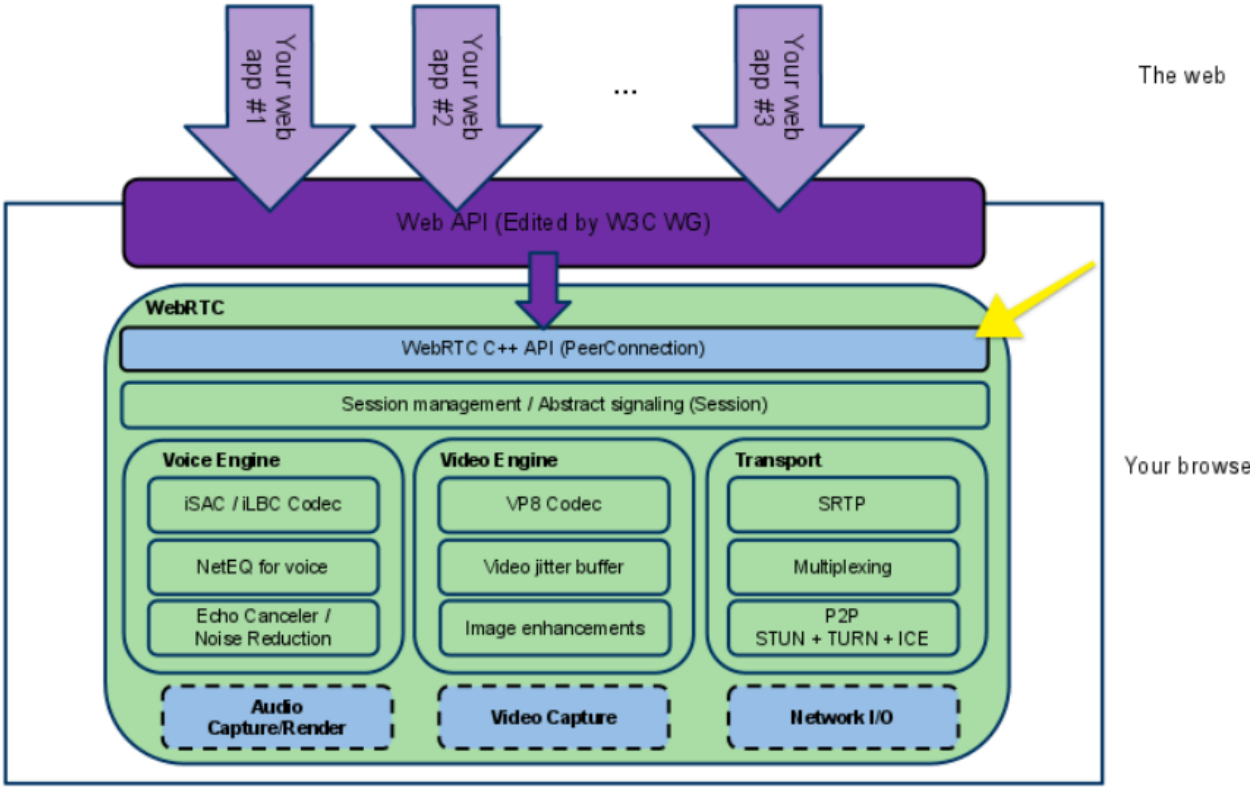
Introduction

Part I: WebRTC Definition

What is WebRTC? Web Real-Time Communication is a technology that enables Web applications and sites to capture and optionally stream audio and/or video media, as well as to exchange arbitrary data between browsers without requiring an intermediary. The set of standards that comprise WebRTC makes it possible to share data and perform teleconferencing peer-to-peer, without requiring that the user install plug-ins or any other third-party software.

WebRTC consists of several interrelated APIs and protocols which work together to achieve this. The documentation you'll find here will help you understand the fundamentals of WebRTC, how to set up and use both data and media connections, and more.



Part II: What is WebRTC used for

The goal of WebRTC is to facilitate real-time P2P communications over the internet. There are several use cases for WebRTC, including the following:

- ☐ WebRTC is used for video chats and meetings on video calling platforms, such as Zoom, Microsoft Teams, Slack or Google Meet.
- ☐ Industries, including healthcare, surveillance and monitoring, and internet of things, use WebRTC. For example, WebRTC use in Telehealth enables doctors to conduct virtual office visits with a patient over a web browser.
- ☐ In the field of home and business security and surveillance, WebRTC is used as a connecting agent between browsers and security cameras.
- ☐ WebRTC is heavily used for real-time media.
- ☐ WebRTC provides the underlying connection between instructors and students for online education.

