|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **C:\Users\Admin\AppData\Local\Microsoft\Windows\INetCache\Content.Word\VIT new logo.png**  **School of Electronics Engineering (SENSE)** | | | | |
| **PROJECT BASED LEARNING (J Component) - REPORT** | | | | |
| **COURSE CODE / NAME** | ECM2001– Data Communication Networks | | | |
| **PROGRAM / YEAR** | B.Tech (Electronics and Computer Engineering) | | | |
| **LAST DATE FOR REPORT SUBMISSION** | 11 December 2021 | | | |
| **DATE OF SUBMISSION** | 11 December 2021 | | | |
| **TEAM MEMBERS**  **DETAILS** | **REGISTER NO.** | **NAME** | | |
| 19BLC1037 | Shivansh | | |
| 19BLC1046 | Pratyush Motha | | |
|  |  | | |
|  |  | | |
|  |  |  | | |
| **J TITLE** | **Video Calling Application using WebRTC** | | | |
| **COURSE HANDLER’S NAME** | **Dr. T. Jayavignesh** | | **REMARKS** |  |
| **COURSE HANDLER’S SIGN** |  | |
|  |  | |  |  |

**Table of Content**

Abstract3

Team Members4

[1.](#_1fob9te) Introduction 4

[2.](#_2et92p0) Algorithm 6

[3.](#_3dy6vkm) Implementation 7

[3. Coding](#_3dy6vkm) 8

[3.1](#_1t3h5sf) Implementation for backend 8

[3.2](#_4d34og8) Frontend, WebRTC, SocketIo implementation 10

4[.](#_3dy6vkm) Results and Inferences 16

5[.](#_3dy6vkm) APPLICATION ORIENTED LEARNING 17

6. Conclusion 22

7. References 22

**ABSTRACT:**

We have built a video calling application, the application is similar to google meet or zoom. Just like google meet or zoom, the users can generate a unique meeting id and share it with other people to join the video meeting together. The application uses an open source library called webRTC for real time audio video transmission from one node to another node. For this real time communication to take place there is a peer id, which needs to be shared with all participants of the meeting that the new user wants to join. This could be done through various different approaches but the modern and efficient one is using web sockets. The whole idea is to transfer the peer id of a node to others nodes using web sockets which will happen over the private server that we will own. To implement this operation we have Socket.IO, which is also an open source library. Now we also need to build a user interface and power the backend server for the application. For this purpose we have used React.Js which is also an open source UI manipulation library provided by Facebook. React makes it really easy to implement the UI part using Javascript as the primary programming language because it works on the principle of components which are reusable, so it prevents developers from writing the same code again and again. To power the server for the application, over which the unique meeting id generation and passing the peer id will take place. The whole algorithm that we have implemented to connect users at a particular meeting ID is explained in the ALGORITHM section. To access the microphone and camera of the user we have used the navigation module provided by browser APIs. The underlying principles for using the core tools are as follows. Unlike all other browser communication which use Transmission Control Protocol (TCP), WebRTC transports its data over User Datagram Protocol (UDP). The requirement for timeliness over reliability is the primary reason why the UDP protocol is a preferred transport for delivery of real-time data. Socket.IO primarily uses the WebSocket protocol with polling as a fallback option, while providing the same interface. Although it can be used as simply a wrapper for WebSocket, it provides many more features, including broadcasting to multiple sockets, storing data associated with each client, and asynchronous I/O.

**TEAM MEMBERS:**

****

**Shivansh Pratyush Motha**

**19BLC1037 19BLC1046**

**INTRODUCTION:**

The ongoing situation of Covid -19 had everyone of us work from home it’s during this time we got more familiar with video conferencing platforms and application like Zoom, Google meet, and Microsoft Teams and have use them more often during this time for various purposes whether to attend online classes or to call our friends or to attend meetings online, this technology help the world to stay connected during pandemic and now is a part of our daily life , So out of curiosity we had for this technology we researched for it wanting to know how it works from the inside handling our daily lives with the click of a button, as we searched we got to know about WebRTC. This technology is the backbone for major video conferencing applications, now before beginning we must understand how simple chat applications work. When two browsers need to send messages to each other, they typically need a server in between for coordination and passing the messages. But having a server in the middle results in a delay in communication between the browsers. This delay hardly affects the utility of the chatting app. Even if this delay is (say) 5 secs, we would still be able to use this chatting application.

