EasyChair Preprint

№ 4304

Peer to Peer Audio and Video Communication using WebRTC

Kushtrim Pacaj, Kujtim Hyseni and Donika Sfishta

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

October 1, 2020

Peer to Peer Audio and Video Communication using WebRTC

Kushtrim Pacaj - Mobile
kp48560@ubt-uni.net

Kujtim Hyseni - Web and server side
kh48497@ubt-uni.net

Donika Sfishta - Mobile
ds50723@ubt-uni.net

University for Business and Technology, 10000, Prishtine

Abstract. In this paper we introduce a simple way on how you can build your own "end-to-end" audio and video chat. You can expect to learn how to add new features on it, as well as building your own signaling server, your mobile UI and mobile calls logic implemented in a perfect coexistence between NodeJS, Java and Kotlin. What makes this application special is the possibility to express yourself with different annotations while you're on a video chat.

1. Introduction

Audio and video communication has had surges in usage numbers in the last couple of months. Whether we measure by new users acquired per month, or by minutes of meetings held, all providers have seen a boom in popularity due the COVID pandemic.
Building a video communication app or device is seen by many as a complicated task, containing many difficult to master components such as the initial communication setup via peers (especially when needing to traverse NATs), stream negotiation, encoding/decoding of audio and video streams.
And while many of those components are difficult, there exists an open-source project called WebRTC[1] which aims to make it simpler.

Our project uses the WebRTC API and it's library built for Android, in addition to a simple signaling server built using Socket.IO in order to achieve the P2P video/audio call functionality. And to make it a little more fun, we decided to add a new feature, which will allow the user to "draw" on top of the call view, and those drawings will be shown on the other side as well.

The paper will first explain some background on how WebRTC works, and then we will detail how our app uses it.

2. Design and Implementation

To understand how a WebRTC call works there are several components that are important:

1. Signaling Server for pre-call communication
2. SDP and offer/answer flow
3. ICE candidate exchange
4. Setting up MediaTracks and rendering them

2.1 Signaling Server

Even though WebRTC gives the possibility of peer to peer connection we need to exchange some data between peers firstly and "ask" some questions and address some issues first like:

- Does the other person want to answer the call?
- Can you answer (are you on another call?)
- What will the call be (audio/video?) What codecs will be used?
- How to reach the other person? Where to send the streams?

