Controlling an IP-Based Media Server with the Media Server Markup Language

Executive Summary

The Media Server Markup Language (MSML) can be used to remotely control IP-based Media Servers, such as the Dialogic® PowerMedia™ Extended Media Server (PowerMedia XMS), with MSML commands and events being transported between an Application Server and a Media Server via SIP INFO messages. This application note describes the design and implementation of Application Server software that provides voice mail, video mail, and conferencing features using an MSML Media Server provided by PowerMedia XMS.

Table of Contents

Introduction

IP-based Media Servers, such as the Dialogic® PowerMedia™ Extended Media Server (PowerMedia XMS), may be remotely controlled using the Media Server Markup Language (MSML). An Application Server provides endpoint call control, establishes RTP media streams between the endpoints and the Media Server, and controls the Media Server using MSML commands and events transported via SIP INFO messages. This application note describes the design and implementation of Application Server software that provides voice mail, video mail, and conferencing features using an MSML Media Server provided by PowerMedia XMS.

IP-Based Media Server System Architecture

A system utilizing an IP-based Media Server typically includes an Application Server and a File/HTTP server. Connection to the PSTN network is accomplished by using an IP Gateway and connection to the Internet, and/or an enterprise IP network is established through a router/firewall. Figure 1 is a diagram of a simplified IP-based Media Server system utilizing MSML for media control. Not shown are typical SIP infrastructure elements, such as proxy and registration servers.
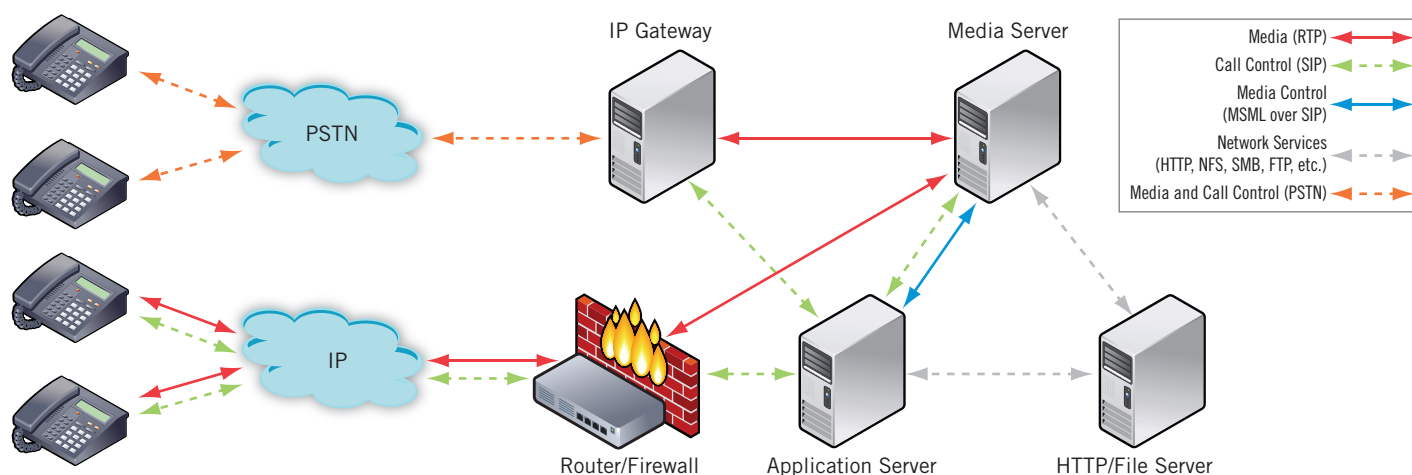

Figure 1: IP-Based Media Server System Architecture

Incoming calls are handled by the Application Server, which establishes the RTP path between the endpoint and the Media Server. The Application Server then controls the media operations using resources on the Media Server.

Role of the Application Server

The role of the Application Server is to provide call-based services to endpoints. Typical examples of the services provided by the Application Server include IVR, voice and video mail, conferencing, etc. Most of these services include a media processing element, which the Application Server provides through the use of a Media Server. The Application Server is responsible for call control, Media Server control, and the governing business logic. There is a distinct separation of call and Media Server control, and the media stream, as shown in Figure 2.
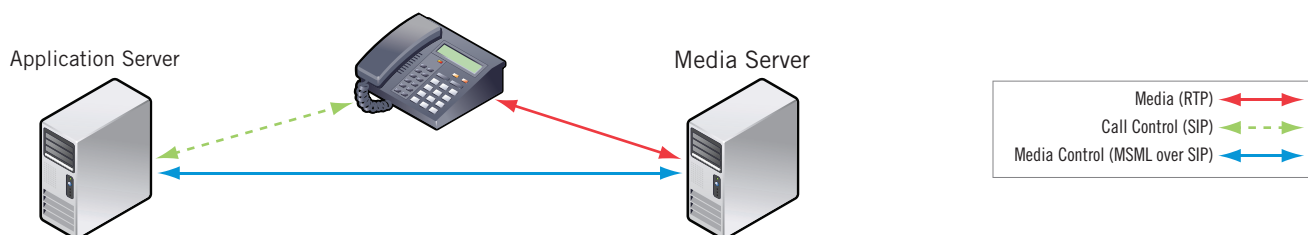

Figure 2: Separation of Control and Media

A typical call session starts with the endpoint calling the Application Server. The Application Server coordinates establishing the RTP media stream directly between the endpoint and the Media Server. Then according to the business logic implemented by the Application Server, the Application Server interacts with the Media Server to provide the media processing operations needed to support the business logic.

Media Server Control with MSML

The Media Server Markup Language is an XML language used to control media services provided by a Media Server. Many different kinds of services may be controlled with MSML including DTMF collection, play, record, conferencing, and dialogs. MSML is itself transport independent, but SIP INFO messages are typically used.

MSML commands are sent to the Media Server by the Application Server as XML within a SIP INFO message body. Events are returned to the Application Server using the same mechanism.