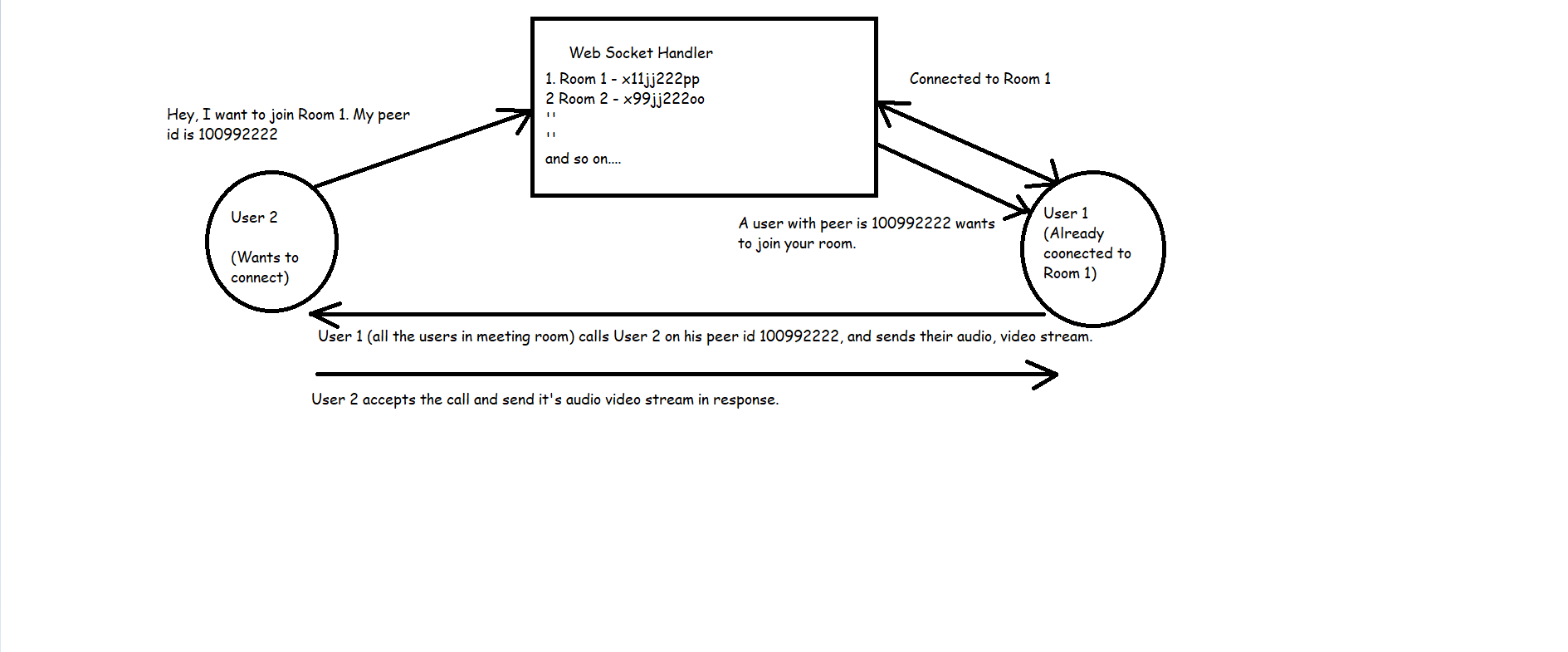
However, in the case of a video conferencing application, this delay is significant. It will be extremely difficult to talk to someone using such an application. Imagine yourself talking to someone who receives your voice 5 secs later. You can realize how annoying it will be.

Hence, for video conferencing, we require Real-Time Communication between the browsers. Such communication is possible if we eliminate the server from between. This is why we will have to use WebRTC — an open-source framework providing web browsers and mobile applications with real-time communication via simple APIs.

WebRTC stands for Web Real-Time Communication. It enables peer-to-peer communication without any server in between and allows the exchange of audio, video, and data between the connected peers. With WebRTC, the role of the server is limited to just helping the two peers discover each other and set up a direct connection.

To build the application we have used WebRTC and Socket.io for real time communication purposes. WebRTC is a free and open-source project providing web browsers and mobile applications with real-time communication via simple application programming interfaces. WebRTC also provides several APIs that make it easy to contact browsers to get access to microphone, camera and several other output devices. Socket.IO is a JavaScript library for realtime web applications. It enables real-time, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Both components have a nearly identical API.