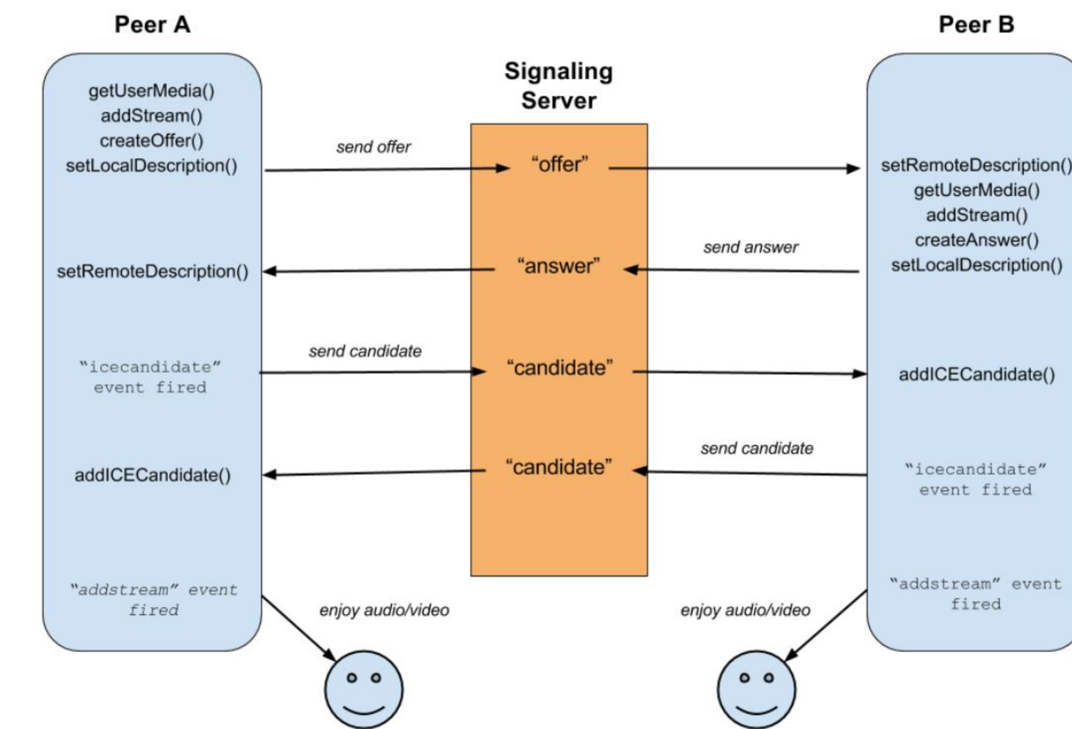

Figure 1. Architecture of Signaling Server in a peer to peer connection

So, while the WebRTC is a peer to peer technology, it can’t work by itself.
It first needs an existing medium of communication between the peers in order to set up the call. This medium of communication is called a Signaling server.
The WebRTC specs *intentionally* don't specify what this signaling channel is ; it’s up to the application developer to implement one. It could theoretically be anything, even using the postal system and sending letters, or IPoAC (RFC 1149)... Though that would make setting up a call take weeks…
The signaling channel we implemented is based on Socket.IO, which is a protocol for real-time communication between server and client, allowing two-way communications between them.
Socket.IO internally uses websocket technology whenever possible, and fallbacks to long-polling if websockets aren’t available.

The methods implemented by our signaling server are shown in the table below

Listens on this event from Peer A	It then emits this event to Peer B
saveUserInfo	onUserOnline
callUser	onIncomingCall
answerCall	onCallAnswered
declineCall	onCallDeclined
endCall	onCallEnded
sendOffer	onReceivedOffer
sendAnswer	onReceivedAnswer
sendIceCandidate	onReceivedIceCandidate
sendFabricPath	onReceivedFabricPath
disconnect	onUserDisconnected

Our implementation of signaling is a barebones one, it only contains things that are absolutely necessary to set up a call. A signaling server in a *real* app will most certainly implement more functions, including ability to set up a group call, to toggle audio/video streams on/off etc. It would also likely have some in memory DB (such as REDIS) with multiple servers, and proxies between user and server.
For our example, the methods above suffice.

2.2 SDP - Session description Protocol

SDP is used to describe the session (what streams, how many, what kinds, what codecs etc.).
The structure of SDP is shown below:[2]

Session description
v= (protocol version number, currently only 0)
o= (originator and session identifier : username, id, version number, network address)
s= (session name : mandatory with at least one UTF-8-encoded character)
i=* (session title or short information)
u=* (URI of description)
e=* (zero or more email address with optional name of contacts)
p=* (zero or more phone number with optional name of contacts)
c=* (connection information—not required if included in all media)
b=* (zero or more bandwidth information lines)
*One or more **Time descriptions** ("t=" and "r=" lines; see below)*
z=* (time zone adjustments)
k=* (encryption key)
a=* (zero or more session attribute lines)
*Zero or more **Media descriptions** (each one starting by an "m=" line; see below)*

Time description (mandatory)
t= (time the session is active)
r=* (zero or more repeat times)

Media description (if present)
m= (media name and transport address)
i=* (media title or information field)
c=* (connection information — optional if included at session level)
b=* (zero or more bandwidth information lines)
k=* (encryption key)
a=* (zero or more media attribute lines — overriding the Session attribute lines)

For example, in order to specify that we want to encode a video stream and that we support H264 and H263 then we would the SDP would look similar to this:

m=video 16446 RTP/AVP 98 99

a=rtpmap:98 H264/90000
a=rtpmap:99 H263-1998/90000

We should note that in most cases, sdp is not modified manually, but using a higher-level WebRTC API. Though in more complex applications, it is sometimes done in cases where the high-level API is lacking or non-existent. This is called SDP munging.
A use-case of SDP munging would be if we want to prioritize a specific video encoding algorithm, then the only way to do that currently is by modifying the m line and changing the numbers that refer to encoders (in our example by swapping 98 <-> 99).

2.3 ICE - Interactive Connectivity Establishment

While SDP is used to describe the session, the ICE is used to describe how that "session" will be connected.
In short, it's used to know how to connect to the other peer.
When a PeerConnection is created for a peer, a series of ICE candidates are generated. These describe ways to reach us. They are sent to the other peer via signaling.

The contents of the ICE candidates are described in the picture below:

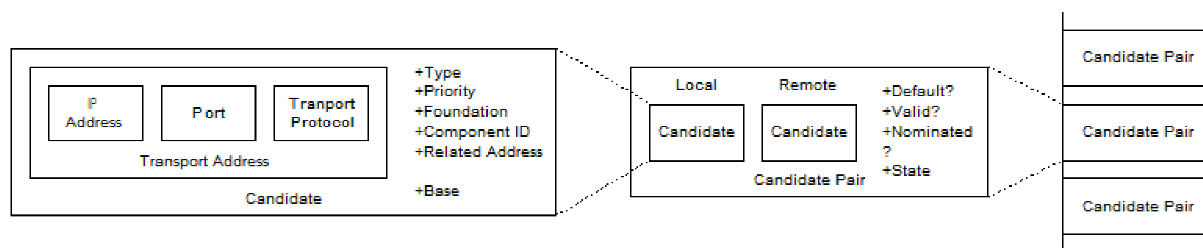

Figure 2. Candidate Pair Components

There are three types of candidates:

Host – candidate showing data about local interface (local IP)
Reflexive – candidate showing *IP:port* of us as seen from outside NAT (calculated by sending a request to STUN server)
Relayed – candidate telling the TURN server that can be used if P2P can't be connected.

