The CDEV Generic Server

A CDEV Extension Library for Building Client/Server Systems

Version 1.5 December 9, 1996

Walt Akers, Chip Watson, Jie Chen

TJNAF - Thomas Jefferson National Accelerator Facility

DOCUMENT DATE: Table of contents generated: May 30, 2000 8:58 am

TRADEMARKS: UNIX is a registered trademark of AT&T in the USA and other countries.

VxWorks is a register trademark of Wind River Systems.

The X Window System is a trademark of Massachusetts Institute of Technology.

OSF/Motif and Motif are trademarks of Open Software Foundation, Inc.

Ultrix and DEC are registered trademarks of Digital Equipment Corporation.

HPUX is a registered trademark of Hewlett Packard.

SURA/CEBAF: The Southeastern Universities Research Association (SURA) operates the Continuous Electron Beam Accelerator Facility (CEBAF) for the United States Department of Energy under contract DE-AC05-84ER40150.

Table of Contents

List of Figures

1. Overview of the CDEV Generic Server Engine

Purpose of This Document

This document is designed to provide an overview and tutorial of how to implement client/server applications by using the CDEV Generic Server Engine. Adherence to the structure and syntax that is specified in this document will improve the likelihood that the CDEV service/server developer's application will be compatible with other similar applications using CDEV.

The class library was designed to be as efficient as possible and still maintain the flexibility to allow CDEV client/server developers to use it with minimal modification. In addition to describing the conceptual behavior of the server, this document will also discuss the C++ classes and how inheritance and overloading may be used to build the best server for your application.

Intended Audience

This document is intended for anyone who will be developing a CDEV server or will be developing CDEV applications that will communicate with one another over a network. This document will also be useful for software developers who wish to develop a non-CDEV application that can communicate with an existing server that uses this class library.

What is the CDEV Generic Server Engine

The CDEV Generic Server Engine is a collection of C++ classes that may be used to quickly develop a client/server application. The communications component of the library is based on the Adaptive Communications Environment (ACE), a freeware product developed by Douglas Schmidt that is provided with the CDEV distribution.

CDEV servers use a global CDEV Name Server (provided with the source code distribution) to register themselves. The client services can then use this Name Server to locate servers by type, name or host. The Name Server insures that each server name is unique within its type or domain. Servers that have not reregistered within a specific time period (usually 60 seconds) are automatically removed from the Name Server.

Clients and servers that are developed using this library will use the CDEV Linear Internet Protocol to communicate. The documentation for this protocol is provided with the CDEV distribution and its use ensures that the developer's server will be accessible by all CDEV compliant applications.

Why Use the CDEV Generic Server Engine

The CDEV Generic Server Engine provides a robust and reliable mechanism for quickly developing client/server applications. Because all of the network communications intricacies are isolated by the C++ classes, the developer's server can easily be modified and upgraded without significant modification to the network internals. Additionally, by using CDEV, the client does not need to be 'network-aware', the client C++ class library does all of the communications work.

The CDEV Generic Server Engine also provides a myriad of features that would require a significant investment in time to develop for each new server. These features are described in detail below.

Features

- The developer is only required to create a subclass of the cdevServer C++ class and overload a single method in order to generate the communications component of his server.

- The complete client communications portion of the application is accomplished by inheriting a CDEV service class from the cdevClientService C++ class and writing a boiler plate service loader.
- The engine uses the CDEV Linear Internet Protocol (CLIP) to communicate. This protocol uses cdevData objects (a self-describing data structure) to transfer data allowing unique, application specific data structures to be transferred without modifying the protocol.
- The client and server side of the application use a global CDEV Name Server to register and locate various servers by their type, name or host.
- The socket utility classes use embedded buffering to optimize asynchronous communications and increase communications speed.
- The communications interface is completely abstracted from the client application. Because the application has only a CDEV view of the world, the underlying communications engine can be modified or upgraded without breaking the program.
- Clients automatically reconnect to server following a disconnect or communications error.
- Communications integrity is ensured by using TCP/IP and the Adaptive Communications Environment (ACE) C++ library.
- The server supports multiple concurrent client connections. Because the inbound data is read incrementally and buffered, a slow client will not cause the server to block while waiting for a transmission to be completed.
- The server is completely event driven. It is activated whenever a client submits a packet or packets, otherwise, it sleeps until it has inbound data to process.
- An individual tag map is maintained for each connection. When data is received the server will map the contents of the clients cdevData objects to the server's representation prior to processing. The cdevData objects are remapped to the client's representation prior to returning results.
- The server has built-in mechanisms for storing, executing and removing client specified monitors on server data objects. This monitoring capability easily allows application developers to create event-driven client programs that respond to changes in the server.
- A timer-based CDEV 'polling' class is provided that allows the server to attach to other CDEV servers or services to obtain information.

2. Building the CDEV Generic Server Engine

Steps for Compiling and Testing the CDEV Generic Server Engine

1. **Install and compile the CDEV source code distribution**. See the CDEV distribution for specific instructions for compiling these libraries.
2. **Compile the Adaptive Communications Environment (ACE) Library.** ACE is located at the same level as the 'src' directory in the CDEV distribution tree. The README file located in that directory will provide specific instructions on building this library on your system. The ACE libraries should be automatically copied to the CDEV library directory.
3. **Setup the Makefile for your platform.** In the directory *include/makeinclude* there are a collection of makefiles that are followed by the name of the platform for which they were developed. Link the makefile associated with your platform to the file Makefile.OS by typing the following command: *"ln -s Makefile.XXXX Makefile.OS".*

   If a makefile for your platform does not already exist, you may have to create one in that directory.
4. **Compile the cdevGenericServer Library**. The source code tree for this distribution is located in the directory $CDEV/extensions/cdevGenericServer.

   The makefile for this library requires the same environment variables that are used by the main CDEV makefile.

   CDEV This is the root directory of the CDEV distribution.

   CDEVVERSION This is the version number of the CDEV class library.

   CDEVSHOBJ This is the directory for the CDEV shared objects.

   CDEVLIB This is the directory where the CDEV libraries reside.

   CDEVINCLUDE This is the directory where the CDEV include files reside.

   To compile the libraries, go to the cdevGenericServer directory and type make. All libraries and the associated test and example applications should be built.
5. **Run the test applications to ensure that the code is working correctly.** These applications are located in the test sub-directory of the cdevGenericServer tree. The test applications require that a special DDL file is specified and that the CDEV Name Server is operating. Perform the following steps to test the library.

   5a. **Start the Name Server.** The NameServer application is located in the bin directory of the cdevGenericServer distribution tree. The NameServer should produce no output and can be executed in the background by typing: *"NameServer &".*

   5b. **Specify the host name of the Name Server**. Because all applications will need to access the Name Server, the host where it is executing should be specified in the *CDEV_NAME_SERVER* environment variable. This variable must be specified in the environment of each shell that will need to access the Name Server. If the Name Server is running on host *foo.cebaf.gov*, the Name Server environment variable can be set by typing: *"setenv CDEV_NAME_SERVER foo.cebaf.gov".*

5c. **Specify the CDEV Device Definition Language file for the test programs.** The DDL file for the test programs is named *TestService.ddl* and is located in the test sub-directory of the cdevGenericServer distribution tree. In order for this DDL file to be used as the default, it should be specified in the *CDEVDDL* environment variable. This can be accomplished by moving to the test directory and typing the following command*: "setenv CDEVDDL $PWD/ TestService.ddl"*.

5d. **Specify the CDEVSHOBJ directory.** The CDEVSHOBJ directory is the directory that contains the versioned subdirectories for the service shared objects. By default the makefile will place the file TestService.so in the directory *$CDEV/lib/PLATFORM-OSVERSION.XX/1.5/TestService.so,* where PLATFORM is the name of your platform and OSVERSION is the major operating system. The following example shows the location of the TestService.so on a Solaris 5.5 system and the correct setting for the CDEVSHOBJ variable.

Location: *$CDEV/lib/solaris-5.XX/1.5/TestService.so*
CDEVSHOBJ: *$CDEV/lib/solaris-5.XX*

The CDEVSHOBJ variable may point to the directory that actually contains the service shared objects, however, CDEV will always attempt to locate the files in the version subdirectory first in order to support multiple CDEV versions.

5e. **Start the Test Server.** The environment is now correct to start the TestServer. From the bin sub-directory type the command: "*TestServer*". The TestServer should print a message indicating that it is ready to process user requests.

5f. **Specify the Client Tag Map.** In order to test all capabilities of the server, the client should use a tag map that is different from the one that is in use on the server side of the connection. A special tag map has been provided that can be used to test this feature. The tag map is located in the test sub-directory and can be specified by moving to the test subdirectory and typing: *"setenv CDEVTAGMAP $PWD/cdevTagMap.txt"*.

5g. **Start the Test Client.** In a new window, set the CDEV_NAME_SERVER and the CDEVDDL environment variables as previously described. The test client may then be started by typing the following command *"TestProgram"*.

5h. **Examine Test Server and Test Client output.** The server and the client should periodically print a line indicating that the packets that they are exchanging are correctly matched. If a mismatch occurs, both sides of the connection will print out a verbose description of what differences were detected.

5i. **Terminate the Test Server and the Test Client.** The TestServer and TestProgram applications are terminated using CTRL-C. When the applications are terminated, they should display a disconnecting message and exit gracefully.

3. The *Reflector* Server - A Simple Client/Server System

Overview

The Reflector client/server system is a simple CDEV service that returns the cdevData object unmodified to the caller. The Reflector server can be used as a skeleton for any other server that the developer may wish to create. The source code that is provided in the following sections is available in text form in the examples sub-directory of the cdevGenericServer directory tree. A more complex example is provided in section 11 of this document.

Reflector Server Source Code

The server for the Reflector system is instituted as a single C++ file. The source code for this application is listed below.

Figure 1: ReflectorServer.cc - Source Code for the Reflector Server

```
#include <cdevServer.h>

// ****************************************************************
// * class ReflectorServer:
// *   This is the server class for the reflector. It simply
// *   receives messages from a client and immediately returns them.
// *
// *   The constructor passes the domain, server, port and rate to the
// *   underlying cdevServer class to be processed. The cdevServer
// *   constructor will add this server to the Name Server and will
// *   begin processing messages when the cdevServer::runServer()
// *   method is executed.
// *
// *   The processMessages method is the servers interface to the
// *   world... Each time a complete message is received or the time
// *   specified in rate expires, that method will be called.
// ****************************************************************
class ReflectorServer : public cdevServer
{
public:
    ReflectorServer ( char *domain, char *server,
                      unsigned int port, double rate )
        : cdevServer(domain, server, port, rate)
        {
        }

    virtual void processMessages ( void )
        {
        cdevMessage * message;
        while(dequeue(message)==0)
            {
            enqueue(message);
            delete message;
            }
        }
};

void main()
{
ReflectorServer server("REFLECTOR", "TestServerX", 9120, 60);
cdevServer::runServer();
}
```

ReflectorServer Header Files

In the source code for the Reflector server, the only header file that must be included is the one for the cdevServer class. This header contains all of the definition information that is required for the Adaptive Communications Environment (ACE) and the CDEV Linear Internet Protocol.

The ReflectorServer Class

The ReflectorServer class inherits directly from the cdevServer class. Because cdevServer defines all of the functionality necessary to establish a listening socket and accept connections, the developers start-up is limited to initializing the cdevServer class object with the domain name, server name, listening socket number and the time-out rate.

The developer is required to create a processMessages method which will perform whatever message processing that is required of the server. In this case, the processMessages method will merely remove an entry from the queue and then re-enqueue it for return to the client. Note that the enqueue method does not delete the cdevMessage object, so it is the responsibility of the developer to delete the cdevMessage object when it is no longer needed.

The main Function

The main function is responsible for starting the server when the application is started. In order to perform this task, main must first create a new ReflectorServer object. The ReflectorServer in this example has the Name Server domain “REFLECTOR” and the server name “TestServerX”. It will be listening for connections on socket 9120 and will automatically process messages at least once every 60 seconds.

When the ReflectorServer was created it automatically registered itself with the ACE Reactor that is embedded in the cdevServer class. In order to begin accepting connections and processing messages the main function must call the static runServer method of the cdevServer class. This method will continue servicing requests until the static Finished flag of the cdevServer class is set to non-zero.

