to form loops. Instead, servers need to be designed such that request chains are linear (typically organized into a tree). Such a design will lend itself to a natural hierarchy of servers. In real-world control systems, servers lower in the hierarchy typically provide lower-level services (hardware drivers, etc.) while more abstract tasks (overall system process control, user interfaces, etc.) are handled by servers higher up in the hierarchy. Figure 3 below shows an example of a fictitious control system that has its server processes arranged in a typical hierarchy.

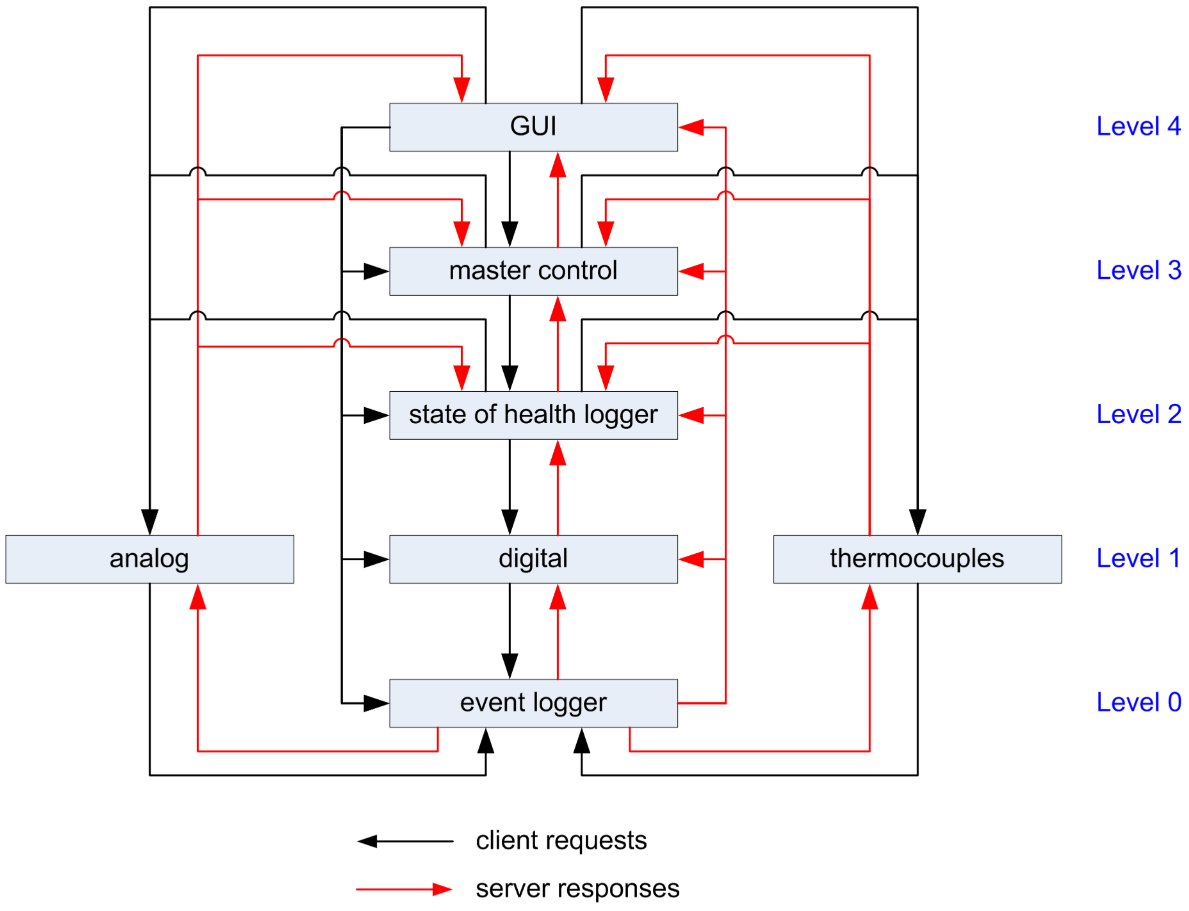

Figure 3: Control system servers arranged in a typical hierarchy. In this example, *GUI* is a pure client and *event logger* is a pure server.

Note in the above example, servers only send requests to servers that are below them in the hierarchy. This is a required attribute for a proper design. So long as this rule is adhered to, deadlock will be avoided.

3.6 Message Serialization

In our client/server model, clients and servers run simultaneously and independently of one another. A situation often arises where two or more clients (multiple, remotely located GUIs or system monitor processes for example), send requests to the same server at or near the same time. There is a potential for conflict here if the server does not completely finish handling one request before the next request arrives.

To avoid this problem, each server maintains its own message queue. Messages sent to the server are placed on the server's message queue and are processed out of the queue, in full, in first-in/first-out order. In effect, the message queue serializes concurrent requests, and guarantees that the server can fully process one message before dealing with the next. Under the QNX operating system, this ability to queue messages is built in. Under Linux, we have had to implement our own message queues.

4 Server Implementation

4.1 Messaging Overview

Message passing is a central topic to any discussion about the client/server model. Because the details of our chosen message-passing mechanism play an important role in the design of the system servers (and clients), we will spend some time looking at the design details of our message-passing mechanism before we delve into the specifics of the actual system server design.

4.1.1 Message Transport

Clients and servers communicate with one another using the Internet standard *transmission control protocol* (TCP) as the underlying message transport layer. TCP was chosen over other Linux IPC mechanisms (shared memory or Unix sockets for example) because TCP can just as easily transport messages between processes running on separate computers as it can between processes running on the same computer. This was a feature of QNX's message-passing implementation as well, and it is a feature we have taken advantage of in the past. We did not want to lose the option of multi-computer solutions for future projects. Also, TCP is a reliable delivery protocol, so, when using TCP, no additional work is needed checksumming messages, verifying message sequence numbers, etc. TCP takes care of that itself. Finally, TCP already supports simultaneous connections to a single process from multiple other processes. This is definitely a feature we require, and TCP gives us that for free.

4.1.2 Establishing Connections

Obviously, before a client can communicate with a server, it must first establish a TCP connection to that server, and, before it can do that, it needs to know what IP address and TCP port number the server is listening on for new client connections. In our implementation, servers are known by short text names. A name resolver exists that can map these names to IP address / port number pairs that a client can then use to create the connection. Servers must register themselves with the name resolver facility before clients will be able to connect to them. The name resolver feature is fully described in design document ***DES-0004*, *The Server Name Resolver***. Please look there for details.

Note that there is a certain amount of overhead involved in establishing a TCP connection between client and server. Not only is there the overhead of creating the connection itself (which, in addition to accepting a new TCP connection request, also requires spawning a new thread on the server side), but, before a connection request can even be made, a client must first connect to and query the name resolver. Obviously this is overhead we do not want to incur *every time* a client sends a request message to a server. Instead, once a TCP connection has been established, that connection remains open to be used over and over again throughout the life of the client and its server. The only exception is if the transmission of a request message ever fails, clients automatically attempt a reconnection. It is this mechanism that makes it possible to stop an existing server and then restart it at some later time without needing to stop or restart any of that server's clients.

4.1.3 Message Format

Clients and servers exchange text-based messages encoded using an in-house format that is simple to create and simple to parse. The format is fully described in design document ***DES-0002*, *Data Serialization Protocol***. The reader is urged to review that document to better understand the following discussion.

Although messages are encoded using our Data Serialization Protocol (DSP), not just any DSP format string constitutes a valid client/server message. Real client/server messages have the following restrictions.

Request messages (that is, messages sent by clients to servers) all have the following format.

```
(<clientName>,<commandName>,(<parameters>))
```

In the same way that all system control servers are given short text names, system control clients also assign themselves short text names. Mostly this is done for logging purposes. In the above format description, `<clientName>` is the text name the client uses to refer to itself, `<commandName>` is the name of the command the client wants the server to perform, and `<parameters>` is a DSP element list (possibly empty) containing any additional parameters the server will need to process the command.