Figure 3 is a sequence diagram of a typical IVR session involving a SIP endpoint, Application Server, and a Media Server.
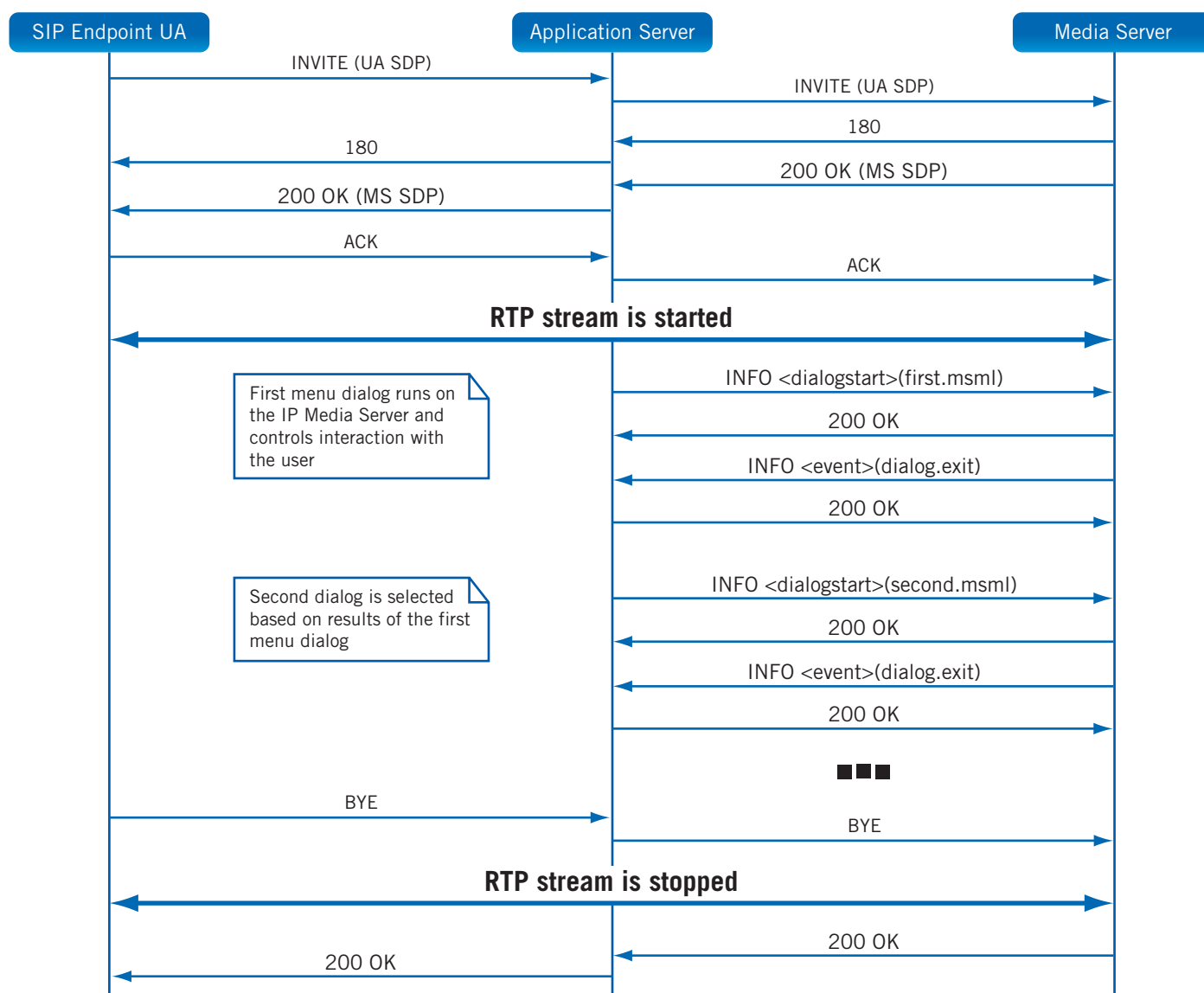

Figure 3: Typical IVR Session

Media Operations

The MSML Sever provided by PowerMedia XMS includes a rich set of media and stream control operations including play/record, DTMF detection and generation, stream, conferencing, dialog, and auditing.

An important element of MSML is the dialog facility that allows much of the work involved in executing complex media operations to be handled by the MSML Server rather than by the Application Server. Examples include prompt and collect DTMF for IVR menus, prompt collect DTMF and record/playback for voice mail, collect DTMF and volume control for conference participants, etc.

Refer to the *MSML Media Server Software User's Guide* or the Dialogic release being used for information on supported commands. Refer to the IETF RFC5707 - Media Server Markup Language (MSML) for details on command and attribute usage. For document locations on the web, see *For More Information* at the end of this document.

Dialogs

MSML dialogs are used to run scripts on the media stream associated with connections and conferences. The script executes on the MSML server, notifying the Application Server of the results.

An MSML dialog may have multiple operations, possibly executing in parallel. One common example is simultaneous play and collect DTMF operation. This is accomplished by using an MSML group executing the play and DTMF collection operations in parallel.

The MSML <dialogstart> command starts a script on the server by specifying the connection/conference target and a URI for the script. Alternatively, the script can be included directly inline. The following example starts the MSML script "menu.moml" on a target, which is the media connection "abcd1234". The target media connection identifier is obtained from the SIP "To:" header tag attribute received from the MSML server when the connection is established.

```
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
    <dialogstart
        target="conn:abcd1234"
        type="application/moml+xml"
        src="http://server.example.com/scripts/menu.moml"/>
</msml>
```

The MSML server will respond to the SIP INFO request with a SIP 200 OK message. The message body will contain the MSML response, which includes the result response code and the dialog identifier.

```
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
    <result response="200"/>
    <dialogid>conn:abcd1234/dialog:1</dialogid>
</msml>
```

At this point the MSML server is executing the script, the results of which will be sent to the Application Server in SIP INFO messages.

The following script plays a menu prompt and waits five seconds for input. If the DTMF is detected, the dialog returns the detected DTMF digit. If DTMF is detected before the prompt has finished playing, the play is terminated and the detected DTMF digit is returned. If a DTMF digit is not detected, the dialog returns a "noinput" result.

```
<?xml version="1.0" encoding="UTF-8"?>
<moml version="1.0" id="menu">
    <group topology="parallel">
        <play>
            <audio uri="file://Menu.wav" format="audio/wav"/>
            <playexit>
                <send target="collect" event="starttimer"/>
            </playexit>
        </play>
        <collect fdt="5s">
            <pattern digits="x">
                <send target="source" event="done" namelist="dtmf.digits
                    dtmf.end"/>
            </pattern>
            <detect>
                <send target="play" event="terminate"/>
            </detect>
            <noinput>
                <send target="source" event="done" namelist="dtmf.end"/>
            </noinput>
        </collect>
    </group>
</moml>
```

The example script has a single <group> defined where <play> and <collect> primitives execute in parallel.

The <play> primitive has two elements. The first is the URI of audio file to be played on the stream. The second is an action to be taken when the play completes, which in this case is to start the first digit timer on the <collect> primitive.

The <collect> primitive has one attribute and three elements defined. The first digit timer (fdt) attribute is set so that a no input event is fired if a DTMF digit is not detected within five seconds, and by default if the timer is not started when <collect> is initially invoked. The <pattern > element specifies the DTMF digit pattern to be matched, in this case any single DTMF digit, and to send an event to the Application Server that includes the detected DTMF digit. The <detect> element sends the <play> primitive a terminate event when DTMF is detected, in which case the play stops. The <noinput> element sends an event to the Application Server when the first digit timer elapses without detecting a DTMF digit.