**ALGORITHM:**

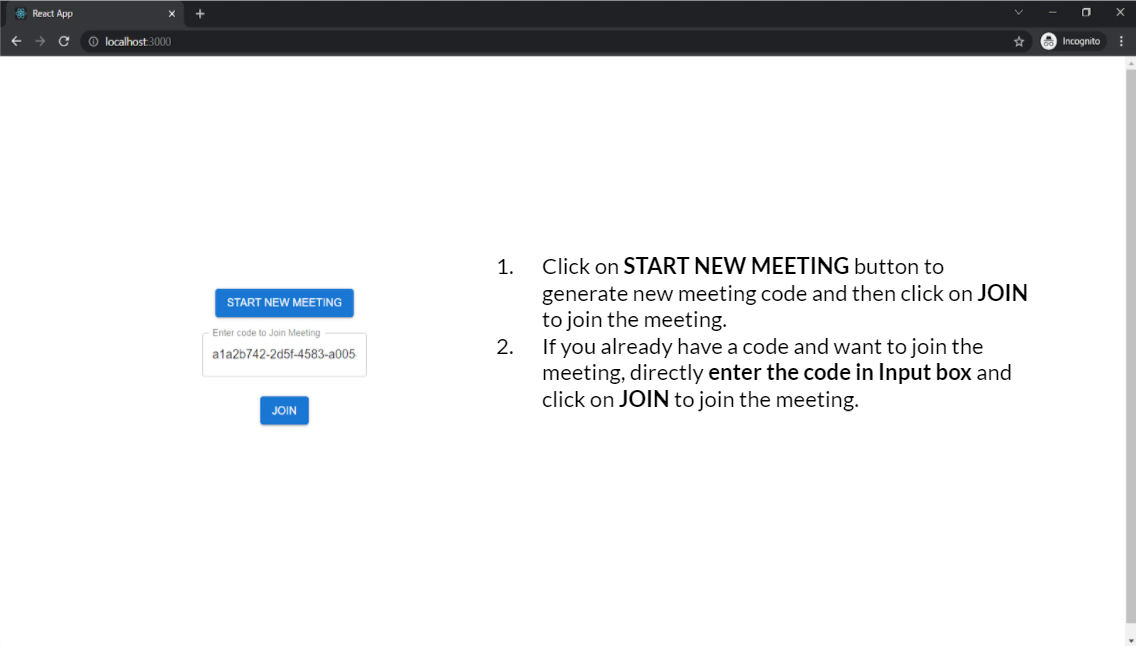


**ALGORITHM STEPS**

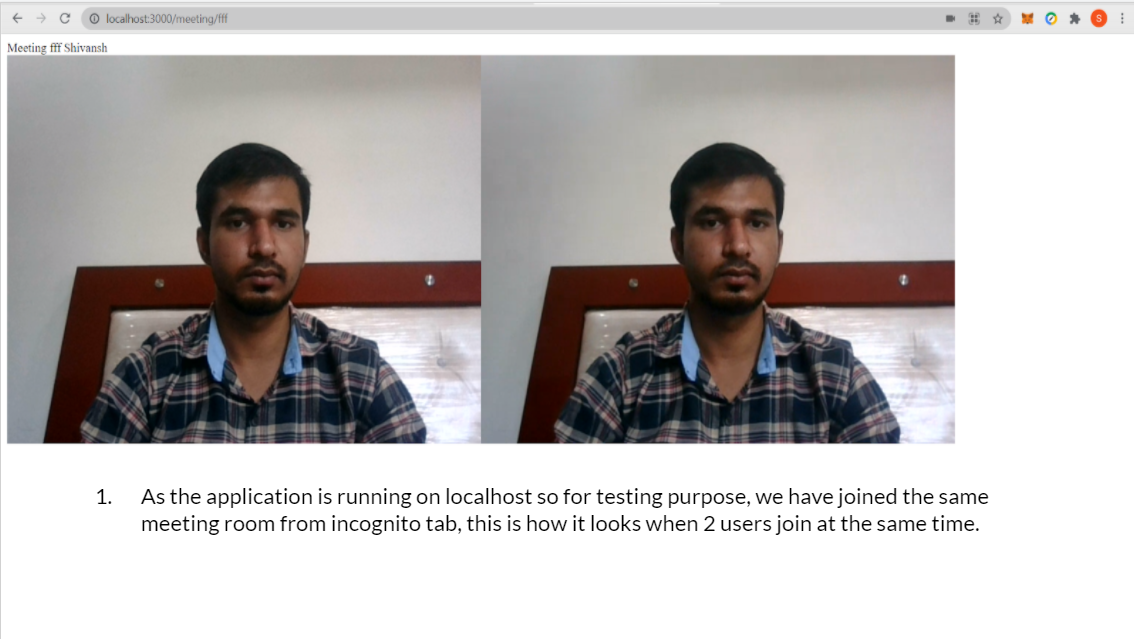
1. Web Socket handler manages rooms, where the events are broadcasted as per requirement.
2. To simulate it's working, let's assume that User 1, User 2 and User 3 are already connected in Room 1.
3. Whenever some new user (User 5) requests the web socket handler to join Room 1, it will broadcast an event (data: <-peer-id-of-User-5->) in Room 1 stating that User 5 wants to join.
4. Then all the users will call User 5 individually, establishing a peer-2-peer connection.