Part III: How does WebRTC work under the hood

In order to understand how WebRTC works under the hood, let's get familiar with some terminologies:

SDP

The configuration of an endpoint on a WebRTC connection is called a **session description**. The description includes information about the kind of media being sent, its format, the transfer protocol being used, the endpoint's IP address and port, and other information needed to describe a media transfer endpoint. This information is exchanged and stored using **Session Description Protocol** (SDP)

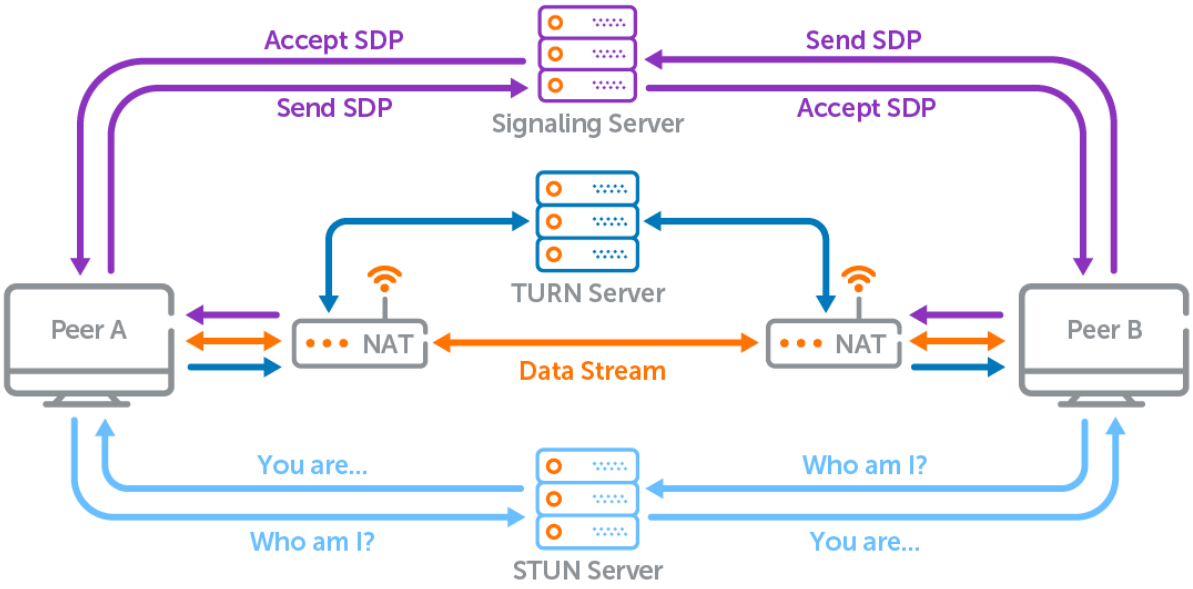
When a user starts a WebRTC call to another user, a special description is created called an offer. This description includes all the information about the caller's proposed configuration for the call. The recipient then responds with an answer, which is a description of their end of the call. In this way, both devices share with one another the information needed in order to exchange media data. This exchange is handled using Interactive Connectivity Establishment (ICE), a protocol which lets two devices use an intermediary to exchange offers and answers even if the two devices are separated by Network Address Translation (NAT).

Signaling:

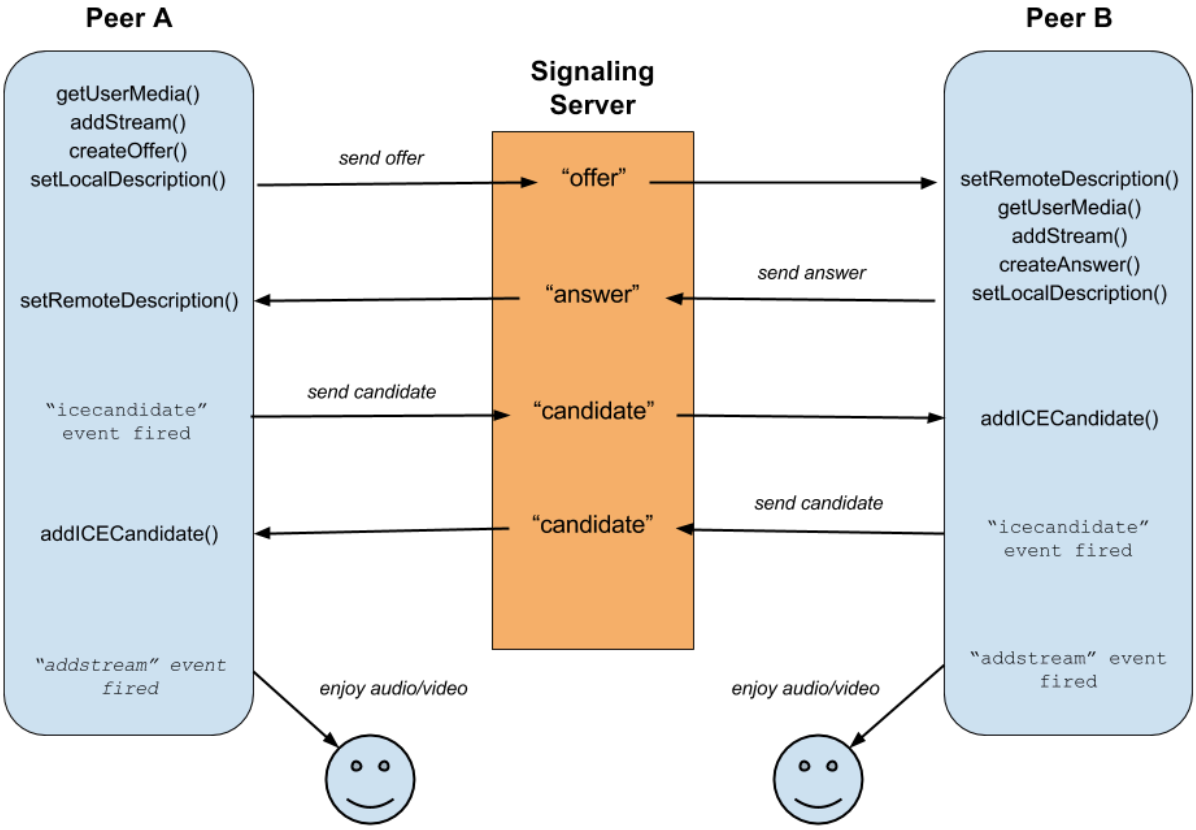
Unfortunately, WebRTC can't create connections without some sort of server in the middle. We call this the **signal channel** or **signaling service**. It's any sort of channel of communication to exchange information before setting up a connection, whether by email, postcard, or a carrier pigeon. It's up to you. The information we need to exchange is the Offer and Answer which just contains the SDP.