The example script has two paths to completion depending on whether a DTMF digit is detected. If a DTMF digit is detected, the MSML Server sends an event to the Application Server:

```
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
    <event name="done"
        id="conn:abcd1234/dialog:1">
        <name>dtmf.digits</name><value>1</value>
        <name>dtmf.end</name><value>dtmf.match</value>
</event>
```

In addition to the event sent to the Application Server, the script also sends a terminate event to the <play> primitive when a DTMF is detected to end the play, if it has not already completed.

If the <play> completes and the <collect> primitive is still executing, a start timer event is sent to the <collect> primitive to start the first digit timer. If no DTMF digits are detected before the timer elapses, the MSML Server sends the following to the Application Server:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
    <event name="done"
        id="conn:abcd1234/dialog:1">
        <name>dtmf.end</name><value>dtmf.noinput</value>
</event>
```

Finally, the MSML Server sends an event in a SIP INFO message signaling that the dialog has completed:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
    <event name="msml.dialog.exit"
        id="conn:abcd1234/dialog:1"/>
</msml>
```

Conferencing

MSML conferences are resources that mix audio and/or video streams from multiple connections. A conference is established on the MSML server by using <createconfence> and specifying initial properties such as the identifier, when to delete the conference, active speaker notification report interval, and video layout.

The object identifier for a conference can be specified with the name attribute, or if not specified, it will be assigned by the MSML Server and returned in the SIP INFO message final response.

Deleting a conference releases the conference resources to the server and can be accomplished in several ways. The default method is to automatically delete the conference when the last media connection is removed. Other available methods are to automatically delete the conference when the control channel is terminated and to only delete the conference explicitly with <deleteconference>.

The active speaker notification report interval is set using the <ri> attribute of the <audiomix> element. It is set to "0" in this example so that notifications are not sent to the Application Server.

The following creates a conference using the default method to delete conferences when the last media connection is broken and to have the MSML Server assign an object identifier:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
 <createconference>
 <audiomix>
 <asn ri="00s"/>
 </audiomix>
 </createconference>
</msml>
```

The MSML server will respond to the SIP INFO request with a SIP 200 OK message. The message body will contain the MSML response, which includes the result response code and the conference identifier:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
    <result response="200"/>
    <confid>conf:hostname-2-48</confid>
</msml>
```

Connections are added to a conference using <join> and supplying the connection and conference identifiers. The identifiers id1 and id2 can be for either the connection or conference:

```
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
    <join id1="conn:abcd1234" id2="conf:hostname-2-48" />
</msml>
```

Connections may be removed from a conference either explicitly or implicitly. Connections are explicitly removed from a conference using the <unjoin> command. When a connection is terminated, it is implicitly and automatically removed from the conference.

Sample Application Server

The MSML Application Server sample application provides a means of demonstrating and exercising features of an MSML IP Media Sever as supported on PowerMedia XMS. The sample Application Server provides voice mail, video mail, and conferencing functionality accessed by voice and multimedia menus presented to the caller. For code download locations on the web, see *For More Information* at the end of this document.

The menu structure is:

Main Menu:

    1 – Voice mail
    2 – Video mail (available only on a video call)
    3 – Voice conference (available only if supported by MSML Server)
    4 – Video conference (available only if supported by MSML Server)
    * - Quit

Voice Mail and Video Mail have the same menu structure. The Video Mail menu is only available if a video capable SIP phone is detected:

```
1 – Record
        Enter 3 digit mailbox number between 101 and 299
                2 – Start Record
                        2 – Stop Record
                                2 – Re-record
                                3 – Save
                                4 – Play
                                * - Quit
                        * - Quit
                * - Quit
        * - Quit
2 – Play
        Enter 3 digit mailbox number between 101 and 299
                1 - Re-play
                2 - Delete
                * - Quit
        * - Quit
* - Quit
```

Voice Conference and Video Conference menu structure:

> 1 - Create conference
>> Enter 3 digit conference number between 101 and 299
>>> * - Quit
> 2 - Join Conference
>> Enter 3 digit conference number between 101 and 299
>>> * - Quit
>> * - Quit
> * - Quit

System Requirements

Hardware and OS requirements are those of the Dialogic® PowerMedia™ Extended Media Server and PowerMedia Dialogic® PowerMedia™ Host Media Processing Software Release 4.1LIN highlighted below. For specific requirements, please refer to the Dialogic® PowerMedia™ Extended Media Server (XMS) Quick Start Guide and the Dialogic® Host Media Processing Software Release 4.1LIN Release Guide.

MSML Media Server

- Dialogic® PowerMedia™ Extended Media Server with licenses to support the desired number of sessions, conferences and codecs.

Application Server

- Network file system access to media files on the Media Server or access to an HTTP server

- Dialogic® PowerMedia™ HMP Software requirements

  — License

 2 IP Call Control per channel

  — 3PCC Support

 Dialogic® PowerMedia™ Host Media Processing Software Release 4.1LIN (PowerMedia HMP 4.1)

- SIP phones

  — Hardware- or software-based

- HTTP server (optional)

  — If an HTTP server is used for script and/or media file storage, it must support GET, PUT and DELETE.

Configuring the Sample Application Server

The MSML Application Server sample application configuration is contained in the text file app_server.cfg included in the data directory of the source code download. See the readme.txt file in the docs directory of the source code download for more information. The following configuration items are available:

Logging level by log object type

      log:<log object type>=<log level>

          where the log object types are:

             all

             system

             main

             eventmgr

             resource

             resourcemgr

             resourcegrp

             resourcegrpmgr

             callflow

             callflowmgr

          and the log levels are:

             debug

             info

             warn

             error

      The default log level for all log object types is "info".

SIP port used by the Application Server

      port:<sip port> the default SIP port is 5060

URI of the Application Server

      app:<URI> where the URI is a SIP address.

      Example: 1234@192.168.1.61:5070

URI of the MSML Media Server

      msml:<URI> where the URI is a SIP address.

      Example: 5678@192.168.1.51

Maximum number of channels

      maxchan:<max channels> where max channels is the maximum number of SIP resources to use. Each session requires two SIP resources. The default maximum number of channels is 4.

URI to the media file storage

      media:<URI> where the URI is for the directory where media files are stored. file:// and http:// are supported.

      file:// is relative to the Media Server.

      The default media file storage URI is file:///media.

File path mapping to the media file storage

      mediafilemap:<path> where the path is the local Application Server path to the directory where media files are stored.

      This value of path replaces file:// in media URIs, if used.

      The default map to file based media file storage is /xms.

