Dialogic®

Tech Note

Dialogic® PowerMedia™ Extended Media Server (XMS) RESTful C# Demo

NETWORK FUEL™

Introduction

This article showcases using Microsoft's .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, its composition allows for it to be easily enhanced with more functionality.

Program Features

The PowerMedia XMS RESTful demo is a GUI-based 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 to de-serialize incoming XML text to objects.

Dialogic® PowerMedia™ Extended Media Server (XMS) RESTful C# Demo

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 the 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;
       }
       // parsing hex to int
       int buf_length = Int32.Parse(line, NumberStyles.AllowHexSpecifier);
       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 to the owner for further processing:

```
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 : Static. 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 is 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. 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.

Glossary of Acronyms

HTTP - HyperText Transfer Protocol

REST – Representational State Transfer: a client/server architecture using HTTP

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

C# - programming language used within .NET Framework

Dialogic®

www.dialogic.com

Dialogic Inc
1504 McCarthy Boulevard
Milpitas, California 95035-7405
USA

All contents of this document are furnished for informational use only and are subject to change without notice and do not represent a commitment on the part of Dialogic Inc. and its affiliates or subsidiaries ("Dialogic"). Reasonable effort is made to ensure the accuracy of the information contained in the document. However, Dialogic does not warrant the accuracy of this information and cannot accept responsibility for errors, inaccuracies or omissions that may be contained in this document.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH DIALOGIC® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in certain safety-affecting situations. Please see http://www.dialogic.com/company/terms-of-use.aspx for more details.

Due to differing national regulations and approval requirements, certain Dialogic products may be suitable for use only in specific countries, and thus may not function properly in other countries. You are responsible for ensuring that your use of such products occurs only in the countries where such use is suitable. For information on specific products, contact Dialogic Inc. at the address indicated below or on the web at *www.dialogic.com*.

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic's legal department at 6700 de la Cote-de-Liesse Road, Suite 100, Borough of Saint-Laurent, Montreal, Quebec, Canada H4T 2B5. **Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.**

Dialogic, Dialogic Pro, Dialogic Blue, Veraz, Brooktrout, Diva, BorderNet, PowerMedia, ControlSwitch, I-Gate, Mobile Experience Matters, Network Fuel, Video is the New Voice, Making Innovation Thrive, Diastar, Cantata, TruFax, SwitchKit, Eiconcard, NMS Communications, SIPcontrol, Exnet, EXS, Vision, inCloud9, NaturalAccess and Shiva, among others as well as related logos, are either registered trademarks or trademarks of Dialogic Inc. and its affiliates or subsidiaries. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 6700 de la Cote-de-Liesse Road, Suite 100, Borough of Saint-Laurent, Montreal, Quebec, Canada H4T 2B5. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

The names of actual companies and products mentioned herein are the trademarks of their respective owners. 03/13