Peer A who will be the initiator of the connection, will create an Offer. They will then send this offer to Peer B using the chosen signal channel. Peer B will receive the Offer from the signal channel and create an Answer. They will then send this back to Peer A along the signal channel.

So with knowing the concept of signaling server and SDP, lets focus on WebRTC connectivity:

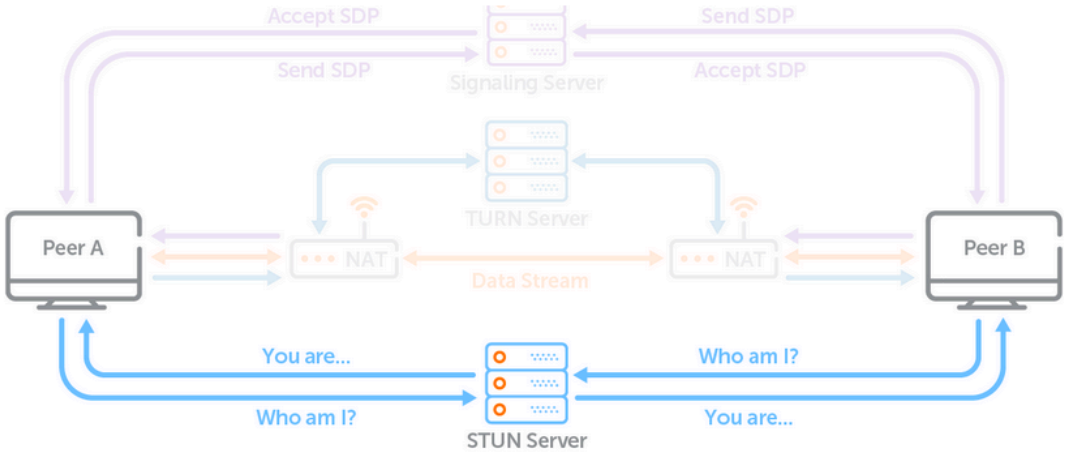


A. Signaling and Connectivity Phase: Signaling in WebRTC is the process by which client devices establish a connection. Basically, these devices need to agree to talk to one another before they can send and receive data. And to come to an agreement, they need to know how to “find” each other. A device sends a session description protocol (SDP) containing certain identifying information (otherwise known as internet connectivity establishment or ICE candidates), such as port and IP information, to a signaling server. This server sends the SDP along to the other device. It also relays SDP acceptance signals between the peers.

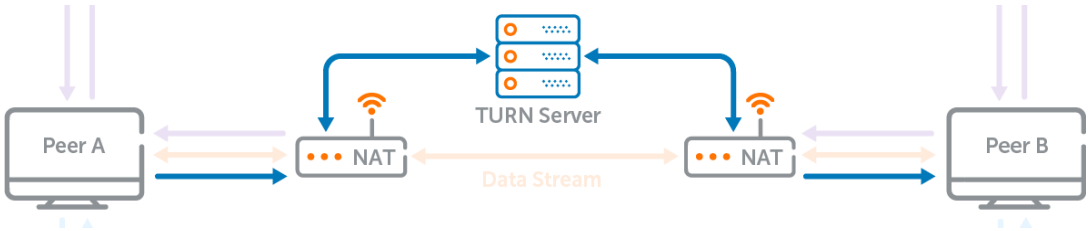


Note: What Is a WebRTC NAT Traversal Server?
It sounds like it should be simple — connecting two or more peers remotely. However, the process is more complicated than it first seems thanks to Network Address Translation (NAT) devices. These devices block client devices from locating their own internet protocol (IP) addresses. Before sending an SDP request, a computer must know its IP address. That's where NAT traversal comes in.

WebRTC STUN Server
The first method of NAT traversal is known as Session Traversal Utilities for NAT (STUN). Put simply, a client device pings a STUN server, asking for a connection. This server is located on the public internet and requires an IP address for any device that tries to communicate with it. Therefore, when a device pings it, it responds with that device's IP address. The information received from the STUN server can be used in the SDP sent over the signaling server.



WebRTC TURN Server
If your NAT device is particularly strict, then STUN may not work for you. That's where Traversal Using Relays around NAT (TURN) comes in. In this case, you forgo ICE candidates and SDP protocol connections and just go around the NAT firewall. TURN servers have public IP addresses, making them easy to connect to. When two clients connect, they can send media to one another using the TURN server as an intermediary.