      This configuration is only applicable if media file URIs use file://. It is ignored for media file URIs using http://.

URI to the script file storage

      scripts:<URI> where the URI is for the directory where script files are stored.

      "file://" and "http://" are supported.

      file:// is relative to the Media Server.

      The default script file storage URI is file:///scripts.

Storing MSML Scripts and Media Files

The MSML Application Server sample application distribution includes media and script directories that contain audio and video media files and the MSML scripts. The MSML scripts and media files may be stored on the MSML Media Server or on an HTTP server. If the scripts and media files are to be stored on the MSML Media Server, "file://" will be the URI base. If the script and media files are to be stored on an HTTP server, then "http://" will be the URI base.

The script files must be modified to specify the appropriate URI base for accessing the media files. A search and replace utility can be used to replace "%MEDIA_URI_BASE%" with the appropriate value in the script files.

MSML Application Server sample application configurations for file based storage:

> media:file://media
> mediafilemap:/xms
> scripts:file://scripts

This configuration example requires that:

- Script files have been changed with %MEDIA_URI_BASE% replaced by file://media.
- The Media Server media directory contains the media and scripts directories and that the directory is shared on the network.
- The Application Server has the network drive mapping /xms mapped to the Media Server media directory.

HTTP-based file storage requires that the HTTP server support GET, PUT, and DELETE commands. The Microsoft Internet Information Server (IIS) can be configured to support PUT and DELETE by enabling write permission for the media directory. The Apache HTTP Server supports PUT and DELETE by enabling the mod_dav module and configuring the container for the media directory.

Sample Application Server example configurations (app_server.cfg) for HTTP Server-based storage:

> media:http://httphost/msml/media
> scripts:http://httphost/msml/scripts

This configuration example requires that:

- Script files have been changed with %MEDIA_URI_BASE% replaced by http://httphost/msml/media.
- The HTTP server has been configured so that HTTP PUT and DELETE are supported in the directory /msml/media.

Running the Sample Application Server

The MSML Application Server sample application is started from a command prompt using the following usage:

```
Usage: app_server <options>
options:
        [-h] [?] - display command line help
        [-maxchan:<max channels>] - maximum number of SIP channels, default: 4
        [-log:<objectType>=<log level> - set logging levels
                <objectType>= [system | main | eventmgr | resource | resourcemgr |
                        resourcegrp | resourcegrpmgr
                        callflow |callflowmgr]
                <log level> = [debug | info | warn | error>]
        [-port:<sip port>] - sip port, default:5060
        [-app:<URI>] - application server SIP address
                        example: 1234@192.168.1.61:5070, default: 1@localhost
        [-msml:<URI>] - MSML Server SIP address
```

example: 5678@192.168.1.51, default: 2@media_server
[-media:<path>] - uri for media files from the Media Server(file:// or http://)
default: file:///media
[-mediafilemap:<path> - local path/dirve mapping to replace file:/// if used
default: /xms
[-scripts:<uri>] - uri for script files(file:// or http://)
default: file:///scripts

Note that the command line parameters are also available as configuration items in app_server.cfg.

While app_server is running, commands may be entered in the console window. The available commands are:

Ctrl-C, 'q', "quit" — quit the application
log:<objectType>=[debuglwarnlinfolerror] — change the log level

Logging is output to the console window as well as to the logfile.

Sample Application Server Design

The design for the MSML Application Server sample application is described in terms of the software architecture, design features, how Design Patterns have been applied, and a high-level block diagram. This section of the document includes information for readers interested in designing applications. Those only wanting to run the sample application server can skip this section and jump to the "Building the Sample Application Server" section below.

High Level Design

The high level design consists of a description of responsibilities for each major component class and its relationships with other major component classes.

Figure 4 shows the major component classes of the MSML Application Server sample application and relationships between them.

Figure 4: MSML Application Server Sample Application and Relationships
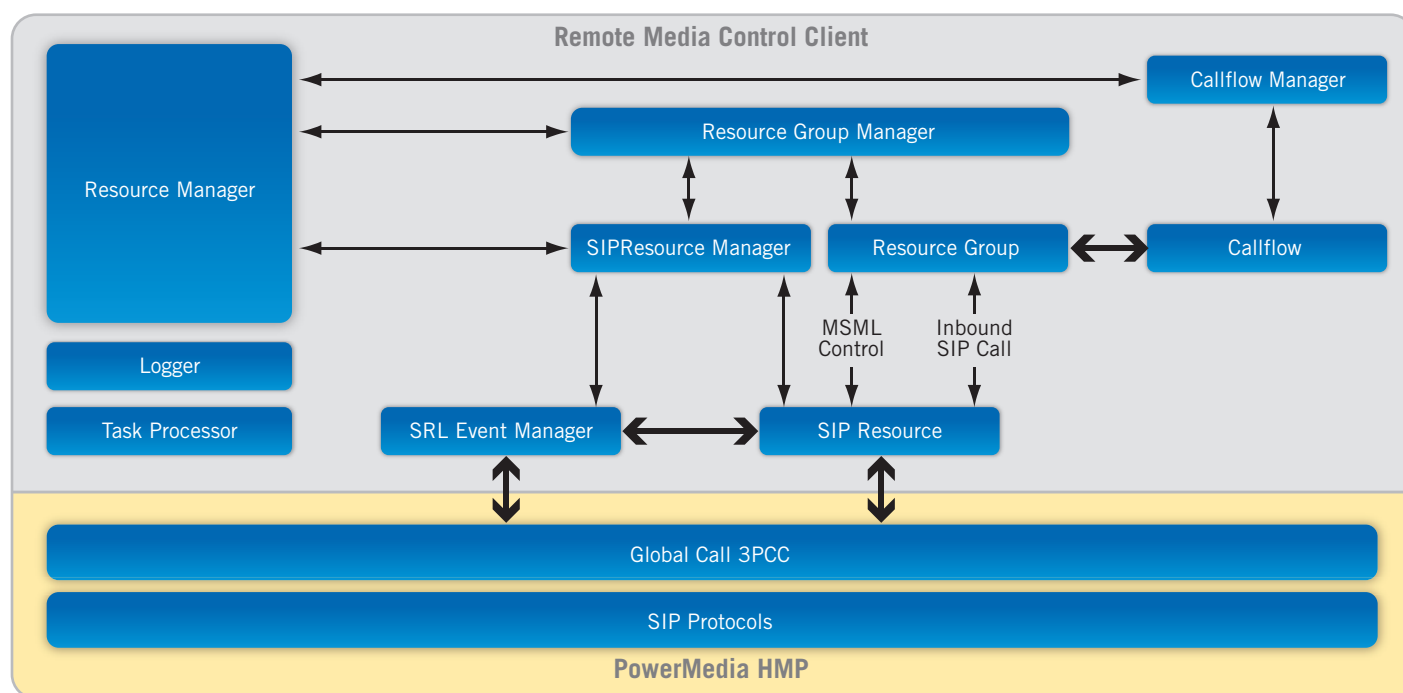
Major Component Classes

Remote Media Control Client

The Remote Media Control Client is a singleton responsible for start-up and shutdown of the resources required to run the application.