Reply messages (that is, messages sent by servers to clients) all have one of two possible formats, depending on whether or not the server is reporting success or failure. Success response messages are formatted as follows:

```
(ok,(<response>))
```

Here, "ok" is literally the string "ok." `<response>` is a DSP element list (possibly empty) containing any additional parameters the server needs to report back as part of the response. Notice that `<response>` is enclosed in parentheses. The parentheses and the `<response>` portion taken together constitute a complete, valid DSP message that must be parsed by the DSP parser at least once more to get at the individual fields in the response.

Failure response messages are formatted as follows.

```
(error,<textDescription>)
```

Here, "error" is literally the string "error." `<textDescription>` is a text string containing a human-readable description of the error condition.

4.2 The `BaseServer` Class

All real-world servers have certain attributes in common. They all…

- are assigned a short text name by which they are known to the rest of the system
- need to register themselves by name with the name resolver to become available to accept messages from clients
- need to listen for and handle new TCP client connection requests
- need to have their own message queues
- need to be able to receive, parse and understand client request messages formatted as described in section 4.1.3
- need to be able to receive, parse and act on internal messages generated by the servers themselves
- need to be able to send clients success and error response messages formatted as described in section 4.1.3
- share a small set of standard messages which all clients can generate and all servers can respond to

In the code base, these common characteristics have been captured by the C++ `BaseServer` class defined in *cliserv.h* and implemented in *cliserv.cpp*. By providing a base class from which all real-world servers are expected to be derived, we can guarantee that all real-world servers handle the above tasks in a consistent manner, and relieve server authors from worrying about the underlying details involved in message passing and message queueing. More importantly, if changes need to be made in the future (switching to a priority-based message queue, for example), those changes only need to be made in the `BaseServer` class, and all derived servers will inherit the changes automatically, without needing to be directly modified.

The `BaseServer` class definition is rather large. A version of it without the in-code comments appears below. Readers are directed to the full class definition in *cliserv.h*, and the associated *Doxygen* documentation for the `BaseServer` class.

```
class BaseServer {
public:
   BaseServer(const string name);
   virtual ~BaseServer();
   void Run();

protected:
   bool ReplyMessage(const string resp = "()");
   bool ReplyError(const string errMsg);
   void QueueServerMessage(const int msgId);
   virtual void ClientMessageHandler();
   virtual void ServerMessageHandler(const int msgId);
   virtual void GetServerStats();
   virtual void GetNumSohParams();
   virtual void GetSohParamInfo();
   virtual void GetSohParams();
   virtual void Shutdown();
   bool                shutdown;
   string              serverName;
   CS_ServerStats      stats;
   string              lastErrorMessage;

   string             clientName;
   string             command;
   string             payload;

   int                currentClientSockId;

 private:
   static void* SignalHandlingThread(void *data);
   static void* ClientListenThread(void *data);
   static void* ClientConnectionThread(void *data);

   MessageParser      msgParser;

   time_t startTime;

   struct SessionFrame {
      int         sockId;
      BaseServer *base;
   };

   struct QueueElement {
      bool   clientMessage;
      int    value;
      string message;
   };
   typedef deque<QueueElement> MessageQueue;

   pthread_mutex_t statsMutex;
```

```
    pthread_mutex_t queueMutex;
    pthread_cond_t  cond;
    MessageQueue    messageQueue;
};
```

4.3 Client Connections

When a new server class is instantiated, its constructor performs a number of tasks, one of which is to spawn a new thread, called the *client listen thread* (in the code, this thread is implemented by the `ClientListenThread()` static method). The client *listen* thread, registers the server with the name resolver (doing this in the base class relieves server authors from having to worry about the name resolver facility), and creates a TCP socket on which to listen for new client connections.

With the listening TCP socket created, the thread goes to sleep waiting for new client connection requests to arrive. When a client connects, the client *listen* thread creates another thread, called a *client connection thread* (implemented in the code by the static `ClientConnectionThread()` method) and attaches the newly created client TCP connection to it. At that point, the client *listen* thread goes back to sleep, waiting for the next client connection request. The newly created client *connection* thread handles all further communication with the client. Note that one new client *connection* thread gets created every time a client connects to the server, and the thread goes away when the client disconnects.

4.4 The Message Queue

4.4.1 Server Request Messages

Servers actually receive messages from two sources. The most common source is from clients sending request messages, but servers also have the ability to generate messages to themselves. For lack of a better term, we will call these internally generated messages *server request messages*. This ability for servers to send messages to themselves is primarily useful in two situations — timers and software signals. Much more will be said about timers and signals in sections 4.5 and 4.6, but here is a brief overview.

Quite often it is useful for a server to be able to wake itself up periodically to perform some task. For example, a server responsible for regulating the temperature of a heater may want to wake itself up once a second, check the current temperature and adjust the heater power accordingly. This can be accomplished by use of a timer facility. During server initialization, the server sets up a timer to expire at one second intervals and then goes to sleep, waiting for client requests. When the timer fires, it places a message on the server's message queue. The server reacts in the same way that it would when handling incoming client requests. Namely, it wakes up, takes the message off the queue, recognizes it as a request to check on the heater power, adjusts the heater power as necessary and goes back to sleep.

Server request messages differ from standard client request in a few ways.

- They don't come from a client; They come from the server itself.
- They do not carry any payload other than a single integer identifier that indicates the source of the message (i.e., a specific timer).
- Because they are generated by servers to themselves, they don't need to be encoded as DSP format messages. Instead, they are written directly into the server's own message queue.
- Unlike client requests, which are *always* responded to, server request messages are acted upon, but they are *never* responded to (after all, there's no corresponding client to receive a response). This also prevents deadlock[2]. Servers can never send regular client messages to themselves because client requests always block waiting for a response. There is no need to block for server messages, because server messages *never* generate responses.

The other place server request messages are commonly used is in response to software signals. As an example of a software signal, consider the Linux *kill* command. When a user uses the *kill* command from the command line to terminate a running application, what actually happens is a POSIX software *SIGTERM* signal is generated and sent to the program by the operating system. On receipt of this signal, most programs simply exit. However, that is not mandatory. Programs that care to can set up signal handlers that intercept the *SIGTERM* signal (as well as many others) and act on it in other ways. Commonly, a program may use this ability to perform some clean-up tasks before they exit. Other programs may choose to ignore the signal altogether, making them immune from being killed by a *SIGTERM* signal.

Being able to intercept signals and operate on them in ways that make sense to a specific application is extremely useful. The system servers take advantage of this internally by setting up signal handlers for several common signals (*SIGTERM* included). When they receive one of these signals, the signal handler packages the signal into a server request message and queues the message to the server. At this point, the message becomes like any other message in the server's message queue, and it will be handled in the order received. Typically a server will respond to a *SIGTERM* message by performing any clean-up tasks that are necessary and exiting. Different signals may cause different behavior, and authors of servers are free to override the defaults for their specific server.

4.4.2 Message Queue Implementation

Internally, the `BaseServer` class maintains a message queue that is used to hold client request messages and internally generated server request messages waiting their turn for processing. Messages placed on the queue are serviced in the order they are received.

The queue itself is implemented as a standard STL `deque` structure (this is the `messageQueue` field defined in the class definition above). Individual elements in the queue are of type

[2]See section 3.3 for a detailed description of the deadlock problem.

`QueueElement`, (also defined above), which is a simple `struct` containing three fields – `clientMessage`, `value`, and `message`.

- `clientMessage` — This is a Boolean field. If `true`, it indicates that the message contained within this element is a normal DSP-encoded message that originated from a client. Otherwise, this element represents a server request message.
- `value` — This is an integer field that can be interpreted in two different ways. As noted earlier, server request messages are identified by a single integer identifier. If `clientMessage` is `false`, then the element contains a server request message and this field contains that identifier. If `clientMessage` is `true`, then this field holds the socket ID of the socket the client is using to communicate with the server. It is necessary to save this socket ID so the server will know on which TCP connection the response should be sent.
- `message` — This is an STL string field. If `clientMessage` is `true`, then the element contains a standard client message and this field contains the DSP-encoded version of that message. If the element contains a server request message instead, then this field is unused.