B. Data Stream Exchanging: The data stream exchange phase in WebRTC is where actual peer-to-peer (P2P) communication happens. Once connectivity is established, encrypted media streams (audio/video) are transmitted via SRTP, while arbitrary data (e.g., files or messages) is exchanged over data channels using SCTP over DTLS. WebRTC ensures low latency, dynamic network adaptation (e.g., adjusting bitrate or resolution), and secure communication, enabling smooth and efficient real-time interactions.

Below links are useful for get familiar with the flow of WebRTC connectivity and data exchange:
https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API
https://webrtc.org/getting-started/overview
https://www.wowza.com/blog/webrtc-server-what-it-is-and-why-you-need-one
Also the below series is useful for both understanding the webRTC and its implementation:
https://youtube.com/playlist?list=PLF_SZiztoCWHJAohyZKl8QnXYUHLeh11t&si=92o5NWHUnhsAtReG

WebRTC in practice
Part I: project Implementation details

Now that we are familiar with the WebRTC itself, let's focus on the implementation of this project, first let's see the tree of our files and the role of each:

app directory: which contains the structure (html file) alongside with the style (css file) and functionality (js file) of our client side, let's go through each file and get deeper insight about each:

index.html: It includes a chat box where users can send messages via an input field and a send button, and a video call section with a grid for video streams and controls for muting audio and toggling the camera. External resources include Google Fonts for styling, a CSS file for custom styles, and the Socket.IO library for real-time communication. A JavaScript file (`script.js`) is linked to handle the application's functionality, loaded asynchronously with `defer`.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Minichat</title>
   <link href="https://fonts.googleapis.com/css2?family=Poppins:wght@400;600&display=swap" rel="stylesheet">
   <link rel="stylesheet" href="styles.css">
   <script src="https://cdn.socket.io/4.5.4/socket.io.min.js"></script>
</head>
<body>
   <h1>Minichat</h1>
   <div id="communicationSection">
       <div id="chatBox">
           <div id="messages"></div>
           <div id="chatBoxText">
               <input type="text" id="chatInput" placeholder="Hey Chat!" />
               <button id="sendMessage">Send</button>
           </div>
       </div>
       <div id="callBox">
           <div id="videoGrid"></div>
           <div id="controls">
               <button id="muteVoice">🔊</button>
               <button id="CameraEnable">📹</button>
           </div>
       </div>
   </div>
   <script src="script.js" defer></script>
</body>
</html>
```

style.css: The styles.css file defines the layout and appearance of the chat and video call interface. It styles elements like the chat box, video grid, and controls with proper alignment, spacing, and responsive behavior. Buttons and input fields are customized for usability, while sections like the message display area may include scrollable features for better user interaction.

script.js: This script handles the core functionality of a real-time video chat and messaging application using WebRTC and Socket.IO. It manages user connections, media streams, and peer-to-peer communication, including signaling for WebRTC connections and exchange of ICE candidates. The script also enables toggling audio/video, tracks voice levels to indicate speaking users, and facilitates a chat interface with color-coded messages for different users. Let's breakdown this file and see what's inside it:

getLocalStream Function:

```
async function getLocalStream() {
   try {
       localStream = await navigator.mediaDevices.getUserMedia({ video: true, audio: true });
       localVideo.srcObject = localStream;
       localVideo.play();

       audioAnalyzers[socket.id] = voiceLevelAdapter(localStream, localVideo);

       socket.emit('join-call');
   } catch (error) {
       console.error('error media devices:', error);
   }
}
```

The getLocalStream function is responsible for capturing the user's audio and video streams by accessing the device's camera and microphone. It uses the navigator.mediaDevices.getUserMedia API to obtain these streams, which are then linked to the localVideo element for a live preview. By setting the srcObject of this element to the captured media stream, the function ensures the user's video is displayed on their screen.

Additionally, the function integrates the voiceLevelAdapter to monitor the local audio stream, providing feedback on audio levels and enabling visual cues when the user is speaking. Finally, it notifies the server of the user's participation in the call by emitting the join-call event through the socket connection, allowing the signaling process to begin.

createPeerConnection Function:

```
function createPeerConnection(peerSocketId) {
   const peerConnection = new RTCPeerConnection({
       iceServers: [
           { urls: 'stun:stun.l.google.com:19302' },
     {
       urls: "turn:5.34.195.146:3478",
       username: "my-turn-server",
       credential: "this-is-credential",
     }
       ],
   });

   localStream.getTracks().forEach(track => {
       peerConnection.addTrack(track, localStream);
   });

   peerConnection.onicecandidate = event => {
       if (event.candidate) {
           socket.emit('ice-candidate', event.candidate, peerSocketId);
       }
   };

   const remoteStream = new MediaStream();
   peerConnection.ontrack = event => {
       remoteStream.addTrack(event.track);

       if (!peers[peerSocketId]?.remoteVideo) {
           const remoteVideo = document.createElement('video');
           remoteVideo.srcObject = remoteStream;
           remoteVideo.autoplay = true;
           remoteVideo.classList.add('video');
           videoGrid.appendChild(remoteVideo);
           peers[peerSocketId].remoteVideo = remoteVideo;

