JAVA CLIENT MANUAL

Beam Jive Consulting

Table of contents

1 Introduction

1.1 About the document

This document will show you how to work with the Offbeat Java client library. All classes will be described and simple examples provided.

It is recommended to read also the Offbeat Manual, which describes the protocol and the concepts of the server side programming with the Offbeat server.

1.2 What can I do with the Offbeat server?

You can connect to the Offbeat server from Java applications and applets and from Flash applications. The XML based protocol makes it possible to have only one server-side application and use it from both Flash and Java.

- Create data driven applications
- Create communication applications (chats, whiteboards…)
- Create controlling and monitoring applications
    - computer monitoring and remote control
    - software monitoring and remote controlling
    - device monitoring and controlling
    - real-time statistics
- Any kind of distributed systems

1.3 What can I not do with it?

It is not possible to create video or voice communication applications with the Offbeat server. For file uploads and file downloads, some HTTP-server should be used.

1.4 How does it work?

1.4.1 Communication

The Offbeat server is a TCP socket server that uses XML as the communication protocol. In a TCP socket connection, the client-server connection, unlike in HTTP, is continuously open. This means that the client may receive data from the server as push messages. This makes it possible to create real-time communication applications, such as monitoring apps, chats and whiteboards.

The response times are very fast when there is no latency of creating the connection for each request. A typical round-trip time (send request -> process request -> receive response) for an Offbeat request is only a few milliseconds. The fastest round-trip times in the tests were as small as one millisecond. This means that the client may send up to 1000 requests in one second (depends on the hardware and the application design).

1.4.2 The communication model

The Offbeat communication model is based on a request-response model. A server programmer creates a server application, which consists of one or more Java class files. The Offbeat Java and Flash clients can then call the server applications. It is possible to send variables to the server application. The server application can read the request variables and create the response message dynamically. The response messages are always XML documents.

The Offbeat Java client uses an asynchronous communication model and works almost in the same way than the Flash client. This means that you always define a handler function for the response messages. You can process the response as soon as it is received from the server. There are six types of messages defined in the Offbeat protocol (see the Offbeat manual for more details):

- Requests
- Responses
- Push messages
- Client disconnect messages
- Log messages
- Acknowledgement messages

There are detailed descriptions of each message type in the following chapters.

2 Installing the client library

The Offbeat Java client is located in the <offbeat>/clients/java directory as a jar file (OffbeatClient-vX_X_X.jar). You can use the client library if you have the file in your CLASSPATH environment variable or it resides in the same directory with your application.

One way to make the client library visible, is to copy it to following directory under your Java distribution: <JAVA_HOME>/jre/lib/ext.

3 Compiling an application

To compile a Java application that uses the Offbeat Java client library, you will need to have the com.bjc.offbeat.client package in your CLASSPATH (see chapter 2).

If you don’t have the package set in your CLASSPATH environment variable, you can still compile your application if OffbeatClient-v1_0_0p.jar is available. Use the –classpath switch of the javac tool to specify the location of the library.

```
javac –classpath OffbeatClient-v1_0_0p.jar SimpleChat.java
```

4 How to use the client library

After installing the Offbeat Server successfully, you can start using the client components. First thing you will need to do in your Java application, is to import the library:

```
import com.bjc.offbeat.client.*;
```

Usually only one instance of the client is needed in application. The following code listing shows how to create an instance of the OffbeatClient class:

```
import com.bjc.offbeat.client.*;

// Your own class
public class MyClass
{
   // Class constructor
   public MyClass()
   {
      OffbeatClient ob = new OffbeatClient( this, “127.0.0.1”, 8384 );
   }
}
```

In the previous example we created an instance of the OffbeatClient that will connect to localhost (127.0.0.1) port 8384 when the connect method is called.

The first parameter to the OffbeatClient class constructor is a reference to the object that will contain all the event handler methods. Usually the keyword “this” should be used. We will discuss this later in this document.

5 Connection handling

This chapter will show you how to connect to a server, how to close the connection and how to handle the events correctly.

