**PowerMedia XMS RESTful C# demo**

**Introduction**

This article showcases using .NET 3.5 framework with C# programming language for creating HTTP RESTful applications for controlling Dialogic® PowerMedia™ Extended Media Server (XMS).   Although the demo demonstrates a simple feature set, like inbound and outbound calls, playing and recording audio data, it's composition allows for it to be enhanced with more functionality.

**Program Features**

The PowerMedia XMS REST demo is a GUI C# demo written to work with PowerMedia XMS. The demo currently has the following functionality:

* Controls PowerMedia XMS resources for making and receiving SIP audio calls.
* Plays and records audio files.
* Implements asynchronous HTTP processing on answering and releasing SIP calls to minimize delays and improve performance.
* Uses XMS XSD schema to build XML message payload from call-related C# class objects and deserialize incoming XML text to objects.

The application is ready to receive incoming calls at any time, as long as the server has free call resources. Simultaneously, a user can make an outbound call from XMS using demo’s GUI. Upon connection the demo plays or plays/records audio files; although the call flow is currently hardcoded, the application provides means for developing more complex call scenarios.

**Code Composition**

The program consists of three threads. The main thread (GUI form), event processing thread, and a thread for handling HTTP responses in asynchronous manner. Below is a brief description of main classes and their roles in the application.

* XmsDemoForm:Form : Static (singleton in terms of C++). User interface for controlling the application. Allows users to specify IP address/port of the PowerMedia XMS system, start/stop the program and make outbound calls. Contains a list box, where top-level messages, errors, and events are displayed.
* XmsInterface : Static. Utility class, helping to establish connection to the server, create HTTP requests and read responses from the PowerMedia XMS. Instantiated in a Form class at start-up. Carries PowerMedia XMS connection information.
* EventHandler : Static. Launched by the main thread at initialization time, creates event handler. Then spawns a thread which waits for incoming PowerMedia XMS events in a loop. When an event arrives, the handler defines its type (call, conference, keep alive) and passes data to consumers for processing.

An overview of the program is detailed below:

1) Creating event handler by sending POST to PowerMedia XMS

HttpWebRequest l\_request = XmsInterface.CreateRequest("POST", "eventhandlers", "");

Upon success, XMS responds with status code 201 Created:

HttpWebResponse l\_response = (HttpWebResponse)l\_request.GetResponse();

if (l\_response.StatusCode != HttpStatusCode.Created)

{

// process negative response here

}

 2) Launching event reader in a separate thread

The XML payload of the response is getting deserialized to the eventhandler\_response class object, and the EventHandler fills up the handler ID and starts the event processing thread:

Type tp = m\_ws.Item.GetType();

switch (tp.Name)

{

case "eventhandler\_response":

eventhandler\_response l\_ehr = (eventhandler\_response)m\_ws.Item;

EventHandler.m\_href = l\_ehr.href;

EventHandler.m\_hId = l\_ehr.identifier;

m\_eventThread = new Thread(EventHandler.EventThread);

m\_eventThread.IsBackground = true;

m\_eventThread.Start(); break;

}

In the event thread routine, the program first sends a GET request to XMS, which causes XMS to start streaming chunked XML data to the application:

HttpWebRequest l\_request = XmsInterface.CreateRequest("GET", "eventhandlers", m\_hId);

Then the data reader is started in the loop:

HttpWebResponse l\_response = (HttpWebResponse)l\_request.GetResponse();

…

StreamReader l\_evtStream = new StreamReader(l\_response.GetResponseStream());

string line = "";

while (m\_isRunning)

{

{ try

{

line = l\_evtStream.ReadLine(); // skipping unreadable

line = l\_evtStream.ReadLine(); // reading event length

if (line == null || line.Length < 1) // may happen upon exit

{

continue;

}

}

catch (IOException ex)

{

continue;

}

int buf\_length = Int32.Parse(line, NumberStyles.AllowHexSpecifier); // parsing hex to int

char[] r\_buf = new char[buf\_length];

l\_evtStream.Read(r\_buf, 0, buf\_length);

string str = new string(r\_buf);

Logger.Log(str, "", false);

if(str.Length > 0) // not to pass empty payload

ProcessEvent(str); // dispatches events to consumers

}

}

The Read() operation is blocking when reading junked input, this is why a separate thread is needed for the event reader.

3. Dispatching Events to Consumers

The ProcessEvent() method defines the type of the signaling resource and sends the event for further processing to the owner:

Type tp = l\_ws.Item.GetType();

@event l\_event = (@event)l\_ws.Item;

switch (l\_event.resource\_type)

{

case "call":

CallManager.ProcessEvent(l\_event);

break;

case "conference":

ConfManager.ProcessEvent(l\_event);

break;

case "keepalive":

Logger.Log("Keepalive from XMS", false);

break;

default:

Logger.Log("ERROR - Unknown event type", false); break;

}

**OTHER KEY CLASSES**

* CallManager : Static. Answers and makes calls. Maintains a hash table of all active calls, advances call states according to the events and commands, implements a call flow. Interacts with EventHandler, GUI and XmsCall classes.
* ConfManager : Static. Reserved for implementation.
* Logger Static. Simple logger, prints HTTP – XML messages to a file and topmost messages and errors to the list box on the form.
* XmsCall : Represents call objects. Stateless, run in a context of event thread. Provides methods for answering and making calls, play-record, collecting digits etc. Since some call control operations, like SIP answer or SIP teardown are not atomic and may be delayed by network, this class implements asynchronous HTTP interactions on such procedures, so the event thread is never blocked. This is done with BeginGetResponse/EndGetResponse .NET HTTP API.