           audioAnalyzers[peerSocketId] = voiceLevelAdapter(remoteStream, remoteVideo);
       }
   };

   return peerConnection;
}
```

The createPeerConnection function establishes a WebRTC peer connection with another participant in the call, identified by their peerSocketId. It initializes an RTCPeerConnection object, configured with ICE servers, including a public STUN server and a TURN server for NAT traversal and relaying media when direct connections are not possible. Once created, the function adds the tracks from the local media stream (localStream) to the connection, enabling the local audio and video to be transmitted to the remote peer. It also sets up an onicecandidate handler to capture ICE candidates generated by the connection and emits them to the server using the ice-candidate event, ensuring connectivity between peers. The function creates a new MediaStream to represent the incoming remote media tracks. As remote tracks are added via the ontrack event, they are appended to the remote stream. If no video element exists for the remote peer, it dynamically creates one, sets its source to the remote stream, and appends it to the video grid for display. Additionally, it invokes the voiceLevelAdapter to monitor the remote audio, enabling visual feedback for their voice levels. Finally, the peer connection object is returned, providing the caller with full control over the WebRTC connection.

voiceLevelAdapter Function:

```
function voiceLevelAdapter(stream, videoElement) {
   const audioContext = new (window.AudioContext || window.webkitAudioContext)();
   const analyser = audioContext.createAnalyser();
   const source = audioContext.createMediaStreamSource(stream);

   source.connect(analyser);
   analyser.fftSize = 256;

   const dataArray = new Uint8Array(analyser.frequencyBinCount);

   function checkAudioLevel() {
       analyser.getByteFrequencyData(dataArray);
       const volume = dataArray.reduce((acc, val) => acc + val, 0) / dataArray.length;

       if (volume > 15) {
           videoElement.classList.add('speaking');
       } else {
           videoElement.classList.remove('speaking');
       }

       requestAnimationFrame(checkAudioLevel);
   }

   checkAudioLevel();
   return audioContext;
}
```

The voiceLevelAdapter function is responsible for monitoring the audio volume levels of a given media stream and visually indicating whether the user is speaking, based on the audio levels.It starts by creating an AudioContext, which is a part of the Web Audio API, enabling the processing of audio data. An AnalyserNode is then created within the context, which is used to extract frequency and volume data from the stream. The audio stream is connected to this analyser, and the analyser's FFT (Fast Fourier Transform) size is set to 256, which determines the resolution of the frequency data. An array (dataArray) is created to hold the frequency data that the analyser will generate. The checkAudioLevel function is defined to continuously monitor the audio levels in real-time. It uses the getByteFrequencyData method of the analyser to populate dataArray with frequency data, representing the audio levels across different frequencies. The volume level is calculated by averaging the values in the dataArray. If the volume exceeds a threshold (in this case, a value of 15), the function adds a speaking class to the video element, visually indicating that the user is speaking. If the volume falls below this threshold, the speaking class is removed. The requestAnimationFrame method ensures that the checkAudioLevel function runs continuously, providing smooth updates to the audio level monitoring. Finally, the function returns the audioContext, allowing the caller to manage the audio processing context if needed.

```
socket.on('user-connected', peerSocketId => {
   const peerConnection = createPeerConnection(peerSocketId);
   peers[peerSocketId] = { peerConnection };

   peerConnection.createOffer()
       .then(offer => {
           peerConnection.setLocalDescription(offer);
           socket.emit('offer', offer, peerSocketId);
       });
});
```

When a new user joins, a peer connection is created using createPeerConnection. Then, an offer is generated and sent to the new peer via the socket.

```
socket.on('offer', async (offer, peerSocketId) => {
   const peerConnection = createPeerConnection(peerSocketId);
   peers[peerSocketId] = { peerConnection };

   await peerConnection.setRemoteDescription(new RTCSessionDescription(offer));
   const answer = await peerConnection.createAnswer();
   await peerConnection.setLocalDescription(answer);

   socket.emit('answer', answer, peerSocketId);
});
```

When receiving an offer from another user, a peer connection is created. The offer is set as the remote description, and an answer is generated and sent back to the original peer.

```
socket.on('answer', async (answer, peerSocketId) => {
   await peers[peerSocketId].peerConnection.setRemoteDescription(new
RTCSessionDescription(answer));
});
```

This event handles the answer from another peer after an offer is made. The answer is set as the remote description for the peer connection.

```
socket.on('ice-candidate', async (candidate, peerSocketId) => {
   const peerConnection = peers[peerSocketId]?.peerConnection;
   if (peerConnection) {
       await peerConnection.addIceCandidate(new RTCIceCandidate(candidate));
   }
});
```

When an ICE candidate (network information for connection) is received, it is added to the peer connection to help establish the connection.

```
socket.on('user-disconnected', peerSocketId => {
   if (peers[peerSocketId]?.remoteVideo) {
       videoGrid.removeChild(peers[peerSocketId].remoteVideo);
   }
   if (peers[peerSocketId]?.peerConnection) {
       peers[peerSocketId].peerConnection.close();
   }
   if (audioAnalyzers[peerSocketId]) {
       audioAnalyzers[peerSocketId].close();
       delete audioAnalyzers[peerSocketId];
   }
   delete peers[peerSocketId];
});
```

This event cleans up when a user disconnects by removing their remote video element, closing their peer connection, and cleaning up any audio analyzer.

