Communication Library for iOS User Guide

Communication Library for iOS User Guide

Printed: July 2011 in (whereever you are located)

Publisher

Midnight Coders, Inc.

Special thanks to:

All the people who contributed to this document, to everyone who has helped us out with the vision for the product, feature suggestions and ideas for improvements. Special thanks to our families for your support, encouragement and patience.

Table of Contents

1 Overview

Communication Library for iOS - is a reusable, native iOS library component enabling developers to quickly and easily connect the iOS applications with server-side technologies. Most of the enterprise and consumer mobile applications require client-server connectivity. The most trivial scenario is when the data is loaded from the server by the client, however, most modern applications call for more advanced integration. This includes instant or real-time data availability, data push and sophisticated, multi-user interaction. The library facilitates all of these scenarios. Using the communication library for iOS developers can easily accomplish the following tasks:

- Integrate iPhone and iPad applications with existing server-side infrastructure which may be developed in Java, .NET or server-side ActionScript.
- Easily leverage server-side APIs and connect to them from the iOS devices.
- Efficiently transfer data from the client application to services running in private data centers on in the cloud.
- Implement real-time data push strategies and deliver real-time updates from Java, .NET or server-side ActionScript code to iOS devices.
- Create multi-user applications where the data can be easily exchanged between multiple application instances.
- Integrate mobile applications using the library with their browser or desktop-based counterparts. This makes it possible to enable users using the mobile application to interact in real-time with the users using the browser or desktop-based version of the same application.

The library provides significant advantages over the competing approaches:

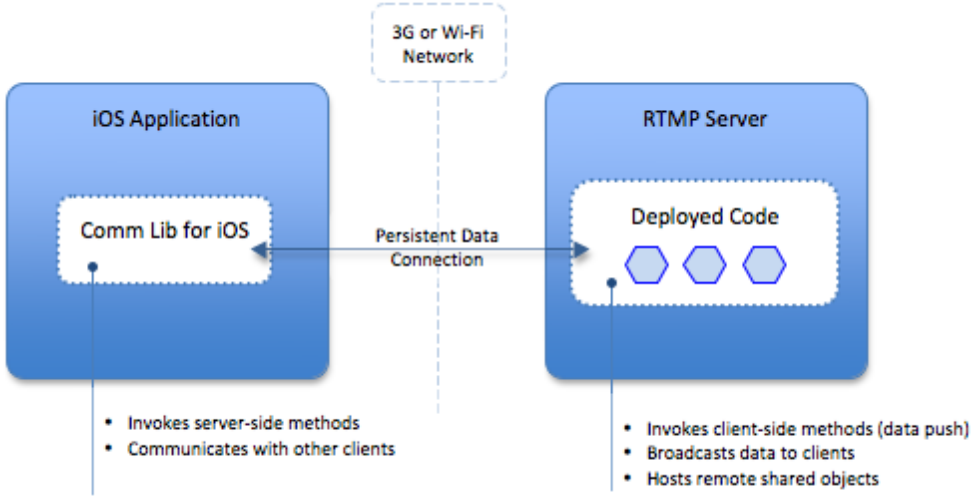
- The on-the-wire protocol implemented by the library is highly optimized and very efficient. This guarantees small payload for the messages which enables efficient communication not only over wi-fi, but also via the 3G networks.
- No vendor lock-in. The library is built in a neutral way without tying the application to a particular server-side technology or vendor.
- Intuitive, well-documented API.

2 Integration Types

Currently the library enables client-server integration via the RTMP protocol. That means to enable client-server communication, the library creates a persistent connection with an RTMP server. The connection is used by both the library and the server to exchange data and handle client-to-server and server-to-client method invocations. There are multiple implementations of the RTMP servers available on the market. See the Supported Servers section for additional information.

Note: A future version of the library will provide an alternative integration type which will enable client-server connectivity via HTTP-based remote method invocation.

The diagram below provides a visual overview of how the library fits into the client application and communicates with an RTMP server:



3 Requirements

The library imposes the following requirements on the iOS applications utilizing it:

- iOS Deployment Target - 4.2 or above
- `Foundation.framework` and `Security.framework` must be added to the list of libraries linked to the binary

To add the library to an Xcode 4 project:

1. Select the project node in Xcode
2. Select the target
3. Switch to the Build Phases view
4. Expand the "Link Binary with LIbraries" panel
5. Click the "+" button and then the "Add Other" button.
6. Navigate to the folder containing the Communication Library for iOS distribution and locate libRTMPClient.a. Click "Open" to add the library

For the integration via RTMP, the library requires an RTMP server. See the Supported Servers section for the configuration instructions.

4 Creating RTMP Connection

Creating a connection with an RTMP server requires three parameters:

1. **RTMP server host name or IP address** - the host name or IP address must be visible from the network the device is connected to. For the public Wi-Fi and 3G networks, the host name or IP address of the server must be publicly accessible. For the private Wi-Fi connections, the server must be on the same subnet.
2. **RTMP server port** - the default port for many RTMP servers is 1935, some other servers may use 2037 or define a custom port.
3. **Application name** - all RTMP servers support the concept of an RTMP application. Typically this is a scope which maintains a collection of client connections and any RTMP resources created within the scope (for instance remote shared objects, media streams, etc).