The ReflectorService Source Code

The ReflectorService is the CDEV interface to the ReflectorServer that is described above. The source code for the ReflectorServer is implemented as a single source file and its associated header file. The source code for the ReflectorService is as follows.

Figure 2: ReflectorService.h - Header File for the Reflector Service

```
#include <cdevClientService.h>

// ****************************************************************
// * Function called to create initial instance of ReflectorService
// ****************************************************************
extern “C” cdevService *newReflectorService ( char *, cdevSystem *);

// ****************************************************************
// * class ReflectorService :
// *   This class simply inherits from the cdevClientService and must
// *   define only a constructor and destructor.
// ****************************************************************
class ReflectorService : public cdevClientService
{
public:
    ReflectorService (  char * name, cdevSystem & system =
                        cdevSystem::defaultSystem());
protected:
    virtual ~ReflectorService ( void ) {};
};
```

Figure 3: ReflectorService.cc - Source Code for the Reflector Service

```
#include <ReflectorService.h>

// ****************************************************************
// * newReflectorService:
// *   This function will be called by the cdevSystem object to create
// *   an initial instance of the ReflectorService.
// ****************************************************************
extern "C" cdevService * newReflectorService
   (char * name, cdevSystem * system)
   {
   return new ReflectorService(name, *system);
   }

// ****************************************************************
// * ReflectorService::ReflectorService :
// *   This is the constructor for the ReflectorService.  It
// *   initializes the underlying cdevClientService by specifying
// *   that it is in the domain of REFLECTOR.
// ****************************************************************
ReflectorService::ReflectorService
   ( char * name, cdevSystem & system)
   : cdevClientService("REFLECTOR", name, system)
   {
   system.reportError(CDEV_SEVERITY_INFO, "ReflectorService", NULL,
            "Constructing a new ReflectorService");
   }
```

The ReflectorService.h Header File

While the server is not required to have a specific header file, a CDEV service must have a header file that may be used to create a loader for the archive version of the library. This file is always named xxxxxService.h, where xxxxx is the name of the service as it will be specified in the CDEV DDL file. This file must contain the complete class definition for the service class.

The newReflector-Service Function

Each service in CDEV must have a function that the cdevSystem object can call to create the initial instance of the object. In the case of the ReflectorService, this method will create a new instance of the ReflectorService using the provided name and cdevSystem object. This new object will then be returned as a pointer to a cdevService object.

The ReflectorService Class

Because of the simplicity of the ReflectorService, all of the functionality of this class is inherited from the cdevClientService class. The ReflectorService class is only required to initialize its parent classes to be fully operational.

The CDEV DDL File

After compiling this source code into a server application and a CDEV shared library using the makefile that is provided in the examples sub-directory, the developer is ready to generate a CDEV DDL file that will map certain device/message combinations to the Reflector service. The following simple CDEV DDL file can be used to map device “device1” and message “get attrib1” to the Reflector Service. Note that by entering the server tag in the service data section, the default server name that the message will be sent to may be specified. In this case all messages associated with the “attrib1” attribute will be sent to “TestServerX”.

Figure 4: Reflector.ddl - A Simple CDEV DDL File

```
service Reflector
   {
   tags {server}
   }

class Reflectors
   {
    verbs {get}
   attributes
      {
      attrib1     Reflector {server=TestServerX};
      }
   }

Reflectors :
   device1,
;
```

Testing the Reflector Server

After compiling the server and the service components of the Reflector system, the developer can test the functionality of the client/server application by performing the following steps.

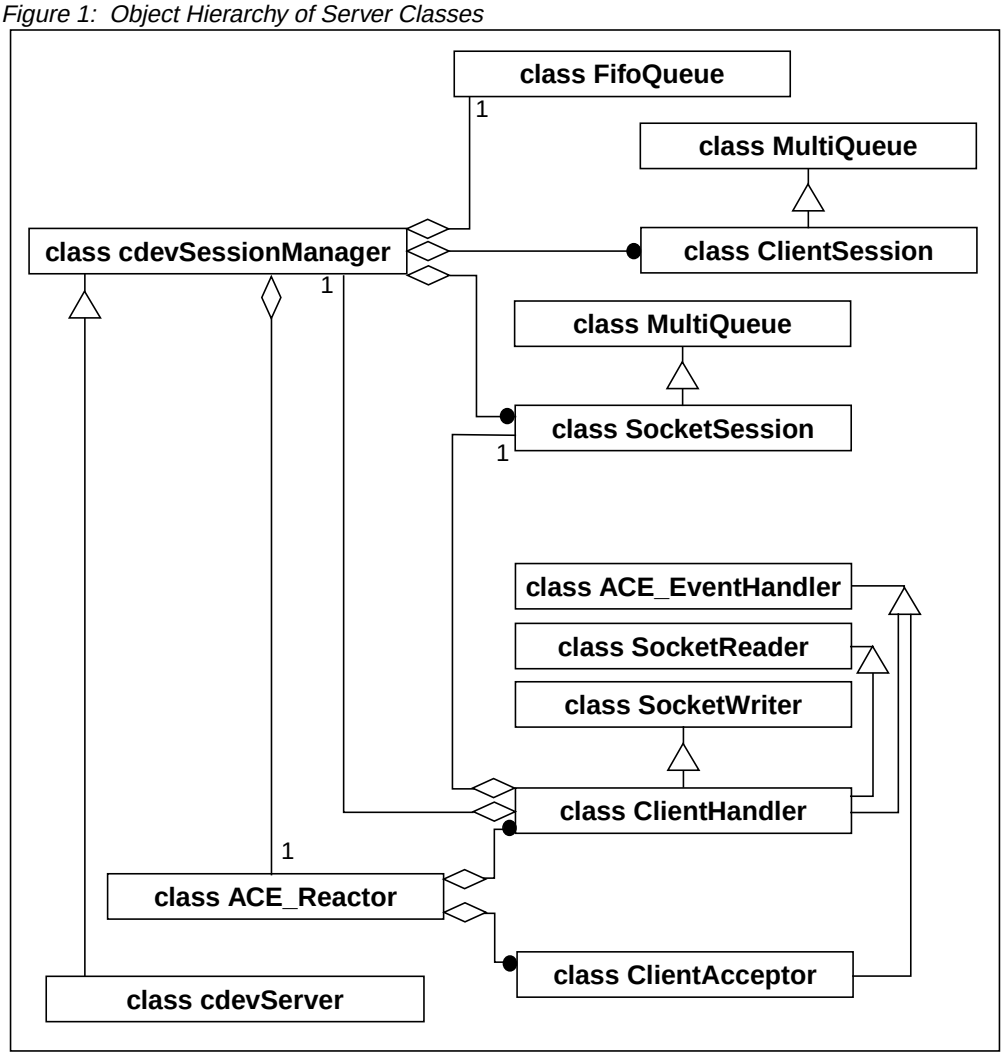
1. Start the Name Server.
2. Set the CDEV_NAME_SERVER environment variable in the shell where you will execute the server and the client to indicate the host where you started the Name Server. For instance: *setenv CDEV_NAME_SERVER cebaf1.cebaf.gov.*
3. Set the CDEVSHOBJ environment variable in the shell where you will start the client application to indicate the directory where the ReflectorService.so file is stored.
4. Set the CDEVDDL environment variable in the shell where you will start the client application to indicate the absolute path to the CDEV DDL file where the Reflector definitions have been created.
5. Start the server.
6. Start the cdevUtil application that is provided with the CDEV distribution and send messages to the server by typing: *device1 get attrib1.* You may want to enter additional output code in the processMessages method of the server to report each time a message is received.

4. Server Class Hierarchy

Server Classes

The server side of the CDEV Generic Server Engine library is composed of a series of classes that are used in conjunction with the ACE Reactor class to provide event driven socket management. The following diagram shows an object diagram of the classes that are used on the server side of the connection.

Figure 1: Object Hierarchy of Server Classes



FifoQueue Class

This class is a simple queue that is used to enqueue inbound messages that are received from a socket. Because all messages will be processed by the same method, there is only one FifoQueue object that is used by all client sockets. This object resides in the cdevSessionManager class.

MultiQueue Class

A MultiQueue object is a special type of FifoQueue that allows the caller to create an object that is to be placed into the queue and then place it into several queues at once. If the message is removed from any queue, it will automatically be removed from all other queues where it exists. This mechanism is used to provide the capability of removing all of a specific client's outbound packets without stepping through all of the packets in the associated socket's queue.

ClientSession Class

The ClientSession class inherits its queue functionality from the MultiQueue class. It is used to hold all outbound packets that are associated with a specific client identifier. The ClientSession object also holds supplemental data that the cdevSessionManager will need to manage the client. This class be subclassed by the developer in order to associate more data with the client identifier. The following information is stored in the ClientSession object:

Attributes of the ClientSession Class

localID *short localID;*

This is the client identifier that will be used on the server to uniquely identify the client.

clientID *int clientID;*

This is the client identifier that was provided by the client combined with the socket identifier.

socketID *int socketID;*

This is the socket identifier (file descriptor) to which the client is connected.

SocketSession Class

The SocketSession class inherits its functionality from the MultiQueue class. It is used to hold all packets that are destined for a specific socket. The SocketSession object is also used to store supplemental data that the cdevSessionManager will need to maintain the connection. This class be subclassed by the developer in order to associate more data with the socket identifier. The following information is stored in the SocketSession object:

Attributes of the SocketSession Class

socketID *int socketID;*

This is the socket identifier (file descriptor) to which the remote client is attached.

ClientAcceptor Class

The ClientAcceptor class is used by the ACE Reactor to listen to the server socket and accept each inbound client connection. When a connection is accepted, this class will create a ClientHandler object that will manage the connection throughout its lifetime.

SocketReader Class

The SocketReader class has the embedded mechanisms to read buffered packets from a socket. The ClientHandler inherits the functionality of this class to read data that is received on its associated socket.

SocketWriter Class

The SocketWriter class has the embedded mechanisms to write buffered packets to a socket. The class maintains a 56 kilobyte buffer that it uses to enqueue as many outbound messages as possible before executing a network write. The ClientHandler inherits the functionality of this class to write data to its associated socket.

ClientHandler Class

A ClientHandler object is created each time a new connection is accepted by the server. This class is used by the ACE Reactor to manage the input and output events on the specific socket. When data is received from the socket by the handle_input method, the ClientHandler object will enqueue the inbound packet in the FifoQueue provided by the cdevSessionManager class. When the cdevSessionManager class

enqueues outbound packets, the handle_output method of the ClientHandler class will write them to the socket using as many write operations as required to transmit all data.

When the ClientHandler object is destroyed it will notify the cdevSessionManager object which will remove its associated queues and will remove it from the ACE Reactor.

cdevSessionManager Class

The cdevSessionManager class maintains all of the queues, ClientSession and SocketSession objects that are used to operate a server. This class also defines the enqueue and dequeue methods that are used by the cdevServer to obtain inbound packets and to submit outbound packets.

cdevServer Class

The cdevServer class inherits the queue management functionality that is provided by the cdevSessionManager class and then implements the ClientAcceptor and ClientHandler classes to accept and process connections. The cdevServer class also introduces the concept of timed execution of the server function and automatic registration of the server with the CDEV Name Server.

In order to construct a new CDEV Server application, the developer only needs to inherit his server class from the cdevServer class and then define the *processMessages* method. This method will be called whenever the server timer expires or when data is ready to be processed in the inbound queue. Once called the *processMessages* method should use the *dequeue* method to remove the inbound cdevMessage object, process the message, and then use the *enqueue* method to return the processed cdevMessage object to the client.

