Dialogic®

XMS Java Client Library Programming Guide

Programming Guide

Date DocNumber

Copyright and Legal Notice

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 9800 Cavendish Blvd., Suite 500, Montreal, Quebec, Canada H4M 2V9. **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, Diva ISDN, Mobile Experience Matters, Making Innovation Thrive, Video is the New Voice, VisionVideo, Diastar, Cantata, TruFax, SwitchKit, SnowShore, Eicon, Eiconcard, NMS Communications, NMS (stylized), SIPcontrol, Exnet, EXS, Vision, PowerMedia, PacketMedia, BorderNet, inCloud9, I-Gate, ControlSwitch, NaturalAccess, NaturalCallControl, NaturalConference, NaturalFax 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 9800 Cavendish Blvd., Suite 500, Montreal, Quebec, Canada H4M 2V9. 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.

<Include only if open source product(s) in document>This document discusses one or more open source products, systems and/or releases. Dialogic is not responsible for your decision to use open source in connection with Dialogic products (including without limitation those referred to herein), nor is Dialogic responsible for any present or future effects such usage might have, including without limitation effects on your products, your business, or your intellectual property rights.

Table of Contents

Revision History

Revision	Release date	Notes
0.1	11-Jul-2012	Initial Draft
0.2	26-Jul-2012	Updates to error handling
Last modified: 26-Jul-2012		

Refer to www.dialogic.com for product updates and for information about support policies, warranty information, and service offerings.

1. Overview

This publication provides guidelines for building a client application that interfaces with the Dialogic® PowerMedia™ Extended Media Server using a java library , i.e. XMS java client library.

The Dialogic® PowerMedia™ Extended Media Server (PowerMedia XMS) is a powerful software media server that enables standards-based, real-time multimedia communications solutions for mobile and broadband environments. For more information on PowerMedia XMS please visit http://www.dialogic.com/en/products/media-server-software/xms.aspx.

The XMS java client library is consider demo code and allows a programmer to build asynchronous and synchronous sample applications in the java environment. The resulting program can be executed on nearly any device that supports the java runtime environment.

The XMS java client library abstracts the implementation details of the protocol interface, e.g. RESTful, MSML, … etc. XMS java client library version 1.0 supports RESTful interface. Future versions will expand the protocols supported.

2. XMS Java Client Library Architecture

The XMS Java Client Library is comprised of three main components:

- **XMS Connector** – this establishes and manages commands and events being sent to the PowerMedia XMS server.
- **XMSCall** – object that provides call and media functionality, e.g. ability to make a call, drop a call, call progress analysis, play a media file, record media, … etc..
- **XMSConf** – object that provide conferencing capabilities including