Below is a sample code for connecting to the "MyApp" application on a server running at 192.168.0.11, on port 2037:

Header file:

```
RTMPClient *rtmpClient;
```

Class implementation file:

```
-(void)doConnect:(id)sender {
    rtmpClient = [[RTMPClient alloc] init];
    rtmpClient.delegate = self; // must be an instance of IRTMPClientDelegate
    [rtmpClient connect:@"192.168.0.11" port:2037 app:@"MyApp"];
}
```

The reference to assign the `delegate` property to `self`, enables the client code to receive callbacks from the library for the events delivered via `IRTMPClientDelegate`. In the example above, the implementation class implements the `IRTMPClientDelegate` protocol, thus is the reference to `self`.

The library also provides several overridden methods for establishing a connection. Specifically, it supports the following use-cases:

- Connecting to the root application on a server via the standard RTMP port. API reference.

```
-(void)connect:(NSString *)server;
```

- Connecting to the root application on a server via specific port API Reference.

```
-(void)connect:(NSString *)server port:(int)port;
```

- Connecting to a named application on a server via specific port (the example is shown above). API Reference.

```
-(void)connect:(NSString *)server port:(int)port app:(NSString *)application;
```

- Connecting to a named application on a server via specific port with additional connection arguments. API Reference.

```
-(void)connect:(NSString *)server port:(int)port app:(NSString *)application
params:(NSArray *)params;
```

5 Connection Events

The library can inform the client application about the status of the connection via the IRTMPClientDelegate protocol:

```
@protocol IRTMPClientDelegate <NSObject, IPendingServiceCallback>
-(void)connectedEvent;
-(void)disconnectedEvent;
-(void)connectFailedEvent:(int)code description:(NSString *)description;
@end
```

Since the protocol inherits from IPendingServiceCallback, there is an additional method not listed above - resultReceived.

To implement the protocol declare its usage in a header file:

```
#import "RTMPClient.h"

@interface ClientInvokeViewController : UIViewController
<IRTMPClientDelegate> {
     RTMPClient    *socket;
}
@end
```

And provide an implementation in the implementation file:

```
-(void)doConnect:(id)sender {
    rtmpClient = [[RTMPClient alloc] init];
    rtmpClient.delegate = self;
    [rtmpClient connect:@"192.168.0.11" port:2037 app:@"MyApp"];
}

-(void)connectedEvent {
    NSLog(@" $$$$$$ <IRTMPClientDelegate>> connectedEvent\n");
}

-(void)disconnectedEvent {
    NSLog(@" $$$$$$ <IRTMPClientDelegate>> disconnectedEvent\n");
}

-(void)connectFailedEvent:(int)code description:(NSString *)description {
    NSLog(@" $$$$$$ <IRTMPClientDelegate>> connectFailedEvent: %d = '%@'\n", code,
description);

    if (code == -1)
        NSLog( @"Unable to connect to the server. Make sure the hostname/IP address
and port number are valid\n" );
    else
        NSLog( [NSString stringWithFormat:@" !!! connectFailedEvent: %@ \n",
description] );
}

-(void)resultReceived:(id <IServiceCall>)call {
  // see the Invoke Server Methods section
}
```

6 Invoke Server Methods

The library supports remote method invocation via RTMP. The methods the client application invokes through the library must be available in the remote application the client is connected to (see the Application Name parameter in Creating RTMP Connection). The client-side API for invoking the methods is the same regardless of the server the client connects to. However, the mechanism for exposing server-side methods varies between available servers. See the Supported Servers section for the details on exposing server-side methods to the clients.

Remote method invocation is available through the invoke method in the RTMPClient class. A connection must be established before the client can call a server-side method. The example below demonstrates an invocation of the `echoInt` method accepting an argument of type `NSNumber`. The corresponding server-side type will vary depending on a specific server-side implementation. Various implementations of the method are also provided below.

Objective-C Client:

```
-(void)echoInt {
      // set call parameters
      NSMutableArray *args = [NSMutableArray array];
      NSString *method = [NSString stringWithString:@"echoInt"];
      [args addObject:[NSNumber numberWithInt:12]];

      // send invoke
      [rtmpClientInstance invoke:method withArgs:args];
}
```

echoInt with WebORB for .NET

```
using System;
using Weborb.Messaging.Server.Adapter;

namespace Weborb.Examples.ClientInvoke
{
  public class AppHandler : ApplicationAdapter
  {
    public int echoInt( int i )
    {
      return i;
    }
  }
}
```

echoInt with WebORB for Java

```
package examples.weborb;

import weborb.messaging.WebORBApplication;

public class AppHandler extends WebORBApplication
{
   public int echoInt( int i )
   {
     return i;
   }
```

```
}
```

echoInt with Wowza Media Server

```
package examples.weborb.ios;

import com.wowza.wms.amf.*;
import com.wowza.wms.client.IClient;
import com.wowza.wms.module.ModuleBase;
import com.wowza.wms.request.RequestFunction;

public class MyWowzaModule extends ModuleBase
{
  public void echoInt( IClient client, RequestFunction function, AMFDataList
params )
  {
    int value = getParamInt( params, PARAM1 );
    sendResult( client, params, value );
  }
}
```

echoInt with Flash Media Server

```
Client.prototype.echoInt = function( intValue )
{
  trace( "received invocation of enchoInt with arg value - " + intValue );
  return intValue;
}
```

For instructions about deploying server-side code see the Supported Servers section.