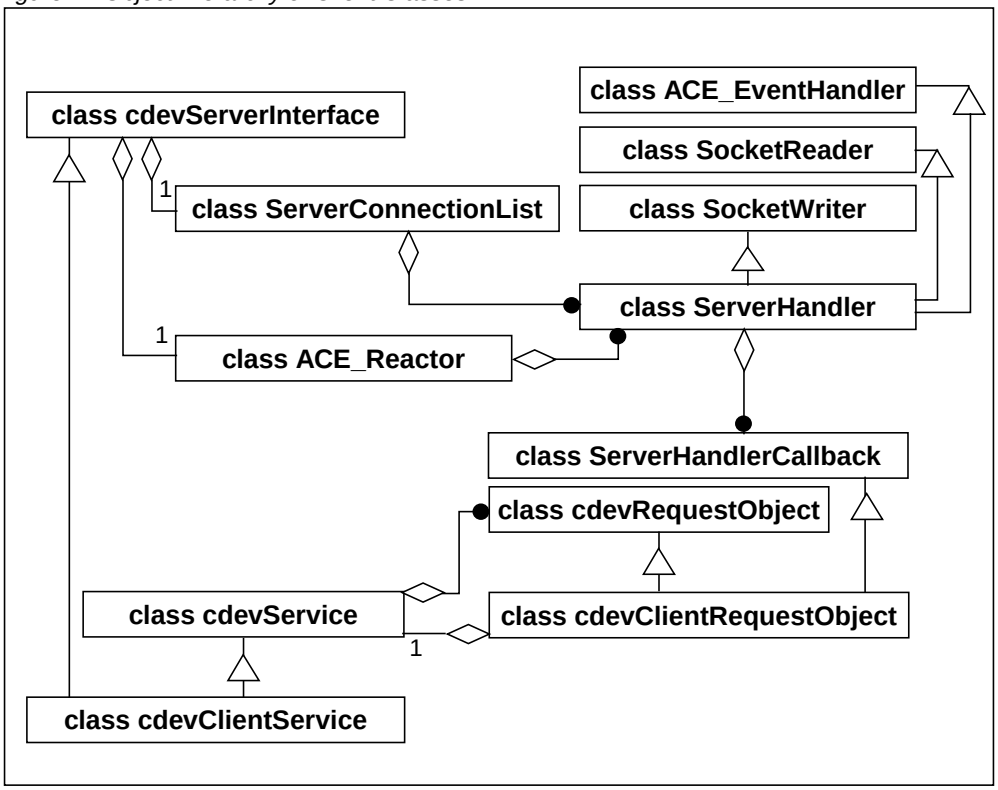
The encodePacket and decodePacket methods of this class are responsible for coordinating the tables of context objects, performing tag mapping and converting between the local client identifier and the foreign client identifier.

5. Client Class Hierarchy

Client Classes

The client side of the CDEV Generic Server Engine consists of a collection of classes that are used in conjunction with the CDEV service architecture and the ACE Reactor to provide pollable event driven behavior. The following diagram shows the object structure of the classes used by the client.

Figure 2: Object Hierarchy of Client Classes



SocketReader Class

The SocketReader class has the embedded mechanisms to read buffered packets from a socket. The ServerHandler inherits the functionality of this class to read data that is received on its associated socket.

SocketWriter Class

The SocketWriter class has the embedded mechanisms to write buffered packets to a socket. The class maintains a 56 kilobyte buffer that it uses to enqueue as many outbound messages as possible before executing a network write. The ServerHandler inherits the functionality of this class to write data to its associated socket.

ServerHandler Class

A ServerHandler object is created for each server that the cdevServerInterface wishes to communicate with. This object has an embedded FifoQueue object that is used to store the outbound packets until the ServerHandler's data is flushed. If the ServerHandler receives a sufficient number of packets or volume of data it will automatically flush its buffer to the socket. The ServerHandler inherits much of its communications functionality from the SocketReader and SocketWriter classes which provide buffered communications methods.

The cdevClientRequestObject may obtain a pointer to the ServerHandler that is associated with the server that it is communicating with. By referencing this pointer when enqueueing messages to the server through the cdevService, the request object can increase its performance significantly because it does not have to locate the associated ServerHandler on each transmission.

A cdevClientRequestObject that is utilizing a ServerHandler will register itself in a list of ServerHandlerCallback objects that are maintained in the ServerHandler object. When the ServerHandler is destroyed it will notify each ServerHandlerCallback object in the list to allow them to clear their pointers to it, avoiding the inadvertent use of an invalid object.

ServerHandler-Callback Class

The ServerHandlerCallback class is a virtual base class that any class that uses a ServerHandler object may use to detect when the object is destroyed. When an object that is inherited from this class is registered with the ServerHandler, it will be called prior to deleting the ServerHandler object. The inherited object should then set the associated ServerHandler pointer to NULL to prevent inadvertent access to a deleted object. The cdevClientRequestObject inherits from this class in order to support this functionality.

ServerConnection-List Class

The ServerConnectionList is a table of all ServerHandler objects that are currently connected to servers. The cdevServerInterface uses this table to locate a ServerHandler by the name of its associated server. This table prevents multiple connections from inadvertently being established to the same server.

cdevServerInter-face Class

This class has a ServerConnectionList that references all ServerHandlers that are connected to servers for the specific service. This object provides the mechanisms that are used to enqueue, dequeue, flush, poll and pend on the outbound connections.

The cdevServerInterface class contains an ACE Reactor that is used to respond to input/output events that occur on the sockets within the ServerHandlers. Because this Reactor is static within the cdevServerInterface class, it will handle events for all of the server connections in all of the services that are inherited from it when it is called.

cdevClientRe-questObject Class

This class inherits the majority of its functionality from, the cdevRequestObject class and is used to associate a device with a specific message. The identity of the server that supports the specific device/message combination may be specified in several ways: through the *server* tag in the context, through the CDEV DDL file, or by the server specified by a prior call to *set default*. If a specific server has been named, the cdevClientRequestObject will obtain a pointer to the associated ServerHandler object and will use that as a reference when submitting enqueue messages to the cdevClientService object. If no server has been specified, then the cdevClientRequestObject will set the ServerHandler pointer to NULL and will rely on the cdevClientService to use the default server. If no default server has been specified then the transmission will fail.

cdevClientService Class

This class inherits its functionality from the cdevServerInterface class. While the cdevServerInterface class deals exclusively with binary streams and their associated lengths, this class is called to enqueue information in the form of device, message, data and context. The class will encode this information into the appropriate binary format and submit it to the cdevServerInterface for enqueueing. When data is received

from the server, the cdevServerInterface will provide it to the cdevClientService in the form of a binary stream, and the service will decompose it into its CDEV components and dispatch the caller specified callback.

Messages are enqueued in the service in the form of cdevTranObjs. The cdevClientService manages these messages by maintaining a list of numbered transactions. The transaction number associated with a message is embedded in the outbound binary packet. When a server responds the same transaction number is embedded in the response packet. This transaction number is then used to locate and dispatch the associated callback. Once the callback has been executed, the transaction object is removed from the list and deleted.

In the event that a message indicates a monitoring operation, the transaction object is marked as permanent until a *monitorOff* operation is executed.

6. Properties of the cdevSessionManager Class

Overview

As described earlier the cdevSessionManager class is responsible for managing the queues that are used to read and write data to a collection of sockets that are communicating with the server. The public interface to the cdevSessionManager class is described below.

Attributes of the cdevSessionManager Class

Reactor

static ACE_Reactor Reactor;

This is the ACE Reactor object that is used to respond to input/output events on the individual sockets and to respond to time triggered events that are specified by the developer.

trigger

FD_Trigger trigger;

This class contains an embedded pipe that is used to trigger events in the object's ACE_Event_Handler. Each time a new packet is enqueued in the cdevSessionManager, a byte is written to the pipe within the trigger object, this causes the Reactor to call the processMessages method of the cdevSessionManager object to handle its input.

rate

ACE_Time_Value rate;

Because the cdevSessionManager's processMessages method may be called on a periodic basis rather than just when a new message has arrived, the rate variable contains the period in seconds between each subsequent execution.

localIdx

static IntHash localIdx;

This is a table of ClientSession objects that are hashed based on the local client index. The local client index is short integer that is unique within the server and is used to identify a specific client that is communicating through a socket. The ClientSession object is the queue that is used to store outbound packets that are destined for a specific client. Additional client specific information is also stored in the ClientSession object.

inbound

FifoQueue inbound;

This is the queue into which all inbound messages that are destined for a specific server are placed. A message is enqueued as a binary stream and its associated length.

clients

IntHash clients;

This is the table of ClientSession objects that are indexed by the client specified client identifier. While the localIdx table is global in scope, this table is specific to this instance of the cdevSessionManager class. The ClientSession object that is referenced by a specific client identifier is used to enqueue messages that are destined for that client ID.

sockets *IntHash sockets;*

This is the table of SocketSession objects that are currently in use by this instance of the cdevSessionManager class. The SocketSession that is referenced by the socket identifier is used to enqueue messages that are destined for that client. The ClientHandler object dequeues messages from the queue and writes them to the socket using multiple writes if necessary.

Methods of the cdevSessionManager Class

getNextLocalID *static short getNextLocalID (void);*

Because local indexes are short integers that monotonically increase, this method is used to obtain the next local index from the list.

newClientSession

```
ClientSession * newClientSession (   int SocketID,
                                     int ClientID,
                                     int LocalD);
```

This method is used by the cdevSessionManager whenever it needs to create a new ClientSession object. By overriding this method the developer may return subclassed ClientSession objects that contain additional, server-specific information that is associated with the client identifier.

newClientSession *SocketSession * newSocketSession (int SocketID);*

This method is used by the cdevSessionManager whenever it needs to create a new SocketSession object. By overriding this method the developer may return subclassed SocketSession objects that contain additional, server-specific information that is associated with the socket.

findLocalClient *virtual ClientSession * findLocalClient (short localID);*

This method is used to obtain a pointer to the ClientSession object that is associated with a local client identifier. Because the cdevServer object deals exclusively with the local client identifier, this method allows it to obtain the ClientSession without converting between the local and foreign client ID.

findClient *virtual ClientSession * findClient (int clientID);*

This method allows the caller to obtain a pointer to the ClientSession associated with a specific client identifier. This method uses the remotely specified client ID to locate the ClientSession object.

findSocket *virtual SocketSession * findSocket (int socketID);*

This method allows the caller to locate the SocketSession associated with a specific socket identifier. The socket identifier is the same as the file descriptor for a specific socket. The SocketSession object is the queue that is used by the ServerHandler object to dequeue messages that are destined for the client.

addClient *virtual ClientSession * addClient (int socketID, int clientID);*

This method is used to add a new ClientSession object to the clients list and obtain a pointer to the new object. If a ClientSession object already exists for the socketID/clientID combination, then a pointer to the existing ClientSession object will be returned. If a new ClientSession object is created, then this method will cause a “register” message to be generated and enqueued in the inbound queue for the cdevServer object.

addSocket *virtual SocketSession * addSocket (int socketID);*

This method is used to add a new SocketSession object to the sockets list and obtain a pointer to the new object. If a SocketSession object already exists for the specific socketID, then the existing SocketSession object will be returned.

removeClient *virtual void removeClient (int clientID, int unregisterFlag=1);*

This method removes the ClientSession object associated with a specific client identifier. If the unregister flag is non-zero, then this method will compose and send an “unregister” message to the inbound queue to alert the cdevServer object that the client (local client identifier) is being removed. The server should then remove any monitors that are associated with the local client identifier.

This method will remove and delete any outbound messages that are destined for the specified client identifier from the outbound queues.

removeSocket *virtual void removeSocket (int socketID);*

This method removes the SocketSession object associated with the specified socket identifier from the sockets list. When a socket is removed the ClientHandler associated with the socket identifier will be disconnected and deleted and then all ClientSession objects that are associated with the socket will also be deleted. This message will delete all outbound messages that are in the SocketSession queue or in any of its associated ClientSession queues.

enqueue *virtual int enqueue (int socketID, char * binary, unsigned len);*

This method is used by the ClientHandler object to enqueue messages that are destined for the processMessages method. The method will create a SocketSession for the ClientHandler if one does not already exist and will place the message into the inbound queue.

If the message is from a new client identifier, then a new ClientSession object is created.

enqueue *virtual int enqueue (cdevPacket * packet);*

This method is used to enqueue a packet that is to be returned to a client. The cdevPacket structure as described in the CLIP literature contains a client identifier that is used to determine to which socket the packet should be enqueued.

The enqueue method will call the encodePacket method in order to convert the packet into a cdevPacketBinary object to be enqueued. The developer may overload that method in order to perform any post-processing that may be necessary before converting the packet to binary format.

Note, this cdevPacket object remains the property of the caller and must be deleted when it is no longer needed.

dequeue *virtual int dequeue (cdevPacket * &packet);*

This method is used to dequeue messages that have been placed in the inbound queue. The processMessages method will then process the message and return the result using the enqueue method. Note that the inbound message contains a client identifier and other components that must be returned to the client unmodified.