Start-up tasks include reading the configuration, and starting the Logger, Task Processor, Callflow Manager and Resource Manager. Starting the Resource Manager causes the application to begin processing of inbound calls.

Shutdown tasks include stopping the Resource Manager and Callflow Manager, and releasing the Logger and Task Processor.

The Remote Media Control Client is started and stopped from the process's main() function.

Resource Manager

The Resource Manager is a singleton responsible for initializing the call control, media, and application logic resources required for handling inbound calls.

On start-up, the Resource Manager initializes the Callflow Manager, Resource Group Manager, and SIP Resource Manager. Capacities and characteristics are based on configuration and are included as initialization parameters.

When an inbound call is detected, the Resource Manager obtains a Resource Group from the Resource Group Manager and a Callflow from the Callflow Manager, provides the Resource Group to the Callflow, and starts the Callflow to handle the call.

When a call disconnect event is detected, the Resource Manager resets the Callflow and Resource Group, and resources are returned to the managers.

The Resource Manager is started and stopped by the Remote Media Control Client.

Callflow Manager

The Callflow Manager is a singleton responsible for providing objects that will control the application logic for a single inbound call based on configuration and capabilities associated with the inbound call.

The Resource Manager requests a Callflow from the Callflow Manager. A reference to a Resource Group containing the Inbound Call and Media Resource is provided in the request so that the Callflow Manager can select the appropriate Callflow.

The Callflow Manager is started and stopped by the Resource Manager.

Callflow

Callflow is responsible for the context and state machine that controls the processing of an inbound call.

Callflow objects are initialized by the Callflow Manager with a Resource Group that provides the interface for executing dialogs on a Media Server. Using this interface, Callflow issues StartDialog commands to the Resource Group and responds to DialogStarted and DialogComplete events in accordance with the state machine.

SIP Resource Manager

The SIP Resource Manager is a singleton responsible for managing system level SIP resources used for call and media control. This includes discovering available devices, managing them during start-up and shutdown, and starting and stopping system level event delivery for the devices.

The SIP Resource Manager is started and stopped by the Resource Manager, and used to detect incoming calls and disconnected calls.

The Resource Group Manager uses the SIP Resource Manager to reserve SIP Devices to construct Resource Groups.

SRL Event Manager

The SRL Event Manager is a singleton that is responsible for receiving the SRL events, packaging event data, and dispatching event tasks to registered SIP resource objects.

SIP Resource objects register for events from the SRL Event Manager.

The SIP Resource Manager starts and stops the SRL Event Manager.

SIP Resource

SIP Resource objects are responsible for making system level device API calls and responding to events needed to perform call and media control.

SIP Resource objects are used by the Resource Manager to detect incoming calls and disconnected calls.

Resource Groups use SIP Resource objects for call control of incoming calls, and as the connection to an MSML Media Server for transport of MSML commands and events.

Resource Group Manager

The Resource Group Manager is a singleton responsible for constructing Resource Group objects based on configuration and capabilities associated with an inbound call.

Resource Group objects are constructed by initializing a Resource Group object with SIP Resources obtained from the SIP Resource Manager.

The Resource Manager starts and stops the Resource Group Manager, and uses it to obtain Resource Groups.

Resource Group

A Resource Group is responsible for a set of resources and for providing an interface to a Callflow that does not require knowledge of the underlying resources. Responsibilities also include coordinating setup and teardown of resources that make up the group.

Resource Groups are used by Callflows to perform media dialog operations on a Remote Media Server and by the Resource Manager as a means to associate an inbound call and a Media Resource, and maintain the context of the resources within the Resource Group.

Logger

The Logger is a singleton responsible for logging messages to a console window and a file based on a dynamically adjustable logging level.

The Logger is started by the Remote Media Control Client.

Task Processor

Task Processors are responsible for maintaining an input queue of tasks to execute, and executing, in the order received, each task in a worker thread.

Software Architecture

The MSML Application Server sample application software architecture uses a layered approach consisting of resource, resource group, and business logic layers. Each layer consists of stateful context objects communicating via messages. An application services layer provides facilities used by the other layers. Figure 5 is a diagram of the software model.
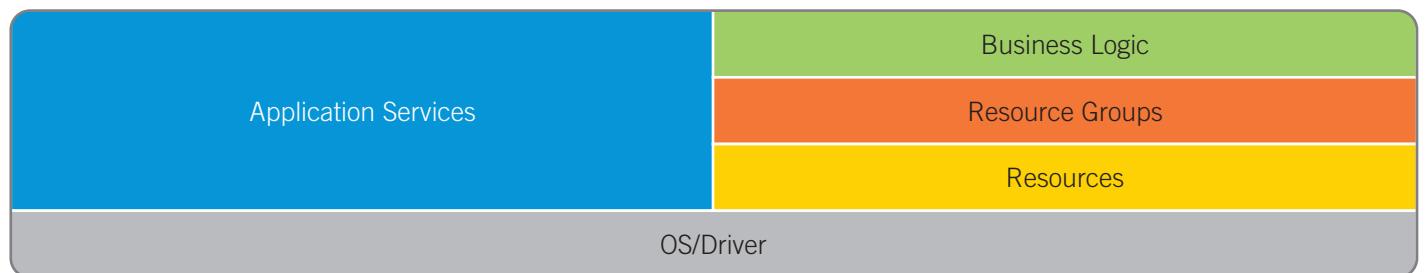
Figure 5: Software Model

The result of this software model is an architecture that allows a high degree of decoupling between the application business logic, and the underlying resource technologies and implementations.

An event-driven, multi-threaded asynchronous programming model supports the message passing mechanism and provides scalability.

OS/Driver Layer

This OS/Driver layer represents the operating system, drivers and libraries as provided by the Dialogic® PowerMedia™ Host Media Processing Software and other components, such as third party libraries. Access to this layer is through published APIs, and the layer is not part of the application code base.

Application Services Layer

This layer consists of modules that provide services to the other layers, including runtime configuration, logging, and resource management. Classes common to the other layers, such as those used to support worker threads, message passing, and portability are also included in the Application Services layer.

Resource Layer

The resource layer includes classes that wrap functionality of OS/Driver layer entities and provide convenient interfaces for use by the resource group layer objects.