New queue elements get added to the message queue from two different places, corresponding to the two different kinds of messages that can be queued (client requests and server requests).

Server request messages are added to the queue by the `QueueServerMessage()` method.

```
void QueueServerMessage(const int msgId);
```

This method takes as its only parameter, an integer identifier that will be copied to the `value` field of a `QueueElement`. The method is simple. All it does is construct a new queue element and adds it to the server's message queue.

Client request messages are added to the queue by the static `ClientConnectionThread()` method. Remember there is one of these threads for every client that is connected to the server.

`ClientConnectionThread()` is also quite simple. It uses an instance of the `ReadStream` class (please see design document ***DES-0003*, *The ReadStream Utility Class***, where this class is fully described) to read DSP-formatted messages arriving from the client on the client's TCP connection. Once a full message has arrived, it is added to an appropriately configured `QueueElement` element and added to the message queue. Then the thread goes back to waiting for further client request messages.

Elements are pulled off the message queue and processed by the server's `Run()` method. The `Run()` method acts as the server's queue processor. It is implemented in the server's main-line thread, and it spends most of its time asleep, waiting for new messages to appear in the queue. When a new element appears on the queue, the `Run()` method wakes up, removes the front element, and passes it on for further processing. Each additional queue element is likewise processed (if there are any additional queue elements) until the queue is empty.

Once the queue is empty, the thread goes back to sleep until another message appears on the queue.

Processing is slightly different depending on whether the element contains a client request or a server request. If it contains a client request, the message is extracted from the element's `message` field and a top level parse of the DSP format message is performed (using the `MessageParser` utility class described in design document ***DES-0002*, *Data Serialization Protocol***). All valid client requests must be of the format described in section 4.1.3, which means they contain three parameters — client name, command name, and a parameter list (possibly empty). These three parameters are extracted and stored in the `BaseServer` class variables `clientName`, `command`, and `payload` respectively. Then the client message dispatcher is called to further decipher the message, perform the requested action and send a reply back to the client. The client message dispatcher is further described in section 4.4.3.

If the element pulled from the message queue contains a server request, then the server message dispatcher is called instead, passing to it the `value` field from the queue element. The server message dispatcher will be further described in section 4.4.5.

4.4.3 Client Message Dispatcher

The client message dispatcher is implemented in the code by the `ClientMessageHandler()` method.

```
virtual void ClientMessageHandler();
```

This method serves as a clearing house for all client requests. If the request can be serviced by a very small amount of code, it may be handled directly by this method. However, more typically the request is sent on to a message handler that is written to deal with the specific command in the message. Technically, `ClientMessageHandler()` requires three things — the name of the client that sent the message, the name of the command the client wants performed and the list of additional parameters (if any) that the server will need to perform the command—yet the method takes no parameters. That is because those three bits of information have already been copied to internal class variables, as described in the previous section. The dispatcher as well as all specific-purpose message handlers expect to find the information in those class variables.

Notice that `ClientMessageHandler()` is a *virtual* method. All real-world servers override this method in order to handle their own server-specific commands. A typical real-world version of this method will compare the `command` variable to a list of messages the server is prepared to handle, and process the message if a match is found. If no match is found, it is assumed the message is one of the common messages shared by all servers (see section 6), and the `BaseServer` class' version of the method is called to deal with it.

4.4.4 Client Message Handlers

As described in the previous section, typically the client message dispatcher passes received client messages on to message handler methods for final processing. See the definitions and implementations of the `BaseServer` methods `GetServerStats()`, `GetNumSohParams()`, `GetSohParamInfo()`, `GetSohParams()` and `Shutdown()` in the files *cliserv.h* and *cliserv.cpp* for examples of client message handlers implemented in the base server.

Client message handlers have several attributes in common.

- When client message handlers are called, the class internal variables `clientName`, `command`, and `payload` are still valid. Client message handlers rely on those to get at the content of the client's request. For that reason, client message handlers do not take any parameters.
- It is the responsibility of the message handlers to further parse the client request as necessary to extract whatever parameters were passed as part of the message. This is typically done using one or more local instances of a `MessageParser` class (for complete details on this class, see the design document ***DES-0002, Data Serialization Protocol***), or one of the parsing helper functions described in section 8.
- All client request messages *must* be responded to, and it is the responsibility of the individual client message handlers to send those response messages back to the client.
- Because success and failure indications are returned directly to the client, client message handlers do not need to return a return value.

Two methods exist to assist with constructing and sending response messages back to clients — `ReplyMessage()` and `ReplyError()`. These are described below.

```
bool ReplyMessage(const string resp = "()");
```

The `ReplyMessage()` method (defined in *cliserv.h* and implemented in *cliserv.cpp*) is used by client message handlers when returning a success result to a client. It takes a single STL string parameter called `resp`. This parameter contains a DSP-encoded message containing any parameters that need to be passed along with the response beyond the success indication itself. That is, the `resp` string contains the `(<response>)` portion (including the parentheses) of the response message described in section 4.1.3. The "ok" as well as the outermost enclosing parentheses are automatically added by `ReplyMessage()`.

Note that if the response has no additional parameters to send, then `resp` should be set to the empty DSP message "()". As a convenience, the `resp` parameter defaults to the empty DSP message if no value is explicitly provided for the parameter.

The `ReplyMessage()` method returns a Boolean value. If `true` is returned, it indicates that the message was delivered to the client successfully. If `false` is returned, it indicates that the response message could not be delivered to the client (perhaps the client is no longer running or the TCP link to the client has been broken). In this case, the `BaseServer` class variable `lastErrorMessage` will contain a text description of the failure.

```
bool ReplyError(const string errMsg);
```

The `ReplyError()` method (defined in *cliserv.h* and implemented in *cliserv.cpp*) is used by client message handlers when returning a failure result to a client. The method takes a single STL string parameter called `errMsg`. This parameter contains a human-readable text description of the error. Note that, unlike `ReplyMessage()`, which requires its parameter to be encoded as a DSP message, the `errMsg` string is bare. The `ReplyError()` method will make sure that the text is properly DSP-encoded before sending it to the client.

The `ReplyError()` method returns a Boolean value. If `true` is returned, it indicates that the message was delivered to the client successfully. If `false` is returned, it indicates the response message could not be delivered to the client. In this case, the `BaseServer` class variable `lastErrorMessage` will contain a text description of the failure.

4.4.5 Server Message Dispatcher

The server message dispatcher is implemented in the code by the `ServerMessageHandler()` method.

```
virtual void ServerMessageHandler(const int msgId);
```

Like `ClientMessageHandler()`, this is a virtual method that is expected to be overridden by any real-world server implementation that expects to generate server request messages. Unlike `ClientMessageHandler()`, this method does take a parameter, `msgId`, which corresponds to the `value` field from the message queue element. The `msgId` parameter identifies the source of the server request (a timer expiring for instance) and allows it to take the appropriate action. Like `ClientMessageHandler()`, `ServerMessageHandler()` may choose to handle the message directly if it can be done in a small amount of code, but typically the message is handed off to a specific-purpose message handler to be dealt with.

4.4.6 Mutexes and Condition Variables

As we have seen, all system servers are multi-threaded applications. Each server has its main-line thread, which processes new messages that appear on the message queue: one thread to listen for new client connections, and one additional thread for each currently connected client. Each server also has a *signal handling thread*, which will be described in greater detail in section 4.5.

The main-line thread and all client connection threads need access to the server's message queue. Because the threads all run independently of each other, there is the danger that two or more threads could try to make changes to the queue simultaneously, which would likely result in a corrupted message queue, which would spell disaster.

A way is needed to protect access to the queue such that no more than one thread can manipulate it at a time. Linux (via the POSIX *pthreads* library) provides a mechanism for doing that called a *mutex*. A mutex can be thought of as an access token. If you have the mutex, you can access the queue, otherwise, you have to wait for whoever does have the mutex to give it up so you can take it.