The dequeue method will call the decodePacket in order to convert the binary cdevPacketBinary object into a cdevPacket object. The developer may override that method in order to perform any pre-processing that may be necessary before returning the packet.

Note, this cdevPacket object becomes the property of the caller and must be deleted when it is no longer needed.

decodePacket *virtual cdevPacket * dequeue (cdevPacketBinary * binary);*

This method converts a cdevPacketBinary object (as stored in the inbound queue) into a cdevPacket object. The developer may override this method in order to perform any special pre-processing that may be necessary prior to returning the cdevPacket object via the dequeue method.

encodePacket *virtual cdevPacketBinary * enqueue (cdevPacket * packet);*

This method converts a cdevPacket object into a cdevPacketBinary object (for submission to the client/socket queue). The developer may override this method in order to perform any special post-processing that may be necessary.

get_handle *virtual int get_handle (void) const;*

This method returns the file descriptor that is used by the FD_Trigger object. This method is called by the ACE Reactor in order to obtain a file descriptor for polling.

handle_input *virtual int handle_input (ACE_HANDLE);*

This method is called by the ACE Reactor whenever the file descriptor in the FD_Trigger object has a read event ready. This method will, inturn, call the processMessages method to dequeue the message from the inbound queue and process it.

handle_close *virtual int handle_close (int, ACE_Reactor_Mask);*

This method is called by the ACE Reactor to close the connections associated with this object.

handle_timeout *virtual int handle_timeout (const ACE_Time_Value&, const void *);*

This method is called by the ACE Reactor when the period specified by the Rate parameter has expired. This method will, inturn, call the processMessages method to handle events.

set_rate *virtual void set_rate (double d);*

This method is used to alter the rate at which the cdevSessionManager is periodically triggered.

get_rate *virtual ACE_Time_Value& get_rate (void);*

This method will return the rate at which the cdevSessionManager object is periodically triggered.

processMessages *virtual void processMessages (void);*

This is a user defined method that dequeues messages, processes them and then enqueues the result.

7. Properties of the cdevServer Class

Overview

The cdevServer class is the developer's primary interface to the server side of the CDEV Generic Server Engine. In most cases the developer will only be required to overload the processMessages method with his own method that dequeues a message, processes it, and then enqueues the result. The cdevServer class has the following properties.

Attributes of the cdevServer Class

Finished

static sig_atomic_t Finished;

This flag is used to indicate that the server should shutdown all connections and exit. This is a public flag that may be set by signal handlers or by the developer.

serverName

*char * serverName;*

This is the name of the server that was specified when the cdevServer object was created. This name will be used to identify the server when reporting errors or events.

acceptor

*class ClientAcceptor * acceptor;*

This is the ClientAcceptor class that will be used to listen for incoming connections on the specified listening socket.

timer

*cdevNameServerTimer * timer;*

This is a timer object that will be registered with the ACE Reactor and will reregister the service with the Name Server periodically. If a server does not update its registration with the Name Server at least once per minute, the Name Server will remove its name from its list of available servers.

status

int status;

The status variable is set to 0 if the ClientAcceptor was successfully opened to listen for incoming connections, or -1 if an error occurred while posting the listening socket.

Methods of the cdevServer Class

cdevServer

*cdevServer (char * Domain, char * Server,*
unsigned short Port, double Rate);

This method is the constructor for the cdevServer class. It will register the server with the CDEV Name Server using the specified Domain and Server names. It will then post a listening socket using a ClientAcceptor object on the specified Port. The ACE Reactor will use the rate parameter to establish the frequency in seconds in which the processMessages method should be called.

newClientSession

*ClientSession * newClientSession (int SocketID, int ClientID, int LocalD);*

This method overrides the cdevSessionManager's newClientSession method and returns a *CLIPClientSession* object. The CLIPClientSession allows the cdevServer object to associate the most recent CDEV context received with an individual client identifier.

newSocketSession

*SocketSession * newSocketSession (int SocketID);*

This method overrides the cdevSessionManager's newSocketSession method and returns a *CLIPSocketSession* object. The CLIPSocketSession object allows the cdevServer object to associate a cdevTagMap object and a cdevContextMap object with each socket identifier.

dequeue

*int dequeue (cdevMessage * &message);*

Because the cdevServer deals only with the cdevMessage type cdevPackets, this method will dispose of any other packet type that is received and will return the next available cdevMessage object from the inbound queue.

decodePacket

*cdevPacket * decodePacket (cdevPacketBinary * input);*
*cdevPacket * decodePacket(cdevMessage * message);*

These methods allow the cdevServer class to perform special processing whenever a cdevMessage object is dequeued.

encodePacket

*cdevPacketBinary * encodePacket (cdevPacket * input);*
*cdevPacketBinary * encodePacket(cdevMessage * message);*

These methods allow the cdevServer class to perform special processing whenever a cdevMessage object is enqueued.

operational

virtual int operational (void);

This method returns a boolean integer indicating whether or not the listening socket has been posted. If the return value is zero, then the cdevServer cannot receive new connections.

8. Properties of the cdevServerInterface Class

Overview

The cdevServerInterface class is responsible for managing all server connections for a specific service on the client side of the connection. In most cases the developer will not need to modify any of the code associated with this class. The properties of the cdevServerInterface class are described below.

Attributes of the cdevServer Interface Class

Reactor *static ACE_Reactor Reactor;*

This is the ACE Reactor object that is used to respond to input/output events on the individual sockets. The developer is responsible for calling the poll or pend method periodically in order to force events to be handled.

connections *ServerConnectionList connections;*

This is a list of ServerHandler objects that are used to handle input/output events on all currently connected servers.

domain *char * domain;*

This is the name of the domain in which this cdevServerInterface class is operating. When a server name is specified, the cdevServerInterface object will poll the Name Server for the location of the specified server name within this domain.

defaultServer *char * defaultServer;*

This is the name of the defaultServer to which messages will be sent if no other server has been specified. The caller must specify the name of the default server by using the "*set default*" message with the server name in the "*value*" tag of the outbound cdevData object.

defaultServerHandler *ServerHandler * defaultServerHandler;*

This is the ServerHandler object for the default server. It is maintained separately from the ServerConnectionList in order to reduce lookup times when the default server is accessed.

maxFd *int maxFd;*

This is the allocated size of the fdList array.

fdList *int * fdList;*

This is a an array of integers that contains the file descriptors that are in use in the ServerHandler objects. The cdevSystem object will request this list of integers in order to execute the select system call to determine which file descriptors have waiting input events.

Methods of the cdevServer Interface Class

cdevServerInterface *cdevServerInterface (char * Domain);*

This is the constructor for the cdevServerInterface class. The Domain is the name of the domain that all servers must be registered with in the CDEV Name Server.

getDefault *virtual char * getDefault (void);*

This method returns the name of the default server.

getDomain *virtual char * getDomain (void);*

This method returns the name of the default Name Server domain.

setDefault *virtual void setDefault (char * Default);*

This method allows the caller to set the name of the default server. This method will cause the cdevServerInterface to attach to the specified server name within the default Name Server domain.

connect *virtual ServerHandler * connect (char * server);*

This method allows the caller to connect to a specified server within the default Name Server domain. Once connected, the ServerHandler associated with the connection will be returned to the caller. If the server is already connected, its current ServerHandler will be returned.

disconnect *virtual ServerHandler * disconnect (char * server);*

This method allows the caller to terminate a connection to the specified server within the default Name Server domain.

enqueue *int enqueue (ServerHandler *handler, char *binary, size_t len);*

This method enqueues a binary data stream in the outbound queue for the specified ServerHandler. When the ServerHandler has reached a high-water mark (or when the flush method is called), the data in the outbound queue will be written to the socket.

getFd *virtual int getFd (int * &fd, int &numFd);*

This method allows the cdevSystem object to get the file descriptors that are in use by the cdevServerInterface in order to poll for read events using the select system call.

flush *virtual int flush (void);*
int flush (int fd);

This method forces the data in the outbound queues for all ServerHandlers (or the specified ServerHandler if the fd parameter contains a file descriptor) to be flushed to their associated sockets. The system will wait for up to five seconds for all outbound data to be written.

pend

virtual int pend (double seconds, int fd = -1);

This method calls the handle_events method of the ACE Reactor to poll for inbound events. If data is ready on any of the supported sockets, then the ServerHandler objects will be called to handle the input.

9. Properties of the cdevClientService Class

Overview

The cdevClientService class is the class that the developer will directly inherit from in order to create his new service. This class inherits most of its functionality from the cdevServerInterface class and either inherits or contains all of the methods necessary for a CDEV Service. The properties of the cdevClientService class are described below.

Attributes of the cdevClientService Class

callback *cdevCallback callback;*

This is the callback object that is used to support sendNoBlock requests because the service implements all operations as sendCallback operations. In situations where no callback object has been specified, this one is used by default.

transactions *AddressIndex transactions;*

This is a table that contains a list of all active transaction objects. When a transaction is returned from the server, its associated transaction object is removed from this list and deleted.

contexts *cdevContextMap contexts;*

This is a table that contains a copy of all contexts that have been used by the service. This table is used to allow the cdevRequestObjects to maintain an integer identifier for their current context and to simplify detection of context changes to a specific socket.

tagCallback *cdevClientTagCallback tagCallback;*

This is a callback object that will be called each time a new tag is placed in the cdevGlobalTagTable. This callback causes a new copy of the cdevTagMap to be submitted to each server.

Methods of the cdevClientService Class

cdevClientService

```
cdevClientService ( char * domain, char * name,
                    cdevSystem & system =
                    cdevSystem::defaultSystem());
```

This is the constructor for the cdevClientService object. The domain parameter specifies the name of the Name Server domain where its servers will be found. The name parameter is the name of the service and the system parameter is a reference to the cdevSystem object that it will use.

defaultCallback

```
static void defaultCallback ( int, void *,
                              cdevRequestObject &,
                              cdevData &);
```

This is the default callback function that is used by the callback attribute for processing sendNoBlock messages.

outputError *virtual int outputError (int severity, char *name, char *formatString,...);*

This method is used by the class to display error and warning messages. This method calls the reportError method of the cdevSystem object.

flush *int flush (void);*

This method causes all messages that are waiting in the outbound queues to be flushed to their associated sockets.

pend *int pend (double seconds, int fd = -1);*

This method causes the service to pend for a specified number of seconds and wait for read events on its file descriptors.

poll *int poll (void);*

This method causes the service to pend for a very short period of time and wait for read events on its file descriptors.

pend *int pend (int fd = -1);*

This method causes the service to pend until the next read event occurs on one of its file descriptors.

getNameServer *int getNameServer (cdevDevice * &ns);*

This method is typically used to return a pointer to a service specific local Name Server device. Because this option is not currently supported by this service, the ns parameters is set to NULL and 0 is returned.

getRequestObject *int getRequestObject (char * device, char * message, cdevRequestObject * &req);*

This method is used by the cdevSystem object to obtain a specific cdevRequestObject associated with the specified device and message.

enqueue

*int enqueue (char * server, cdevData * in, cdevTranObj & xobj);*

*int enqueue (ServerHandler * handler, cdevData * in, cdevTranObj & xobj);*

These methods are called by the cdevClientRequestObject to enqueue messages to be sent to a specific server. The server may be specified either by the server name or by the associated ServerHandler object.

cancel *int cancel (cdevTranObj & xobj);*

This message is used to cancel a transaction that has already been sent. Since a transaction cannot be canceled once sent to the server, this method simply removes its transaction number from the list of transactions and deletes its transaction object.

enqueue

*int enqueue (int status, ServerHandler * handler,*
*char * binary, size_t binaryLen);*

This method is called by the cdevServerInterface to enqueue an inbound packet that has been received from a server. The status indicates whether the message was successfully sent and the ServerHandler indicates the server that the message was destined for. This method will call the fireCallback method to dispatch the message.

fireCallback

void fireCallback (int status, cdevTranObj &xobj,
*cdevData *resultData,*
int partialTransaction = 0);

This method will execute the callback method associated with the specified transaction object. If non-zero, the partialTransaction flag indicates that the request that is being serviced will generate multiple responses.

10. Properties of the cdevClientRequestObject Class

Overview

The cdevClientRequestObject is a cdevRequestObject class that has been optimized to operate with the cdevClientService class. The cdevClientRequestObject class has the following properties.

