webrtcpeer-android Documentation

Release 1.0.4

Jukka Ahola

Jul 17, 2017

Contents

Table of Contents:

CHAPTER 1

Overview

webrtcpeer-android is a Java library for Android. This library can be used to make p2p multimedia connections between peers connected to the same Kurento server. This documentation provides help on how to install and use the library.

Source code is available at https://github.com/nubomedia-vtt/webrtcpeer-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/webrtcpeer-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:webrtcpeer-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/webrtcpeer-android

and

https://bintray.com/nubomedia-vtt/maven/webrtcpeer-android

Developer installation guide

This documents provides information how to compile the webrtcpeer-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/webrtcpeer-android.git
```

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

https://github.com/nubomedia-vtt/webrtcpeer-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 1.0.10+ 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:webrtcpeer-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/webrtcpeer-android

and

https://bintray.com/nubomedia-vtt/maven/webrtcpeer-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" />
```

Create NBMWebRTCPeer object

First, necessary WebRTC classes have to be imported to the project:

```
import fi.vtt.nubomedia.webrtcpeerandroid.NBMMediaConfiguration;
import fi.vtt.nubomedia.webrtcpeerandroid.NBMWebRTCPeer;
import fi.vtt.nubomedia.webrtcpeerandroid.NBMPeerConnection;
// Also import NBMMediaConfiguration content for ease-of-reading when making␣
↪configuration
import fi.vtt.nubomedia.webrtcpeerandroid.NBMMediaConfiguration.*;
```

Class `NBMWebRTCPeer.Observer` defines interfaces for callback functions from `NBMWebRTCPeer`. One option is to have main class implement this interface:

```
public class MyWebRTCApp implements NBMWebRTCPeer.Observer {
    /* Class declaration */
}
```

Or it can be inherited to a separate class:

```
public class MyObserver implements NBMWebRTCPeer.Observer {
    /* Class declaration */
}

public class MyWebRTCApp {
    MyObserver myObserver;
    /* Class declaration */
}
```

Preparing local video rendering

Changes are that your application will render some video. If you wish to render the video your own device is sending (imagine a small thumbnail video of yourself at the corner of the screen), you can do the following:

```
import org.webrtc.SurfaceViewRenderer;

...
private SurfaceViewRenderer localView;
...

localView = (SurfaceViewRenderer) findViewById(R.id.gl_surface_local);
localView.init(EglBase.create().getEglBaseContext(), null);
```

Assuming that you have a widget `gl_surface_local` of type `SurfaceViewRenderer` in your activity layout. Now you have a widget initialized which can be used to show your own video.

Creating NBMWebRTCPeer

Before creating `NBMWebRTCPeer`, the main component used to setup a WebRTC media session, a configuration object (`NBMMediaConfiguration`) must be defined:

```
/*
    Audio codec: Opus audio codec (higher quality)
    Audio bandiwidth limit: none
    Video codec: Software (VP8)
    Video renderer: OpenGLES 2.0
    Video bandwidth limit: none
```

```
    Video format: 640 x 480 @ 30fps
*/
mediaConfiguration = new NBMMediaConfiguration();
```

Default values can be changed by using an alternative constructor. Different image formats are declared in module `android.graphics.ImageFormat`.

```
import android.graphics.ImageFormat;

...

NBMVideoFormat receiverVideoFormat = new NBMVideoFormat(1280, 720, ImageFormat.YUV_
↪420_888, 30);
NBMMediaConfiguration mediaConfiguration = new NBMMediaConfiguration(NBMRendererType.
↪OPENGLES, NBMAudioCodec.OPUS, 0, NBMVideoCodec.VP8, 0, receiverVideoFormat,␣
↪NBMCameraPosition.FRONT);
```

`NBMWebRTCPeer` is the main component used to setup a WebRTC media session, it must be initialized with a media configuration object (`NBMMediaConfiguration`):

```
NBMWebRTCPeer nbmWebRTCPeer = new NBMWebRTCPeer(mediaConfiguration, this, localRender,
↪ myObserver);
nbmWebRTCPeer.initialize();
```