**IMPLEMENTATION (REAL TIME):**

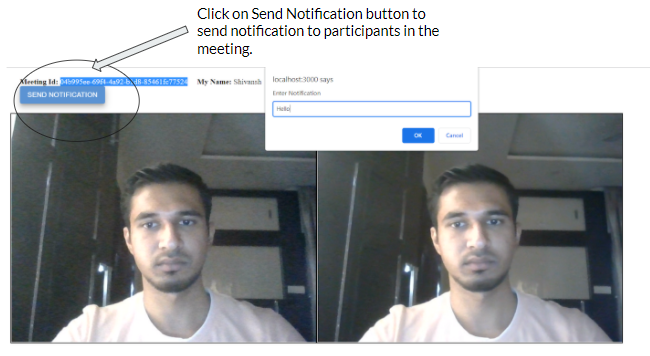
Join Meeting/ Create new unique meeting ID.

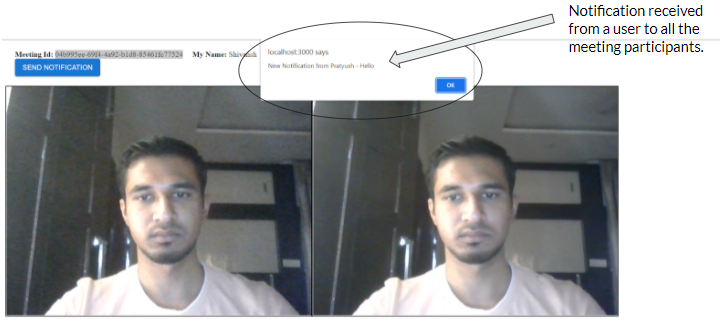


2. The meeting window



3. Sending Messages during meetings.





**CODING:**

Implementation of backend service:

const { v4: uuidv4 } = require('uuid');

const server = require('http').createServer((req, res) => {

res.setHeader('Content-Type', 'application/json')

const headers = {

"Access-Control-Allow-Headers": "Content-Type, Authorization",

"Access-Control-Allow-Origin": req.headers.origin, //or the specific origin you want to give access to,

"Access-Control-Allow-Credentials": true

};

res.writeHead(200, headers);

if (req.url === '/newCall') {

let id = uuidv4()

res.end(JSON.stringify({ id: id, message: "success" }))

} else {

res.end(JSON.stringify({ error: "not found" }))

}

});

const io = require("socket.io")(server,

{

cors: {

origin: "\*",

methods: ["GET", "POST"]

}

});

server.listen(5000, 'localhost', () => {

console.log('Server Listening at PORT 5000')

})

io.on('connection', client => {

client.on('joinroom', data => {

client.join(data.room);

io.to(data.room).emit("addpartcipant", data)

console.log('Room ', data)

client.on('send-message', ({ from, message })=>{

console.log("Message ", message)

client.broadcast.to(data.room).emit('new-message', {from, message})

});

});

});

**Frontend, WebRTC, SocketIo implementation:**

Homepage:

import React, { useState } from 'react';

import Button from '@mui/material/Button';

import TextField from '@mui/material/TextField';

import { useHistory } from "react-router-dom";

import URL from '../config/baseurl'

import './css/home.css'

export default function SimpleContainer() {

const [code, setCode] = useState('')

let history = useHistory()

const newMeeting = async()=>{

let url = URL + 'newCall'

try{

const response = await fetch( url, {

method: 'GET',

headers: {

'Content-Type': 'application/json'

