Oct 31st, 9:00 AM - 10:30 AM

Peer to peer Audio and Video Communication

Kushtrim Pacaj
University for Business and Technology - UBT

Kujtim Hyseni
University for Business and Technology - UBT

Donika Sfishta
University for Business and Technology - UBT

Peer to peer

Audio and Video

Communication

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

Even though today we have many mobile audio and video chat applications, we decided to make a new one since they all seem to be the same meaning that, despite some better or worse call performance, there's nothing innovative in it. We added a cool feature so video calls will not just be simple video calls but you can annotate on top of it. So, our application mainly targets young people who we think will find this feature cool. Of course, this feature is in addition to low latency and good quality call. We decided to use WebRTC for audio and video calls as in our opinion and not only, it is the best technology to use for peer to peer calls providing good quality and low latency. Even though the call is peer to peer, first you need a server to exchange data between participants. That's why we made a signaling server using NodeJS and SocketIO.

2.Design and Implementation

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?

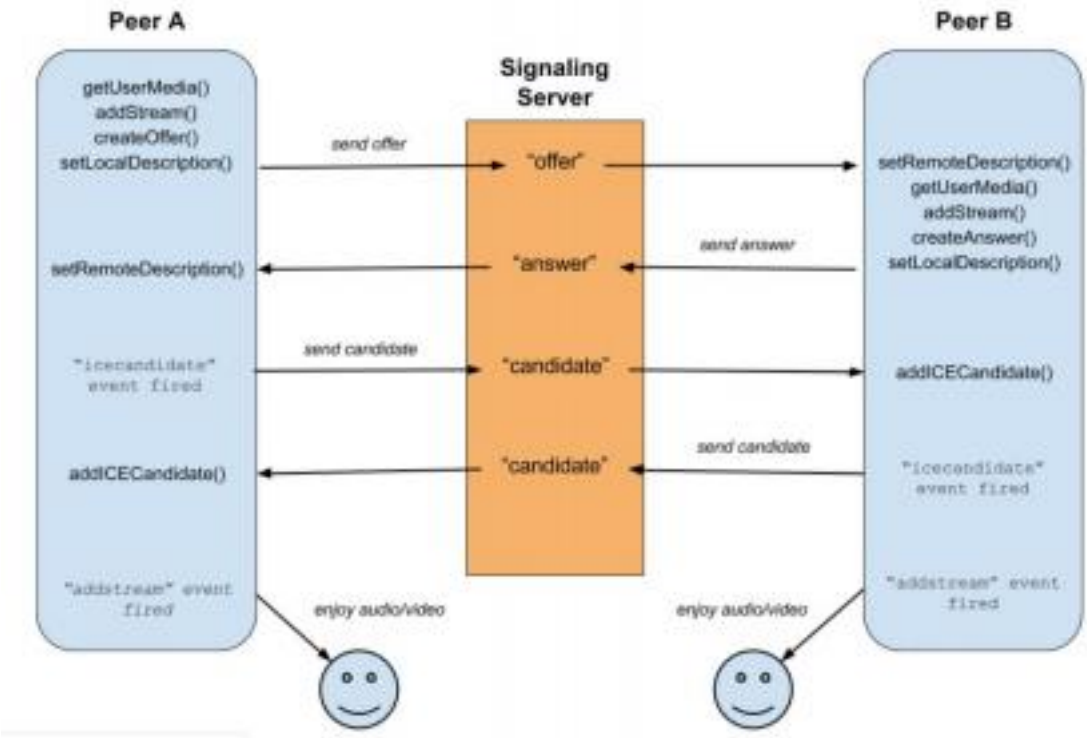


Figure 1. Architecture of Signaling Server in a peer to peer connection

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… 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.