```
socket.on('mute-state', ({ peerId, audio, video }) => {
   const peerVideo = peers[peerId]?.remoteVideo;
   if (peerVideo) {
       if (audio !== undefined) {
           peerVideo.dataset.audioMuted = audio ? 'true' : 'false';
       }
       if (video !== undefined) {
           peerVideo.style.opacity = video ? '0.5' : '1';
       }
   }
});
```

When a user's mute state changes for audio or video, this event updates the relevant properties. If audio is muted, the video element's dataset is updated, and if video is muted, its opacity is adjusted.

```
function sendMessage() {
   const message = chatInput.value.trim();
   if (message) {
       addMessageToChat(`Me: ${message}`, 'self');
       socket.emit('chat-message', message);
       chatInput.value = '';
   }
}

function addMessageToChat(message, sender = 'other') {
   const messageElement = document.createElement('div');
   messageElement.classList.add('message');
   const userColor = getUserColor(sender);
   messageElement.style.backgroundColor = '#1c1c1c';

   if (sender === 'self') {
       messageElement.style.backgroundColor = '#6055ffc0';
       messageElement.style.color = '#fff';
       messageElement.style.alignSelf = 'flex-end';
   }

   messageElement.textContent = message;
   messagesContainer.appendChild(messageElement);
   messagesContainer.scrollTop = messagesContainer.scrollHeight;
}
```

These two functions manage sending and displaying chat messages in a real-time communication system: The sendMessage function captures the user's input from chatInput, ensures it's not empty, and adds the message to the chat display using addMessageToChat. It also sends the message to the server via the chat-message socket event, then clears the input field. The addMessageToChat function creates and styles a message element based on the sender. For messages sent by the user (self), it uses a distinct background color, text color, and alignment. The message is appended to messagesContainer, and the container scrolls to show the latest message.

Now let’s get into the signaling server code.

server.js:

```
import express from 'express'
import { createServer } from 'http'
import { Server } from 'socket.io'

const app = express()
const httpServer = createServer(app)
const io = new Server(httpServer, {
   cors: {
       origin: '*',
   },
})

const users = new Map()

io.on('connection', (socket) => {
   console.log(`A new user connected with socket ID: ${socket.id}`);


   socket.on('ice-candidate', (iceCandidate, targetSocketId) => {
       console.log(
           `Ice candidate received on server to be able to send to ${targetSocketId} `,
           iceCandidate
       );

       io.to(targetSocketId).emit('ice-candidate', iceCandidate, socket.id);
   });

   socket.on('offer', (offer, targetSocketId) => {
       console.log(
           `Offer received on server from ${socket.id} to be able to send to ${targetSocketId}`,
           offer
       );

       io.to(targetSocketId).emit('offer', offer, socket.id);
   });

   socket.on('answer', (answer, targetSocketId) => {
       console.log(
           `Answer received on server to be able to send to ${targetSocketId}`,
           answer
       );

       io.to(targetSocketId).emit('answer', answer, socket.id);
```

```
    });

    socket.on('chat-message', (message) => {
        console.log(`Chat message from ${socket.id}: ${message}`);
        socket.broadcast.emit('chat-message', message, socket.id);
    });

    socket.on('join-call', () => {
        console.log(`${socket.id} joined the call`);
        socket.broadcast.emit('user-connected', socket.id);
    });

    socket.on('disconnect', () => {
        console.log(`${socket.id} disconnected`);
        socket.broadcast.emit('user-disconnected', socket.id);
    });
});


const port = process.env.PORT || 8000

httpServer.listen(port, () => {
    console.log(`Server started on port ${port}`)
})
```

This code creates a signaling server with Express and Socket.IO for WebRTC communication and chat. It handles WebRTC events like ice-candidate, offer, and answer, relaying them between peers to establish peer-to-peer connections. Users joining a call trigger the join-call event, notifying others, while disconnections emit user-disconnected. Chat messages are broadcast to all users except the sender. The server listens on port 8000 (or an environment-specified port) and allows cross-origin requests.

Part II: Project Deployment

I use Arvancloud object storage in order to server my static index.html, style.css and script.js behind the https server and you can check it by following the links below:
https://minichat.s3-website.ir-thr-at1.arvanstorage.ir/

which serves my files, but how does it interact with my signaling server and where is it ?

I deployed my signaling server on cloudflare server (where I don't need any domain to buy and also utilize the https), the address of my signaling server is:
https://signaling.faranegareh.com

and in the client code the address of the socket is linked to this address.