Resource Group Layer

The resource group layer consists of classes that provide a high-level, resource-focused interface to the business logic layer. Resource group layer classes may simply wrap a resource class, or may coordinate between two or more resource classes to provide the high-level interface.

Business Logic Layer

The business logic layer consists of classes that implement the call flow for the MSML Application Server sample application. The business objects use interfaces provided by the application services and resource group layer classes.

Message Passing

Communication between the software layers, and between objects in the same layer, is performed using a message passing mechanism. Commands are invoked on an object by queuing a task to the target. Results are received by means of notification events. Commands and events are encapsulated in task objects that include associated command parameters and event data.

Multithreading

Command tasks and event notifications are processed in worker threads by queuing task/event objects to the worker thread's queue. Each object that accepts tasks and events is assigned a worker thread, and all tasks and events are processed for that object in the assigned worker thread. This has the advantage that tasks and events for an object are processed in the order they are queued and additional data synchronization is not required.

The main application process, SRL event handler, and logger run in separate threads.

Software Design Features and Application of Design Patterns

The design for objects in software model layers follows an asynchronous event/command driven state machine model. This is accomplished by using a combination of the State, Command, and Observer Design Patterns.

Complexity is managed by limiting class responsibilities. For classes that handle events and commands, abstract classes are responsible for public interfaces and the state machines, and concrete subclasses are responsible for implementing a set of primitive functions.

Memory management is addressed by the use of smart pointers. In cases where shared-ownership or transfer-of-ownership is not required, Boost scoped_ptr is used; otherwise Boost shared_ptr is used.

State Machines

State machines in the MSML Application Server sample application are implemented using the State Pattern. The State Pattern is described in terms of context and state objects. The context object delegates handling of commands and events to the current state object, which takes action appropriate for the state. State objects further delegate handling of commands and events back to the context object via *primitive* functions. Changing the state of a context object is accomplished by changing the state object to one of a different class.

The state machine for each context object is implemented as a set of concrete state classes, each of which implements the behavior for commands and events valid for that state. The context object state is changed based on the result of command and event handling. An abstract state class, from which concrete state classes are subclassed, implements default behavior for commands and events. In general, the default behavior is to handle them as error conditions with no change to the state. Error handling in the abstract state is event/task specific and may require posting a task failed event or may only require logging an error.

The responsibilities of context classes are divided between an abstract context class and concrete subclasses. The abstract context class is responsible for the public interface, the state objects which make up the state machine, and for delegating commands and events to the current state object. The concrete context subclasses are responsible for implementing the *primitive* functions called by the state classes.

The responsibilities of the state classes are to delegate commands and events back to the context primitive functions and to change the context state based on event/command and the results of the primitive functions.

Figure 6 shows the static class diagram of the State Pattern implementation used in this design. Figure 7 is a sequence diagram showing how commands and events are handled by context and state objects.
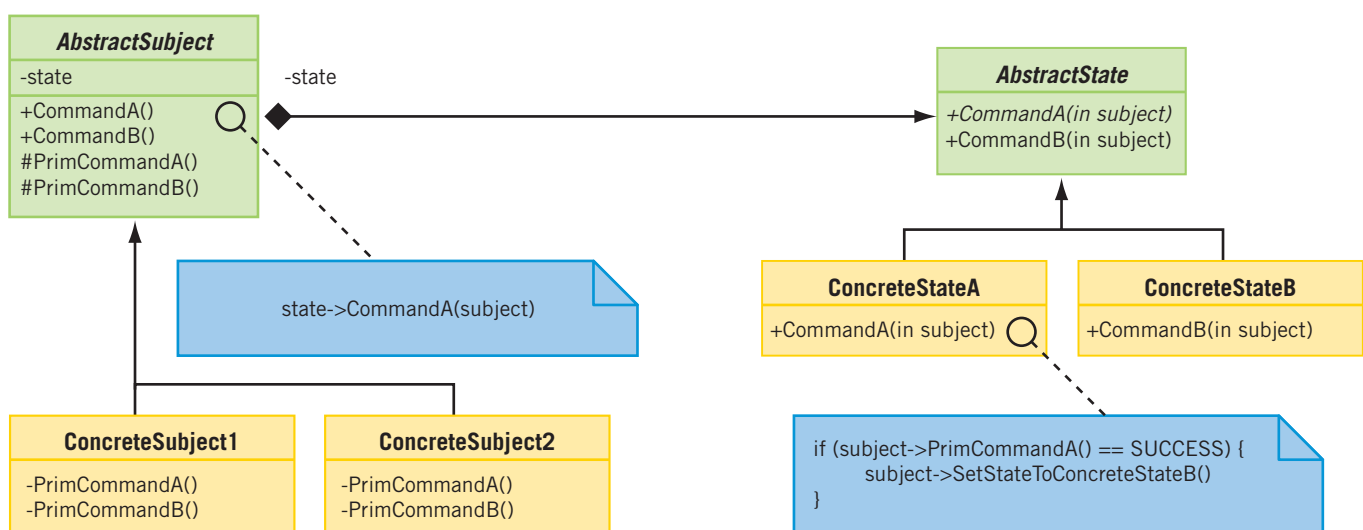

Figure 6: Static Class Diagram — State Machine Implementation Using the State Pattern
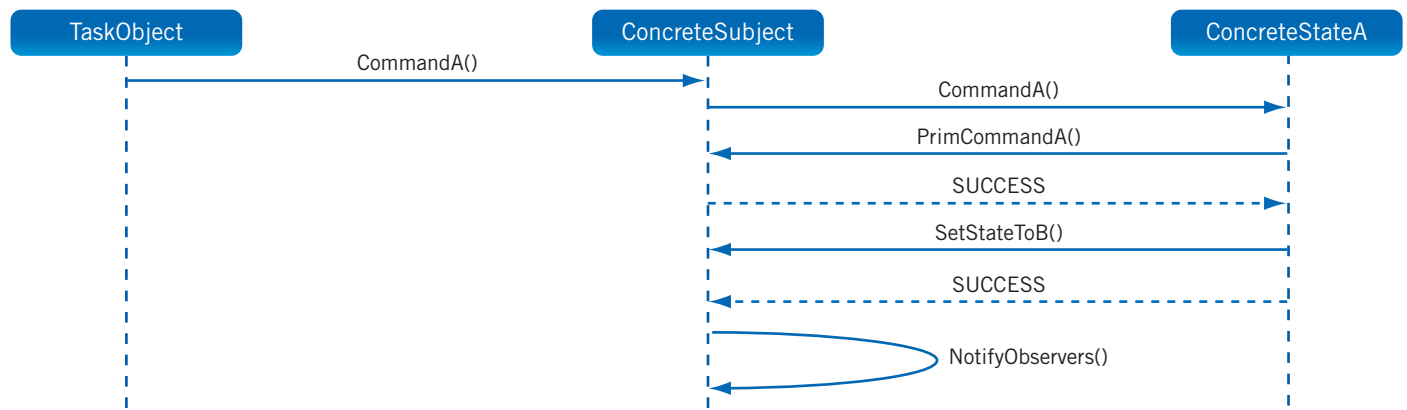
Figure 7: Sequence Diagram — State Machine Implementation Using the State Pattern

Each concrete state class encapsulates the behavior for valid commands and events only for that state which results in code being localized in each class. State classes can be hierarchical, which reduces the code duplication typical in nested switch statements and state table implementations of state machines.