In the code, the mutex used to protect the server's message queue is called `queueMutex`. It is defined as

```
pthread_mutex_t queueMutex;
```

In those places where the message queue is manipulated (that is to say, in the `Run()`, `QueueServerMessage()`, and `ClientConnectionThread()` methods), you will see code similar to the following:

```
pthread_mutex_lock(&queueMutex);
<do some stuff with the queue here>
pthread_mutex_unlock(&queueMutex);
```

In this code, "locking" the mutex means "acquiring the token." If a thread attempts to lock a mutex that is already held by some other thread, the operating system puts the thread to sleep until such time as the mutex becomes available. If the thread is executing code that follows the lock request, it means that, at that point, it owns the mutex, and it can safely modify the queue without having to worry about other threads conflicting with it. Once a thread finishes modifying the queue, it must "unlock" the mutex, which is to say "relinquish the token so another thread can take it."

Although the message queue mutex handily solves the problem of multiple threads trying to access the message queue at the same time, there is another problem that has to do with the queue processor (the `BaseServer::Run()` method, which runs in the main-line thread). Specifically, the queue processor needs to know when there are new messages on the queue so that it can handle them.

There are about two ways to handle this — the good way, and the poor way. We will describe the poor way first, so that you can see the problems with it and how those problems are overcome by the good way.

One solution might be to have the queue processor sit in a tight loop, constantly polling the status of the queue, so as soon as a new message is available, it can parse and dispatch it. This is called a *busy wait* loop. A busy wait loop is undesirable because most of the time it does nothing useful (because most of the time there are no new messages on the queue waiting to be processed). Even so, the thread locks the mutex, checks the message queue, and unlocks the mutex over and over and over again, as fast as the CPU and operating system allow. Doing this is hugely inefficient and a tremendous drain on the CPU.

The burden placed on the CPU could be partially alleviated by introducing a sleep in the loop. For instance, the loop could check the queue, and, if there are no messages waiting, go to sleep for one second before checking again. Although this would work, it introduces an unnecessary and probably unacceptable delay between the time a client request is added to the queue and the time the server actually starts processing the request.

What is really needed is a way for the individual client connection threads to notify the queue processing thread when they add a new message, and leave the queue processing thread asleep, consuming no CPU cycles, until that notification arrives. Fortunately, the operating system (via the POSIX *pthreads* library) offers just such a capability in the form of, what the library calls, a *condition variable*.

Condition variables are tied to a mutex. In the code base, the same mutex that is used to protect the message queue from simultaneous access is also used with the condition variable. To see how this works, we will examine some code.

```
pthread_mutex_lock(&queueMutex);
while(!shutdown) {
   pthread_cond_wait(&cond, &queueMutex);
   ...
   while (messageQueue.size() > 0) {
      // process front message off the queue
      ...
      pthread_mutex_lock(&queueMutex);
   }
}
```

The above code snippet is a stripped-down version of the `BaseServer::Run()` method's queue processor. For clarity, most everything but the use of the condition variable has been removed. The code begins by locking the mutex. This guarantees exclusive access to the condition variable (called `cond` in the code). With the mutex locked, a call to `pthread_cond_wait()` is made, passing to it both the mutex and the condition variable. In effect, `pthread_cond_wait()` tells the OS to put the thread to sleep until some other thread notifies it that there is work to be done. As its last action before going to sleep, `pthread_cond_wait()` also unlocks the mutex so that the condition variable is available to other threads.

The important things to take away from this are:

- The mutex *must* be locked before the call to `pthread_cond_wait()` (`pthread_cond_wait()` will unlock the mutex itself before putting the thread to sleep)
- While asleep, the message processor thread consumes no CPU cycles
- The client connection threads can cause the message processor to wake up by signalling through the condition variable whenever they put new messages on the queue

This last item is demonstrated by the following code snippet taken from the part of the code that handles client connections.

```
while (readStream.Read(sf->sockId, element.message)) {
   pthread_mutex_lock(&sf->base->queueMutex);
   sf->base->messageQueue.push_back(element);
   pthread_cond_signal(&sf->base->cond);
   pthread_mutex_unlock(&sf->base->queueMutex);
}
```

Here, the client connection thread uses the ReadStream utility class (see design document ***DES-0003*, *The ReadStream Utility Class*** for full details), to wait for a complete message to arrive from a client. Then it locks the mutex (to gain exclusive access to the message queue and the condition variable), places the new message on the queue, and calls `pthread_cond_signal()`. This causes the operating system to wake up the queue processor thread to handle the new message. If new messages arrive while the queue processor is handling the current message, they will be enqueued properly and handled, in the order received, as quickly as the queue processor can process them out of the queue.

The code that adds server request messages to the queue, interacts with the condition variable in exactly the same way.

4.5 Signal Handling

As mentioned in section 4.4.1, in addition to messages sent by clients, servers can also receive messages they send to themselves. This commonly happens in two cases — timed events, which are discussed in section 4.6, and software signals.

POSIX signals are a way for events to be generated, sent to, and handled by a process asynchronously. They are somewhat akin to hardware interrupts, in that a signal event can occur at any time, and, when it does, it is handled by a special function that runs outside of the normal code execution path, so that it is always immediately available to handle the event. For hardware interrupts, the special handler function is called an *interrupt handler*. As you might expect, for software interrupts (signals), the handler function is called a *signal handler*.

All processes running on a Linux system have the ability to receive signals, although few of them have explicit signal handlers. If the code author does not explicitly provide a signal handler, then default signal handling behavior takes over. By default, signals are typically ignored or cause the running process to terminate, depending on the nature of the signal.

In server applications based on the `BaseServer` class, the situation is somewhat more complicated, because `BaseServer`-based applications are multi-threaded. We only want one thread to respond to signals. All other threads should just ignore them. This is accomplished in two steps. First, in the `BaseServer` constructor, there is code that causes the application to ignore *all* incoming signals.

```
sigfillset(&set);
```

```
sigprocmask(SIG_BLOCK, &set, NULL);
```

Because all threads inherit their signal handling ability from their parent, and the parent of all threads is the thread that runs the `BaseServer` constructor, all threads in a `BaseServer` derived application will ignore all signals. . . except for one.

Once signals are set to be globally ignored, a special signal-handling thread is created (implemented by the static `SignalHandlingThread()` method).

```
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&thread, &attr, SignalHandlingThread, (void*)this);
```

The signal handling thread uses a call to the system library function `sigwaitinfo()` to, in effect, say "Put me to sleep now, but wake me up if *any* signal is delivered to this application."

```
sigfillset(&set);
for (;;) {
   sigwaitinfo(&set, &info);
   \\ process signal here
   ...
}
```

Although this is not the normal way of registering a signal handler, in effect, the signal handling thread becomes *the* signal handler for the entire server application. The thread sleeps until a signal is delivered to the server, at which time the thread wakes up and examines the signal. The thread responds to the signal by queueing a *server request message* (see section 4.4.1) and then going back to sleep waiting for the next signal to arrive.

The server request message that gets queued contains an integer identifier that indicates to the queue processor the type of signal that was received or the source of the signal (depending on the signal). The queue processor hands these queued signal events off to the appropriate message handler for further processing. For instance, if a SIGTERM signal was generated, (a user could do this from the command line using the *kill* or *killall* shell commands for example), the default server request handler (`ServerMessageHandler()`) responds by setting the server's internal `shutdown` flag to true, which, in turn, causes the queue processor (the `Run()` method) to exit and the server's destructor to run, hopefully performing whatever clean-up needs to be performed before finally exiting.

As mentioned previously, the `ServerMessageHandler()` method can be overridden by real-world server authors to implement whatever behavior they deem appropriate for various signals.

4.6 Timers

The other common source of server request messages (that is messages that servers queue to themselves) is timers. These are used frequently for those servers that need to wake up periodically and perform some task (like adjust a heater's output power as described in section 4.4.1).