Result Handling

When the invoked server-side method returns a value, the library delivers it via the `IPendingServiceCallback` protocol inherited through `IRTMPClientDelegate`. The implementation of the protocol is the `resultReceived` method. Consider the following implementation of the method:

```
-(void)resultReceived:(id <IServiceCall>)call {

    int status = [call getStatus];

    if (status != 0x01 ) // this call is not a server response (0x01 is used to
differentiate this from a data push)
        return;

    NSString *method = [call getServiceMethodName];
    NSArray *args = [call getArguments];

    if (args.count)
        NSLog(@"resultReceived <---- status=%d, method='%@', return value=%@\n",
status, method, [args objectAtIndex:0]);
    else
        NSLog(@"resultReceived <---- status=%d, method='%@',, no return value -
void = 0\n", status, method);
}
```

For additional information see the API documentation for the `IPendingServiceCallback` and

IServiceCall protocols.

7 Server Data Push

RTMP servers provide the API for calling client-side functions, thus effectively executing data push since the client is connected to the server via a persistent connection. The library delivers server-to-client invocation requests via the `IPendingServiceCallback` protocol.The protocol is inherited by `IRTMPClientDelegate`. An implementation of that protocol must be assigned to the `delegate` property on the RTMPClient instance. Consider an implementation of the `resultReceived` method below:

```
-(void)resultReceived:(id <IServiceCall>)call {

    NSString *method = [call getServiceMethodName];
    NSArray *args = [call getArguments];
    int status = [call getStatus];

    if (status != 0x02) // this call is not a server to client call
        return;

    NSLog(@" received invocation request from server <---- status=%d, method='%@',
arguments=%d\n", status, method, args.count);
}
```

For additional information see the API documentation for the `IPendingServiceCallback` and `IServiceCall` protocols.

8 Remote Shared Objects

Remote Shared Objects is a standard feature for all RTMP Servers and is quite common in Flex/Flash applications. It is not well-known outside of that domain for the reason that non-Flash RTMP implementations are relatively new. Nevertheless, it is a very powerful feature enabling creation of interactive applications. The Remote Shared Objects feature supports the following use-cases:

- Shared data in real-time between multiple connected clients
- Simulate peer-to-peer method invocation
- Push data from the server via server-side remote shared object update

The video below provides an explanation of how Remote Shared Objects work with a demonstration of the functionality. If the video does not appear in the document, it is available in the Midnight Coders YouTube Channel.

Connecting to a RSO

To connect to a Remote Shared Object, the client application must establish an RTMP connection first. Once connected to the server, use the `getSharedObject` method to connect to the RSO. The method returns a reference to the local representation of the RSO. The object implements the `IClientSharedObject` protocol.

RSO declaration in a header file:

```
id <IClientSharedObject> clientRSO;
```

Connecting to RSO in the implementation file:

```
NSString *name = [NSString stringWithString:@"MyRSO"];
clientRSO = [rtmpClientInstance getSharedObject:name persistent:NO];
```

The `getSharedObject` method accepts two arguments: the shared object name and the persistence flag. If the flag is set to YES, the shared object is persisted on the server.

Managing Connection State

The `IClientSharedObject` protocol provides a mechanism to manage the connection state. The `isConnected` method checks if the client-side reference to the RSO is connected. The `connect` and `disconnect` methods provide a way to manage the connection's state.

```
-(void)changeConnectionState {

    if (![clientRSO isConnected])
        [clientRSO connect];
    else
        [clientRSO disconnect];
}
```

Updating RSO Property

The `IClientSharedObject` reference inherits from the `IAttributeStore` protocol which provides access to all the methods operating on the shared object properties. To set a property value use the `setAttribute` method available in `IClientSharedObject` (through the `IAttributeStore` inheritance). For example, the following code snippet sets an RSO property ("`coordinates`") to a value containing object with fields "x" and "y":

```
NSMutableDictionary *myCoordinates = [NSMutableDictionary dictionary];
[myCoordinates setValue:[NSNumber numberWithFloat:point.x] forKey:[NSString
stringWithString:@"x"]];
[myCoordinates setValue:[NSNumber numberWithFloat:point.y] forKey:[NSString
stringWithString:@"y"]];
[clientSO setAttribute:@"coordinates" object:myCoordinates];
```

The API also supports updating all remote shared object properties in a single call by using the `setAttributes` method.

Remote Shared Objects Events

The library delivers RSO events through the `ISharedObjectListener` protocol. In the current version in order to receive RSO events, the `delegate` assigned to `RTMPClient` must also implement the protocol. The protocol defines the following methods/events:

`onSharedObjectConnect` - invoked when the client connects with the remote shared object.
`onSharedObjectDisconnect` - invoked when the client disconnects from the remote shared object.
`onSharedObjectUpdate` - invoked when a remote shared object property is updated. The update may be originated by any other RSO client (or the server).
`onSharedObjectDelete` - invoked when a remote shared object property is deleted. The property may be removed by any other RSO client (or the server).
`onSharedObjectClear` - invoked when all shared objects properties have been removed (cleared).
`onSharedObjectSend` - invoked when another RSO client (or the server) sends a message through the RSO.

For example, the update notification event for property assignment shown above can be handled with the following code:

```
-(void)onSharedObjectUpdate:(id <IClientSharedObject>)so withKey:(id)key andValue:
(id)value {

    if ([key isEqualToString:@"ballCoordinates"]) {
      NSDictionary * myCoordinates = (NSDictionary *)value;
      NSLog( @"x coord %@", [myCoordinates valueForKey:@"x"] );
      NSLog( @"y coord %@", [myCoordinates valueForKey:@"y"] );
  }
}
```

9 Supported Servers

Enter topic text here.