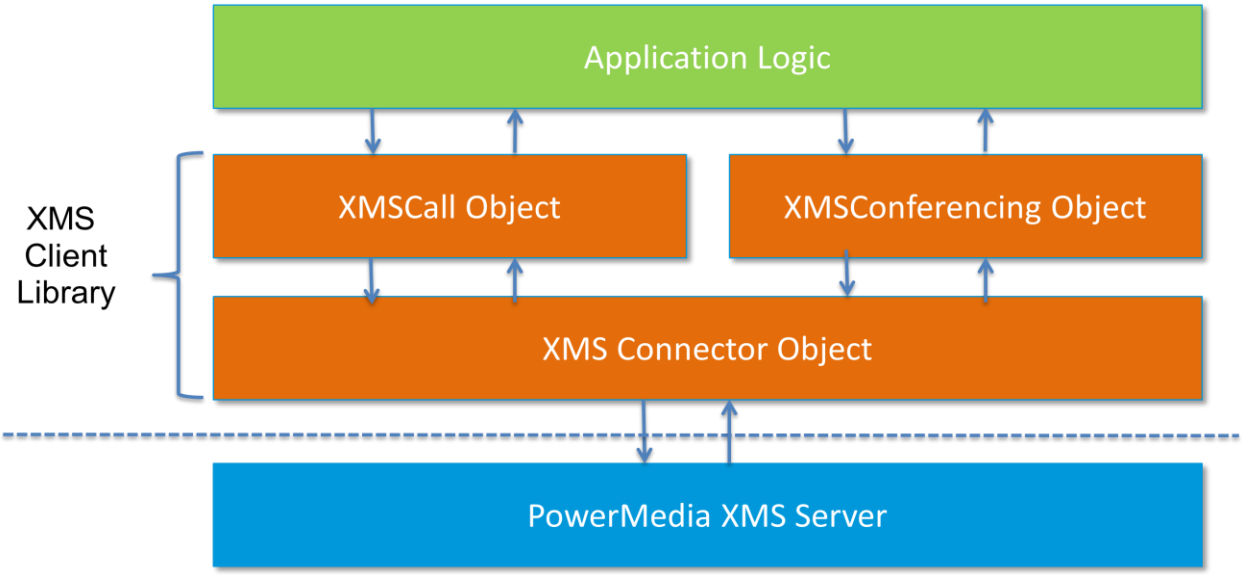
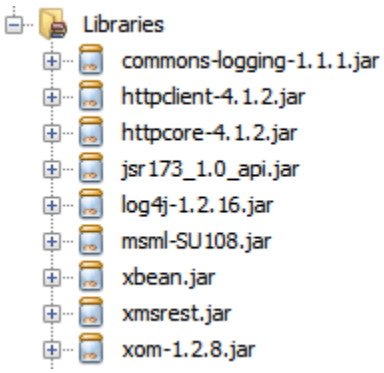


Figure 1 - High Level Application Overview

3. Setup and using the XMS Java Client Library

The libraries first needed to be added to the project being worked. Below is a snapshot of the files being used:



Once added by sure to add the following import statement to the project:

```
Import com.dialogic.XMSClientLibrary.*;
```

4. Object Concepts

XMS Connector Object

The XMS connector object establishes a connection to a specific PowerMedia XMS server. The XMS connector can take a configuration file to set up information on technology type and connection information to the PowerMedia XMS server.

Instantiating the XMS connector object

Using an object factory the programmer needs to instantiate a version of the object and set up configuration information for the instantiated object to use:

```
XMSObjectFactory myFactory = new XMSObjectFactory();

XMSConnector myConn = myFactory.CreateConnector("xmsconn.xml");
```

Configuration file that sets up defaults such as server address, technology type, … etc.

XMS Connector Configuration File

The configuration file is an XML based file that contains information on connection to be set up to a specific PowerMedia XMS server. There are specific tags set up

```
<xmsconfig>

<techtype>REST</techtype>

<baseurl>http://192.168.0.1:81/default/</baseurl>

<appid>app</appid>

</xmsconfig>
```

Tag attributes:

- <*xmsconfig*> : encapsulates the configuration information.
- <*techtype*> : specifies the technology type being used with the PowerMedia XMS Server. 'REST' is the current technology environment used.
- <*baseurl*> : (REST) base address and port address.

- <*appid*> : PowerMedia XMS allows you to define discrete application IDs (appid) so that multiple applications may be simultaneously run on a single XMS server.

NOTE: The appid is pre-defined in the Routing Screen on the PowerMedia XMS web-based GUI which is used for post-operating system installation and configuration tasks. Refer to the Dialogic® PowerMedia™ XMS Installation and Configuration Guide for detailed information about the PowerMedia XMS Admin Console. There, new appids may be added or unwanted appids can be removed. It also allows pattern matching on SIP URIs to direct calls to a specific appid.

XMS Call Object

The XMS Call object is used manage call control and media functions for the PowerMedia XMS server.

Thefollowing are key methods for the XMSCall object. It is not a comprehensive list and please refer to the javadoc API documentation for full details and usage.