Attributes of the cdevClient RequestObject Class

sendStatus *SendStatus sendStatus;*

This is a structure that is used as the user argument to the default callback for the cdevClientRequestObject. Whenever a *send* method is executed, the request object can detect that the operation has completed by polling this value.

server *char server [256];*

This is the name of the server that the cdevClientRequestObject is currently connected to. This value is maintained in order to reestablish the connection if a communication error occurs.

DDL_server *char DDL_server[256];*

This is the server name that is specified in the CDEV DDL file as the default server for this device/message combination.

syncCallback *cdevCallback syncCallback;*

This is the callback object that is used to receive the callback when the *send* method is executed. The callback used by this method expects to receive a SendStatus structure as its user argument.

handler *ServerHandler * handler;*

This is the ServerHandler object for the server to which the cdevClientRequestObject is currently attached.

contextID *int contextID;*

This is the index of the current context from the cdevContextMap that is maintained in the service. This identifier is used during transmission to determine if the context has changed since the last transmission.

commandCode *int commandCode;*

This is an integer that holds an enumerated type identifying the command or verb that the cdevClientRequestObject supports. The standard verbs are “get”, “set”, “monitorOn”, and “monitorOff”.

messageCode *int messageCode;*

This is an integer that holds an enumerated type identifying the message that the cdevClientRequestObject supports. Typically only messages that are intrinisic to the service layer will be

enumerated here. The following messages are currently supported: “get servers”, “get default”, “set default”, and “disconnect”.

Methods for cdevClientRequestObject Class

constructor

*cdevClientRequestObject (char * device, char * message, cdevSystem & system = cdevSystem::defaultSystem());*

This is the constructor for the cdevClientRequestObject. It will check the CDEV DDL file to determine if a default server has been established for this device/message combination.

setContext

virtual int setContext (cdevData & ctx);

This method is used to set the context for the cdevClientRequestObject. The context may be used to establish the default server that the request object will communicate with if the server tag has been set. If the server tag is unspecified, then the server specified in the CDEV DDL file will be used. If no server has been specified in the CDEV DDL file, then the request object will rely on the cdevClientService to use the default server.

send

virtual int send (cdevData & in, cdevData & out);
*virtual int send (cdevData * in, cdevData & out);*
*virtual int send (cdevData & in, cdevData * out);*
*virtual int send (cdevData * in, cdevData * out);*

This method will synchronously transmit the device/message combination to the server. See the setContext method for details on how the target server is selected.

sendNoBlock

virtual int sendNoBlock (cdevData & in, cdevData & out);
*virtual int sendNoBlock (cdevData * in, cdevData & out);*
*virtual int sendNoBlock (cdevData & in, cdevData * out);*
*virtual int sendNoBlock (cdevData * in, cdevData * out);*

This method will asynchronously transmit the device/message combination to the server. The caller is required to poll the system in order to allow the transmission to be processed. See the setContext method for information on how the target server is selected.

sendCallback

virtual int sendCallback (cdevData & in, cdevCallback &);
*virtual int sendCallback (cdevData * in, cdevCallback &);*

This method will asynchronously transmit the device/message combination to the server and will call a developer specified callback function when the message has been processed. The caller is required to poll the system in order to provide time for the transmission to be processed. See the setContext method for information on how the target server is selected.

className *virtual const char * className (void) const;*

This method returns the name of the class.

defaultCallback *static void defaultCallback (int status, void * user, cdevRequestObject &, cdevData &);*

This is the callback function that will be executed when the synchronous send method has been used. It will set the value of the SendStatus structure that was passed as its user argument to indicate completion and the status of the call.

executeServerHandlerCallback
virtual void executeServerHandlerCallback (ServerHandler);*

This method is inherited from the ServerHandlerCallback method and will be called by the ServerHandler that is currently in use by this request object prior to its destruction. This allows the request object to set the pointer to NULL to avoid accessing an invalid or deleted data item later.

getServerHandler *virtual int getServerHandler (ServerHandler ** Handler);*

This method allows the caller to get a pointer to the ServerHandler that is in use by the cdevClientRequestObject. When called, this method will check the class parameters and attach or reattach to a server if necessary before returning the ServerHandler object.

getContextID *int getContextID (void);*

This method returns the context index that uniquely identifies the context that is in use within this cdevClientRequestObject. This value is used by the service to determine if the context data needs to be retransmitted to the server.

getCommandCode *int getCommandCode (void);*

This method returns the command code that identifies the verb portion of the message used by this request object. This code is used by the service to avoid having to perform string comparisons to identify the message content.

getMessageCode *int getMessageCode (void);*

This method returns the message code that identifies the message in use by this request object. This code is used by the service to identify messages that may be processed locally rather than being transmitted to the server.

11. Implementing Monitoring on the cdevServer

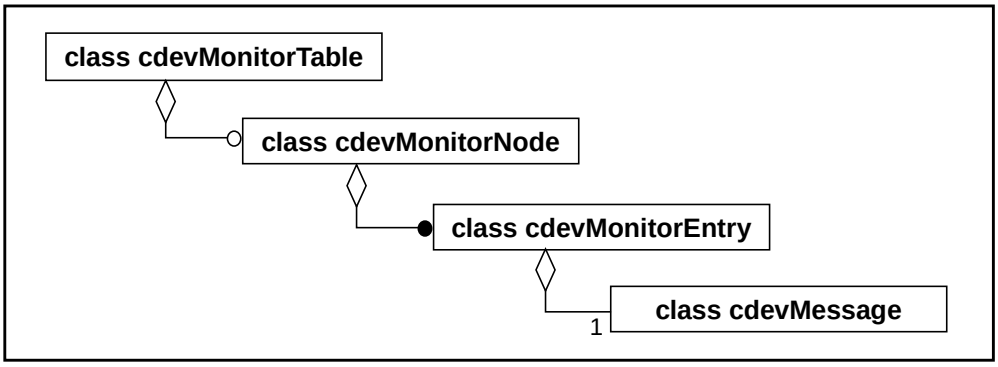
Overview

Monitoring is implemented on the cdevServer by the use of a ***cdevMonitorTable*** class. This class stores a collection of ***cdevMonitorNode*** objects that are represented by a device name and its associated attribute name. Within each of the cdevMonitorNode objects there is a list of ***cdevMonitorEntry*** objects that contain information regarding an individual monitor request - this information is stored in the form of the cdevMessage object that was used to place the request.

The cdevMonitorTable provides the methods to insert and remove monitors and to retrieve the cdevMonitorNode objects that are used by individual device/attribute pairs. By using the node directly to trigger monitors, the application can greatly speed the delivery of messages when a monitored value changes.

The following object model describes the general structure of the cdevMonitorTable.

Figure 3: General Structure of the cdevMonitorTable



Special Notes

Because a monitor generates many responses from the request, the service has to be able to differentiate it from the transaction that generate a single response. To accomodate this, the cdevMonitorTable uses the operationCode member of the cdevMessage structure to indicate that the result is one message of many messages that may be generated. If the first bit of the operationCode is non-zero, then the response is one of many. If the first bit of the operationCode is zero, then this is the last response that will be generated by the monitorOn request.

Attributes of the cdevMonitorTable Class

monitors

StringHash monitors;

This is a hash table that is hashed on a string representation of the device/attribute combination. Each hash entry points to the cdevMonitorNode for that specific hash string combination.

Methods of the cdevMonitorTable Class

insertMonitor

*int insertMonitor (cdevMessage * request, cdevData * data);*

*int insertMonitor (cdevMessage * request, cdevData ** data, size_t dataCnt);*

This method allows the caller to insert a new monitor for the device and attribute that are specified in the cdevMessage object. The data that is provided with the call contains the

current settings for all properties of the device/attribute combination and will be used to dispatch the initial callback. If the cdevMessage object contains a list of multiple devices, then the second insertMonitor method is used to provide an array of cdevData objects containing the settings for each device.

The cdevMessage object that is provided to this method becomes the property of the cdevMonitorTable and should not be accessed again by the caller.

removeMonitor *int removeMonitor (cdevMessage * request);*

This method allows the caller to remove a monitor that was previously installed using the insertMonitor method.

removeClientMonitors *int removeClientMonitors(short clientID);*

This method will remove all monitors that are associated with a specific client identifier. This method is typically employed to remove all of a client's monitors when it is disconnected.

findMonitor

```
cdevMonitorNode * findMonitor (     char * device,
                                    char * attrib);
```

This method is used to locate the collection of monitors that is associated with one device/attribute pair. An object that manages the values of that device/attribute pair may use this cdevMonitorNode to directly trigger monitors when one or more its properties are changed.

fireMonitor

```
int fireMonitor (     char * device, char * attrib,
                      char * property, cdevData * data);
int fireMonitor (     char * device, char * attrib,
                      int property, cdevData * data);
```

This method is used to trigger the callbacks for the monitors that are associated with the specified device/attribute pair. The property that has been changed is provided in the property parameter and a list of the current values of all properties should be provided in the data parameter. When this method is called it will locate all monitors that are triggered by the specified property and will use the context associated with the monitor to load the desired property values into the outbound data object.

fireCallback *int fireCallback (cdevMessage * message);*

This is a pure virtual method that the developer must provide. This method is called when the cdevMonitorTable has a monitor that needs to be dispatched. The data is provided to the method in the form of a cdevMessage object which must be dispatched to the client using the user provided mechanisms.

Attributes of the cdevMonitorNode Class	**parent**	*class cdevMonitorTable * parent;* This is a pointer to the cdevMonitorTable that contains this cdevMonitorNode. This pointer will be used to access the user defined fireCallback method when a monitor must be dispatched.
	node	*cdevMonitorEntry * nodes;* This is a list of all of the monitors that are associated with this cdevMonitorNode.
	hashString	*char * hashString;* This is a unique string that is composed of the device name and attribute that is used to identify this cdevMonitorNode within the list of all cdevMonitorNodes that a cdevMonitorTable may be managing.
Methods of the cdevMonitorNode Class	**fireMonitor**	*int fireMonitor (char * property, cdevData * data);* *int fireMonitor (int property, cdevData * data);* This method is called in order to trigger all monitors that are associated with the specified property. When the method is called it will walk through the cdevMonitorEntry objects and locate each one that is associated with the specified property. It will then evaluate the context for that entry and populate the outbound data with the appropriate properties before calling the fireCallback method of its parent cdevMonitorTable object.
	isMonitored	*int isMonitored (void);* This method returns a boolean value that indicates if there are any active monitors that have been placed on this cdevMonitorNode object.

12. VirtualService: A Complex Client/Server Implementation

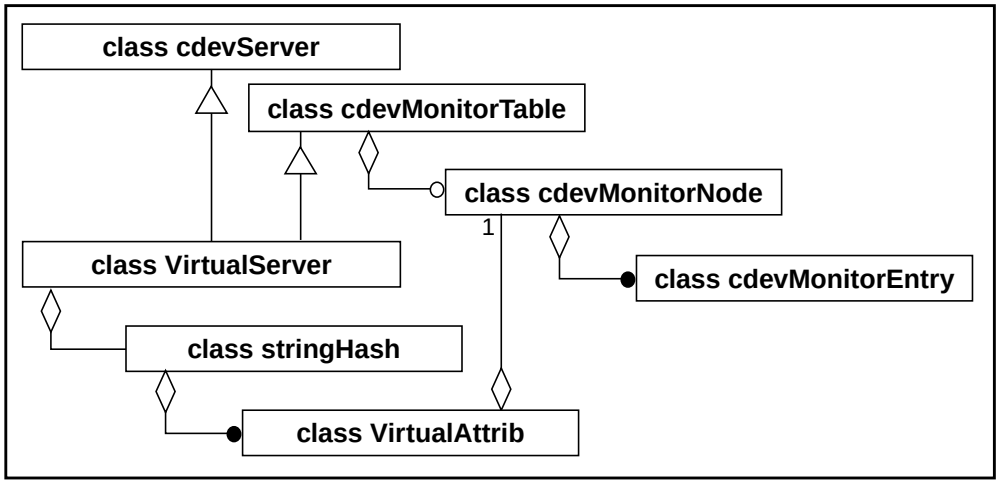
Overview

The VirtualService example gives the developer a complex illustration of how to create a client/server system that provides for getting, setting and monitoring specific properties of a virtual device/attribute pair. By examining the source code, the developer can also get an understanding of the different approaches used to return message completion codes to the client and how to establish and trigger monitors using the components that are provided with the distribution.