9.1 WebORB for .NET

WebORB for .NET is an RTMP server implemented natively in .NET. Any custom server-side functionality can be implemented in any CLR language and target versions of the .NET framework starting at 2.0 and above. When WebORB is hosted in IIS, the RTMP server is started from the Global.asax file with a very first request to the virtual directory. Additionally, the WebORB RTMP server can run standalone both as a service or a console application by running the weborbee.exe executable located in the root of the installation directory. WebORB for .NET uses the port 2037 for all RTMP connections.

Creating an RTMP application

A WebORB for .NET messaging application can be set up by creating a directory under the `[WEBORB-HOME-DIR]/Applications/`. Once a directory is created, make sure the WebORB server is restarted so it recognizes the application. When connecting to the server, the name of the folder is the name of the application in the `connect` call.

Custom Server-Side Code

A WebORB RTMP application can be extended with custom code containing the application logic. The code must be a .NET class inheriting from `Weborb.Messaging.Server.Adapter.ApplicationAdapter`. Any public method added to the class can be invoked by the connected clients. To deploy the custom code, compile it as a .NET class library and copy the assembly into the /bin folder of the application where WebORB is deployed (or the /bin folder of the default WebORB installation). The RTMP application must also contain a configuration file in its folder. The configuration file associates the application handler class with the application. See the WebORB Documentation for additional details.

Data Push (server-to-client calls)

WebORB for .NET supports server to client invocation via the `IServiceCapableConnection` interface. The interface represents a connection between an RTMP client the current application. The code snippet shown below demonstrates how to invoke a method on all client connections when a client disconnects. Specifically, the code invokes the client-side "clientDisconnected" method with one argument - client's ID.

```
using System;
using System.Collections.Generic;
using System.Text;

using Weborb.Messaging.Api;
using Weborb.Messaging.Api.Service;
using Weborb.Messaging.Server.Adapter;

namespace Example
{
  public class AppHandler : ApplicationAdapter
  {
    public override void appDisconnect( IConnection conn )
    {
      IEnumerator<IConnection> connections = scope.getConnections();

      while( connections.MoveNext() )
      {
```

```
        IConnection connection = connections.Current;
        Object[] args = new Object[] { conn.getClient().getId() };

        if( connection is IServiceCapableConnection )
         ((IServiceCapableConnection)connection).invoke( "clientDisconnected", args
);
      }
    }
  }
}
```

Remote Shared Objects

WebORB's server-side remote shared objects provide support for the following use-cases:

- Obtaining a reference to a remote shared object
- Adding shared object listener to receive notifications for updates, deletes and sent messages
- Updating shared object property
- Clearing/deleting all shared object properties
- Sending a message to all shared object clients

Obtaining ISharedObject Reference

To obtain a reference to a remote shared object, use the `getSharedObject` method in the `ApplicationAdapter` subclass. The method returns an instance of the `Weborb.Messaging.Api.SO.ISharedObject` interface. The object is a server-side representation of the shared object, it provides access to all the RSO functionality, including setting RSO properties, delivering messages to the clients, etc:

```
using System;

using Weborb.Messaging.Api;
using Weborb.Messaging.Api.SO;
using Weborb.Messaging.Server.Adapter;

namespace Examples
{
  public class AppHandler : ApplicationAdapter
  {
    private ISharedObject myRSO;

    public override bool  appStart(IScope app)
    {
      bool appStarted = base.appStart( app );

      if( appStarted )
        myRSO = getSharedObject( app, "ChatSO", true );

      return appStarted;
    }
  }
}
```

Adding Listener

In order to receive notifications about RSO property updates, as well as many other RSO-related events, a listener should be added to a remote shared object reference. RSO listeners must implement the `Weborb.Messaging.Api.SO.ISharedObjectListener` interface. A listener can be added to an instance of `ISharedObject` using the `addSharedObjectListener` method:

```
using System;

using Weborb.Messaging.Api;
using Weborb.Messaging.Api.SO;
using Weborb.Messaging.Server.Adapter;

namespace Examples
{
  public class AppHandler : ApplicationAdapter
  {
    private ISharedObject myRSO;

    public override bool  appStart(IScope app)
    {
      bool appStarted = base.appStart( app );

      if( appStarted )
      {
       myRSO = getSharedObject( app, "ChatSO", true );
       myRSO.addSharedObjectListener( new MyRSOListener() );
      }

      return appStarted;
    }
  }

  public class MyRSOListener : ISharedObjectListener
  {
    public void onSharedObjectConnect( ISharedObjectBase so )
      {
       // a client has connected to the shared object
      }

    public void onSharedObjectDisconnect( ISharedObjectBase so )
      {
       // a client has disconnected from the shared object
      }

    public void onSharedObjectUpdate( ISharedObjectBase so, string key, object
value )
      {
       // a shared object property with the 'key' has been updated. The value is
'value'
      }

    public void onSharedObjectUpdate( ISharedObjectBase so, IAttributeStore
values )
      {
       // shared object properties have been updated. The updated key/value pairs
are in 'values'
      }

    public void onSharedObjectUpdate( ISharedObjectBase so, IDictionary<string,
object> values )
      {
       // shared object properties have been updated. The updated key/value pairs
are in 'values'
      }
```

```
    public void onSharedObjectDelete( ISharedObjectBase so, string key )
      {
       // a shared object property with the key 'key' has been removed
      }

    public void onSharedObjectClear( ISharedObjectBase so )
      {
       // all shared object properties have been cleared
      }

    public void onSharedObjectSend( ISharedObjectBase so, string method, IList
parms )
      {
        // a client sent a message 'method' with parameters 'parms' to all the
clients of the shared object identified by so
      }
   }
}
```