On Linux systems, a convenient way to get timer functionality is to use POSIX standard per-process timers, which are implemented in a series of library functions, including `timer_create()`, `timer_delete()`, and `timer_settime()`. POSIX timers are versatile, and their versatility makes them somewhat more complicated to use than need be for the task at hand. For that reason, a wrapper class called `Timer` (described below) has been created to simplify the interface for server authors (see *cliserv.h* for the class definition and *cliserv.cpp* for its implementation). The class definition appears below.

```
class Timer {
 public:

   enum TimerType_e {
      ONE_SHOT,
      PERIODIC
   };

   Timer(const int sid);
   ~Timer();
   bool Set(const int ms, const TimerType_e type, const int msgId);
   bool Start();
   bool Stop();

   string lastErrorMessage;

 private:

   bool        timerCreated;
   int         ms;
   TimerType_e type;
   timer_t     tid;
   int         sigId;
};
```

As can be seen, `Timer`-based timers can be started and stopped, and their expiration interval can be set. In addition, timers can run either in *one shot* mode (after they expire once, they don't expire again unless manually restarted), or *periodic* mode (timers continue to expire at the expiration interval until manually stopped).

Notice that there are no explicit methods for creating a timer or deleting timer. The associated POSIX timer is created internally, as necessary when the `Set()` method is called, and it is deleted when the class destructor runs.

`Timer`-based timers notify the process that owns them that they have expired by sending a POSIX signal to the process. This signal is handled using the server's standard signal handling mechanism (see section 4.5) — with one complication. `Timer` based timers use POSIX *real-time* signals. Real-time signals differ somewhat from the signals discussed in section 4.5. For our purposes, the main differences are:

- They have no pre-defined meaning. This is different from signals like SIGHUP, SIGTERM and SIGKILL which mean specific things. For example, SIGTERM is a request for a process to terminate and SIGKILL is a *command* to terminate immediately (cannot be overridden). The interpretation of POSIX real-time signals, on the other hand, is left entirely to the application writer.
- POSIX real-time signals can carry an additional bit of data (either an integer or a pointer) with them in addition to the signal number itself.

The following code snippet, taken from the base server's signal handling thread's static method, shows how `BaseServer`-based servers handle timer signals specially.

```
for (;;) {
   sigwaitinfo(&set, &info);
   if (info.si_code == SI_TIMER) {
      base->QueueServerMessage(info.si_value.sival_int);
   }
   else {
      switch (info.si_signo) {
      case SIGTERM:
         base->QueueServerMessage(BASE_SIGTERM_MSG);
         break;
         ...
```

As you will recall from section 4.5, all `BaseServer`-based servers have a special signal handling thread that handles incoming signals for the entire server application. It does this by using the `sigwaitinfo()` library function to receive information on all delivered signals. When a signal is delivered, `sigwaitinfo()` populates a structure with information about the signal. For normal signals, the only bit of information available is the number of the signal itself (found in the field `si_signo`). For real-time signals, the struct contains a couple of other bits of useful information. The field `si_code` contains the value `SI_TIMER` for signals that were generated as the result of a POSIX timer expiring. Also, the field `si_value.sival_int` contains whatever integer value was attached to the signal when the author created it. It is this value that gets placed in the message queue and used by the queue processor to determine which message handler to call to handle the timer expiration.

4.6.1 The `Timer` Constructor

```
Timer::Timer(const int sid)
```

The `Timer` class constructor takes one parameter — an integer value called `sid`. This is the ID of the *signal* to be attached to this timer, and should not be confused with the value the timer will deliver to the message queue when it expires. The number of POSIX real-time signals that are available on Linux and other Unix-like operating systems varies from system to system but the range of legal values is specified by two symbolic constants — SIGRTMIN and SIGRTMAX. It is recommended that servers use SIGRTMIN as the signal associated with their first timer. If additional timers are required, use SIGRTMIN+1, SIGRTMIN+2, etc.

The constructor does not actually create a POSIX timer. Instead, it simply initializes some internal class parameters and stores away the ID of the real-time signal to be used when a POSIX timer is actually created.

4.6.2 The `Timer` Destructor

```
Timer::~Timer()
```

The `Timer` class destructor checks to see if a real POSIX timer has been created for the class, and, if so, deletes it.

4.6.3 The `Set()` Method

```
bool Timer::Set(const int ms, const TimerType_e type, const int msgId)
```

The `Set()` method takes three parameters with the following meanings.

- `ms` — An integer parameter that contains the timer's expiration interval specified in milliseconds
- `type` — This parameter should contain either `Timer::PERIODIC` or `Timer::ONE_SHOT`. It determines whether or not the timer will automatically restart after it expires.
- `msgId` — This is an integer message ID to be delivered to the server's message queue each time the timer expires.

This method returns `true` if the call was successful or `false` otherwise. If the call fails, the class variable `lastErrorMessage` will contain a text description of the error.

It is the `Set()` method that actually creates a POSIX timer for the object. If a POSIX timer has already been allocated to the object, the old timer is first deleted, then a new timer is created. The ID of the real-time signal to be used with the timer and the ID of the message to be delivered to the server's message queue are both associated with the timer at the time the timer is created.

4.6.4 The `Start()` Method

```
bool Timer::Start()
```

The `Start()` method starts a timer previously initialized by the `Set()` method. It takes no parameters. The method returns `true` if the call was successful or `false` otherwise. If the call fails, the class variable `lastErrorMessage` will contain a text description of the error.

4.6.5 The `Stop()` Method

```
bool Timer::Stop()
```

The `Stop()` method can be used to stop a timer previously initialized by the `Set()` method. Calling `Stop()` on a timer that is already stopped is harmless. This method takes no parameters. The method returns `true` if the call was successful or `false` otherwise. If the call fails, the class variable `lastErrorMessage` will contain a text description of the error.

4.7 Server Statistics

Internally, the `BaseServer` class (and therefore all real-world servers as well) maintain a number of statistics on the number of clients that have connected, have disconnected, are currently connected, message queue depth, message counts, per-message processing timings, etc. These statistics are meant to help developers better characterize the behavior of their systems. The statistics are available to all clients via the standard `CS_GET_SERVER_STATS` message (more fully described in section 6).

Each server maintains its statistics counters in a class variable called `stats`, which is a `struct` of type `CS_ServerStats`. This structure itself contains an STL map of another structure called `CS_MessageStats` which is used to contain count and timing information for each individual message that the server handles. The data fields of both structures are shown below.

```
struct CS_MessageStats {
   string messageId;
   int    num;
   double minTime;
   double avgTime;
   double maxTime;
};
typedef map<string, CS_MessageStats> CS_MSMap;
```

```
struct CS_ServerStats {
   int      upTime;
   int      numClientConnects;
   int      numClientDisconnects;
   int      currentNumClients;
   int      maxNumClients;
   int      numClientMessages;
   int      numServerMessages;
   int      currentQueueDepth;
   int      maxQueueDepth;
   CS_MSMap msMap;
};
```

Both of these structures are defined in *cliserv.h*. For more information on these, refer to the *Doxygen*-generated documentation for the structures and the comments in the source code itself.

5 State-of-Health Data

NOTE: The following discussion describes support for sensor state-of-health reporting, but only insofar as it concerns the `BaseServer` class. For a full discussion on system-wide state-of-health reporting, see design document ***DES-0006, The State-of-Health Server***.

Long experience has shown that many or most servers that go into a real-world control system function as device drivers. That is to say, they provide the software interface to some piece of hardware like a pressure sensor, a temperature controller, a gas concentration analyzer, etc. One task that has been generally useful on all control systems implemented using our client/server architecture is state-of-health (SoH) monitoring.

One design goal for our real-world control systems is to provide a generic way to log all sensor data to disk at periodic intervals. The trouble is, different control systems have different sets of hardware for which SoH information is desired. For a generic SoH monitoring task to exist, all sensors (that is to say, the servers that are responsible for the sensors) need to be able to report their sensor information in a consistent way. One way to guarantee this consistency is to make it a part of the `BaseServer` class, so all real-world servers inherit the same SoH reporting mechanism.