}

})

let ans = await response.json()

setCode(ans.id)

} catch(err){

console.log('Cannot get meeting code')

}

}

const joinMeeting = async()=>{

if(!code) return

history.push(`/meeting/${code}`)

}

return (

<div className="homepage\_main">

<div className="homepage\_main\_left">

<div className="homepage\_main\_left\_button">

<Button variant="contained" onClick={newMeeting}>Start New Meeting</Button>

</div>

<div className="homepage\_main\_left\_input" style={{ margin: 20 }}>

<TextField label="Enter code to Join Meeting" variant="outlined" value={code} onChange={(event)=>setCode(event.target.value)} />

</div>

<div className="homepage\_main\_left\_button2" style={{ margin: 5 }}>

<Button variant="contained" onClick={joinMeeting}>Join</Button>

</div>

</div>

<div className="homepage\_main\_right">

<div></div>

</div>

</div>

);

}

Meeting Page:

import React, { useEffect, useState, useRef } from 'react'

import {

useParams

} from "react-router-dom";

import { io } from 'socket.io-client';

import { useHistory } from "react-router-dom";

import Peer from 'peerjs';

import URL from '../config/baseurl'

import Button from '@mui/material/Button';

const socket = io(URL)

function Meeting() {

let { id } = useParams();

const [name, setName] = useState('')

const [peers, setPeers] = useState({})

const history = useHistory()

const myVideo = useRef() // reference to local video

// constraints

function setConstraints() {

const constraints = {

// 'audio': {

// 'echoCancellation': true,

// },

'audio': false,

'video': true

}

return constraints

}

// open stream

async function openStream() {

let cons = setConstraints()

return await navigator.mediaDevices.getUserMedia(cons);

}

// add video stream to grid

function add(videoStream, name) {

let Scenary = document.getElementById('Dish');

let Camera = document.createElement('video');

Camera.className = 'Camera';

Camera.setAttribute('id', name)

Camera.srcObject = videoStream

Camera.addEventListener('loadedmetadata', () => {

Camera.play()

})

Scenary.appendChild(Camera);

}

// remove

function less(name) {

let removeCamera = document.getElementById(name)

removeCamera.parentNode.removeChild(removeCamera)

}

useEffect(async () => {

if (socket.connected === false) {

history.push(`/`)

alert("Couldn't connect to socket server at the movement")

return

}

let n = window.prompt('Enter Name')

setName(n)

socket.emit("joinroom", { name: n, room: id });

socket.emit("send-message", { from: n, message: `${n} Joined the Meeting` });

let myStreamGlobal = await openStream()

myVideo.current.srcObject = myStreamGlobal

let peer = new Peer(n)

// listen for incoming calls

peer.on('call', call => {

// metadata from call

let metadata = call.metadata

let name = metadata.name

// send you stream

call.answer(myStreamGlobal)

// append user stream to grid

call.on('stream', (stream) => {

console.log('Getting Peers stream')

add(stream, name)

})

})

socket.on("addpartcipant", (data) => {

console.log('New Participant ', data)

connectPeers(data.name)

});

socket.on("new-message", (data)=>{

alert(`New Notification from ${data.from} - ${data.message}`)

})

// when someone joins our room, call them using their peer id

function connectPeers(name) {

// send userId in metadata of call

let options = { metadata: { "name": name } }

const call = peer.call(name, myStreamGlobal, options)

call.on('stream', userVideoStream => {

console.log('Getting User video stream')

add(userVideoStream, name)

}

)

// close video

call.on('close', () => {

less(name)

})

}

}, [])

const sendMessage = ()=>{

let message = window.prompt('Enter Notification')

socket.emit("send-message", { from: name, message: message });

}

return (

<div>

<div style={{margin: 20}}>

<b>Meeting Id:</b> {id}

&nbsp; &nbsp; &nbsp;<b>My Name:</b> {name}

<br></br>

<Button variant="contained" onClick={sendMessage}>Send Notification</Button>

</div>

<div id="Dish">

<video ref={myVideo} autoPlay className="Camera" muted="muted"></video>

</div>

</div>

)

}

export default Meeting

**RESULTS & INFERENCES:**

As we did accomplish a real-time video conferencing application using WebRTC on small scale operations we wanted to explore the disadvantages and limitations that would occur on a large scale operation so we did researched on that and got to now following points