Updating Property

When a shared object property is updated, WebORB delivers a notification event to all clients connected to the shared object. To update a property, use the `setAttribute` method on the RSO reference. Consider the following example:

```
Hashtable messageObject = new Hashtable();
messageObject[ "imText" ] = messageText;
messageObject[ "username" ] = ".NET Client";
messageObject[ "color" ] = 0;
messageObject[ "isBold" ] = false;
messageObject[ "isItalics" ] = false;
messageObject[ "isUnderline" ] = false;
myRSO.setAttribute( "UserMessage", messageObject );
```

The code above updates the "UserMessage" property of a remote shared object with an untyped object (represented as Hashtable). Alternatively, the code could be rewritten to use a strongly typed object:

```
InstantMessage messageObject = new InstantMessage();
messageObject.imText = messageText;
messageObject.username = ".NET Client";
messageObject.color = 0;
messageObject.isBold = false;
messageObject.isItalics = false;
messageObject.isUnderline = false;
myRSO.setAttribute( "UserMessage", messageObject );

public class InstantMessage
{
  public String imText {get; set;}
  public String userName {get; set;}
  public int color {get; set;}
  public bool isBold {get; set;}
  public bool isItalics {get; set;}
  public bool isUnderline {get; set;}
}
```

Deleting All Shared Object Properties

To delete all shared object properties use the `clear` method on the `ISharedObject` reference. See Obtaining ISharedObject Reference for details. When the method is invoked, WebORB notifies all other connected clients that the properties have been deleted.

Sending Messages To All RSO Clients
To delete all shared object properties use the `sendMessage` method on the `ISharedObject` reference. See Obtaining ISharedObject Reference for details. When the method is called, WebORB sends a notification event to invoke the method with the specified arguments on all clients connected to the remote shared object. See Server Data Push for additional details.

9.2 WebORB for Java

WebORB for Java embeds the Red5 product which is a fully functional Java RTMP server. Any custom server-side functionality can be implemented in Java and deployed with WebORB into any Java web container. Alternatively, WebORB for Java can be executed standalone thus avoiding the container deployment step. To listen for the RTMP connections, WebORB for Java uses the port 1935.

Creating an RTMP application

Creating and registering an RTMP application is a two step process:

1. Create an XML file in `[WEBOB-HOME-DIR]/WEB-INF/classes` with the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/
dtd/spring-beans.dtd"
<beans>
    <bean id="XXXXXXX.context" class="org.red5.server.Context" autowire="byType" />
    <bean id="XXXXXXX.scope" class="org.red5.server.WebScope" init-
method="register">
        <property name="server" ref="red5.server" />
        <property name="parent" ref="global.scope" />
        <property name="context" ref="XXXXXXX.context" />
        <property name="handler" ref="XXXXXXX.handler" />
        <property name="contextPath" value="/XXXXXXX" />
        <property name="virtualHosts" value="*,localhost, localhost:1935,
localhost:8080, 127.0.0.1:8080" />
    </bean>
    <bean id="XXXXXXX.handler" class="weborb.messaging.WebORBApplication"
singleton="true" />
</beans>
```

Make sure to put the application name everywhere where it shows "`XXXXXXX`". It is recommended to save the file with the name of `YOURAPPNAME-web.xml`.

2. Register the file in `[WEBOB-HOME-DIR]/WEB-INF/classes/beanRefContext.xml` by adding the following line in the `<list>` element:

```
<value>YOURAPPNAME-web.xml</value>
```

3. Restart WebORB for Java. You can verify that the application is registered in the WebORB Management Console. Open the console at http://localhost:8080, select the Messaging Server tab. The application should appear in the list of applications. When connecting to the server, the name of the application (as specified in step 1) should be used in the `connect` call.

Custom Server-Side Code

A WebORB RTMP application can be extended with custom code containing the application logic. The code must be a Java class inheriting from `weborb.messaging.WebORBApplication`. Any public method added to the class can be invoked by the connected clients. To deploy the custom code, the compiled classes can be copied either into the WEB-INF/classes folder (the folder hierarchy must match the

package structure) or in a jar file into WEB-INF/lib. The RTMP application configuration file must also reference the custom code class. See the first step in Creating an RTMP Application. The following line in the file must contain the fully qualified name of the class:

```
<bean id="XXXXXXX.handler" class="weborb.messaging.WebORBApplication"
singleton="true" />
```

Where "XXXXXXX" is the name of the application.

Data Push (server-to-client calls)

WebORB for Java supports server to client invocation via the `IServiceCapableConnection` interface.The interface represents a connection between an RTMP client the current application. The code snippet shown below demonstrates how to invoke a method on all client connections when a client disconnects. Specifically, the code invokes the client-side "clientDisconnected" method with one argument - client's ID.