The idea behind this is that we need to find the other peer, but preferably use the less costly and shortest route.
Now in the case that the other peer is on our local network, when they receive our ICE of type Host, they will try to "ping" us on that IP, and actually get through. In this case the P2P communication would be established without going through the internet at all.

If that first step doesn't succeed, then WebRTC would try to use the *Reflexive* ICE, meaning try to connect to our PublicIP:port combo.
It is estimated[3] that about 80% of communication can succeed this way.

The rest are cases when the Peer is behind a symmetric NAT (which uses different outgoing ports for different endpoints). If we fall into this category then we will try the *Relayed* ICE, which is actually some media server that both peers connect to and use it to relay their feeds. In this case the connection is no longer truly Peer-to-peer.

In our example we added public STUN servers, but no TURN server (you have to buy a server and run it).

2.4 Android app client: Call flow

A simple class diagram of the app is shown below.

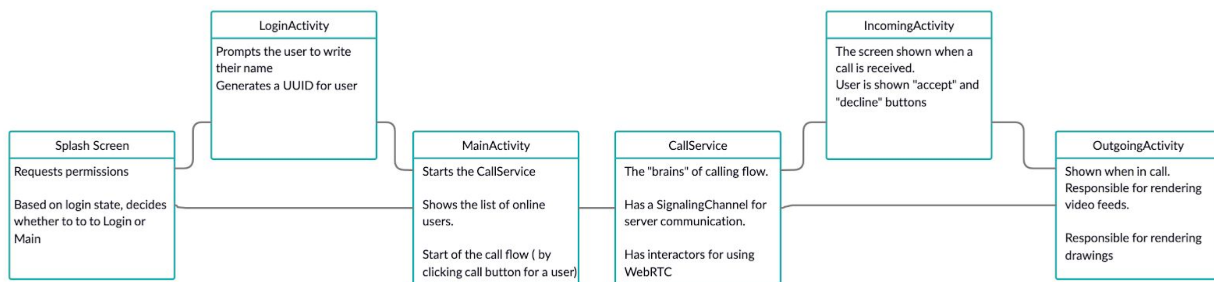

Figure 3. Android app class diagram

The first screen shown is the SplashScreen.
Since the app needs *CAMERA* / *RECORD_AUDIO* to make a call, we ask the user to grant these permissions at this stage.
If they are granted, then at first, the user is sent to a Login screen to fill out their name.
In this stage, this is enough, since we don't need authentication for the demo app. In a production app, we would need to implement an identity server too, and signup/login using it.

After the user logs in, we show the MainActivity screen.
This is where a list of online users is shown. To call one of them, you just have to click the "Call user" button next to their name.

CallService is the main part of the app in regards to calls. When started, it keeps a live connection with the server using SignalingChannel (in our case Socket.IO).

When a user wants to start a call, the main screen triggers a local event to CallService to kickstart the calling flow.
Service starts the call view, and using signaling, sends a "*callUser*" event to the other peer.

When the other peer receives that, it shows an incoming call screen, and two buttons (“accept”/”decline”). When the user accepts, then an *onCallAnswered* event is sent to the initiator, which begins the “WebRTC flow”.

The WebRTC flow is shown in the graph above, under signaling server.
To summarize, media streams are created and added to PeerConnections, SDP offer / SDP answers are exchanged between peers (using signaling), and ICE candidates as well.

2.4.1 Code snippets about WebRTC part

Code is structured such as we have an WebRtcInteractor class than handles communication with WebRTC APIs.

```
public WebRtcInteractor(SignalingChannel signalingChannel) {

   this.signalingChannel = signalingChannel;
   factory = createPeerConnectionFactory();
   localMediaStreamInteractor = new LocalMediaStreamInteractor(factory);
   peerConnectionInteractor = new PeerConnectionInteractor(factory);
   iceCandidateInteractor = new IceCandidateInteractor();

}
```

Factory is part of WebRTC API that is used to create peer connections and tracks.
LocalMediaStreamInteractor is a class that we wrote in that manages internally the creation of media tracks.
PeerConnectionInteractor manages the currently connected peers.
IceCandidateInteractor is used to set or queue ice candidates if they’re received before the offer/answer is exchanged.