The `BaseServer` SoH mechanism expects all reporting sensors to be identifiable by a unique (system-wide), human-readable, short text name, and each sensor to provide exactly one value. For example, a temperature sensor might be called “heat1” and its one piece of reported information would be its temperature, say 37.4 degrees Celsius. Sensors that report multiple values per sensor are handled with pseudo-sensors. For example, a pressure sensor called “ps101” that reports both a pressure and a temperature, might provide two SoH records called “ps101.press” and “ps101.temp,” each of which reports only a single value.

For each reporting sensor (or pseudo-sensor), two pieces of information exist — an information record that describes the sensor but contains no sensor data, and a data record that contains a sensor’s value but no description. In the code, the information record is defined by the structure `SOH_ParamInfo` and the data record by `SOH_Param`. Both structures are defined in *cliserv.h* and are reproduced below.

```
struct SOH_ParamInfo {
   string  name;
   string  units;
   bool    analog;
   string Serialize();
   void   Deserialize(const string s);
};
typedef map<string, SOH_ParamInfo> SOH_ParamInfoMap;
```

```
struct SOH_Param {
   string  name;
   double  value;

   string Serialize();
   void   Deserialize(const string s);
};
typedef map<string, SOH_Param> SOH_ParamMap;
```

Access to this information comes in the form of three public methods of the `BaseServer` class, which are described below. See the method implementations in *cliserv.cpp* for complete details.

```
virtual void GetNumSohParams();
virtual void GetSohParamInfo();
virtual void GetSohParams();
```

The first thing to note is that all of these methods are virtual. That is because these methods, as implemented in `BaseServer`, do nothing. In that sense, these three methods serve more as a template for server authors than useful functions. Authors of servers that need to provide SoH information must override these three methods with their own versions appropriate to the specific server being written.

Also note that these three methods are client-request message handlers. They take no parameters because, when they run, client message handlers expect to have the complete details of the client request message available in the `BaseServer` internal class variables `clientName`, `command`, and `payload` (see section 4.4.2). They also provide no return value because client message handlers report success or failure directly back to clients via the `ReplyMessage()` or `ReplyError()` methods. They do not report results back to the client message dispatcher that calls them (see section 4.4.3).

5.1 The `GetNumSohParams()` Method

Authors of real-world servers that override this method are expected to return the number of reporting SoH sensors (including pseudo-sensors) the server maintains. The response message should be formatted as follows.

```
(ok,(<num>))
```

Here `<num>` is the number of reporting SoH sensors the server maintains.

The version of `GetNumSohParams()` implemented in the `BaseServer` class always returns zero (0).

5.2 The `GetSohParamInfo()` Method

Authors of real-world servers that override this method are expected to return to the client a series of one or more sensor (or pseudo-sensor) information records (`SOH_ParamInfo` structs) as described above. The records must be DSP-encoded using the following format:

```
(<rec>,<rec>,...,<rec>)
```

Here each `<rec>` represents a DSP-encoded version of one `SOH_ParamInfo` struct. The number of `<rec>` entries in the returned list should match the number returned by the `GetNumSohParams()` method. Rather than building the DSP representations of the individual `<rec>` entries manually, the server author should call the `SOH_ParamInfo` struct's `Serialize()` method, which will return a proper DSP encoding of the struct.

The version of `GetSohParamInfo()` implemented in the `BaseServer` class always returns an empty list.

5.3 The `GetSohParams()` Method

Authors of real-world servers that override this method are expected to return to the client a series of one or more sensor (or pseudo-sensor) data records (`SOH_Param` structs) as described above. The records must be DSP-encoded using the following format.

```
(<rec>,<rec>,...,<rec>)
```

Here each `<rec>` represents a DSP-encoded version of one `SOH_Param` struct. The number of `<rec>` entries in the returned list should match the number returned by the `GetNumSohParams()` method. Rather than building the DSP representations of the individual `<rec>` entries manually, the server author should call the `SOH_ParamInfo` struct's `Serialize()` method, which will return a proper DSP encoding of the struct.

The version of `GetSohParams()` implemented in the `BaseServer` class always returns an empty list.

6 Standard Server Messages

All servers derived from the `BaseServer` class (that is to say, all real-world servers) support a standard subset of commands in addition to any server-specific commands they may also offer. All server commands, including the standard commands, are represented by short text strings assigned to symbolic constants. The constants are used throughout the code (as opposed to the bare strings) and we will use only the constants in this document as well.

The symbolic constants for the standard server commands are defined near the top of *cliserv.h* with the definitions reproduced below:

```
const string CS_PING                = "basePing";
const string CS_GET_SERVER_STATS    = "baseGetServerStats";
const string CS_GET_NUM_SOH_PARAMS = "baseGetNumSohParams";
const string CS_GET_SOH_PARAM_INFO = "baseGetSohParamInfo";
const string CS_GET_SOH_PARAMS      = "baseGetSohParams";
const string CS_SHUTDOWN            = "baseShutdown";
```

The `BaseServer` class provides message handlers for all of these messages, so all real-world servers will automatically be able to respond to at least this set of commands without the server author having to handle them locally. However, most of the `BaseServer` client message handler methods are *virtual*, so server authors do always have the option of overriding the default response for these commands if circumstances warrant.

6.1 The `CS_PING` Command

The `CS_PING` command is a request from a client simply to see if the server is responsive. Handling of this command is so simple, that the `BaseServer` class does not even provide an explicit message handler for it. Instead, it is handled directly out of the client message dispatcher (see section 4.4.3) with a call to `ReplyMessage()`.

6.2 The `CS_GET_SERVER_STATS` Command

This command is serviced by the `GetServerStats()` message handler. It provides a way for clients to request the current set of statistics counters maintained internally by the server (see section 4.7 for details).

6.3 The `CS_GET_NUM_SOH_PARAMS` Command

This command is serviced by the `GetNumSohParams()` message handler.

For full details on this and the other state-of-health oriented commands, see section 5.

6.4 The CS_GET_SOH_PARAM_INFO Command

This command is serviced by the `GetSohParamsInfo()` message handler.

For full details on this and the other state-of-health oriented commands, see section 5.

6.5 The CS_GET_SOH_PARAMS Command

This command is serviced by the `GetSohParams()` message handler.

For full details on this and the other state-of-health oriented commands, see section 5.

6.6 The CS_SHUTDOWN Command

This command simply tells the server to shut itself down in a clean manner. It is handled directly out of the client message dispatcher (see 4.4.3), so it has no corresponding message handler.

7 The `BaseClient` Class

In the same way that all real-world servers are derived from the `BaseServer` class, all real-world client interfaces are derived from the `BaseClient` class. The `BaseClient` class exists for a few reasons:

- It automatically manages the communication link to the server. That is, the `Base·Client` class (and it's derived classes) can initially establish a connection to the server (if it exists), and can automatically re-establish the connection if the server is shut down and restarted.
- It provides a standard mechanism (the `SendMessage()` method) for sending messages to the server and receiving responses back from the server
- It provides one API method for each of the standard server messages (see section 6). This frees the user of the class from having to worry about how requests and response messages are encoded or transmitted. Real-world client interface classes derived from `BaseClient` are expected to continue this paradigm, providing one API method for each server-specific message supported by the corresponding server.

7.1 Client Thread Safety

Before we begin an in-depth examination of the `BaseClient` class, we should take a closer look at the need for thread safety. As previously discussed (see section 4.4.6), servers are always multi-threaded applications, so the `BaseServer` class needs a mechanism to ensure that various structures shared by those threads (namely the message queue and the server statistics structures) are protected from simultaneous access. For that, we use a POSIX standard *pthread* library mutex.