Performance of state machines using the State Pattern is typically better than that of state machines implemented with state tables, as performance is independent of the number of states. Memory utilization is efficient when the state classes are designed without class member variables.

Command and Event Processing

Command and event processing in the MSML Application Server sample application uses an implementation of the Command Pattern. Commands and events are handled identically by packaging up the necessary data and function information into a task object and processing the task in a worker thread.

The Command Pattern is described in terms of client, receiver, and invoker. The client is the source of a command, the receiver is the target of the command and the invoker is the mechanism by which the command is executed. A client creates a command object and hands it off to the invoker. The invoker executes the command on the receiver, possibly at a later time and in a different thread of execution. Application of the Command Pattern allows a straightforward way to support multithreading and to minimize coupling both between classes and at runtime between objects.

In the MSML Application Server sample application design, task objects are used to hold commands as function objects along with data objects required as function parameters. Events are mapped to event handling functions, packaged into task objects and treated the same as other tasks. The design of the invoker is that of a worker thread with a task queue to hold pending tasks. Tasks are pushed on the queue by clients and popped from the queue for execution in the worker thread. A mutex makes the queue thread-safe and a semaphore mechanism signals the worker thread when a task is ready to be processed.

Objects that participate in task processing are assigned a worker thread that will execute the tasks for that object. This has the advantage of providing a single point of data synchronization.

Several Boost libraries are used to implement task processing. Using Boost has the advantages of being platform independent and of a modern C++ design and implementation. Specifically, the Boost.Thread library is used for threads and data synchronization, Boost.Function and Boost. Bind are used for function objects, and Boost shared pointer is used for memory management of the task objects. The Boost.Function library is a set of function object wrapper class templates. The Boost.Bind library provides for binding free functions, member functions and function parameters to function objects.

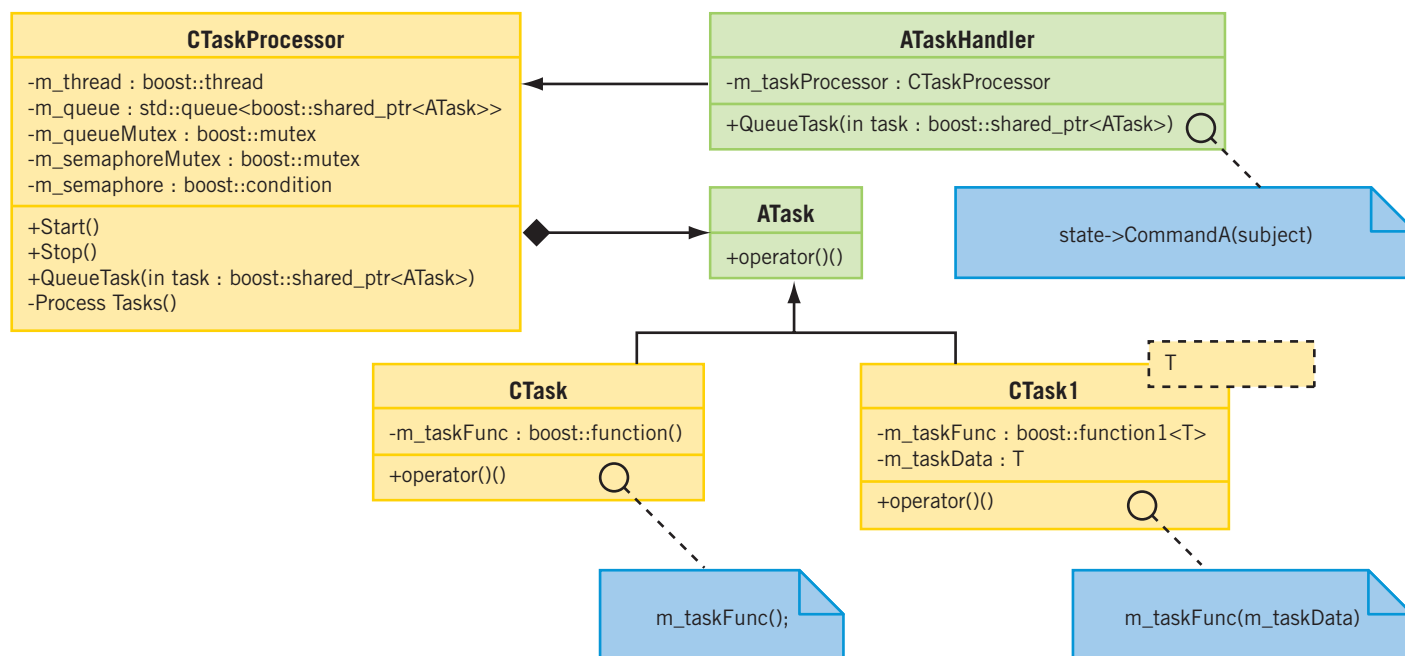Figure 8 is a diagram of the task processing classes used in this design.

Figure 8: Static Class Diagram — Task Processing

Collaboration Between Objects

Message passing between two objects in the MSML Application Server sample application is the process of one object queuing a command task to a second object and receiving an event task in return. The Command Pattern addresses the mechanism used for queuing tasks to another object, and the Observer Pattern addresses the mechanism by which objects can receive event tasks.

In general, the Observer Pattern describes the process of one object registering for, and receiving notification of, state changes in another object in a manner that keeps the two objects decoupled. It is described in terms of observers and subjects. Observers register with subjects to be notified of state changes. When the state of a subject changes, the registered observers are notified. Observers may also unregister to stop receiving notifications. The consequences of using the Observer Pattern are that the subject can notify any number observers of state changes without specific knowledge of the observers. This enables multicasting of events and minimizes coupling between observers and subjects.