* As WebRTC is browser based, browsers cannot synchronize multiple incoming streams, you need a video conferencing server for audio and video mixing to run group audio or video conferences.
* Group conference mixing and transcoding requires large computing resources. Group video conference support for WebRTC usually requires paid subscription (in case of cloud solutions) or big infrastructure investments, as each conference layout usually requires at least one logical CPU core on the server.
* WebRTC solutions are incompatible with each other. The standard generally applies only to the methods of video and audio transmission, while vendors are free to decide on signalling, messaging, file transfer, conference scheduling, etc.
* Next questions was how many peers can connect to webRTC so we got the answer theoretically speaking as many as we like but practically it comes down to speed and feeds and large computing resources done by browser.

1. **Speeds** – the resolution and bitrate we’re expecting in our service
2. **Feeds** – the stream count of the single session

**APPLICATION ORIENTED LEARNING:**

WebRTC applications offer easy peer-to-peer voice and video communication in situations where a standard phone call isn't optimal. To give you a sense of what WebRTC is capable of, and how it can be used, here are some applications that leverage WebRTC technology to deliver some awesome user functionality

**WhatsApp**

[borrows heavily from WebRTC](https://webrtchacks.com/whats-up-with-whatsapp-and-webrtc/) (such as the use of acoustic echo cancellation and active gain control from the WebRTC voice engine)

**Facebook**

Facebook's early rollout of WebRTC-based communications in 2015 caused a stir among the tech community as one of the largest bets to date on the emerging technology

**Google Hangouts**

Google Hangouts offers phone calls, SMS, video conferencing, and messaging capability all within the browser. There are other applications that are more popular than Google Hangouts, but Google's software is still a solid benchmark for demonstrating the scope and capabilities of WebRTC.

**Google Duo**

The app was [launched in 2016](https://www.theverge.com/2016/8/16/12474996/google-duo-review-video-chat-app-launch) as a WebRTC-based competitor to Apple's ubiquitous FaceTime, but has seen low adoption in the intervening years. Nonetheless, the app's WebRTC foundations allow for more reliable peer-to-peer connectivity and default-on end-to-end encryption,

**CONCEPTS LEARNED:**

WebRTC consists of several interrelated APIs and protocols which work together to achieve Real Time Communication. The most important APIs that we will use in this tutorial series are — click links to see demos

* getUserMedia(): capture audio and video.
* MediaRecorder: record audio and video.
* RTCPeerConnection: stream audio and video between users.
* RTCDataChannel: stream data between users.

## Signaling

Before the two peers can start communicating with each other, they need to know a lot of information about each other like —

* If there is any other peer available for communication.
* Network data, such as a peer’s IP address and port as seen by the outside world
* Session-control messages — used to open or close communication
* Error messages
* Media metadata, such as codecs, codec settings, bandwidth, and media types that will be sent by a peer
* Key data used to establish secure connections

Don’t worry if you do not understand what the above information represents. The important thing is to realize that a lot of information needs to be exchanged before a direct connection can be set-up. Such information can be termed as metadata.

Signaling refers to the mechanism which coordinates initial communication and enables sending of metadata between the peers (browsers). Hence, initially, the peers communicate with each other using the signaling mechanism — primarily, for discovering other peers and sharing the information needed to create a direct connection between them. Once the direct connection has been established, there is no role of signaling thereafter.

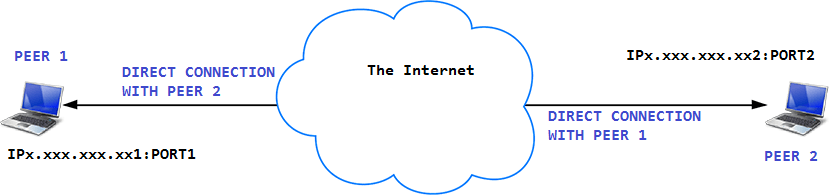
For signaling, we need a server.