Although client interface classes are not by themselves multi-threaded, the applications that use them frequently are. Theoretically, each multi-threaded application that uses a `BaseClient`-derived client interface could implement their own mutex(es) to ensure a single client interface is used by only a single thread at a time. In fact, multi-threaded client applications are actually pretty common, but thread protection is tricky to do right, and it is easy to get wrong. Getting it wrong can mean deadlocks and/or highly intermittent (indeed seemingly random) data corruption that is very difficult to diagnose and correct. To relieve the application writer of this burden, all `BaseClient`-derived interface classes provide internal mutex protection of those portions of the interface class (primarily the communication link to the server) that could potentially fail if accessed simultaneously by more than one thread.

The specifics of the `BaseClient` mutex protection implementation as well as other design decisions that affect thread safety (primarily concerning the `SendMessage()` method) will be discussed in more detail later on.

7.2 `BaseClient` Class Method Details

A cleaned-up version of the `BaseClient` class definition appears below, with detailed method descriptions following. In the code, the class is defined in *cliserv.h* and implemented in *cliserv.cpp*.

```
class BaseClient {
public:
   BaseClient(const string clientName, const string serverName);
   virtual ~BaseClient();
   string ServerExists();
   string GetServerStats(CS_ServerStats &stats);
   string ShutdownServer();
   string GetNumSohParams(WORD &num);
   string GetSohParamInfo(SOH_ParamInfoMap &infoMap);
   string GetSohParams(SOH_ParamMap &paramMap);

protected:
   string SendMessage(const string command, string &response, const string params = "()");

   string          clientName;
   string          serverName;
   int             sockId;

private:
   pthread_mutex_t mutex;
   bool            connected;

   virtual string  Connect();
};
```

7.2.1 The `BaseClient` Constructor

```
BaseClient(const string clientName, const string serverName);
```

The `BaseClient` constructor takes two parameters:

- `clientName` — An STL string containing a short text name by which the client will identify itself to the system (client names are used mostly for logging purposes).
- `serverName` — An STL string containing the short text name of the server with which the client will communicate. This string must match the name the server uses to register itself with the name resolver (see design document ***DES-0004*, *The Server Name Resolver*** for complete details).

The constructor does surprisingly little. It merely stores away the client and server names, marks the communication link to the server as disconnected, and initializes the mutex used to help ensure thread safety.

7.2.2 The `BaseClient` Destructor

```
virtual ~BaseClient();
```

The `BaseClient` destructor also does little. It only closes the communication link to the server (if it is open) and destroys the mutex.

7.2.3 The `connected` Class Variable

```
bool connected;
```

Internally, the `BaseClient` class maintains a private Boolean variable called `connected`, that serves as the class' best guess as to whether or not the class currently has a valid connection to the server. This is generally misunderstood, so we will clarify the point here.

If `connected` is `true`, that does not necessarily mean that the client/server connection is actually usable, it just means the connection was valid the last time it was used, and the assumption is that it is still valid. In this circumstance, the client will immediately try to send on the connection without trying to establish a connection first. Almost all of the time, this will be successful. However, if it is not successful, the client will set `connected` to `false` and then begin procedures to establish a new connection to the server. All `connected` gains us is a way to quickly bypass the connection process unless it's needed (which happens rarely).

7.2.4 The `Connect()` Method

```
virtual string Connect();
```

The `Connect()` method is a private method used internally to connect to the server. It takes no parameters and returns an STL string as its return value. If the returned string contains the text "ok," it indicates that the call to `Connect()` was successful. Otherwise the returned string contains a text description of why the call failed.

`Connect()` is actually just a helper function to the `SendMessage()` method (it is not called from anywhere else). Discussing `Connect()` now will serve as a segue into the discussion of `SendMessage()` to follow.

`Connect()` does several things. First, it contacts the server name resolver to obtain a current IP address and TCP port number on which the server can be contacted. Then, it checks the client's current concept of the state of the connection (that is the `connected` class variable discussed above). If the client thinks there is already a valid connection to the server, that connection is first closed as a precaution. Finally, `Connect()` attempts to establish a new TCP connection to the server. If all these steps succeed, then "ok" is returned, otherwise an appropriate error message is returned.

7.2.5 The `SendMessage()` Method

```
string SendMessage(const string command,
                   string &response,
                   const string params = "()");
```

`SendMessage()` is the method that the `BaseClient` class (and all derived classes) use in its API wrappers to send requests to servers and receive the resulting responses. It also is responsible for managing the communication link to the server. `SendMessage()` takes three parameters:

- `command` — The name of the command the client is asking the server to perform. In the case of the standard server commands, these are always one of the symbolic constants defined in *cliserv.h* and discussed in section 6. In practice, client authors that derive from `BaseClient()` are expected to maintain this paradigm, so the `command` parameter should always be a symbolic constant for a command string.
- `response` — On successful return of the call to `SendMessage()`, this parameter will contain the payload portion of the response from the server (that is to say, any additional parameters that are part of the server's response beyond the simple "success" or "failure" indication). For server responses that contain no additional parameters, the `response` parameter will just contain the DSP empty message, "()".

- `params` — This parameter is a DSP-encoded message containing any additional parameters (beyond the name of the command) that need to accompany the request to the server. Having this parameter come third, rather than grouping it with the `command` parameter, seems a little out of order. It's done this way so that the `params` parameter can be defaulted to the DSP empty message. This is a convenient addition, because a great number of server commands do not take any additional information, in which case the `params` parameter can be left out entirely.

`SendMessage()` returns an STL string as its return value. If this string is the text "ok," it indicates that the command was successfully delivered to the server and the server processed the command and returned a success indication. In this circumstance, the `response` parameter is valid and will contain any additional parameters that are part of the server's response. If the return value is not "ok," then it is a text description of the failure. In this circumstance, the `response` parameter is not valid and cannot be depended upon to contain anything useful.

With each call to `SendMessage`, several things happen:

1. The various components that go into a server request message (client name, command name, optional parameters), are bundled into a DSP encoded message as described in section 4.1.3
2. The request is now ready to be transmitted to the server. It turns out that the sole shared resource that needs to be protected against simultaneous access between multiple threads communicating with a server through a single client interface is the communication link, so, before transmitting the message, the client class' mutex is locked.
3. The `BaseClient connected` class variable is examined. If there is no current connection to server in place, one is established using the `Connect()` helper method described previously.
4. The message is transmitted to the server, and the reply from the server is read
5. In the event of a failure in either the transmission of the request or receipt of the response, the communication link is torn down and re-established (using the `Connect()` method), and a second attempt is made to exchange messages with the server
6. The mutex that protects the communication link is released
7. A top-level DSP parse is performed on the response. If the server returned a success indication, then any additional parameters present in the response are placed in the `response` parameter and `SendMessage()` exits with a return value of "ok." Otherwise, the text description of the error returned by the server becomes the `SendMessage()` return value, and `response` is undefined.

Of course, the above sequence mostly describes the success path. Failures along the way (the server name resolver has no entry for the server, a TCP link can not be established, the message exchange fails, etc.) will ultimately result in an appropriate error message being returned as the method's return value.

At this point, more should be said about thread safety, because all of the features implemented to provide thread safety are implemented here in the `SendMessage()` method. The first of these features is the mutex used to protect the communication link. As previously stated, the communication link is the only shared resource that needs protection. The `SendMessage()` method is the only place in the client code where the communication link is dealt with, so it is also the only place where the mutex needs to be locked or unlocked. [3]

The communication link is the only resource that needs protection because `SendMessage()` has been carefully designed such that other bits of information that might normally be points of contention between threads are located on the call stack. This is important. Because each thread has its own call stack, as long as data items used by `SendMessage()` exist as local variables, function parameters or return values, each thread will get its own copy of the data items and simultaneous calls will not conflict. It is for this reason that `Connect()`'s interface to the server name resolver is a local variable. This is also the reason that error messages reported back by `SendMessage()` are reported back by way of the method's return value. Finally, having the server response data returned through the `response` parameter, guarantees that each thread must provide its own storage for the response.

7.2.6 User API Methods

The remaining `BaseClient` class methods are API wrappers around the various standard server messages described in section 6. These API methods are what users of the class would use to communicate with the server. There is one method for every client request message the server supports.

