kurento-room-client-android Documentation

Release 1.0.4

Jukka Ahola

Mar 16, 2017

Contents

Table of Contents:

CHAPTER 1

Overview

kurento-room-client-android is a Java library for Android. This library can be used to deploy applications for interaction with Kurento room server. This documentation provides help on how to install and use the library.

Source code is available at https://github.com/nubomedia-vtt/kurento-room-client-android

Support is provided through the Nubomedia VTT Public Mailing List available at https://groups.google.com/forum/#!forum/nubomedia-vtt

License

Licensing information of this software can be found at

https://github.com/nubomedia-vtt/kurento-room-client-android#licensing-and-distribution

CHAPTER 2

Installation Guide

If you want to use the API in your own project through Maven or JCenter, please follow the User installation guide (the first section). If you wish to contribute to the source code or compile the library locally for your project, please follow the Developer installation guide (the next section below).

User installation guide

This library is maintained in both JCenter and Maven. In order to use it in your project, you will need to add either one of these central repositories to your project. Please refer to the documentation of your IDE for more help. To add the dependency to this library, simple add the following dependency:

```
fi.vtt.nubomedia:kurento-room-client-android:[version]
```

Where [version] is the latest edition available in the repository. You may find the versions in

https://mvnrepository.com/artifact/fi.vtt.nubomedia/kurento-room-client-android

and

https://bintray.com/nubomedia-vtt/maven/kurento-room-client-android

Developer installation guide

This documents provides information how to compile the kurento-room-client-android library from the sources. This guide is also valid if you wish to compile the library locally for you project.

First, make sure you have installed Git on your system. Help for Installing Git can be found in:

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Next, clone the project by using the command below:

```
git clone https://github.com/nubomedia-vtt/kurento-room-client-android.git
```

If you wish to work without Git, there is also a ZIP link available:

https://github.com/nubomedia-vtt/kurento-room-client-android/archive/master.zip

Setup the developing environment by importing the project to in your IDE. The project is an Android Studio project and should open easily. For other IDEs and more help, support is provided through the Nubomedia VTT Public Mailing List available at https://groups.google.com/forum/#!forum/nubomedia-vtt

CHAPTER 3

Developer Guide

This documents provides a tutorial on how to utilize the kurento-room-client-android library for your project. The tutorial is made for Android Studio, but the same procedure applies to other IDEs as well.

This documentation is made for library version 1.1.0+ and may not be fully applicable to older versions.

Setting up a new project

Create a new Android Studio project. Then add a dependency to the library with the following Gradle dependency

```
dependencies {
    compile 'fi.vtt.nubomedia:kurento-room-client-android:[version]'
}
```

Where [version] is the latest edition available in the repository. Depending on the repository system you decide to use, you may find the versions in

https://mvnrepository.com/artifact/fi.vtt.nubomedia/kurento-room-client-android

and

https://bintray.com/nubomedia-vtt/maven/kurento-room-client-android

The library naturally requires a permission to access the Internet, so add the following permission to `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Connecting to server

KurentoRoomAPI is the main class for handling room connectivity. To use it, first import the following classes

```
import fi.vtt.nubomedia.kurentoroomclientandroid.KurentoRoomAPI;
import fi.vtt.nubomedia.kurentoroomclientandroid.RoomError;
import fi.vtt.nubomedia.kurentoroomclientandroid.RoomListener;
import fi.vtt.nubomedia.kurentoroomclientandroid.RoomNotification;
import fi.vtt.nubomedia.kurentoroomclientandroid.RoomResponse;
import fi.vtt.nubomedia.utilitiesandroid.LooperExecutor;
```

Next you will need to have a class which owns the `KurentoRoomAPI` instance. In this example we use the main activity as a base class for implementation. Below is the base structure for our MainActivity:

```
public class MainActivity extends Activity implements RoomListener {

    private LooperExecutor executor;
    private static KurentoRoomAPI kurentoRoomAPI;

    @Override
    protected void onCreate(Bundle savedInstanceState) {    }

    @Override
    public void onRoomResponse(RoomResponse response) {    }

    @Override
    public void onRoomError(RoomError error) {    }

    @Override
    public void onRoomNotification(RoomNotification notification) {    }

    @Override
    public void onRoomConnected() {    }

    @Override
    public void onRoomDisconnected() {    }
}
```