The Observer Pattern implementation used in the MSML Application Server sample application expands on the concept of state change notifications to include notifications for identifiable events and also includes command task result notifications. Observers register with a subject for each notification individually by supplying a task object as part of the registration. The observer notification process consists of the subject making a copy of the task object for each registered task, binding any task data to the task object, and queuing the task to the observer. Figure 9 is a diagram of the classes used to support the Observer Pattern and includes pseudo-code implementation notes.
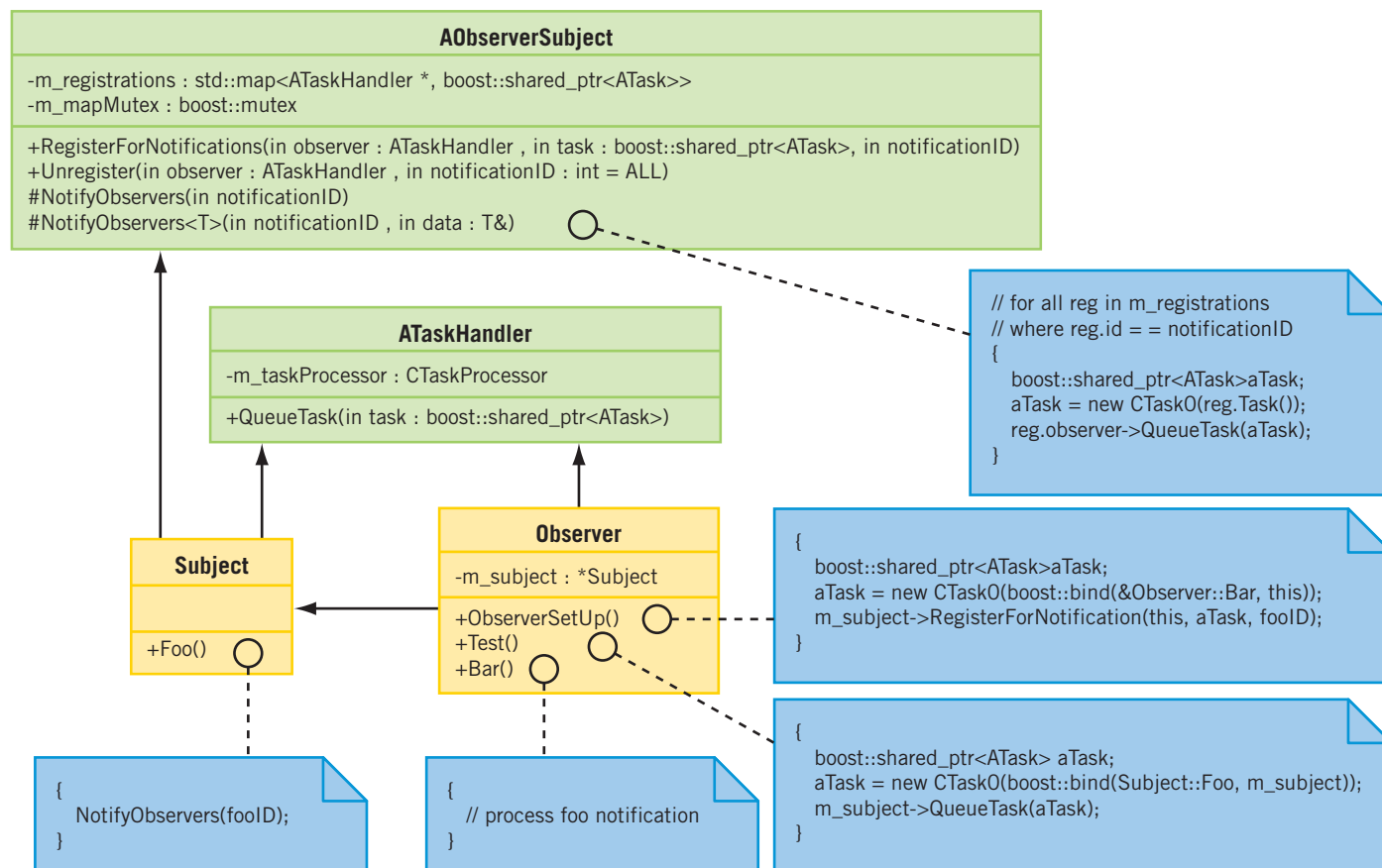
Figure 9: Static Class Diagram — Observer

Building the Sample Application Server

Build Requirements

Dialogic® PowerMedia™ Host Media Processing Software Release 4.1LIN
GNU gcc 3.2.3 or later
Boost Libraries 1.33.1 or later
cURL Library

The source distribution includes a makefile. It is located in the <install>\build directory and intended to be run from there.

The Boost and cURL libraries should have been installed with the Linux Operating System.

The Boost Libraries are obtained from http://www.boost.org. Boost.Jam, an extension of Perforce Jam, is used to build the Boost Libraries. A build executable can be downloaded or it can be built from scratch. Follow the instructions at http://www.boost.org/more/getting_started.html using the *install* method.

For reference, the following Boost Libraries are used in the Sample Application Server:

- Bind — Generalization of the std::bindNxx functions
- Format — Format arguments using a format-string to an internal stream
- Function — Function object wrappers
- Smart Pointers — Automatic ownership and lifetime management
- Threads — Portable threading and synchronization
- Utility — Miscellaneous features including a non-copyable base class

The cURL library is obtained from http://curl.haxx.se/ and built using the supplied project files.

To build the application, change the current directory to <install>\build, and run either *make debug* or *make release*. The command *make clean* will remove all intermediate and executable files for both debug and release.

Acronyms

DTMF	Dual Tone Multi-Frequency
HMP	Host Media Processing
IETF	Internet Engineering Task Force
IP	Internet Protocol
IVR	Interactive Voice Response
MMK	Multimedia Kit
MMP	Multimedia Platform
MSML	Media Server Markup Language
OS	Operating System
PowerMedia HMP	Dialogic® PowerMedia™ Host Media Processing Software
PowerMedia XMS	Dialogic® PowerMedia™ Extended Media Server
PSTN	Public Switched Telephone Network
RTP	Real-time Transport Protocol
SIP	Session Initiation Protocol
SRL	Standard Runtime Library
URI	Uniform Resource Identifier
XML	Extensible Markup Language

For More Information

Dialogic®_MSML Media Server Software User's Guide
http://www.dialogic.com/manuals/docs/msml_mediaserver_userguide_v9.pdf

Dialogic® PowerMedia™ Extended Media Server and Dialogic® Host Media Processing Software Release 4.1LIN Documentation
http://www.dialogic.com/manuals/default.htm

Sample code
http://www.dialogic.com/en/Products/media-server-software/download/xms-demos.aspx

Session Initiation Protocol
http://www.ietf.org/rfc/rfc3261.txt

SIP INFO Message
http://www.ietf.org/rfc/rfc2976.txt

Media Server Markup Language (MSML) RFC
http://tools.ietf.org/html/rfc5707

Boost Libraries
http://www.boost.org

cURL Libraries
http://curl.haxx.se/

Dialogic®

www.dialogic.com