- **Initialize**() : Initialize the XMSConnector
- **Makecall**() : Make a call to the specified destination
- **Join**() : Join / Route 2 Calls together
- **Record**() : Record a file
- **PlayRecord**() : This file will start a Play and a record at the same time
- **Play**(java.lang.String a_file) :Play a file
- **Dropcall**() : This will terminate an call. The call could be in progress or in the middle of being established.
- **Waitcall**() : Puts the call in a state waiting for a new inbound call
- **Answercall**() : This Answer an incoming call.
- **CollectDigits**() : Collect DTMF digits
- **PlayCollect () :** play and collect digits
- **EnableEvent**() : enable the Callback for the specified event
- **DisableEvent**() : Disable the Callback for the specified event

Using XMSCall Object Synchronous Mode

The following describes instantiating an XMSCall object:

```
XMSObjectFactory myFactory = new XMSObjectFactory();  // create xmsconn factor

XMSConnector myConn = myFactory.CreateConnector("xmsconn.xml");

myConn.Initialize();

XMSCall myCall = myFactory.CreateCall(myConn);
```

The above sample will now have created an XMSCall object. Note that XMSCall object supports synchronous mode of operation and an asynchronous callback model. A simple call flow example is shown below:

```
XMSReturnCode RC;

XMSObjectFactory myFactory = new XMSObjectFactory();

XMSConnector myConn = myFactory.CreateConnector("xmsconn.xml");

myConn.Initialize();

XMSCall myCall = myFactory.CreateCall(myConn);

RC = myCall.Waitcall();

System.out.println("Rcvd Call - Waitcall completed with return Code " + RC);

RC=myCall.Play ("verification/test_recording.wav");

System.out.println("Play record complete " + RC);

RC = myCall.Dropcall();

System.out.println("DropCallcompleted with return Code " + RC);
```

This basic sample creates an XMS connector object, then creates and XMS call object. It then waits for a SIP call to arrive to the PowerMedia XMS server. The call is automatically answered and wav file is played to the caller. The call is then dropped.

Setting Options

The XMS client library defaults options for the methods available. A developer can override those options. The javadoc will provide full information on usage and available parameters. The following is a list of available options.

- XMSAnswercallOptions
- XMSCollectDigitsOptions
- XMSMakecallOptions

- XMSPlayCollectOptions
- XMSPlayRecordOptions
- XMSWaitcallOption
- XMSRecordOptions

The Following Example Shows How To Set The Waitcall Options To Specify A Video Call Is Wanted:

```
XMSCall myCall = myFactory.CreateCall(myConn);

myCall.WaitcallOptions.SetMediaType(XMSMediaType.VIDEO);

RC = myCall.Waitcall();
```

Using XMSCall Object Asynchronous Mode

The XMSCall library also supports the use of asynchronous callback programming model. Asynchronous more is more conducive to building more scalable applications. One can enable a callback function for a specific event or enable a callback function for all possible events.

An example of a generic handler is below. A programmer would get notified of an event and the decide what action to take based on that event.

Below you will set the set up the handler to look for all events:

```
        XMSObjectFactory myFactory = new XMSObjectFactory();
        XMSConnector myConnector = myFactory.CreateConnector("XMSConnectorConfig.xml");
        XMSCall myCall = myFactory.CreateCall(myConnector);
        MyCallbacks myCallback = new MyCallbacks();

        //Enable all events to go back to my event handler
        myCall.EnableAllEvents(myCallback);

        //Wait for an inbound call and start the state machine
        myCall.Waitcall();

       /* At this point event handler thread will process all events so this thread
          just waits to complete */

       while(!myCallback.isDone)   {
           Sleep(1000);
       }
```

Below you will see the callback function implemented to handler

```
public class MainCallbackTester implements XMSEventCallback{
    public void ProcessEvent(XMSEvent a_event) {
        // process event
        switch(a_event.getEventType()){

            case CALL_CONNECTED:
               System.out.println("Call Is connected, Playing file");
                a_event.getCall().Play("verification/verification_intro.wav");
                break;

            case CALL_PLAYCOLLECT_END:
                System.out.println("CALL_PLAYCOLLECT_END, dropping call.");
                a_event.getCall().Dropcall();
                break;

            case CALL_PLAY_END:
                System.out.println("Play is done Disconnecting");
                a_event.getCall().Dropcall();
                break;

            case CALL_DISCONNECTED:
                System.out.println("Call Is DISCONNECTED!");
                break;

            default:
                System.out.println("Unknown Event Type!!");
        }
    }


}
```