```
package demo;

import java.util.Collection;
import java.util.Set;
import org.red5.server.api.IClient;
import org.red5.server.api.IConnection;
import org.red5.server.api.IScope;
import org.red5.server.api.service.IServiceCapableConnection;
import weborb.messaging.WebORBApplication;

public class AppHandler extends WebORBApplication
{
  public void disconnect( IConnection conn, IScope scope )
  {
    Collection<Set<IConnection>> connections = scope.getConnections();

    while( connections.MoveNext() )
    {
      IConnection connection = connections.Current;
      Object[] args = new Object[] { conn.getClient().getId() };

      for( Set<IConnection> connectionsSet : connections )
        for( IConnection connection : connectionsSet )
          if( connection instanceof IServiceCapableConnection )
          {
            IServiceCapableConnection cnx = (IServiceCapableConnection) connection;
            cnx.invoke( "clientDisconnected", args );
          }
      }
   }
}
```

Remote Shared Objects

WebORB's server-side remote shared objects provide support for the following use-cases:

- Obtaining a reference to a remote shared object
- Adding shared object listener to receive notifications for updates, deletes and sent messages
- Updating shared object property
- Clearing/deleting all shared object properties
- Sending a message to all shared object clients

Obtaining ISharedObject Reference

To obtain a reference to a remote shared object, use the `getSharedObject` method in the `WebORBApplication` subclass. The method returns an instance of the `org.red5.server.api.so.ISharedObject` interface. The object is a server-side representation of the shared object, it provides access to all the RSO functionality, including setting RSO properties, delivering messages to the clients, etc:

```
package demo;

import org.red5.server.api.IScope;
import weborb.messaging.WebORBApplication;

public class AppHandler extends WebORBApplication
{
  private ISharedObject myRSO;

  public void appStart( IScope app )
  {
    createSharedObject( scope, "chatRSO", false );
    myRSO = getSharedObject( scope, "chatRSO", false );

    return super.appStart( app );
  }
}
```

Adding Listener

In order to receive notifications about RSO property updates, as well as many other RSO-related events, a listener should be added to a remote shared object reference. RSO listeners must implement the `org.red5.server.api.so.ISharedObjectListener` interface. A listener can be added to an instance of `ISharedObject` using the `addSharedObjectListener` method:

```
package demo;

import org.red5.server.api.IScope;
import weborb.messaging.WebORBApplication;

public class AppHandler extends WebORBApplication
{
  private ISharedObject myRSO;

  public void appStart( IScope app )
  {
    createSharedObject( scope, "chatRSO", false );
    myRSO = getSharedObject( scope, "chatRSO", false );
    myRSO.addSharedObjectListener( new MySharedObjectListener() );

    return super.appStart( app );
  }
}

  class MySharedObjectListener : ISharedObjectListener
  {
    public void onSharedObjectConnect( ISharedObjectBase so )
      {
       // a client has connected to the shared object
      }

    public void onSharedObjectDisconnect( ISharedObjectBase so )
      {
```

```
       // a client has disconnected from the shared object
      }

    public void onSharedObjectUpdate( ISharedObjectBase so, string key, object
value )
      {
       // a shared object property with the 'key' has been updated. The value is
'value'
      }

    public void onSharedObjectUpdate( ISharedObjectBase so, IAttributeStore
values )
      {
       // shared object properties have been updated. The updated key/value pairs
are in 'values'
      }

    public void onSharedObjectUpdate( ISharedObjectBase so, IDictionary<string,
object> values )
      {
       // shared object properties have been updated. The updated key/value pairs
are in 'values'
      }

    public void onSharedObjectDelete( ISharedObjectBase so, string key )
      {
       // a shared object property with the key 'key' has been removed
      }

    public void onSharedObjectClear( ISharedObjectBase so )
      {
       // all shared object properties have been cleared
      }

    public void onSharedObjectSend( ISharedObjectBase so, string method, IList
parms )
      {
        // a client sent a message 'method' with parameters 'parms' to all the
clients of the shared object identified by so
      }
   }
}
```

Updating Property

When a shared object property is updated, WebORB delivers a notification event to all clients connected to the shared object. To update a property, use the `setAttribute` method on the RSO reference. Consider the following example:

```
Hashtable messageObject = new Hashtable();
messageObject.put( "imText", messageText;
messageObject.put( "username", "Java Client" );
messageObject.put( "color", 0 );
messageObject.put( "isBold", false );
messageObject.put( "isItalics", false );
messageObject.put( "isUnderline", false );
myRSO.setAttribute( "UserMessage", messageObject );
```

The code above updates the "UserMessage" property of a remote shared object with an untyped object (represented as Hashtable). Alternatively, the code could be rewritten to use a strongly typed object:

```
InstantMessage messageObject = new InstantMessage();
messageObject.imText = messageText;
messageObject.username = "Java Client";
messageObject.color = 0;
messageObject.isBold = false;
messageObject.isItalics = false;
messageObject.isUnderline = false;
myRSO.setAttribute( "UserMessage", messageObject );

public class InstantMessage
{
  public String imText;
  public String userName;
  public int color;
  public boolean isBold;
  public boolean isItalics;
  public boolean isUnderline;
}
```

Deleting All Shared Object Properties
To delete all shared object properties use the `clear` method on the `ISharedObject` reference. See Obtaining ISharedObject Reference for details. When the method is invoked, WebORB notifies all other connected clients that the properties have been deleted.

Sending Messages To All RSO Clients
To delete all shared object properties use the `sendMessage` method on the `ISharedObject` reference. See Obtaining ISharedObject Reference for details. When the method is called, WebORB sends a notification event to invoke the method with the specified arguments on all clients connected to the remote shared object. See Server Data Push for additional details.

9.3 Adobe Flash Media Server

The Communication Library for iOS fully supports integration with the Adobe Flash Media Server. See the following article for details:
http://blog.themidnightcoders.com/index.php/2011/07/11/integrating-native-ios-applications-with-flash-media-server-using-rtmp/

9.4 Wowza Media Server

The Communication Library for iOS fully supports integration with the Wowza Media Server. See the following article for details:
http://blog.themidnightcoders.com/index.php/2011/07/07/ios-applications-wowza-media-server-integration-with-rtmp/

10 Class Library API

Enter topic text here.

10.1 RTMPClient

connect:

Establishes an RTMP connection to the root application running on the specified RTMP server running on the default port (1935). Traditionally the server would be accessible via the URL formatted as: rtmp://server