5.1 Defining host and port

To connect to Offbeat server, you need to know the correct host and port. Host is the IP address or the host name of the computer where the Offbeat server is running. Port is the port number that the server is listening. If the Offbeat server is on the same computer with the client, the host should be 127.0.0.1 or localhost.

```
OffbeatClient oc = new OffbeatClient( this, “127.0.0.1”, 8384 );
```

5.2 Setting the callback methods

The callback methods (event handler methods) should be set before the “connect” method is called. The following code listing shows how to define the callbacks:

```
public class MyClass
{
   private OffbeatClient ob;

   public MyClass()
   {
       ob = new OffbeatClient( this, “127.0.0.1”, 8384 );

       ob.onConnect = “myOnConnect”;
       ob.onClose = “myOnClose”;
       ob.onPushMessage = “myPushHandler”;
       ob.onLogMessage = “myLogHandler”;
       ob.onClientDisconnect = “clientDisconnect”;
   }

   public void myOnConnect( boolean success )
   {
      // The connection is OK
      if( success )
      {
      }
      // Connection failed
      else
      {
      }
   }

   public void myOnClose()
   {
       // Connection was closed
   }

   public void myPushHandler( XML data, String file, StringclientID )
   {
      // Handle push message
   }

   public void myLogHandler( String msg )
   {
      // I got a log message!
   }

   public void clientDisconnect( String clientID, String clientName )
   {
      // A client has disconnected, remove from lists etc...
   }
```

5.3 Opening the connection

When the handler functions have been set, it is time to open the connection. The connection can be opened simply by calling the connect() method. The following example show how to do it:

```
ob.connect();
```

When the connection has been opened, the client will automatically call the handler function specified in the onConnect variable.

5.4 Closing the connection

To close an open connection, the close() method should be called. A call to the close() method will close the connection and call the handler function defined in the onClose property. The following code shows how to call the close() method:

```
ob.close();
```

6 Sending requests

The requests are used to call an application on the server. The connection should be opened before sending the first request. It is possible to pass data to the server application by setting request variables. In the following example we send a basic request to the server:

```
XmlRequest req = ob.newRequest( this, “handler”, “MyApp/MyFile.xma” );
ob.send( req );
```

In the example we sent a request to the application MyApp’s file called MyFile.xma. In the server, there is an application called MyApp that contains a Java class file called MyFile.class. The file extension .xma has to be used when calling the server applications. The first parameter ‘this’ is a reference to the object that contains the event handler method. The second parameter ‘myHandler’ is the function that will be called when the response arrives from the server.

It can be seen, that the connection object is used to create a new request. The method newRequest creates a new com.bjc.offbeat.XmlRequest object that will be converted to XML and sent to the server. The setVar(String name, String value) method of the XmlRequest class can be used to set request variables. The following example shows how set request variables:

```
XmlRequest req = myConn.newRequest( this, “cbk”, “SaveNews.xma” );
req.setVar( “title”, “Offbeat is a server” );
req.setVar( “text”, “Offbeat really is a server!” );
myConn.send( req );
```

In the server application, the request variables can be read, and the news can be saved to a database.

If there is no need to set the handler function, the parameter may be null. This is the case in many chat-like applications.

7 Receiving messages

7.1 General

There are five different kind of messages that a Offbeat Java Client can receive. The most common type of message to receive is a response message. Response messages are generated on the server on your request. Push messages come from other clients.

7.2 Receiving normal response messages

Normal response messages a reply messages to your requests. The response messages are generated on the server by server applications. The response messages are always XML documents. The response may contain any data. The response message may contain for example database query results or a server generated timestamp or what ever you decide to add to the response message on the server.

A response message is handled in the handler function that was defined in the request. The following example shows how to send a request and how to handle the response:

```
// Send some request
public void sendRequest()
{
   XmlRequest req = myConn.newRequest( this, “myHandler”, “Test1.xma” );
   myConn.send( req );
}

// The handler method
public void myHandler( XML response, int errors )
{
   if( errors == 0 )
   {
          // Do something with the data ...
   }
   // There are errors, handle correctly
   else
   {
          // Do some error handling (loop though the errors)
   }
}
```

The first parameter to the handler function contains the response message. The datatype of the response parameter is com.bjc.offbeat.client.XML. The XML class comes with the Offbea Java client and we will show how to use it later in this document.