Virtual Server Structure

The server side of the VirtualService example is structured as described in the object model below.

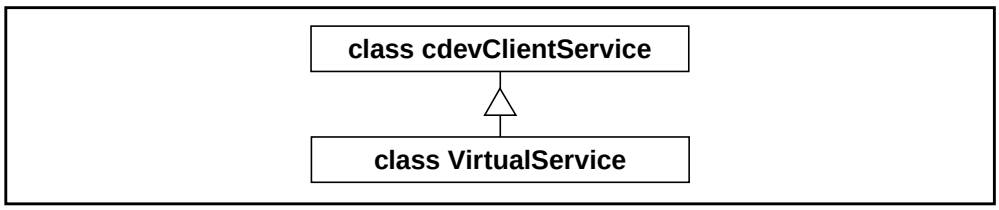
Figure 4: Components of the Virtual Server



Virtual Service Structure

The client side of the VirtualService example is much less complex and inherits almost all of its functionality from the cdevClientService class and declares no specialized request object class.

Figure 5: Components of the Virtual Service



VirtualAttrib.h

The following header file defines the structure of the VirtualAttrib class. A VirtualAttrib object is used to represent a single entity within the VirtualServer. Each VirtualAttrib is represented by a device name and an associated attribute. This device/attribute pair has a collection of properties that may be read, written to, or monitored. All of these properties are set using methods in order to allow the class to ensure that they are set within the specified range and to allow the monitor callbacks to be fired when a value changes.

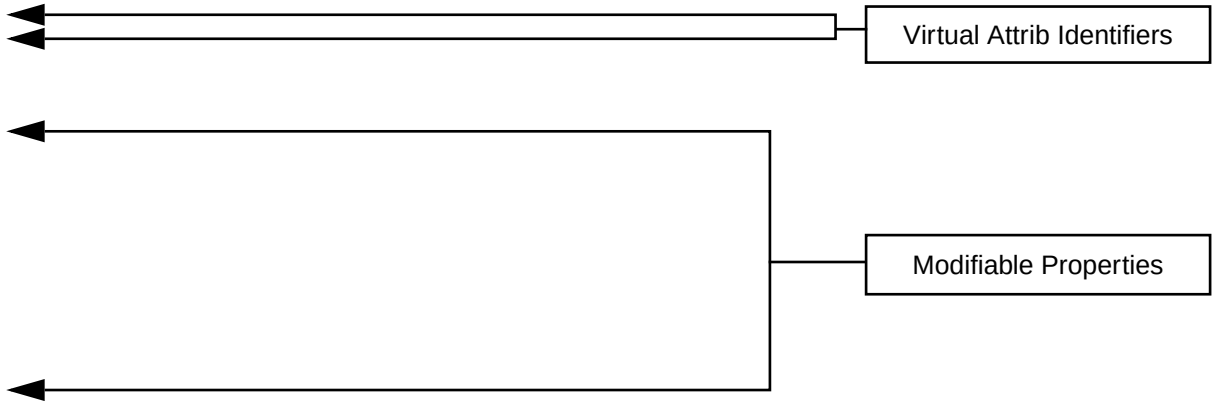
Additionally, helper functions have been added that allow the caller to populate the properties of the VirtualAttrib using a cdevData object that contains tagged data items that specify the new values. Other methods allow the caller to read selected properties into an outbound cdevData object using a caller specified context.

```
#ifndef _VIRTUAL_ATTRIB_H_
#define _VIRTUAL_ATTRIB_H_ 1

#include <cdevData.h>
#include <VirtualServer.h>

// ****************************************************************************
// * class VirtualAttrib:
// *This class maintains a list of items that make-up a Virtual Attrib. And
// *access mechanisms.
// ****************************************************************************
class VirtualAttrib
{
private:
   char             * device;
   char             * attrib;
   cdevMonitorNode * monitors;

   double  value;
   char    status  [255];
   char    severity[255];
   char    units   [255];
   double  alarmHigh;
   double  alarmLow;
   double  warningHigh;
   double  warningLow;
   double  controlHigh;
   double  controlLow;
   int     resultCode;
```



```
public:
   VirtualAttrib          (char * Device, char * Attrib);
   ~VirtualAttrib         ( void );

   int      setFromData    ( cdevData * data );
   void     getToData      ( cdevData * data, cdevData * context = NULL );
   void     getAllToData   ( cdevData * data );

   int      setValue       ( double Value );
   int      setStatus      ( char * Status );
   int      setSeverity    ( char * Severity );
   int      setUnits       ( char * Units );
   int      setAlarmHigh   ( double AlarmHigh );
   int      setAlarmLow    ( double AlarmLow );
   int      setWarningHigh ( double WarningHigh );
   int      setWarningLow  ( double WarningLow );
   int      setControlHigh ( double ControlHigh );
   int      setControlLow  ( double ControlLow );
   void     checkAlarms    ( void );

   double   getValue       ( void ) { return value; }
   char *   getStatus      ( void ) { return status; }
   char *   getSeverity    ( void ) { return severity; }
   char *   getUnits       ( void ) { return units; }
   double   getAlarmHigh   ( void ) { return alarmHigh; }
   double   getAlarmLow    ( void ) { return alarmLow; }
   double   getWarningHigh ( void ) { return warningHigh; }
   double   getWarningLow  ( void ) { return warningLow; }
   double   getControlHigh ( void ) { return controlHigh; }
   double   getControlLow  ( void ) { return controlLow; }
   int      getResultCode  ( void ) { return resultCode; }

   void     insertMonitor  ( cdevMonitorTable * table, cdevMessage * message );
   void     removeMonitor  ( cdevMonitorTable * table, cdevMessage * message );
};

#endif
```

VirtualAttrib.cc This is the source file that defines the implementation details of the VirtualDevice class. The methods below are used to insert and retrieve properties associated with the device/attribute pairs that are represented by each VirtualDevice object.

```
#include <VirtualAttrib.h>

static int VALUE_TAG          = -1;
static int STATUS_TAG         = -1;
static int SEVERITY_TAG       = -1;
static int UNITS_TAG          = -1;
static int ALARMHIGH_TAG      = -1;
static int ALARMLOW_TAG       = -1;
static int WARNINGHIGH_TAG    = -1;
static int WARNINGLOW_TAG     = -1;
static int CONTROLHIGH_TAG    = -1;
static int CONTROLLOW_TAG     = -1;
```

These static integers will be populated with the associated tag identifiers that are used in the cdevData object. By using the tag identifier integer rather than the associated character string, performance is greatly improved when accessing properties stored in the cdevData object.

```
// ************************************************************************
// * VirtualAttrib::VirtualAttrib :
// *This is the constructor for the VirtualAttrib class.  It initializes the
// *internal mechanisms to 0.
// ************************************************************************
VirtualAttrib::VirtualAttrib ( char * Device, char * Attrib )
    : device(strdup(Device)),
      attrib(strdup(Attrib)),
      monitors(NULL),
      value(0.0),
      alarmHigh(0.0),
      alarmLow(0.0),
      warningHigh(0.0),
      warningLow(0.0),
      controlHigh(0.0),
      controlLow(0.0)
    {
    *severity = 0;
    *units    = 0;
    strcpy(status, “NORMAL”);
    }
```

The constructor for the VirtualAttrib object makes a copy of the name of the device and attribute and then initializes all of its internal properties. Note that if the ‘high’ and ‘low’ values for a range specification (such as alarm) are equal, then that range specification is disabled.

```
// ************************************************************************
// * VirtualAttrib::~VirtualAttrib :
// *This is the destructor for the VirtualAttrib class.  It must free the
// *memory associated with the device and attribute names.
// ************************************************************************
VirtualAttrib::~VirtualAttrib ( void
)
   {
   delete device;
   delete attrib;
   }

// ************************************************************************
// * VirtualAttrib::setFromData :
// *This method will populate the VirtualAttrib object with the data contained in
// *the cdevData object.
// ************************************************************************
int VirtualAttrib::setFromData ( cdevData * data )
   {
   double val;
   int    result;
   if(data!=NULL)
      {
      result = CDEV_SUCCESS;
      data->get(UNITS_TAG, units, 255);
      if(data->get(CONTROLLOW_TAG, &val)==CDEV_SUCCESS)  setControlLow(val);
      if(data->get(CONTROLHIGH_TAG, &val)==CDEV_SUCCESS) setControlHigh(val);
      if(data->get(ALARMLOW_TAG, &val)==CDEV_SUCCESS)    setAlarmLow(val);
      if(data->get(ALARMHIGH_TAG, &val)==CDEV_SUCCESS)   setAlarmHigh(val);
      if(data->get(WARNINGLOW_TAG, &val)==CDEV_SUCCESS)  setWarningLow(val);
      if(data->get(WARNINGHIGH_TAG, &val)==CDEV_SUCCESS) setWarningHigh(val);
      if(data->get(VALUE_TAG, &val)==CDEV_SUCCESS)
         {
         result=setValue(val);
         }
      }
   else result = CDEV_ERROR;
   checkAlarms();
   return result;
   }
```

The destructor for a VirtualAttrib object only needs to delete the device and attrib strings that were duplicated when the object was created.

When a “set” cdevMessage object is received in the processMessages method of the VirtualServer, it contains a cdevData object that has a list of properties and values. For each property that is specified this method will call the set method with the new value. If the value is different than the previous value and the property is monitored, then the callback will be fired at that time.

The result of this operation is indicated by success in setting the value property.

```
// ************************************************************************
// * VirtualAttrib::getToData :
// *This method will populate the VirtualAttrib object with the data contained in
// *the cdevData object.
// ************************************************************************
void VirtualAttrib::getToData ( cdevData * data, cdevData * context )
   {
   if(data!=NULL)
      {
      data->remove();
      if(context!=NULL)
         {
         if(context->getType(VALUE_TAG)!=CDEV_INVALID)
            data->insert(VALUE_TAG, getValue());
         if(context->getType(STATUS_TAG)!=CDEV_INVALID)
            data->insert(STATUS_TAG, getStatus());
         if(context->getType(SEVERITY_TAG)!=CDEV_INVALID)
            data->insert(SEVERITY_TAG, getSeverity());
         if(context->getType(UNITS_TAG)!=CDEV_INVALID)
            data->insert(UNITS_TAG, getUnits());
         if(context->getType(CONTROLLOW_TAG)!=CDEV_INVALID)
            data->insert(CONTROLLOW_TAG, getControlLow());
         if(context->getType(CONTROLHIGH_TAG)!=CDEV_INVALID)
            data->insert(CONTROLHIGH_TAG, getControlHigh());
         if(context->getType(ALARMLOW_TAG)!=CDEV_INVALID)
            data->insert(ALARMLOW_TAG, getAlarmLow());
         if(context->getType(ALARMHIGH_TAG)!=CDEV_INVALID)
            data->insert(ALARMHIGH_TAG, getAlarmHigh());
         if(context->getType(WARNINGLOW_TAG)!=CDEV_INVALID)
            data->insert(WARNINGLOW_TAG, getWarningLow());
         if(context->getType(WARNINGHIGH_TAG)!=CDEV_INVALID)
            data->insert(WARNINGHIGH_TAG, getWarningHigh());
         }
      else
         {
         data->insert(VALUE_TAG, getValue());
         data->insert(STATUS_TAG, getStatus());
         data->insert(SEVERITY_TAG, getSeverity());
         }
      }
   }
```

When a “get” cdevMessage object is received in the processMessages method of the VirtualServer, it contains a context that indicates the properties that the caller desires to be returned. This method walks through the context object and copies each property that is specified in the context into the cdevData object pointed to by the data parameter. Once populated, this object will be returned to the caller.