```
-(void)connect:(NSString *)server;
```

Arguments

server

The IP address or hostname of the server where the RTMP server is running

connect:port:

Establishes an RTMP connection to the root application running on the specified RTMP server on the specified port. Traditionally the server would be accessible via the URL formatted as: rtmp://server:port

```
-(void)connect:(NSString *)server port:(int)port;
```

Arguments

server

The IP address or hostname of the server where the RTMP server is running

port

The port of the socket the server is listening on

connect:port:app:

Establishes an RTMP connection to the application on the specified RTMP server running on the given port. Traditionally the server would be accessible via the URL formatted as: rtmp://server:port/app

```
-(void)connect:(NSString *)server port:(int)port app:(NSString *)application;
```

Arguments

server

The IP address or hostname of the server where the RTMP server is running

port

The port of the socket the server is listening on

app

The application name

connect:port:app:params:

Establishes an RTMP connection to the application on the specified RTMP server running on the given port with an array of connection parameters

```
-(void)connect:(NSString *)server port:(int)port app:(NSString *)application
```

```
params:(NSArray *)params;
```

Arguments

server
The IP address or hostname of the server where the RTMP server is running

port
The port of the socket

app
The application name

params
The array of application parameters passed into the application's 'connect' method on the server

delegate

A property with a reference to an implementation of the `IRTMPClientDelegate` protocol. Provides access to the connection-related events as well as a handler for return values from the client-to-server calls and server-to-client invocation requests.

```
@property (nonatomic, assign) id <IRTMPClientDelegate> delegate;
```

invoke:withArgs:

Invokes a method with the array of arguments on the application the client is connected to.

```
-(void)invoke:(NSString *)method withArgs:(NSArray *)args;
```

Arguments

method
The method name

args
The array of the method arguments

getSharedObject:persistent:

Returns a reference to a remote shared object that multiple clients can access. If the remote shared object does not already exist, it is created by the server (if the application the client is connected to allows RSO construction).

```
-(id <IClientSharedObject>)getSharedObject:(NSString *)name persistent:(BOOL)
persistent;
```

Arguments

name
The name of the shared object

persistent
If YES – specifies that the shared object is persistent on the server. The value of NO indicates the shared object is not persistent.

10.2 IRTMPClientDelegate

The protocol is used by the library to inform the client application about various connection-related events. The protocol inherits from the IPendingServiceCallback protocol.

connectedEvent

The delegate receives this method when the socket connection is established

```
-(void)connectedEvent;
```

disconnectedEvent

The delegate receives this method when the socket connection is disconnected

```
-(void)disconnectedEvent;
```

connectFailedEvent:description:

The delegate receives this method when the socket connection has failed with the event code and description

```
-(void)connectFailedEvent:(int)code description:(NSString *)description;
```

Arguments

code

The event code. The value of -1 indicates the connection with the specified host over the specified port could not be established

description

The event description

10.3 IPendingServiceCallback

A protocol used by the library to deliver server-to-client invocations and to process return values from the client-to-server invocations.

resultReceived:

The delegate receives this method when the RTMP client receives a result of a remote method invocation or when the server performs server-to-client remote method invocation. Object parameter call conforms the IServiceCall protocol

```
-(void)resultReceived:(id <IServiceCall>)call;
```

Arguments

call

The object conforming to the IServiceCall protocol

10.4 IClientSharedObject

The `IClientSharedObject` protocol represents a client-side reference to a remote shared object. The protocol inherits from the `IAttributeStore` which provides access to the methods operating on the

shared object properties.

connect

Connects to a remote shared object using the existing connection.

```
-(void)connect;
```

isConnected

Checks if the client is connected to the remote shared object.

```
-(BOOL)isConnected;
```

disconnect

Disconnects from the remote shared object.

```
-(void)disconnect;
```

sendMessage:arguments:

Sends a message to (invokes a method on) other clients connected to the remote shared object.

```
-(void)sendMessage:(NSString *)handler arguments:(NSArray *)arguments;
```

Arguments

handler
The name of the handler/method to call

arguments
An array of objects that should be passed as arguments to the handler/method

clear

Deletes all remote shared object attributes. If the remote shared object is persistent, the data is also removed from the persistent storage. Returns YES if successful; NO otherwise

```
-(BOOL)clear;
```

close

Detaches a reference from the remote shared object and immediately destroys the reference. The method should be called when the remote shared object reference is no longer needed.

```
-(void)close;
```

10.5 ISharedObjectListener

onSharedObjectConnect:

The delegate receives this method when the client has established to a remote shared object.