The second parameter, errors, contains the number of errors that occurred when generating the response on the server. If errors variable is zero, the response is OK and it can be processed. If there is one or more errors in the response, the error can be handled and an appropriate error message can shown to the user. There are two errors in the following response message:

```
<?xml version="1.0" encoding="UTF-8"?>
<MSG TYPE="0" FILE="metafile.xma" REQUEST_ID="123123" ERRORS="2">
```

```
   <ERROR CODE="0">Error description 1</ERROR>
   <ERROR CODE="3">Error description 2</ERROR>
</MSG>
```

7.3 Receiving push messages

The push messages are also generated by the server. The push messages are always sent by some other client that is using the same application. The push messages are also XML documents that can be handled in the same manner as the normal response messages. The following example show how to handle push messages:

```
// Set the push message handler
myConn.onPushMessage = “myPushHandler”;

// ... open connection etc ...

// The handler function
public void myPushHandler( XML data, String file, String clientID )
{
    if( file.equals( “myServerFile1.xma” ) )
    {
        // Do something with the message
    }
}
```

By looking the code listing above, it can be seen that there are three variables that can be used to handle the push message. The first parameter, data, contains the push message data in XML object. **The second parameter, file, can be used to check which server file generated the push message.** The third parameter, clientID, is the unique public client ID of the user who sent the message. If you have got a list of users from the server earlier, the client ID can be used to resolve for example the name of the sender.

7.4 Receiving client disconnect messages

The client disconnect messages are automatically generated by the Offbeat server when a client disconnects (or loses the connection). **To receive the client disconnect messages, client has to register with an application.** This can be done on the server application by calling: application.register(String name). This feature can be used to remove disconnected clients from lists etc. The following example shows how to set the handler function and how to handle the incoming disconnect messages:

```
myConn.onClientDisconnect = “myOnClientDisc”;

public void myOnClientDisc( String clientID, String clientName )
{
   // Remove client from lists etc…
}
```

7.5 Receiving log messages

A client can receive log messages that are generated in the server application. On the server, the method user.receiveLogMessages(true), has to be called. This makes it easy to create simple monitoring features to applications. The following example shows how to set the property and how to define the handler function for incoming log messages:

```
// Set the callback for log message handler
myConn.onLogMessage = "myOnLog";

// Define the handler function
public void myOnLog( String msg )
{
  // I received one log message
}
```

7.6 Receiving acknowledgement messages

Acknowledgement messages are generated and sent by the server when the response message is not sent to the client who did the request. For example, if you send a chat message to another user, the server application probably will not send the same message back to you. The server sends an acknowledgement message back to you.

You do not have to do anything with the acknowledgement messages, they are used internally in the Offbeat clients. When you use the Offbeat Client debug feature, you may see acknowledgement messages coming from the server.

7.7 Handling errors

Only the normal response messages may contain errors. **The push messages will not be sent if an exception or error occurs in the server application.**

8 Using the XML and Node classes

8.1 General

The XML and Node classes in the com.bjc.offbeat.client package have been added to simplify the application development. When you use the XML class, you don’t need to import any additional parsers to your Java project. The XML class basically wraps one or more Node objects and reflects the original XML document.

These classes may look familiar to all Flash developers they are very close to the ActionScript XML object.