The following is a minimalistic example of implementing a class with Android WebRTC configured:

```
import org.webrtc.VideoRenderer;
import org.webrtc.VideoRendererGui;
import org.webrtc.RendererCommon;
import org.webrtc.SessionDescription;
import org.webrtc.IceCandidate;
import org.webrtc.MediaStream;
import org.webrtc.PeerConnection.IceConnectionState;
import fi.vtt.nubomedia.webrtcpeerandroid.NBMMediaConfiguration;
import fi.vtt.nubomedia.webrtcpeerandroid.NBMPeerConnection;
import fi.vtt.nubomedia.webrtcpeerandroid.NBMWebRTCPeer;

public class MyWebRTCApp implements NBMWebRTCPeer.Observer {
    private NBMWebRTCPeer nbmWebRTCPeer;
    private SurfaceViewRenderer localView;

    public MyWebRTCApp()
    {
        localView = (SurfaceViewRenderer) findViewById(R.id.gl_surface_local);
        localView.init(EglBase.create().getEglBaseContext(), null);
        mediaConfiguration = new NBMMediaConfiguration();
        nbmWebRTCPeer = new NBMWebRTCPeer(mediaConfiguration, this, localRender,␣
↪this);
        nbmWebRTCPeer.initialize();
    }

    /* Observer methods and the rest of declarations */
    public void onInitialize() { ... }
    public void onLocalSdpOfferGenerated(SessionDescription localSdpOffer,␣
↪NBMPeerConnection connection) { ... }
    public void onLocalSdpAnswerGenerated(SessionDescription localSdpAnswer,␣
↪NBMPeerConnection connection) { ... }
    public void onIceCandidate(IceCandidate localIceCandidate, NBMPeerConnection␣
↪connection) { ... }
```

```
    public void onIceStatusChanged(IceConnectionState state, NBMPeerConnection␣
↪connection) { ... }
    public void onRemoteStreamAdded(MediaStream stream, NBMPeerConnection connection)
↪{ ... }
    public void onRemoteStreamRemoved(MediaStream stream, NBMPeerConnection␣
↪connection) { ... }
    public void onPeerConnectionError(String error) { ... }
    public void onDataChannel(DataChannel dataChannel, NBMPeerConnection connection)
↪{ ... }
    public void onBufferedAmountChange(long l, NBMPeerConnection connection) { ... }
    public void onStateChange(NBMPeerConnection connection) { ... }
    public void onMessage(DataChannel.Buffer buffer, NBMPeerConnection connection) { .
↪.. }
}
```

As shown in the previous code, you may call `nbmWebRTCPeer.initialize()` immediately after the creation of the instance. However, the initialization process, like most of the functionality of the library, is asynchronous, so you want to wait for event `onInitialize()` and proceed in there.

SDP offers

An Offer SDP (Session Description Protocol) is metadata that describes to the other peer the format to expect (video, formats, codecs, encryption, resolution, size, etc). An exchange requires an offer from a peer, then the other peer must receive the offer and provide back an answer:

Offering your own media

Now that you have everything setup for multimedia session, it's time to offer your own video to the server. You can do this by simply generaring and offer by using your newly created `nbmWebRTCPeer` instance:

```
nbmWebRTCPeer.generateOffer("local", true);
```

The first parameter is the connectionid you want to assign to your local video connection. When offering your local media, you simply set the second parameters to true to indicate that you wish to offer your local media instead of offering to receive something, which we will discuss in the next section further. But for now, when the offer is generated, a `onLocalSdpOfferGenerated` callback is triggered. You can catch

```
private boolean isMyVideoPublished = false;

...

public void onLocalSdpOfferGenerated(SessionDescription localSdpOffer)
{
    if (!isMyVideoPublished) {

        /* Do stuff */

        isMyVideoPublished = true;
    }
}
```

The "stuff" of the previous code block can be for example a call to `KurentoRoomAPI.sendPublishVideo` if you are using the Android room API.

Offering to receive from peers

To offer to receive a media from a peer, you make a similar call as before:

```
nbmWebRTCPeer.generateOffer("MyRemotePeer", false);
```

The first parameter is the connectionid you want to assign to your peer. Usually it is a good idea to assign the username of the peer to connection ID, if you are for example using the Kurento room API for Android.

The second parameters is now set to false to indicate that you wish to receive media instead of transmitting it. A `onLocalSdpOfferGenerated` callback is once again triggered. Let's extend the previous code:

```
private boolean isMyVideoPublished = false;

...

public void onLocalSdpOfferGenerated(SessionDescription localSdpOffer)
{
    if (!isMyVideoPublished) {

        /* Do stuff */

        isMyVideoPublished = true;
    } else {
        // We have generated an offer to receive video and we can for example use
        // KurentoRoomAPI.sendReceiveVideoFrom to transmit the offer to the server
    }
}
```

As you have probably noticed from these examples, this library is intended for p2p connectivity only and you most likely want to use a secondary library for dealing with server signaling. Nubomedia project offers two libraries for Android, depending on your use case:

http://doc-kurento-room-client-android.readthedocs.io/en/latest/

http://doc-kurento-tree-client-android.readthedocs.io/en/latest/

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.