```
-(void)onSharedObjectConnect:(id <IClientSharedObject>)so;
```

Arguments

so
The remote shared object reference

onSharedObjectDisconnect:

The delegate receives this method when the client has disconnected from the remote shared object.

```
-(void)onSharedObjectDisconnect:(id <IClientSharedObject>)so;
```

Arguments

so
The remote shared object reference

onSharedObjectUpdate:

The delegate receives this method when the remote shared object attribute is updated.

```
-(void)onSharedObjectUpdate:(id <IClientSharedObject>)so withKey:(id)key andValue:
(id)value;
```

Arguments

so
The remote shared object reference
key
The name of the updated attribute
value
The value of the updated attribute

onSharedObjectUpdate:

The delegate receives this method when multiple attributes of the remote shared object are updated.

```
-(void)onSharedObjectUpdate:(id <IClientSharedObject>)so withValues:(id
<IAttributeStore>)values;
```

Arguments

so
The remote shared object reference
values
The new attributes of the shared object

onSharedObjectUpdate:

The delegate receives this method when multiple attributes of a shared object are updated.

```
-(void)onSharedObjectUpdate:(id <IClientSharedObject>)so withDictionary:
(NSDictionary *)values;
```

Arguments

so
The remote shared object reference
values

The updated attributes of the remote shared object

onSharedObjectDelete:

The delegate receives this method when an attribute is deleted from the remote shared object.

```
-(void)onSharedObjectDelete:(id <IClientSharedObject>)so withKey:(NSString *)key;
```

Arguments

so
The remote shared object reference
key
The name of the deleted attribute

onSharedObjectClear:

The delegate receives this method when all the attributes of the remote shared object are removed.

```
-(void)onSharedObjectClear:(id <IClientSharedObject>)so;
```

Arguments

so
The remote shared object reference

onSharedObjectSend:

The delegate receives this method when the shared object method call is sent.

```
-(void)onSharedObjectSend:(id <IClientSharedObject>)so withMethod:(NSString *)
method andParams:(NSArray *)parms;
```

Arguments

so
The remote shared object reference
method
The name of the method invoked by other remote shared object client (or server)
params
The arguments

10.6 IAttributeStore

getAttributeNames

Gets the attribute names. Returns an array containing all attribute names

```
-(NSArray *)getAttributeNames;
```

getAttributes

Gets the attributes. Returns the dictionary containing all attributes

```
-(NSDictionary *)getAttributes;
```

setAttribute:object:

Sets an attribute on this object. Returns YES if the attribute value changed, NO otherwise

```
-(BOOL)setAttribute:(NSString *)name object:(id)value;
```

Arguments

name
The name of the attribute to change

value
The new value of the attribute

setAttributes:

Sets multiple attributes on this object

```
-(void)setAttributes:(NSDictionary *)values;
```

Arguments

value
The values of the attributes to set

setAttributeStore:

Sets multiple attributes on this object

```
-(void)setAttributeStore:(id <IAttributeStore>)values;
```

Arguments

value
The values of the attributes to set

getAttribute:

Returns attribute value or `nil` if the attribute doesn't exist.

```
-(id)getAttribute:(NSString *)name;
```

Arguments

name
The name of the attribute to get the value for

getAttribute:object:

Returns the value for a given attribute or assigns the specified default value if the attribute does not exist. Returns the attribute value

```
-(id)getAttribute:(NSString *)name object:(id)defaultValue;
```

Arguments

name
The name of the attribute to get

defaultValue
The value of the attribute to set if the attribute does not exist

hasAttribute:

Checks if the object has the attribute. Returns YES if the attribute exists, NO otherwise.

```
-(BOOL)hasAttribute:(NSString *)name;
```

Arguments

name
The name of the attribute to check

removeAttribute:

Removes the attribute. Return YES if the attribute is found and removed, NO otherwise.

```
-(BOOL)removeAttribute:(NSString *)name;
```

Arguments

name
The name of the attribute to remove

removeAttributes

Removes all attributes

```
-(void)removeAttributes;
```

10.7 IServiceCall

The IServiceCall protocol is used by IPendingServiceCallback to deliver the result value from a remote method invocation or execute server to client (data push) invocation request.

isSuccess

Returns YES if successful; NO otherwise

```
-(BOOL)isSuccess;
```

getServiceMethodName

Returns the name of the service method

```
-(NSString *)getServiceMethodName;
```

getServiceName

Returns the name of the service

```
-(NSString *)getServiceName;
```

getArguments

Returns the array of the service method arguments for server-to-client invocations or get the return value for a client-to-server method invocation. If the invoked server method return "void", the array is empty, otherwise the return value is in the first element of the array.

```
-(NSArray *)getArguments;
```

getStatus

Returns the status of the service method

```
-(uint)getStatus;
```

Returned value is one of the following constants:

```
STATUS_PENDING = 0x01
STATUS_SUCCESS_RESULT = 0x02
STATUS_SUCCESS_NULL = 0x03
STATUS_SUCCESS_VOID = 0x04
STATUS_SERVICE_NOT_FOUND = 0x10
STATUS_METHOD_NOT_FOUND = 0x11
STATUS_ACCESS_DENIED = 0x12
STATUS_INVOCATION_EXCEPTION = 0x13
STATUS_GENERAL_EXCEPTION = 0x14
STATUS_APP_SHUTTING_DOWN = 0x15
```

getException

Returns the exception of the service method if the method call resulted in an exception

```
-(NSException *)getException;
```

Endnotes 2... (after index)

Back Cover