8.2 Using the classes

It is very easy to use the XML and Node classes. When you create a new XML object, you pass in the XML data as String:

```
XML myXML = new XML( “<node attr=”value”><second>data</node>” );
```

Now, we would need a reference to the first node of the document, which is in this case the <node> tag. You can use the “**firstChild**” variable to access the first node of the document:

```
System.out.println( “My node name: “ + myXML.firstChild.getName() );
```

And to print out the value of the attribute “attr”, you would write:

```
System.out.println( “At: “ + myXML.firstChild.getAttribute( “attr” ) );
```

Next thing we need to know is how to loop through all the childNodes of the node tag. We can use the “**length**” variable of the node:

```
for( int i = 0; i < myXML.firstChild.length; i ++ )
{
   System.out.println( “node: “ + i );
}
```

Now it is important to notice that you can use the XML object with the Node object. For example, the myXML.firstChild in the previous example is indeed a Node object.

Each of the node objects have also an array of Node objects called “**childNodes**”. To loop through all the nodes inside of the root node, you could write:

```
for( int i = 0; i < myXML.firstChild.length; i ++ )
{
   Node currentNode = myXML.firstChild.childNodes[i];
   System.out.println( “Node: “ + currentNode.getName() );
}
```

8.3 You can use also your own XML parser

It is also possible to use any other XML parser than the built-in XML object. In the earlier chapters, you have noticed that the XML documents to the responses and push messages are passed as com.bjc.offbeat.client.XML. If you want to pass the XML document as a String to the event handler methods, just set the “mode” variable of the OffbeatClient object to false:

```
OffbeatClient obc = new OffbeatClient( this, “127.0.0.1”, 8384 );
obc.mode = false; // true would use the XML objects...
```

If the mode has been set to false, you should also change the event hander method signatures:

```
// The handler method - the first parameter is String
public void myHandler( String response, int errors )
{
   if( errors == 0 )
   {
         // Do something with the data ...
   }
   // There are errors, handle correctly
   else
   {
         // Do some error handling (loop though the errors)
   }
}

// The push message handler method - mode is false...
public void myPushHandler( String data, String file, String clientID )
{
    if( file.equals( “myServerFile1.xma” ) )
    {
        // Do something with the message
    }
}

```

9 com.bjc.offbeat.client.OffbeatClient class

Class that handles the connections, requests and responses. This class is used with the XmlRequest class.

void OffbeatClient(Object baseObject, String host, int port)

The constructor of the class. Creates a new instance of the OffbeatClient class. Can be used in the following way:

```
OffbeatClient myConn = new OffbeatClient( this, “127.0.0.1”, 8384 );
```

Parameters:

baseObject: A reference to the object that contains the event handler methods

host: Host name or IP address of the Offbeat server

port: The Offbeat server port number

Returns:

Nothing

void send(XmlRequest request)

Sends a request to the Offbeat server. The following example shows how the request can be sent through an open connection:

```
// First create a request
myRequest = myClient.newRequest( this, “myCallback”, “theFile.xma” );
// Then send it
myClient.send( myRequest );
```

Parameters:

request: The request object that will be sent to the server

Returns:

nothing

XmlRequest newRequest(Object cbkObj, String cbk, String file)

Get a new request object. The OffbeatClient initializes the request object so that it is ready to be used.

Parameters:

cbkObj: Object that contains the callback method (the second parameter)

cbk: The callback function that will be called on the response

file: Name of the file that is called. The file extension should be .xma.

Returns:

New XmlRequest object

boolean connect()

Connect to the server. When the connection has been established, the OffbeatClient will call the method that is specified in the onConnect property.

Parameters:

None

Returns:

True if the connection can be opened, false otherwise

void close()

Closes the connection to the server. Calls the method that is defined in the onClose property of the OffbeatClient object.

Parameters:

None

Returns:

Nothing

boolean isConnected()

Can be used to check if the connection to the server is open.

Parameters:

None

Returns:

True if the connection is open, false if the connection is closed.

String onPushMessage

A property that can be used to define the function that is called when a push message has been received. The push message handler function should always take three parameters: data:XML, file:String, clientID:String.

String onConnect

With this property it is possible to set a callback function that is called after the connection has been made. The callback function takes one Boolean parameter that tells if the connection was successfully opened or not.

String onClose

By setting this property, it is possible to call a function when the connection is closed. The callback function will be called also when the OffbeatClient.close() method is called.