If the context is empty, then the “value”, “status”, and

```
// ***************************************************************************
// * VirtualAttrib::getAllToData :
// *This method will populate the VirtualAttrib object with the data contained in
// *the cdevData object.
// ***************************************************************************
void VirtualAttrib::getAllToData ( cdevData * data )
   {
   if(data!=NULL)
      {
      data->remove();
      data->insert(VALUE_TAG, getValue());
      data->insert(STATUS_TAG, getStatus());
      data->insert(SEVERITY_TAG, getSeverity());
      data->insert(UNITS_TAG, getUnits());
      data->insert(CONTROLLOW_TAG, getControlLow());
      data->insert(CONTROLHIGH_TAG, getControlHigh());
      data->insert(ALARMLOW_TAG, getAlarmLow());
      data->insert(ALARMHIGH_TAG, getAlarmHigh());
      data->insert(WARNINGLOW_TAG, getWarningLow());
      data->insert(WARNINGHIGH_TAG, getWarningHigh());
      }
   }

// ***************************************************************************
// * VirtualAttrib::setValue  :
// *This method allows the caller to set the value of the Virtual Attrib.
// *This call will fail if the specified value is outside of the legal
// *range.
// ***************************************************************************
int VirtualAttrib::setValue ( double Value )
   {
   resultCode = CDEV_SUCCESS;

   if(controlHigh>controlLow &&
     (Value<controlLow || Value>controlHigh))
      {
      resultCode = CDEV_OUTOFRANGE;
      }
   else if(value != Value)
      {
      value = Value;
```

Unlike the getToData method, this method will populate the cdevData object will all properties that are currently contained in the VirtualAttrib object.

The setValue method is used to set the value property within the VirtualAttrib object. Since the overall range may be specified using the controlHigh and controlLow properties, this method will ensure that the new value conforms to the range (if specified) and will fail if the value is too high or too low.

```
        checkAlarms();
        if(monitors && monitors->isMonitored())
            {
            cdevData data;
            getAllToData(&data);
            monitors->fireMonitor(VALUE_TAG, &data);
            }
        }
    return resultCode;
    }

// ********************************************************************************
// * VirtualAttrib::setStatus  :
// *This method allows the caller to set the status of the device.
// ********************************************************************************
int VirtualAttrib::setStatus ( char * Status )
    {
    if(strcmp(status, Status))
        {
        strncpy(status, Status, 255);
        status[254] = 0;
        if(monitors && monitors->isMonitored())
            {
            cdevData data;
            getAllToData(&data);
            monitors->fireMonitor(STATUS_TAG, &data);
            }
        }
    return CDEV_SUCCESS;
    }

// ********************************************************************************
// * VirtualAttrib::setSeverity  :
// *This method allows the caller to set the severity flag for the device.
// ********************************************************************************
int VirtualAttrib::setSeverity ( char * Severity )
    {
    if(strcmp(severity, Severity))
        {
        strncpy(severity, Severity, 255);
        severity[254] = 0;
        if(monitors && monitors->isMonitored())
```

The checkAlarms method is called after the value has been set. The checkAlarms method determines if the new value places the VirtualAttrib in a warning or alarm state based on the confines that are specified in the alarmHigh/Low and warningHigh/Low properties.

If the VirtualAttrib is monitored, then all properties will be copied into a cdevData object and the list of monitors for the value property will be fired.

Sets the status property and fires any monitors that may be associated with that value.

Sets the severity property and fires any monitors that may be associated with that value.

```
        {
        cdevData data;
        getAllToData(&data);
        monitors->fireMonitor(SEVERITY_TAG, &data);
        }
     }
  return CDEV_SUCCESS;
  }

// ****************************************************************************
// * VirtualAttrib::setUnits  :
// *This method allows the caller to set the units for the device.
// ****************************************************************************
int VirtualAttrib::setUnits ( char * Units )
  {
  if(strcmp(units, Units))
     {
     strncpy(units, Units, 255);
     units[254] = 0;
     if(monitors && monitors->isMonitored())
        {
        cdevData data;
        getAllToData(&data);
        monitors->fireMonitor(UNITS_TAG, &data);
        }
     }
  return CDEV_SUCCESS;
  }

// ****************************************************************************
// * VirtualAttrib::setAlarmHigh  :
// *This method allows the caller to set the high alarm value of the device.
// ****************************************************************************
int VirtualAttrib::setAlarmHigh ( double AlarmHigh )
  {
  if(alarmHigh!=AlarmHigh)
     {
     alarmHigh = AlarmHigh;
     checkAlarms();
     if(monitors && monitors->isMonitored())
        {
        cdevData data;
```

Sets the units property and fires any monitors that may be associated with that value.

Sets the alarmHigh property and calls checkAlarms to determine if a change in this value will trigger a change in the device alarm status. Dispatches any monitors that are associated with the property.

```
            getAllToData(&data);
            monitors->fireMonitor(ALARMHIGH_TAG, &data);
            }
        }
    return CDEV_SUCCESS;
    }

// ************************************************************************
// * VirtualAttrib::setAlarmLow  :
// *This method allows the caller to set the low alarm value of the device.
// ************************************************************************
int VirtualAttrib::setAlarmLow ( double AlarmLow )
    {
    if(alarmLow!=AlarmLow)
        {
        alarmLow = AlarmLow;
        checkAlarms();
        if(monitors && monitors->isMonitored())
            {
            cdevData data;
            getAllToData(&data);
            monitors->fireMonitor(ALARMLOW_TAG, &data);
            }
        }
    return CDEV_SUCCESS;
    }

// ************************************************************************
// * VirtualAttrib::setWarningHigh  :
// *This method allows the caller to set the high warning value of a device.
// ************************************************************************
int VirtualAttrib::setWarningHigh ( double WarningHigh
)
    {
    if(warningHigh!=WarningHigh)
        {
        warningHigh = WarningHigh;
        checkAlarms();
        if(monitors && monitors->isMonitored())
            {
            cdevData data;
            getAllToData(&data);
```

Sets the alarmLow property and calls checkAlarms to determine if a change in this value will trigger a change in the device alarm status. Dispatches any monitors that are associated with the property.

Sets the warningHigh property and calls checkAlarms to determine if a change in this value will trigger a change in the device warning status. Dispatches any monitors that are associated with the property.

```
            monitors->fireMonitor(WARNINGHIGH_TAG, &data);
            }
        }
    return CDEV_SUCCESS;
    }

// ********************************************************************************
// * VirtualAttrib::setWarningLow   :
// *This method allows the caller to set the low warning value of a device.
// ********************************************************************************
int VirtualAttrib::setWarningLow  ( double WarningLow )
    {
    if(warningLow != WarningLow)
        {
        warningLow = WarningLow;
        checkAlarms();
        if(monitors && monitors->isMonitored())
            {
            cdevData data;
            getAllToData(&data);
            monitors->fireMonitor(WARNINGLOW_TAG, &data);
            }
        }
    return CDEV_SUCCESS;
    }

// ********************************************************************************
// * VirtualAttrib::setControlHigh  :
// *This method allows the caller to set the maximum value for a device.
// ********************************************************************************
int VirtualAttrib::setControlHigh ( double ControlHigh )
    {
    if(controlHigh != ControlHigh)
        {
        controlHigh = ControlHigh;
        checkAlarms();
        if(monitors && monitors->isMonitored())
            {
            cdevData data;
            getAllToData(&data);
            monitors->fireMonitor(CONTROLHIGH_TAG, &data);
            }
```

Sets the warningLow property and calls checkAlarms to determine if a change in this value will trigger a change in the device warning status. Dispatches any monitors that are associated with the property.

Sets the controlHigh property and calls checkAlarms to determine if a change in this value will trigger a change in the device status. Dispatches any monitors that are associated with the property.

```
        }
    return CDEV_SUCCESS;
    }

// ************************************************************************
// * VirtualAttrib::setControlLow   :
// *This method allows the caller to set the minimum value of a device.
// ************************************************************************
int VirtualAttrib::setControlLow  ( double ControlLow )
    {
    if(controlLow != ControlLow)
        {
        controlLow = ControlLow;
        checkAlarms();
        if(monitors && monitors->isMonitored())
            {
            cdevData data;
            getAllToData(&data);
            monitors->fireMonitor(CONTROLLOW_TAG, &data);
            }
        }
    return CDEV_SUCCESS;
    }

// ************************************************************************
// * VirtualAttrib::checkAlarms  :
// *This method allows the caller to read the value in comparison with all
// *of its limits and set the status and severity tag appropriately.
// ************************************************************************
void VirtualAttrib::checkAlarms ( void )
    {
    int done = 0;
    if(controlHigh>controlLow)
        {
        if(value<controlLow)
            {
            setStatus("OUT OF RANGE LOW");
            setSeverity("ERROR");
            done = 1;
            }
        else if (value>controlHigh)
            {
```

Sets the controlLow property and calls checkAlarms to determine if a change in this value will trigger a change in the device status. Dispatches any monitors that are associated with the property.

This method tests the value property against the ranges that may be specified in the warningHigh/Low, alarmHigh/Low and controlHigh/Low properties. If the value is outside of any of these ranges, then the status and severity variables will be set to a corresponding value: “WARNING”, “ALARM”, or “ERROR”

```
        setStatus(“OUT OF RANGE HIGH”);
        setSeverity(“ERROR”);
        done = 1;
        }
    }
if(!done && alarmHigh>alarmLow)
    {
    if(value<alarmLow)
        {
        setStatus(“ALARM LOW”);
        setSeverity(“ALARM”);
        done = 1;
        }
    else if (value>alarmHigh)
        {
        setStatus(“ALARM HIGH”);
        setSeverity(“ALARM”);
        done = 1;
        }
    }
if(!done && warningHigh>warningLow)
    {
    if(value<warningLow)
        {
        setStatus(“WARNING LOW”);
        setSeverity(“WARNING”);
        done = 1;
        }
    else if (value>warningHigh)
        {
        setStatus(“WARNING HIGH”);
        setSeverity(“WARNING”);
        done = 1;
        }
    }
if(!done)
    {
    setStatus(“NORMAL”);
    setSeverity(“\0”);
    }
}
```

```
// ************************************************************************
// * VirtualAttrib::insertMonitor :
// *This message adds a monitor to the cdevMonitorTable for this device/
// *attribute pair.  The message parameter becomes the property of the
// *monitorTable and should not be accessed again by the caller.
// ************************************************************************
void VirtualAttrib::insertMonitor ( cdevMonitorTable * table, cdevMessage * message )
   {
   if(table!=NULL && message!=NULL)
      {
      cdevData data;
      getAllToData(&data);
      table->insertMonitor(message, &data);
      monitors = table->findMonitor(device, attrib);
      if(monitors && !monitors->isMonitored()) monitors = NULL;
      }
   else if(message!=NULL) delete message;
   }
```

The processMessages method of the VirtualServer class calls this method when it receives a “monitorOn” message. This method collects all of the current property values into a cdevData object and then submits the cdevMessage object and the data to the insertMonitor method of the cdevMonitorTable object. The method will then call findMonitor to locate its node in the cdevMonitorTable for later access.

Note that the cdevMessage object becomes the property of the cdevMonitorTable object and should not be accessed again.

```
// ************************************************************************
// * VirtualAttrib::removeMonitor:
// *This method uses the cancelTransIdx to locate and delete a monitor
// *that was previously posted using that transaction index.
// ************************************************************************
void VirtualAttrib::removeMonitor (cdevMonitorTable * table, cdevMessage * message )
   {
   if(table!=NULL && message!=NULL)
      {
      table->removeMonitor(message);
      if(monitors && !monitors->isMonitored()) monitors = NULL;
      }
   }
```

The processMessages method of the VirtualServer class calls this method when it receives a “monitorOff” message. The method will call the removeMonitor method of the cdevMonitorTable to remove the monitor.

If all monitors have been removed that are associated with this VirtualAttrib object, then the monitors pointer will be set to NULL.