Gathering local candidates

After signaling offers, a `NBMWebRTCPeer.Observer.onIceCandicate` callback will be fired each time the ICE framework has found some local candidates. Again if you are using the Kurento Room API for Android you may send the information to the server by issuing to following calls:

```
private boolean isMyIceCandidateSent = false;

public void onIceCandicate(IceCandidate localIceCandidate, NBMPeerConnection␣
↪nbmPeerConnection)
```

```
{
    if (!isMyIceCandidateSent){
        isMyIceCandidateSent = true;
        KurentoRoomAPI.sendOnIceCandidate("MyRoomUserName", iceCandidate.sdp,␣
↪iceCandidate.sdpMid, Integer.toString(iceCandidate.sdpMLineIndex),␣
↪sendIceCandidateRequestId);
    } else {
        KurentoRoomAPI.sendOnIceCandidate(nbmPeerConnection.getConnectionId(),␣
↪iceCandidate.sdp,
                iceCandidate.sdpMid, Integer.toString(iceCandidate.sdpMLineIndex),␣
↪sendIceCandidateRequestId);
    }
}
```

This event handler code will synchronize the local ICE candidates with your signaling server, which in this case is the room server. In practice, now both the server and other users know how to connect to your client, but your client does not yet know how to connect to anything, so we will tackle that in the next section.

Set remote candidates

Whenever you receive a remote candidate from the server, you can process it by simply calling

```
nbmWebRTCPeer.addRemoteIceCandidate(remoteIceCandidate, username);
```

You may refer to the documentation of the signaling server you are using to find out when you will receive ICE candidates.

Set remote renderer

Each connection may invoke `onRemoteStreamAdded` callback function. To display the remote stream, `attachRendererToRemoteStream` can be called inside `onRemoteStreamAdded` callback function. The process is quite the same as with the local renderer. An example for point-to-point video application:

```
private SurfaceViewRenderer remoteRender;

...

public void onRemoteStreamAdded(MediaStream stream, NBMPeerConnection connection){
    (SurfaceViewRenderer) findViewById(R.id.gl_surface);
    masterView.init(EglBase.create().getEglBaseContext(), null);
    nbmWebRTCPeer.attachRendererToRemoteStream(remoteRender, stream);
}
```

Using data channels

WebRTC DataChannel between peers is available via the API. First, in your `MyWebRTCApp` declare a `DataChannel.Init`:

```
DataChannel.Init dcInit = new DataChannel.Init();
```

Please refer to Google WebRTC documentation on configuring `DataChannel.Init`. Once you have configured the object accordingly, create datachannels for your the peer connections of your choice. For example, to create one data channel for all active peers in `MyWebRTCApp`:

```
for(NBMPeerConnection c : connectionManager.getConnections()){
    c.createDataChannel("MyDataChannelLabel", mediaManager.getLocalMediaStream());
}
```

The first parameter of `createDataChannel` is a non-unique label which can be used to identify given data channel. However, all callback events of data channels have `NBMPeerConnection` as function arguments, which may provide also unique per-peer identification of the channel. In a case that multiple data channels are established to the same peer, or a quick verbose identifier is required, label is a good choice, otherwise use the `NBMPeerConnection` instance for identification. You may also save a reference, as `createDataChannel` function returns a pointer to the newly created data channel.

Closing data channels explicitly is not necessary, as they are closed along with connections. But whenever it makes sense for e.g. user experience, call `DataChannel:close()`.

Datachannels use the following callbacks:

```
// Triggered when peer opens a data channel
public void onDataChannel(DataChannel dataChannel, NBMPeerConnection connection) {
    ...
}
// Triggered when a data channel buffer amount has changed
public void onBufferedAmountChange(long l, NBMPeerConnection connection) {
    ...
}
// Triggered when a data channel state has changed. Possible values: DataChannel.State
↪{CONNECTING,OPEN,CLOSING,CLOSED}
public void onStateChange(NBMPeerConnection connection) {
    ...
}
// Triggered when a message is received from a data channel
public void onMessage(DataChannel.Buffer buffer, NBMPeerConnection connection) {
    ...
}
```

API reference

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