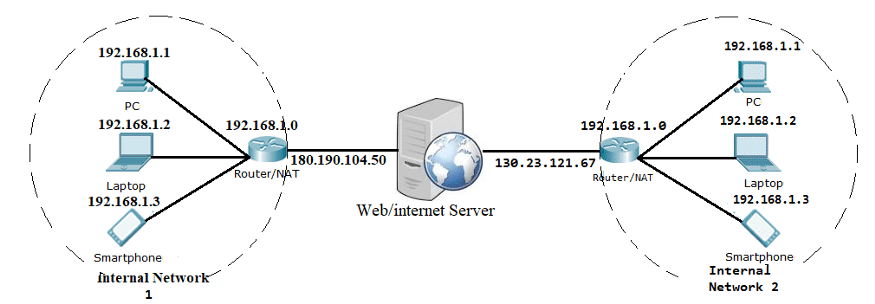
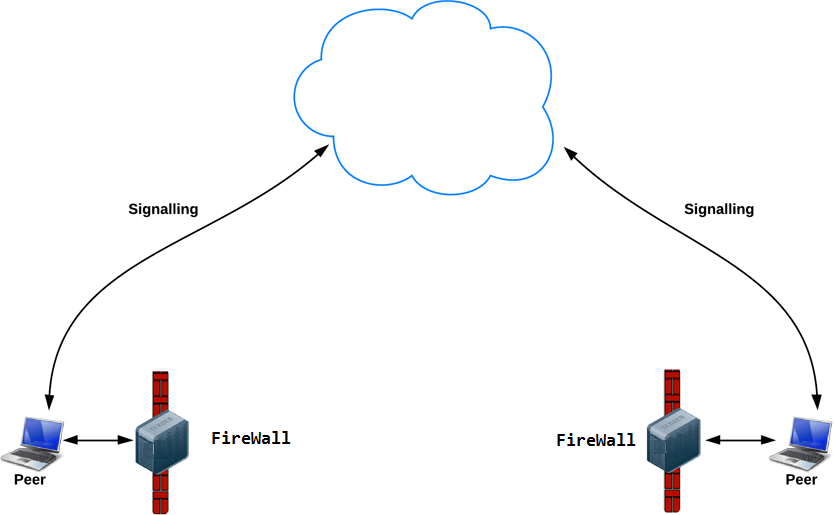
**Session Description Protocol:-**

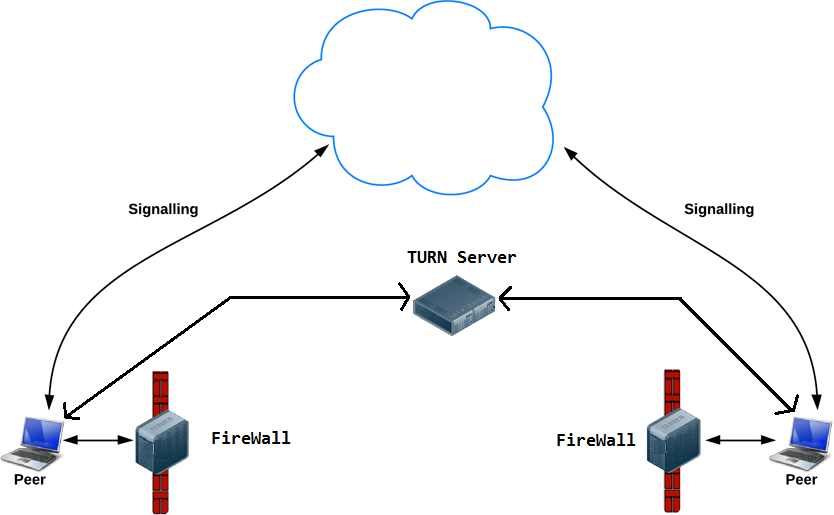
* The signaling mechanism (methods, protocols, etc.) is not specified by WebRTC. We need to build it ourselves. (Although this seems to be a complicated task, believe us — it is not. In this series, we will use Socket.IO for signaling, but there are many alternatives).
* WebRTC only requires the exchange of the media metadata mentioned above between peers as offers and answers. Offers and answers are communicated in Session Description Protocol (SDP) format which looks like the following:-  
   **v=0  
  o=- 7614219274584779017 2 IN IP4 127.0.0.1  
  s=-  
  t=0 0  
  a=group:BUNDLE audio video  
  a=msid-semantic: WMS  
  m=audio 1 RTP/SAVPF 111 103 104 0 8 107 106 105 13 126  
  c=IN IP4 0.0.0.0  
  ...**

## After Signalling — Use ICE to cope with NATs and firewalls

* So, it is natural that you would expect that every WebRTC Connection endpoint would have a unique IP address and PORT number that it could exchange with other peers in order to communicate directly.