VirtualServer.h The following header file defines the structure of the VirtualServer class. The VirtualServer class inherits its functionality from the cdevServer object and consequently has to do very little initialization. When created it calls the populateTable method to generate a list of VirtualAttrib objects that it will support, the service may then manipulate any of the devices that the server has created using the commands “get”, “set”, “monitorOn”, and “monitorOff”.

```
#include <cdevServer.h>
#include <StringHash.h>
#include <cdevMonitorTable.h>
// ****************************************************************************
// * class VirtualServer :
// *   This is the server class for the VirtualDevice.  It simply receives
// *   messages from a client and immediately returns them.
// *
// *   The constructor passes the domain, server, port and rate to the
// *   underlying cdevServer class to be processed.  The cdevServer constructor
// *   will add this server to the Name Server and will begin processing
// *   messages when the cdevServer::runServer() method is executed.
// *
// *   The processMessages method is the servers interface to the world... Each
// *   time a complete message is received or the time specified in rate
// *   expires, that method will be called.
// ****************************************************************************
class VirtualServer : public cdevServer, public cdevMonitorTable
{
private:
   StringHash attribHash;
public:
   VirtualServer ( char * domain, char * server, unsigned int port, double rate )
      : cdevServer(domain, server, port, rate), attribHash()
      {
      populateTable();
      }
   virtual      ~VirtualServer   ( void );
   virtual void processMessages  ( void );
   void         populateTable    ( void );
   virtual int  fireCallback     ( cdevMessage * message );
};
```

VirtualServer.cc This source file implements the classes that are defined in the VirtualServer.h header file. The methods that are contained in this file define the functionality for the VirtualServer that is different from what is provided by default by the cdevServer class.

```
#include <VirtualServer.h>
#include <VirtualAttrib.h>

VirtualServer::~VirtualServer ( void )
   {
   StringHashIterator iter(&attribHash);
   VirtualAttrib    * attrib = NULL;
   char             * key    = NULL;

   iter.first();
   while((key=iter.key())!=NULL)
      {
      attrib = (VirtualAttrib *)iter.data();
      iter++;
      attribHash.remove(key);
      if(attrib!=NULL) delete attrib;
      }
   }
```

This is the destructor for the VirtualServer object. It is responsible for walking through the list of VirtualAttrib objects that were created and deleting each of them from the list prior to terminating.

```
void VirtualServer::populateTable ( void )
   {
   char device[10];
   char attrib[10];
   char key[20];
   for(int i=0; i<10; i++)
      {
      for(int j=0; j<10; j++)
         {
         sprintf(device, “device%i”, i);
         sprintf(attrib, “attrib%i”, j);
         sprintf(key, “device%i attrib%i”, i, j);
         attribHash.insert(key, new VirtualAttrib(device, attrib));
         }
      }
   }
```

The populateTable method is used to generate a collection of device names and their associated attributes that will be used to create a hash table of VirtualAttrib objects. The device names are “device0” through “device9”, and each device has attributes “attrib0” through “attrib9”.

The client may use the “get”, “set”, “monitorOn” or “monitorOff” methods to manipulate the properties associated with any of these VirtualAttrib objects.

```
void VirtualServer::processMessages ( void )
   {
   char            key[255];
   int             saveMessageFlag;
   int             sendMessageFlag;
   cdevMessage   * message;
   VirtualAttrib * attrib;
   cdevData        output;

   while(dequeue(message)==0)
      {
      // ************************************************************
      // * Note at this point a cdevTagMap has already been received
      // * from the client.  This tag map will have initialized all
      // * of the tags that are required by the service.
      // ************************************************************
      if(!strcmp(message->getMessage(), "unregister"))
         {
         sendMessageFlag = 0;
         removeClientMonitors(message->getClientID());
         }
      if(!strncmp(message->getMessage(), "get ", 4))
         {
         output.remove();
         saveMessageFlag = 0;
         sendMessageFlag = 1;

         sprintf(key, "%s %s",
                 message->getDeviceList()[0],
                 &message->getMessage()[4]);

         if((attrib = (VirtualAttrib *)attribHash.find(key))!=NULL)
            {
            attrib->getToData(&output, message->getContext());
            output.insert("resultCode", CDEV_SUCCESS);
            }
         else output.insert("resultCode", CDEV_NOTFOUND);
         }
```

The processMessages method will be called whenever data is waiting to be processed in the inbound queue. When called, this method should process all of the messages that it has available and then return 0.

To process a cdevMessage the method must dequeue it, process it, enqueue the result and then delete the original cdevMessage object.

When a client disconnects, an "unregister" message is automatically generated. When this message is received, the VirtualServer will remove all monitors installed by the client.

A "get" message is sent to read the values of one or more properties as specified by the context. This section locates the VirtualAttrib object using the device name and the attribute portion of the message. It then calls the getToData method of the VirtualAttrib object to read the properties that are specified in the contexts.

The resultCode property is set to CDEV_SUCCESS to indicate that the call completed successfully.

```
else if(!strncmp(message->getMessage(), "set ", 4))
   {
   output.remove();
   saveMessageFlag = 0;
   sendMessageFlag = 1;

   sprintf(key, "%s %s",
           message->getDeviceList()[0],
           &message->getMessage()[4]);


   if((attrib = (VirtualAttrib *)attribHash.find(key))!=NULL)
      {
      output.insert("resultCode",
                attrib->setFromData(message->getData()));
      }
   else output.insert("resultCode", CDEV_NOTFOUND);
   }
else if(!strncmp(message->getMessage(), "monitorOn ", 10))
   {
   saveMessageFlag = 1;
   sendMessageFlag = 0;

   sprintf(key, "%s %s",
           message->getDeviceList()[0],
           &message->getMessage()[10]);

   if((attrib = (VirtualAttrib *)attribHash.find(key))!=NULL)
      {
      attrib->insertMonitor(this, message);
      }
   }
else if(!strncmp(message->getMessage(), "monitorOff ", 11))
   {
   saveMessageFlag = 0;
   sendMessageFlag = 1;

   sprintf(key, "%s %s",
           message->getDeviceList()[0],
           &message->getMessage()[11]);

   if((attrib = (VirtualAttrib *)attribHash.find(key))!=NULL)
```

A "set" message is sent to set the values of specific properties to the values that are specified in the inbound cdevData object. This section uses the device name and attribute portion of the message to locate the appropriate VirtualAttrib object. It then uses the setFromData method of the VirtualAttrib to copy the values into the object. The return value of the setFromData method is inserted into the resultCode property to indicate if the request completed successfully.

A "monitorOn" message is used to establish a monitor that will trigger a callback message each time one or more of the properties specified in the context object is changed.

This method uses the insertMonitor method to install the monitor on the specified VirtualAttrib object.

A "monitorOff" message is used to remove one or more monitors that were previously installed using the monitorOn message. The cancelTransIndex component of the inbound cdevMessage object indicates the monitor that will be terminated.

```
            {
            attrib->removeMonitor(this, message);
            }
         }
      else
         {
         saveMessageFlag = 0;
         sendMessageFlag = 1;
         output.insert("resultCode", CDEV_NOTFOUND);
         }

      if(sendMessageFlag)
         {
         message->setData(&output, 1);
         enqueue(message);
         }
      if(!saveMessageFlag) delete message;
      }
   }
```

The sendMessageFlag indicates whether a return message should be provided to the caller. If the message was a monitorOn request, then the return message has already been automatically dispatched by the cdevMonitorTable object.

If the cdevMessage object was not provided to the cdevMonitorTable to install a monitor, then it should be deleted.

```
int VirtualServer::fireCallback ( cdevMessage * message )
   {
   int        result = CDEV_SUCCESS;
   cdevData * data   = NULL;

   if(message && (data = message->getData())!=NULL)
      {
      data->insert("resultCode", CDEV_SUCCESS);
      result = enqueue(message);
      }
   return result;
   }

void main()
{
VirtualServer server("VIRTUAL", "TestServerX", 9120, 60);
cdevServer::runServer();
}
```

This method is called by the cdevMonitorTable portion of the class whenever a monitored value has changed. It is only required to enqueue the cdevMessage object so that it can be returned to the client with the new value.

The main function creates an instance of the VirtualServer class named "TestServerX", which will have the Name Server domain "VIRTUAL" and will listen for incoming requests on port 9120. The processMessages method will be called automatically at least every 60 seconds.

VirtualService.h The VirtualService.h header file describes the structure of the VirtualService that will be loaded by the cdevSystem in order to accomodate requests made to the VirtualServer.

```
#include <cdevClientService.h>

// ********************************************************************
// * newVirtualService :
// *This function will be called by the cdevSystem object to create an
// *initial instance of the VirtualService.
// ********************************************************************
extern “C” cdevService * newVirtualService ( char * name, cdevSystem * system );

// ********************************************************************
// * class VirtualService :
// *This class simply inherits from the cdevClientService and must define
// *only a constructor and destructor.
// ********************************************************************
class VirtualService : public cdevClientService
{
public:
   VirtualService ( char * name, cdevSystem & system =
cdevSystem::defaultSystem());

protected:
   int RESULT_CODE_TAG;

   virtual ~VirtualService    ( void ) {};
   virtual void fireCallback  ( int status, cdevTranObj &xobj,  cdevData *resultData );
};
```

The newVirtualService function will create the initial instance of the VirtualService class.

The VirtualService class inherits almost all of its functionality from the cdevClientService class.

The fireCallback method has been overloaded in order to allow the service to copy the resultCode property from the returned cdevData object into the completion status for the cdevCallback function.

VirtualService.cc This source file implements the classes that are defined in the VirtualService.h header file. The methods that are contained in this file define the functionality for the VirtualService that is different from what is provided by default by the cdevClientService class.

```
#include <VirtualService.h>

// ********************************************************************************
// * newVirtualService:
// *This function will be called by the cdevSystem object to create an
// *initial instance of the VirtualService.
// ********************************************************************************
extern “C” cdevService * newVirtualService (char * name, cdevSystem * system)
   {
   return new VirtualService(name, *system);
   }

// ********************************************************************************
// * VirtualService::VirtualService :
// *This is teh constructor for the VirtualService.  It initializes the
// *underlying cdevClientService by specifying that it is in the domain of
// *VIRTUAL.
// ********************************************************************************
VirtualService::VirtualService ( char * name, cdevSystem & system)
   : cdevClientService(“VIRTUAL”, name, system)
   {
   // ********************************************************************
   // * Install the RESULT_CODE_TAG at a location of 30 or higher if it
   // * does not already exist.
   // ********************************************************************
   RESULT_CODE_TAG = 0;
   cdevData::tagC2I(“resultCode”, &RESULT_CODE_TAG);

   for(int i=30; RESULT_CODE_TAG==0 && i<65534; i++)
      {
      cdevData::insertTag(i, “resultCode”);
      cdevData::tagC2I(“resultCode”, &RESULT_CODE_TAG);
      }
```

The newVirtualService is called by the cdevSystem to create an instance of the VirtualService object.

The constructor for the class will initialize the underlying cdevClientService object with the name of the Name Server domain and the service name and cdevSystem parameters.

Additionally the constructor must declare any tags that it will be using for communications.

```
    system.reportError(CDEV_SEVERITY_INFO, "VirtualService", NULL,
             "Constructing a new VirtualService");

    }

// ************************************************************************
// * VirtualService::fireCallback :
// *This is the method that will be called to dispatch the callback methods
// *that are associated with the calls to the Virtual Server.  If the
// *message has been processed successfully, then the method will remove
// *the resultCode from the outbound data and use that as the status.
// ************************************************************************
void VirtualService::fireCallback  ( int status, cdevTranObj &xobj,  cdevData *resultData )
    {
    // ********************************************************************
    // * If the message was transmitted successfully, get the result code
    // * from the data that was returned and use that as the status.
    // ********************************************************************
    if(status==CDEV_SUCCESS && resultData!=NULL)
        {
        resultData->get(RESULT_CODE_TAG, &status);
        resultData->remove(RESULT_CODE_TAG);
        }

    cdevClientService::fireCallback(status, xobj, resultData);
    }
```

Once initialized the VirtualService will declare its presence using the reportError mechanism.

Before calling the user defined callback, this method will copy the resultCode property of the returned cdevData object into the status integer that is provided to the callback function.

Virtual.ddl This is the device definition file that is used to map the CDEV requests to the VirtualService for transmission to the VirtualServer.

```
service Virtual
   {
   tags {server}
   }

class Virtuals
   {
    verbs {get, set, monitorOn, monitorOff}
   attributes
      {
      default      Virtual;
      servers      Virtual;
      attrib0      Virtual {server=TestServerX};
      attrib1      Virtual {server=TestServerX};
      attrib2      Virtual {server=TestServerX};
      attrib3      Virtual {server=TestServerX};
      attrib4      Virtual {server=TestServerX};
      attrib5      Virtual {server=TestServerX};
      attrib6      Virtual {server=TestServerX};
      attrib7      Virtual {server=TestServerX};
      attrib8      Virtual {server=TestServerX};
      attrib9      Virtual {server=TestServerX};
      }
   messages
      {
      disconnect   Virtual;
      }
   }


Virtuals :
   device0, device1, device2, device3, device4,
   device5, device6, device7, device8, device9;
```