String onClientDisconnect

Property that can be used to define a callback method that will be called when a client disconnect message has been received. The Offbeat server sends the client disconnect messages automatically to all clients who have registered to the same application as the disconnecting client (See the Offbeat user manual for further details).

String onLogMessage

This property can be set to handle incoming log messages. See the Offbeat user manual to find out more about logging.

int debug

This property can be set to receive debug information from the OffbeatClient class. It produces debug messages by calling the System.out.println method. The default value of the property is 0 (no debugging). The debug can be set to 1 and 2 to receive debug information. The debug level 1 shows only basic information about the events, but the debug level 2 shows also the data that is handled.

String filePrepend

A property that can be used to add text in front of all filenames that are used in the requests. This is handy when the application directory may change. The following example shows how to add a path to all requests:

```
myConn.filePrepend = “myAppDirectory/”;
// Request that will call file “myAppDirectory/myFile.xma”
myRequest = myConn.newRequest( myCbk, “myFile.xma” );
```

10 com.bjc.offbeat.client.XmlRequest class

This class is used only to send new requests to the server. You never call the class constructor, instead you create a new XmlRequest object by calling the newRequest method of the OffbeatClient.

void setVar(String name, String value)

Set a variable to the request.

```
XmlRequest xr = myConn.newRequest( this, “handleResult”, “File.xma” );
xr.setVar( “name”, “John” );
xr.setVar( “age”, “34” );
myConn.send( xr );
```

Parameters:

name: Name of the variable to set

value: Value of the variable to set

Returns:

Nothing

11 com.bjc.offbeat.client.XML class

A class that makes it easier to handle XML documents in Java applications. The XML class wraps XML documents as Java objects.

XML(String document)

The constructor of the class. Creates a new instance of the XML class. Can be used in the following way:

```
String myDocument = new String( “<node>data</node>” );
XML myxml = new XML( myDocument );
```

Parameters:

document: XML document

Returns:

Nothing

Node firstChild

Contains a reference to the first node (root node) of the XML document. Can be used with the Node class to make code look cleaner:

```
String myDocument = new String( “<node>data</node>” );
XML myxml = new XML( myDocument );

Node mynode = myxml.firstChild;
```

12 com.bjc.offbeat.client.Node class

This class is used by the XML class. Represents one node and its subnodes in an XML document. This class should be used only with the XML class, you should never call the constructor of this class.

String getAttribute(String name)

Returns the value of an attribute in the node. Following code listing shows how to get the name attribute of the first node in XML document:

```
String myDocument = new String( “<node name=”tester”>data</node>” );
XML myxml = new XML( myDocument );

System.out.print( myxml.firstChild.getAttribute( “name” ) );
// Prints ‘tester’

// Or...
Node mynode = myxml.firstChild;
System.out.print( mynode.getAttribute( “name” ) );
// Prints ‘tester’
```

Parameters:

name: Name of the attribute to get

Returns:

Value of the attribute, or null if the attribute does not exist

String getValue()

Returns the character data in the node. If the Node has child nodes, they are not included. The following code shows how to read and output the data in the node:

```
String myDocument = new String( “<node name=”tester”>data</node>” );
XML myxml = new XML( myDocument );

System.out.print( myxml.firstChild.getValue() );
// Prints ‘data’
```

Parameters:

none

Returns:

Data in the node as String

Node firstChild

A reference to the first child node. Value is null if the Node does not have any child nodes.

Node[] childNodes

An array of Node objects in this Node object. Can be used to loop through all child nodes. The following code shows the basic way to loop through the nodes:

```
for( int i = 0; i < myXML.firstChild.length; i ++ )
{
   Node currentNode = myXML.firstChild.childNodes[i];
   System.out.println( “Node: “ + currentNode.getName() );
}
```

int length

Number of childNodes in the Node object.

Version history

Version	Date	Author	Description
1.0	12.1.2004	Kai Hannonen	First version
1.1	21.9.2004	Kai Hannonen	Fixed some typos and added the XML and Node class descriptions.