* But it is not so simple. There are two factors which can cause problems here. We must deal with those before we can use our web conferencing application.
* Problem 1 — NAT
* If you are familiar with Computer Networks, you would know what NAT is. If don’t know, do not worry. We will explain it here:
* You already know what IP addresses are. It is an address that identifies a device connected on the internet. Logically, you would expect that each device (which is connected to the Internet) must have a UNIQUE IP Address. But this is not entirely true.
* An IPv4 address is 32 bits long which implies that there are about 4 billion unique address (2³² = 4,294,967,296). At the end of 2018, there were about 22 billion devices connected to the internet. So, you must be wondering — if there are only 4 billion IP addresses, how can 22 billion devices be connected on the internet ? The answer to this is NAT.
* The guys, who maintain the internet, came up with the following solution — They divided the whole IPv4 address range into two groups — public IP Addresses and private IP Addresses. Now, each public IP address can be assigned only to one device but the same is not true for private IP addresses. See the image below for more clarification.
* 
* In the above picture, each router has two IP addresses — one Public IP Address (facing the server) and one Private IP Address (facing the Internal Network). So, if any device inside Internal Network 1 sends a request to the server, the server will see the request coming from the same IP Address i.e. 180.190.104.50
* So this implies that each router maps one Public IP Address to multiple Private IP Addresses of the devices. This also implies that each device (laptop, PC, Smartphone) only knows its private IP Address and not the public IP Address of the router. ( Also, if you search on Google — my IP Address, Google will tell you the public IP Address of the router (you are connected to) because Google sees the Public IP Address of the router and not your Private IP Address.)
* Hence, in a way, we can say that each device has two IP addresses — a Private IP address (assigned to the device) and the Public IP address (assigned to the router to which the device is connected to).
* This can cause problem for WebRTC as the network ICE candidates (generated by the browser) contain the private IP address and not the public IP address of the device. Hence, we must find a way for the browser to know the Public IP Address so that it can create candidates containing the Public IP Address. The solution is a STUN (Session Traversal Utilities for NAT) server. When a device makes a request to the STUN server, the STUN server responds back with a message containing the public IP of the router to which the device is connected to. In this way, the STUN server helps the browser to generate candidates.
* We will see how to integrate STUN with WebRTC later in the tutorial.
* Problem 2 — Firewall
* In reality, most devices live behind one or more layers of firewalls which are like antivirus softwares that blocks certain ports and protocols. A firewall and a NAT may in fact be implemented by the same device, such as a home WIFI router. Since WebRTC uses a number of non-standard Ports, some Firewalls do not allow a direct connection to be made between the two browsers.
* 
* Hence, to solve this, we need a TURN (Traversal Using Relay NAT) server. TURN server basically acts a Relay Server i.e. the relay traffic directly between the two peers if direct (peer-to-peer) connection fails. Following image illustrates:-



**Solution**

* As we discussed before, we need to use STUN and TURN servers while making a peer-to-peer connection using WebRTC. To integrate TURN and STUN with webrtc, we only have to pass a object containing the URLs of TURN and STUN servers to the RTCPeerConnection()

**CONCLUSION:**

WebRTC is an interesting technology that allows us to go beyond the common computer communication protocols between a client and a server like request-response and WebSockets. Of course, none of them replace the others, they may be more suitable for a specific need or requirement but it is helpful to understand their advantages

WebRTC is not easy to start with. It has a lot of different aspects that might be hard to understand, but once you get a hang of it, it actually makes a lot of sense. The topic as a whole is very complex. Fortunately, we are given a good API. Other than that the browser and ICE servers do all of the dirty work for us which makes WebRTC development not bad after all.so

we can consider different alternatives whenever we start a new project.

**REFERENCES:**

We have referred to the documentation of all the tools/ frameworks/ libraries we have used in this project. The references are as below:

1. <https://webrtc.org/>
2. <https://socket.io/>
3. <https://nodejs.org/en/>
4. <https://expressjs.com/>
5. <https://material-ui.com/>
6. <https://trembit.com/blog/how-many-participants-can-we-place-in-one-webrtc-peer-2-peer-room/>
7. <https://bloggeek.me/how-many-users-webrtc-call/>
8. <https://stackoverflow.com/questions/16015304/webrtc-peer-connections-limit#:~:text=Maximum%20peer%20connections%20limit%20is%20256%20(on%20chrome)>.
9. [MDN Web Docs](https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API)