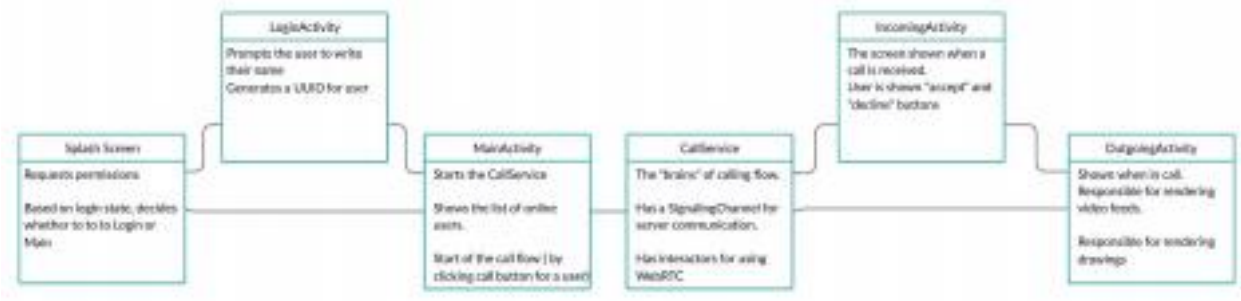
The methods implemented by our signaling server are shown in the table below

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 Android app client.

A simple class diagram of the app is shown below.



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 (https://identityserver4.readthedocs.io/en/latest/), 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.

2.2.1 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 (taken from wikipedia):

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)

2.2.2 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:

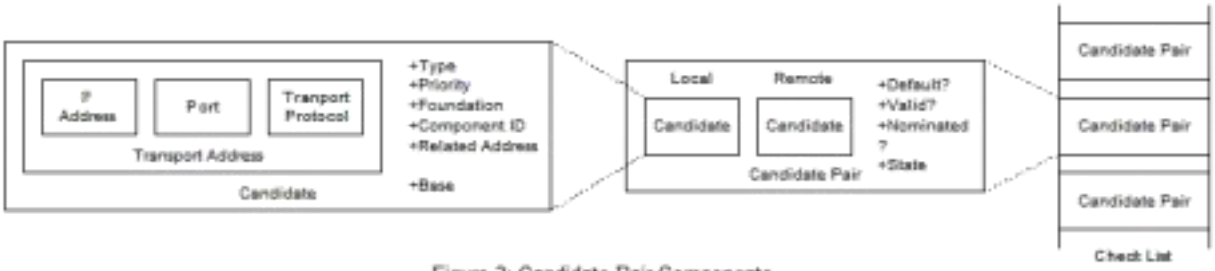


Figure 3: Candidate Pair Components

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.

In our example we added public STUN servers, but no TURN server (you have to buy a server and run it). Besides in our example, at this stage all the communication is done within a Local Network, not even a STUN server would be needed.

2.2.2 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 this library (https://github.com/antwankakki/FabricView) to get user events and draw them, but we simplified it a lot by removing unneeded functionality from it. We also added two features on top of it:

1. Annotations disappear by itself after 2 seconds. The idea was for these annotations to convey a temporary message, and not obstruct the video call.
2. Serializing/deserializing the drawn paths by user and sending them to the other peer to be shown.

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.

2.3 Other apps ..

As a base on how WebRTC works and how to use it, the AppRTC example from Google:
https://webrtc.googlesource.com/src/+/refs/heads/master/examples/androidapp

There are many many video calling apps, and lots of them use WebRTC, including "Google Meet" which was used to present the application.

This app was intentionally kept simple, it's meant only as exposition on how to implement such an app given the time we had to finish it.

Many more time and resources would be needed to make a full-fledged app for audio-video communication. Also, this app for the moment works only in Android, so for the future we would have to implement it for iOS and Web. This way, we need more resources like web developers, iOS developers etc...

3. 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.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 mandatory. You can build features on top of WebRTC, like annotations or drawings. Latency is very small and inside allowed limits.

References:

- https://github.com/satanas/simple-signaling-server:
- https://github.com/socketio/socket.io
- https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity ● https://webrtc.org/getting-started/overview
- https://temasys.io/webrtc-ice-sorcery/
- https://www.vocal.com/networking/ice-interactive-connectivity-establishment/

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