We will go through these events later. But for now, we will need to fill in the `onCreate` method:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    executor = new LooperExecutor();
    executor.requestStart();
    String wsRoomUri = "wss://mykurentoserver:1234/room";
    kurentoRoomAPI = new KurentoRoomAPI(executor, wsUri, this);
    kurentoRoomAPI.connectWebSocket();
}
```

Your `KurentoRoomAPI` has been now created and the websocket has been opened. `connectWebSocket` is an asynchronous process and the code execution continues. Now the events declared previously in `onCreate` come into play. When the websocket gets connected to the server, the event `onRoomConnected` is triggered.

Joining a room

We will make a simple application which joins a room immediately when the websocket is connected. For joining a room, call method `sendJoinRoom`.

```
@Override
public void onRoomConnected() {
    kurentoRoomAPI.sendJoinRoom("MyUsername", "MyRoomName", true, 123);
}
```

The parameters for the method are as follows:

- The username as it appears to all other users
- The name of the room to be joined
- True if data channels should be enabled for this user
- Index number to track the corresponding response message to this request

Once the join is complete, the event `onRoomResponse()` is fired in `MainActivity`.

```
@Override
public void onRoomResponse(RoomResponse response) {
    if (response.getId() == 123) {
        Log.d(TAG, "Successfully connected to the room!");
    }
}
```

The identifier, 123, is just a randomly selected number and you may use any integer you wish. The generic idea is that when you send a request with a specific id number, you will know that a received response is bound to that particular request pointed by the identifier.

Dealing with errors

If you run this example multiple times you will notice that the `onRoomResponse` may not always appear, so what's wrong? Fill in the `onRoomError` event to see if there are error messages:

```
@Override
public void onRoomError(RoomError error) {
    Log.d(TAG, error.toString());
}
```

After doing this and running the example again, you may receive a log message `RoomError: 104 - null`. Supported error codes are in `RoomError.Code`, and below is an example how to use it to compare error codes. But for now, let us refer to the list below and see what is the error. The following error codes are currently supported by the API:

- 101 USER_GENERIC_ERROR_CODE
- 102 USER_NOT_FOUND_ERROR_CODE
- 103 USER_CLOSED_ERROR_CODE
- 104 EXISTING_USER_IN_ROOM_ERROR_CODE
- 105 USER_NOT_STREAMING_ERROR_CODE
- 201 ROOM_GENERIC_ERROR_CODE
- 202 ROOM_NOT_FOUND_ERROR_CODE
- 203 ROOM_CLOSED_ERROR_CODE
- 204 ROOM_CANNOT_BE_CREATED_ERROR_CODE

- 301 MEDIA_GENERIC_ERROR_CODE
- 302 MEDIA_SDP_ERROR_CODE
- 303 MEDIA_ENDPOINT_ERROR_CODE
- 304 MEDIA_WEBRTC_ENDPOINT_ERROR_CODE
- 305 MEDIA_RTP_ENDPOINT_ERROR_CODE
- 306 MEDIA_NOT_A_WEB_ENDPOINT_ERROR_CODE
- 307 MEDIA_MUTE_ERROR_CODE
- 801 TRANSPORT_REQUEST_ERROR_CODE
- 802 TRANSPORT_RESPONSE_ERROR_CODE
- 803 TRANSPORT_ERROR_CODE
- 999 GENERIC_ERROR_CODE

Code 104 means that you are already in the room. This is because our example does not explicitly tell the server that we want to leave the room when the application terminates. So your request was not successful and you are not currently in the room. To find out how to leave a room, refer to section "Leaving a room" below. But in any case, you may add the following handler to your `onRoomError` event:

```
@Override
public void onRoomError(RoomError error) {
    if (error.getCode() == RoomError.Code.EXISTING_USER_IN_ROOM_ERROR_CODE.
↪getValue()) {
        Log.d(TAG, "User with the same name already in the room!");
        // Do some other stuff
    }
}
```

If you receive an error code which is not defined in the Android room API, you can also refer to the room exception codes in the server source:

https://github.com/Kurento/kurento-room/blob/master/kurento-room-sdk/src/main/java/org/kurento/room/exception/RoomException.java

Sending and receiving messages

A room with multiple users is not very useful if you cannot interact with each other. To implement a simple chat feature, you may use `sendMessage` method to transmit simple text messages. Let's extend our example by sending a hello to all participants in the room. Note that the reception of the message to the server will also be noted in the same event handler:

```
@Override
public void onRoomResponse(RoomResponse response) {
    Log.d(TAG, response.toString());
    if (response.getId() == 123) {
        Log.d(TAG, "Successfully connected to the room!");
        kurentoRoomAPI.sendMessage("MyRoomName", "MyUsername", "Hello room!", 125);
    } else if (response.getId() == 125) {
        Log.d(TAG, "The server received my message!");
}
```

To receive messages other users write, it's time to use the `onRoomNotification` method. The difference between these two event handlers, `onRoomNotification` and `onRoomResponse`, is that the former is for receiving events not triggered by you and do not have an id. Instead, they can be identifier by methods:

```
@Override
public void onRoomNotification(RoomNotification notification) {

    if(notification.getMethod().equals(RoomListener.METHOD_SEND_MESSAGE)) {
        final String username = notification.getParam("user").toString();
        final String message = notification.getParam("message").toString();
        Log.d(TAG, "Oh boy! " + username + " sent me a message: " + message);
    }

}
```

You will also receive your own message this way, so you don't need to keep track of messages that you sent.

How to proceed to multimedia

Now begins the tricky part. This API does not contain any WebRTC audio/video encapsulation or streaming mechanisms, it is simply a signalling mechanism between room and its users. To fully establish multimedia connections, you will need to use another library. Below are the links to the Github page and documentation of the project:

https://github.com/nubomedia-vtt/webrtcpeer-android

http://doc-webrtcpeer-android.readthedocs.io/en/latest/

In this tutorial we are going to go through the room part of multimedia connectivity. You will learn how to signal your intention to publish your own video stream to the server. We will also go through how to keep track deal with events and to signal intention to receive video from other users.

Dealing with ICE candidates

ICE refers to Interactive Connectivity Establishment, which is a fundamental concept in establishing connectivity between peers. An ICE candidate refers to a piece of information on how to connect to a certain peer. So in short, ICE candidate is a "connectivity ticket" to a peer.

When developing a client application for Kurento room server, you'll be dealing with two kinds of ICE candidates: your own connection ICE candidates and the ICE candidates of other peers. Most of ICE stuff belongs to `webrtcpeer-android` library, but there is one method which is provided by this library which is `sendOnIceCandidate`.

Since `webrtcpeer-android` library does not have any information on rooms or their users, function `sendOnIceCandidate` is used to bridge these two libraries together. Most likely you will be using this method in `onIceCandidate` of class `NBMWebRTCPeer` in `webrtcpeer-android` library.

```
boolean myIceSent = false;

@Override
public void onIceCandidate(IceCandidate iceCandidate, NBMPeerConnection␣
↪nbmPeerConnection) {
    if (!myIceSent) {
        kurentoRoomAPI.sendOnIceCandidate("MyUsername", iceCandidate.sdp,␣
↪iceCandidate.sdpMid, Integer.toString(iceCandidate.sdpMLineIndex), 126);
        myIceSent = true;
```

```
    } else {
        kurentoRoomAPI.sendOnIceCandidate("MyRoomPeer", iceCandidate.sdp,␣
↪iceCandidate.sdpMid, Integer.toString(iceCandidate.sdpMLineIndex), 126);
    }
}
```

The first call `sendOnIceCandidate` captures the creation of your own connectivity ticket, while the second one handles the connectivity of your peer.

Publish video

When you are ready to send your own video, call `sendPublishVideo` as follows:

```
kurentoRoomAPI.sendPublishVideo(description, false, 127);
```

The call is fundamentally the same what we did before, so you will have to again associate an id for the call. The first parameter, `description`, is a Session Description Protocol (SDP) string describing the features of your media stream. Usually you don't construct these strings yourself, but instead use `webrtcpeer-android` library to generate it.

The second parameter is loopback. If you set it true, the server sends your own video back to you. Usually you want to set it to false and just show the locally generated video, but if you have a use case where the server for example applies a filter to the video streams, you want the video to make a round trip on the server before showing it even to the user themselves.

Unpublish video

To unpublish your own video stream in the room, simply call

```
kurentoRoomAPI.sendUnpublishVideo(128);
```

Receive video

To receive video from another peer in the same room, you will have to use the following call. The first parameter is the name of the user. The second one is the source of the stream. Currently only "webcam" is supported. The third parameter is the session description as in the "Publish video" section, and the last one is the response id.

```
String username = "MyRoomPeer";
kurentoRoomAPI.sendReceiveVideoFrom(username, "webcam", description, 129);
```

Stop receiving video

To tell the server that you no longer want to receive a certain video stream, make the following call:

```
kurentoRoomAPI.sendUnsubscribeFromVideo(username, "webcam", description, 130);
```

Now the parameters are simply the username and a constant string "webcam".

Getting disconnected

If the connection to the server breaks or you are kicked out of the room, the following event is triggered:

```
@Override
public void onRoomDisconnected() {
    Log.d(TAG, "I got disconnected... but why? WHY?!");
}
```

It's up to you how you want to deal with the situation, but the end-user should at least be notified that the room is no longer in use until re-connected.

Leaving a room

To disconnect from a room and to avoid a dangling idle user staying in the room until disconnected by the server, you may send your own command:

```
kurentoRoomAPI.sendLeaveRoom(131);
```

Typically in this kind of scenario you will not wait for the server to respond. However, if your app was not to terminate but instead join another room, you can still handle the responses as usual by using the request id 131:

```
@Override
public void onRoomResponse(RoomResponse response) {
    if (response.getId() == 131) {
        Log.d(TAG, "Successfully left the room!");
    }
}
```

Placement of the `sendLeaveRoom` is a bit tricky if you want to disconnect from the room when the app is destroyed. Due to the asynchronous nature of this API, if you simply call

```
@Override
public void onDestroy() {
    kurentoRoomAPI.sendLeaveRoom(124);
    super.onDestroy();
}
```

You will very likely stay in the room since the threads handling the request will be terminated before the message gets sent. Therefore it is easier just not to leave the room if the application gets destroyed and let the server auto-disconnect your client.

Adding a trusted self-signed certificate

KurentoRoomAPI supports developers to add a trusted self-signed certificate. This allows testing without CA certificate, and moreover if the application uses only one Kurento server, no CA certificate is needed.

Here is an example on how to include a self-signed certificate from assets in Android Studio:

```
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
```

```
...

KurentoRoomAPI kurentoRoomAPI;
CertificateFactory cf = CertificateFactory.getInstance("X.509");
InputStream caInput = new BufferedInputStream(myActivity.context.getAssets().open("my_
↪server_certificate.cer"));
Certificate myCert = cf.generateCertificate(caInput);
kurentoRoomAPI.addTrustedCertificate("MyServersCertificate", myCert);
kurentoTreeAPI.useSelfSignedCertificate(true);
```

Now the application trusts a server which possesses private key of certificate "my_server_certificate.cer".

WSS support on Android 5.0.x (Lollipop) and up

kurento-room-client-android library uses Maven org.java_websocket: http://mvnrepository.com/artifact/org.java-websocket/Java-WebSocket/

However, org.java_websocket version 1.3.0 is not compatible with Android 5.0.x systems due to malfunction in wss protocol handshake. Until a newer version is uploaded to Maven, a workaround is to compile a newer version from git: https://github.com/TooTallNate/Java-WebSocket

This limitation is known to exist only in wss TSL handshake. Android 5.1.x and up should not have this issue.

API reference

The Javadoc is included in the source code and can be downloaded from the link below: https://github.com/nubomedia-vtt/kurento-room-client-android/tree/master/javadoc