Because the standard messages have already been covered in section 6, not much will be said about the individual methods other than what messages they correspond to. However, it should be noted that all of these methods (and similar methods implemented by the authors of real-world client interfaces) do have certain things in common.

- Their main purpose is to hide the details of the underlying message encoding and message passing mechanisms from the user. Users should not care (or necessarily even know) that the input parameters to the method call are being bundled into a DSP-compliant string and transmitted over a TCP connection to be processed.
- All API methods use the `SendMessage()` method to communicate with the server. Therefore, authors of client interfaces (but not users of client interfaces) *do* need to be familiar with the DSP format and be able to build and parse DSP format messages.
- All API methods return STL strings that either hold the text “ok” (the call was successful) or a text description of the problem (if the call was unsuccessful). The returned string is exactly the string returned by the internal call to `SendMessage()`.

[3]Well, actually, the communication link is also manipulated by the `Connect()` method, but `Connect()` is only ever called from within `SendMessage()`, and it relies on `SendMessage()`'s manipulation of the mutex to ensure exclusive access to the link.

- When server responses contain additional information beyond the server's success or failure indication, that information is parsed out of its DSP format container and placed into the method's output parameters.

Short descriptions of the individual `BaseClass` API methods appear below.

```
string ServerExists();
```

The `ServerExists()` method corresponds to the `CS_PING` standard server message. It takes no parameters. A success response indicates that the server is running and responsive.

```
string GetServerStats(CS_ServerStats &stats);
```

The `GetServerStats()` method corresponds to the `CS_GET_SERVER_STATS` standard server message. It has one output parameter. On successful return, the output parameter will contain a copy of the server's internal statistics counters (see section 4.7 for more details).

```
string ShutdownServer();
```

The `ShutdownServer()` method corresponds to the `CS_SHUTDOWN` standard server message. It takes no parameters. A success response indicates that the server acknowledges the request to shutdown and has done so (or at least is in the process of doing so).

```
string GetNumSohParams(WORD &num);
```

The `GetNumSohParams()` method corresponds to the `CS_GET_NUM_SOH_PARAMS` standard server message. It has one output parameter. On successful return, the parameter will contain the number of items the server is configured to report as state-of-health information (this is always zero, unless the server author has explicitly overridden the default behavior of the server).

```
string GetSohParamInfo(SOH_ParamInfoMap &infoMap);
```

The `GetSohParamInfo()` method corresponds to the `CS_GET_SOH_PARAM_INFO` standard server message. It has one output parameter, which is a reference to an STL map of `SOH_ParamInfo` structures. On successful return, the map will contain one entry for each state-of-health item the server supports. Servers that do not override the default behavior will always return empty maps.

```
string GetSohParams(SOH_ParamMap &paramMap);
```

The `GetSohParams()` method corresponds to the `CS_GET_SOH_PARAMS` standard server message. It has one output parameter, which is a reference to an STL map of `SOH_Param` structures. On successful return, the map will contain one entry for each state-of-health item the server supports. Servers that do not override the default behavior will always return empty maps.

8 Message Parsing Helper Functions

Authors of real-world servers and/or real-world client interface classes are required to deal with DSP format messages fairly directly. In client classes, request messages are sent to servers by calling the `SendMessage()` method. The method's `params` and `response` parameters are both encoded as DSP format strings, and it is the author's responsibility to be able to build (in the case of the `params` parameter) or parse (in the case of the `response` parameter) these strings. Similarly, servers receive the parameter portion of client request messages by parsing a DSP format internal class variable called `payload`. Response messages pass additional parameters back to clients by calling the `ReplyMessage()` method, which takes as its one parameter, a DSP-encoded string holding the parameters to be returned. It is the author's responsibility to build that string.

Some helper mechanisms have been developed to make the task of encoding and parsing DSP strings more convenient for the author. The first of these is, of course, the `MessageParser` class, which implements the standard parser for DSP format strings (see design document ***DES-0002*, *Data Serialization Protocol*** for complete details). But there are other mechanisms as well. In many, many cases, the information that needs to be passed to a server or a client consists of just a single parameter. Authors can always explicitly construct the DSP encoding for a single parameter or explicitly use the `MessageParser` utility class to parse them. But for convenience, six helper functions have been written (three to encode, three to decode) for authors to use instead. These six functions are defined in *cliserv.h* and implemented in *cliserv.cpp*. The function definitions are reproduced below.

```
string SerializeInt(const long n);
string SerializeFloat(const double n);
string SerializeString(const string s);

long DeserializeInt(const string s);
double DeserializeFloat(const string s);
string DeserializeString(const string s);
```

Use of these is pretty self-explanatory. The top three functions take an integer, floating point, or string parameter respectively, and return as their return value a DSP-encoded string containing the value of the input parameter. For example, `SerializeInt(1234)` returns the string "(1234)." For string variables, the encoded string is also properly escaped with DSP special characters as required.

The bottom three functions perform the reverse operation. They take a DSP-encoded string containing a single parameter and return the integer, floating point, or string value contained within.

Passing single parameter messages back and forth between clients and servers is probably the most common case (after the zero parameter case, which requires no encoding/decoding on the author's part), and the above six functions make dealing with that case relatively painless. The next-most common situation is probably passing data structures back and forth. Throughout the code base, `struct` and `class` structures that get passed as part of client/server messages have two methods defined called `Serialize()` and `Deserialize()`. They are typically defined as follows:

```
string Serialize() const;
void   Deserialize(const string &s);
```

The `Serialize()` method encodes the contents of the data structure into a DSP format string that is returned. The `Deserialize()` method performs the opposite function, taking a DSP format string of the type created by `Serialize()` and parsing its contents into the fields of the data structure.

9 The `BaseServer` Test Server

The `BaseServer` class is primarily intended to be derived from in order to create real-world server classes. However, it does contain or support all the mechanisms necessary to function as a working server itself. Of course, a server that uses `BaseServer` directly isn't particularly useful, because it cannot do anything other than respond to the few standard server messages (see section 6) that all servers can respond to. Nevertheless, even that much makes building a server based only on `BaseServer` useful for development, testing and demonstration purposes.

The file *baseServer.cpp* in the code base does exactly that. Turning a server class into a stand-alone server is extremely easy. You simply create an instance of the server class and call its `Run()` method. The `main()` function from *baseServer.cpp* is shown below. This is the only code necessary to create the server.

```
#include "cliserv.h"
int main(const int argc, const char *argv[])
{
   BaseServer *server;

   server = new BaseServer(argv[1]);
   server->Run();
   delete server;

   return(0);
}
```

Here the parameter `argv[1]` is expected to be the name the server uses when registering with the name resolver.

Once the server is started, it can be communicated with using the test client described below.

10 The `BaseServer` Test Client

In the same way that the `BaseServer` class can be used as-is to create a working (albeit trivial) server, the `BaseClient` class can be used to create a working demonstration client. Building a useful client application from `BaseClient` takes more effort, though. Keep in mind, that client classes only provide the API methods to send/receive messages to/from the server. It is up to the test client application to actually do something useful with the API methods.

Our test client is implemented by the file *baseMenu.cpp*. The code implements a simple, text-based menu application that allows the user to send any of the standard server messages to the server and view the server's response. It is primarily useful during development and testing but it also serves as a useful demonstration of the client/server model.

The test client is expected to be started from a command-line terminal, and it takes as its one parameter the name of the server it should communicate with (this must be the same name that the server used when registering with the server name resolver). When run, the user is presented with the following menu. Each entry corresponds to a different standard server message (and therefore a different `BaseClient` API method).

```
General Server Items:
-1 - ping server
-2 - get server statistics
-3 - get number of SOH parameters
-4 - get SOH parameter information
-5 - get SOH parameters
99 - shutdown server

 0 - Exit Program

Enter Selection >
```

The test client application is not limited to talking to just the test server. In fact, because *all* servers are capable of responding to the standard server messages, the test client will work with *any* server.