After the call is established, on *SdpObserver* we will receive a MediaTrack instance, which represents the audio or video of the other peer.
In the case of the audio, that is managed by itself and played on phone speakers.
For the VideoTrack we need to render it. This is done by creating a renderer on our screen, and attaching the track to it:

```
<org.webrtc.SurfaceViewRenderer
   android:id="@+id/localRenderer"
   android:layout_width="wrap_content"
   android:layout_height="144dp"
   android:layout_margin="16dp" />
```

```
//…………...

with(binding.localRenderer) {
  setZOrderMediaOverlay(true)
  setScalingType(org.webrtc.RendererCommon.ScalingType.SCALE_ASPECT_BALANCED)
  setMirror(false)
  requestLayout()
  initWithDefaultEglContext()
}
//……………

@Synchronized
fun SurfaceViewRenderer.replaceTrackInRenderer(newVideoTrack: VideoTrack) {

  val oldVideoTrack = this.getTag(R.id.videoTrack) as VideoTrack?

  if (oldVideoTrack != newVideoTrack) {
     oldVideoTrack?.removeSink(this)
     newVideoTrack.addSink(this)
     setTag(R.id.videoTrack, newVideoTrack)
  }
  visibility = View.VISIBLE

}
```

2.5 Android app client extra feature: Annotations on top of live call

Another interesting addition is sending drawings to the other peer. This is a somewhat unique feature, not something generally seen in any video calling applications.
We used FabricView[4] library to get user events and draw them, but we simplified it a lot by removing unneeded functionality from it.
Each time a drawing is made by the user, the drawn path is calculated, and serialized to JSON.
That JSON is sent to the other peer, which deserializes it and renders it on the screen.
Annotations disappear by itself after 2 seconds. The idea was for these annotations to convey a temporary message, and not obstruct the video call.
Sending of the drawings for the moment is done by relaying them through the server for simplicity, but they could've also been implemented using WebRTC data channels in order to make this feature P2P too.

3. Testing and Validation

WebRTC API is one of the most used ones when it comes to low latency peer to peer audio and video calls. WebRTC performance can be measured and seen not only in our application, but in applications like Google Meet, Amazon Chine, Facebook Messenger etc. also. To test our implementation, as of this day a user must clone (download) our repos in github. After that, the user must first start our signaling

server and then make a call. You will find our repositories links and demo videos links in Appendix A. When the users are in a call, they can annotate or draw live in the camera feed and the lines will be shown to the far end user. This can be used to make the call more attractive to the young users, or to show something in the camera feed to the other user, or be used as a board where you can explain different things by drawing. Next step here could be adding AR capabilities while on a call.

4. Related works

There are many Audio-Video communications platforms nowadays, like Zoom, Teams, Meet etc. In a way, they all could be said to be related works.
But not all of those are using WebRTC. Of those mentioned above, only Meet uses WebRTC.
But in the context of open-source WebRTC examples, the only relevant one is AppRTC[5] sample developed by Google

5. Conclusion

In order to develop a peer to peer audio and video call, the best way is to use a ready made API for calls. In our case we used WebRTC, but as can be seen from our code which can be found in github, you need expertise on each field that you want to implement it, meaning that if you want to implement it in Android, you need Android developers. Same goes for iOS, web or windows. Also, it is mandatory to build a server where SDP would be exchanged before an actual call is made, so having some expertise in web development is also preferable. You can build features on top of WebRTC, like annotations or drawings. Latency is very small and inside allowed limits.

List of figures

*Figure 1. Architecture of Signaling Server in a peer to peer connection (diagram by **satanas** : https://raw.githubusercontent.com/satanas/simple-signaling-server/master/doc/RTCPeerConnection-diagram.png)*

Figure 2. Candidate Pair Components https://www.vocal.com/networking/ice-interactive-connectivity-establishment/

Figure 3. *Android app class diagram*

References:

1. The WebRTC project: https://webrtc.org/
2. RFC4566 (SDP: Session Description Protocol): https://tools.ietf.org/html/rfc4566
3. WebRTC turn : https://bloggeek.me/webrtc-turn/
4. FabricView library: https://github.com/antwankakki/FabricView
5. AppRTC: https://chromium.googlesource.com/external/webrtc/+/refs/heads/master/examples/androidapp/

Appendix A

Users who want to test, use or build something on top of our application can find the two related repositories on the links below:

https://github.com/KushtrimPacaj/AP.Project.WebRTC.Signaling
https://github.com/KushtrimPacaj/AP.Project.WebRTC.Mobile

On the links below, you can find our demo videos:

https://www.youtube.com/playlist?list=PL8PtQZl3muhfckHkl6gNahE_cGlHW_l1O&fbclid=IwAR1t4Ve_6Lz5nQKpeY7hYmqGunkklf-uGlSeOvGtHO9mRWJutybP0w1xG-g