In a similar fashion, one can set up a specific callback function for specific event:

```
            XMSObjectFactory myFactory = new XMSObjectFactory();

            XMSConnector myConn = myFactory.CreateConnector("xmsconn.xml");

            XMSCall myCall = myFactory.CreateCall(myConn);

            myCall.EnableEvent(XMSEventType.CALL_OFFERED, CallOfferedCallBack);
```

Event Types

Below you will find a list of events that will be used and delivered by the XMS Client library.

- CALL_CONNECTED - call is connected via Makecall or WaitCall.
- CALL_DISCONNECTED – call has been disconnected, i.e. terminated.
- CALL_OFFERED - newly arrived inbound call is offered to the user application.
- CALL_PLAY_END – play has finished, either play complete or play stopped.
- CALL_COLLECTDIGITS_END - digit collection has completed due one of the following reasons; maxdigits, timeone, tone, or action stopped
- CALL_PLAYCOLLECT_END - play completed due one of the following reasons; maxdigits, timeone, tone, or action stopped CALL_PLAYRECORD_END -
- CALL_RECORD_END – record completed due one of the following reasons; maxdigits, timeone, tone, or action stopped

Checking Event Data

Depending on the event, the programmer can obtain additional information regarding the event that is returned. The key functions that will be used would be :

- getEventType() – returns the event type.
- getReason() – returns the reason for the termination, e.g. max-digits, … etc..
- getData() - obtains any event specific data such as the digit string.
- getInternalData() - Obtain any low level internal information or event contents NOTE : This String will likely contain technology specific data.

An example is shown below:

```
public class MainCallbackTester implements XMSEventCallback{
    public void ProcessEvent(XMSEvent a_event) {
        switch(a_event.getEventType()){
            case CALL_CONNECTED:
                System.out.println("Call Is connected w/ reason: " + a_event.getReason());
                System.out.println("Playing file");
                a_event.getCall().Play("verification/verification_intro.wav");
                break;
            case CALL_PLAYCOLLECT_END:
                System.out.println("CALL_PLAYCOLLECT_END w/ reason: " + a_event.getReason());
                System.out.println("Dropping call.");
                a_event.getCall().Dropcall();
                break;
            case CALL_PLAY_END:
                System.out.println("Play done  w/ reason: " + a_event.getReason());
                System.out.println("Disconnecting");
                a_event.getCall().Dropcall();
                break;
            case CALL_DISCONNECTED:
                System.out.println("Call disconnected w/ reason: " + a_event.getReason());
                break;
            default:
                System.out.println("Unknown Event Type!!");
        }
    }
```

5. Checking for Function Success or Failure

The XMS Java Client library function will typically returned back information on if the command was sent to the PowerMedia XMS via the object XMSReturnCode. Please check the associated javadoc on the specific function call since not all functions will utilize the XMSReturnCode as return code.

The XMSReturnCode from a function call will return one of the following ENUM types:

- SUCCESS – function executed with no issues.
- FAILURE – function failed to execute.
- INVALID_STATE – functioned requested in not supported in the current state of the object.
- NOT_IMPLEMENTED – the function requested has not been implemented.

An example of the using XMSReturnCode is listed below

```
XMSReturnCode myReturnCode;  // defines the object variable for return code


//Instantiate the connector
XMSObjectFactory myFactory = new XMSObjectFactory();
XMSConnector myConn = myFactory.CreateConnector("xmsconn.xml");

//Initialize the connected
myConn.Initialize();

XMSCall myCall = myFactory.CreateCall(myConn);

// Make the call.
myReturnCode = myCall.Makecall(a_dest);

System.out.println("Makecall completed with return Code " + myReturnCode);
```