 Below you will find an overview of XmsCall class and use of the .NET HTTP API:

1) Call manager receives an event indicating a new incoming call. It creates a new XmsCall object and sends an Answer command to it:

// CallManager:

private static void OnOffered(ref XmsCall a\_call)

{

a\_call.CallState = XmsCall.e\_CallState.STATE\_OFFERED;

a\_call.CallDirection = XmsCall.e\_CallDirection.Incoming;

addCall(ref a\_call); // adding the new call to the hash table

if (a\_call.Answer() != 0) // invoking ASYNC Answer procedure

{

a\_call.Drop();

deleteCall(a\_call);

}

}

2) The XmsCall object initiates an answer procedure and returns control to the thread immediately:

// XmsCall:

m\_request = XmsInterface.CreateRequest("PUT", "calls", this.CallId);

using (Stream l\_requestStream = m\_request.GetRequestStream())

{

XmlSerializer x = new XmlSerializer(typeof(web\_service));

x.Serialize(l\_requestStream, l\_ws); // building XML doc to send Answer command

}

// now waiting for a status from the server, the API below returns immediately

IAsyncResult asyncResult = m\_request.BeginGetResponse(new AsyncCallback(onAnswer), this);

// When the XMS response arrives, the onAnswer callback will be invoked by .NET

3) When the operation completes, the .NET invokes a callback indicating end of task. The XmsCall object then raises an unnamed .NET event which can be consumed by any listener in this address space, in this case, by the CallManager:

private static void onAnswer(IAsyncResult a\_result)

{

XmsCall l\_call = (XmsCall)a\_result.AsyncState;

HttpWebRequest l\_request = null;

HttpWebResponse l\_response = null;

try

{

l\_request = l\_call.m\_request; l\_response = (HttpWebResponse)l\_request.EndGetResponse(a\_result);

Logger.Log(XmsInterface.ResponseToString(l\_response), "", false);

if (l\_response.StatusCode != HttpStatusCode.OK)

. . .

l\_call.CallEvent.Set(); // signal to listeners about completion of ASYNC operation

4) The CallManager runs a thread where it waits for completion events from all call objects it owns.

Thread l\_callCompletionThread = new Thread(onAsyncCallEvent);

l\_callCompletionThread.Start();

When one of the calls raises an event, the CallManager defines the call state and acts accordingly:

foreach(XmsCall l\_call in l\_callList)

{

if (l\_call.CallEvent.WaitOne(1))// completion event on this call

{

switch(l\_call.CallState)

{

case XmsCall.e\_CallState.STATE\_FAIL:

deleteCall(l\_call); // delete ther call

break;

case XmsCall.e\_CallState.STATE\_CONNECTED:

OnAnswer(l\_call);

break;

case XmsCall.e\_CallState.STATE\_IDLE:

OnDrop(l\_call);

break;

}

}

}

**Multithreading Considerations**

The three threads comprising the demo may need to access a sharable resource, like a call hash table or ListBox GUI control at same time, which is not safe. To synchronize such tasks, the program uses mutex objects and delegates. For example, each time the hash table needs to be updated, the code first makes sure that there is no one else currently working with the table, and only then proceeds:

private static void deleteCall(XmsCall a\_call)

{

m\_mutex.WaitOne();

try

{

m\_callTable.Remove(a\_call.CallId);

} catch (Exception ex)

{

Logger.Log("{ERROR: Cannot remove a call from call table: " + ex.Message, true);

}

m\_mutex.ReleaseMutex();

}

Another example of handling shared resources is accessing a sharable GUI control from a thread other than the control’s creator. In this demo example, this happens when the event thread needs to print something to the main window. .NET does not allow such operation directly. Instead, the code defines a delegate – in this context, the thread-safe event-processing method, and invokes it (sends the event to the XmsDemoForm class) when needed:

When an object calls the WriteMessage() method of XmsDemoForm class to update the list box, the form first checks whether the request is made from the same thread where the list box was created, and, if not, it stores the data to print and returns a negative response to the caller. The caller then invokes a delegate to complete the printing:

public int WriteMessage(string a\_message, bool a\_request)

{

m\_request = a\_request;

m\_txtMessage = a\_message;

if (txtRequest.InvokeRequired) // if it is safe to print directly?

{

return -1; // the calling thread will try to invoke the delegate

}

ToScreen();// otherwise go ahead and add the message to a list box return 0;

}

**Building and Running the Demo**

The demo was developed with VS 9.0 (Microsoft Visual Studio 2008) and does not use any external references besides those provided by the .NET 3.5 framework. Some definitions around HTTP functionality may be incompatible with .NET 2.0, so as of October 2012,  .NET 3.5 is recommended. The demo does not require any arguments when started. A user starts the demo, fills in the XMS system's IP address and port, XMS Application ID and press “Start” button. The application now is ready to receive and make calls.

**Product List**

Dialogic® PowerMedia™ Extended Media Server / PowerMedia XMS

**Glossary of Acronyms**

HTTP - HyperText Transfer Protocol

REST - client server architecture using HTTP

.NET Framework -  is a software framework developed by Microsoft that runs primarily on Windows

C# - programming language used within .NET Framework