```
const socket = io('https://signaling.faranegareh.com');
```

Scenario 1) Hosts in same network (in the ice servers configuration of my client code I removed the turns server part and use the public free google stun servers):

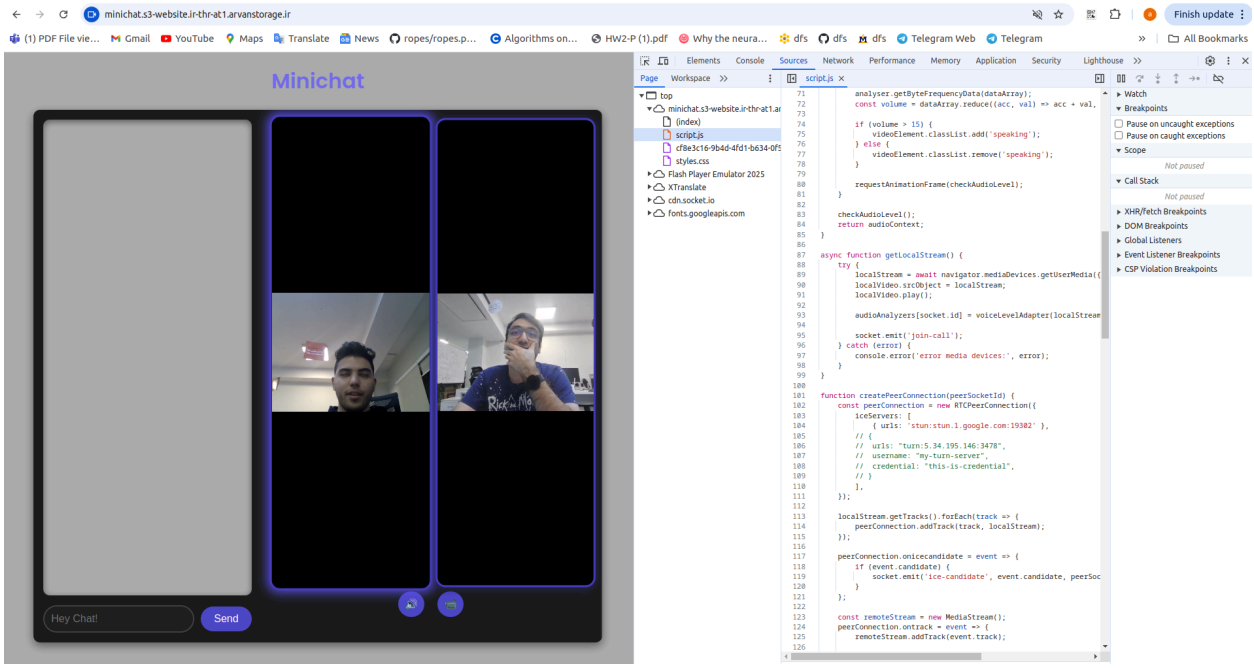
host A private ip: 10.100.10.48

```
arya@arya-ThinkPad-T14-Gen-2i ~/Desktop/uni/99521379_Arya_Shahsavar_Final_Project/app ip a | grep inet | grep 10.100
    inet 10.100.10.48/24 brd 10.100.10.255 scope global dynamic noprefixroute wlp0s20f3
```

host B private ip and the tcpdump that shows that its a p2p communication between these hosts:

```
naeem@naeem-laptop:~$ sudo tcpdump -pnni any host 10.100.10.48 -c 10
tcpdump: data link type LINUX_SLL2
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
22:41:43.186468 wlp46s0 Out IP 10.100.90.26.37082 > 10.100.10.48.35496: UDP, length 113
22:41:43.192973 wlp46s0 Out IP 10.100.90.26.37082 > 10.100.10.48.35496: UDP, length 42
22:41:43.193194 wlp46s0 In  IP 10.100.10.48.35496 > 10.100.90.26.41210: UDP, length 211
22:41:43.199803 wlp46s0 In  IP 10.100.10.48.35496 > 10.100.90.26.41210: UDP, length 1117
22:41:43.199827 wlp46s0 In  IP 10.100.10.48.35496 > 10.100.90.26.41210: UDP, length 1117
22:41:43.199831 wlp46s0 In  IP 10.100.10.48.35496 > 10.100.90.26.41210: UDP, length 1118
22:41:43.199837 wlp46s0 In  IP 10.100.10.48.35496 > 10.100.90.26.41210: UDP, length 1118
22:41:43.199840 wlp46s0 In  IP 10.100.10.48.35496 > 10.100.90.26.41210: UDP, length 1118
22:41:43.199842 wlp46s0 In  IP 10.100.10.48.35496 > 10.100.90.26.41210: UDP, length 1118
22:41:43.206405 wlp46s0 Out IP 10.100.90.26.37082 > 10.100.10.48.35496: UDP, length 1074
10 packets captured
40 packets received by filter
0 packets dropped by kernel
naeem@naeem-laptop:~$ ip a s dev wlp46s0
3: wlp46s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether b8:1e:a4:c6:fd:59 brd ff:ff:ff:ff:ff:ff
    inet 10.100.90.26/24 brd 10.100.90.255 scope global dynamic noprefixroute wlp46s0
       valid_lft 602766sec preferred_lft 602766sec
    inet6 fe80::d3f8:2df1:6190:7218/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
naeem@naeem-laptop:~$
```

and here you can see both hosts interacting with each other (and the stun server is only used):



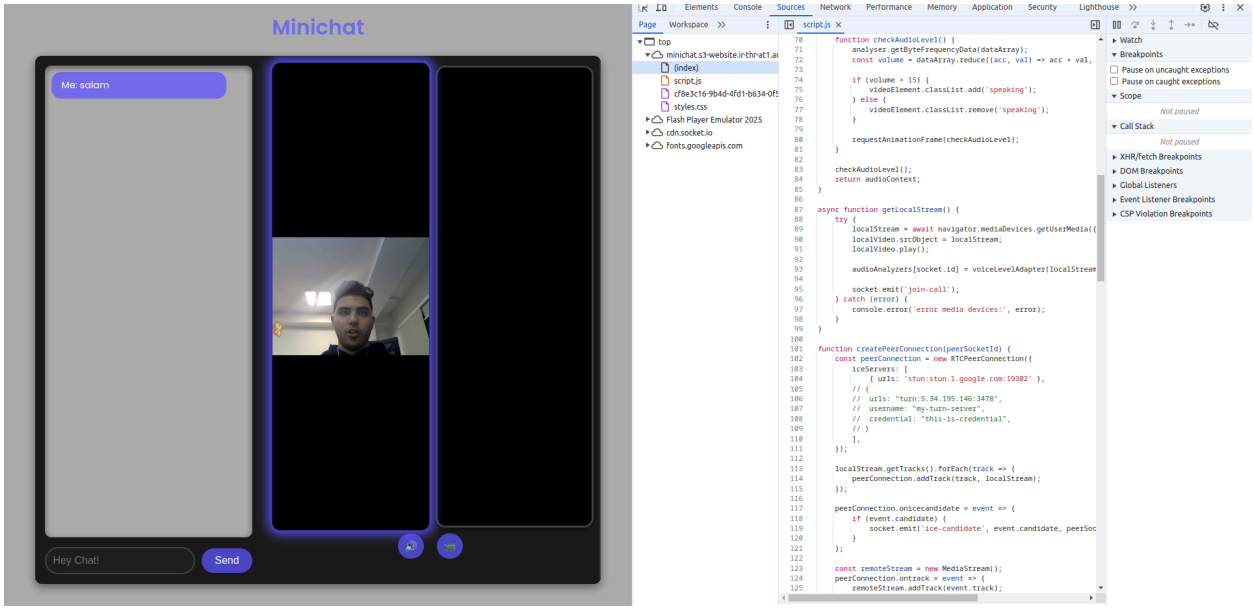
Scenario 2) Hosts in different networks:
Since the public free turns server are not stable and does't work correctly, I manually configure a turn server in a host in arvancloud server. Here is the turn server configuration:

```
listening-port=3100
fingerprint
use-auth-secret
static-auth-secret=596a61dc4165c2a71ba497f4570a8929
realm=5.34.195.146
no-stdout-log
user=my-turn-server:this-is-credential
```

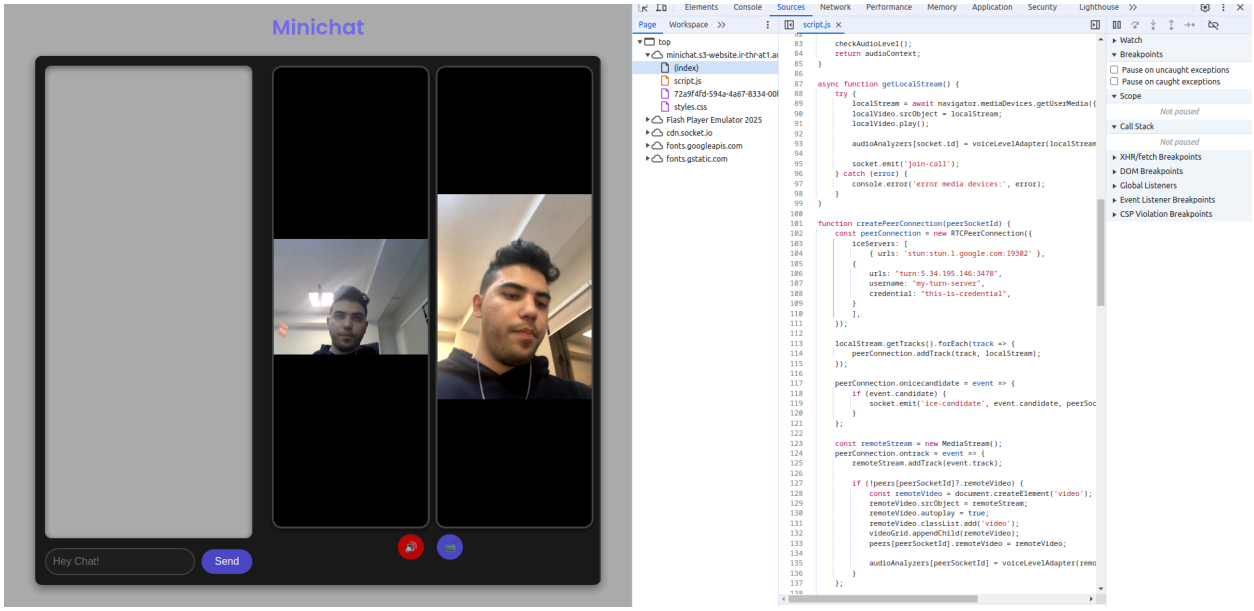
and here is the command to start the turn server:

```
turnser+    9808  0.0  0.1 635340 19916 ?        Ssl  Jan24   0:51 /usr/bin/turnserver -c /etc/turnserver.conf --pidfile=
```

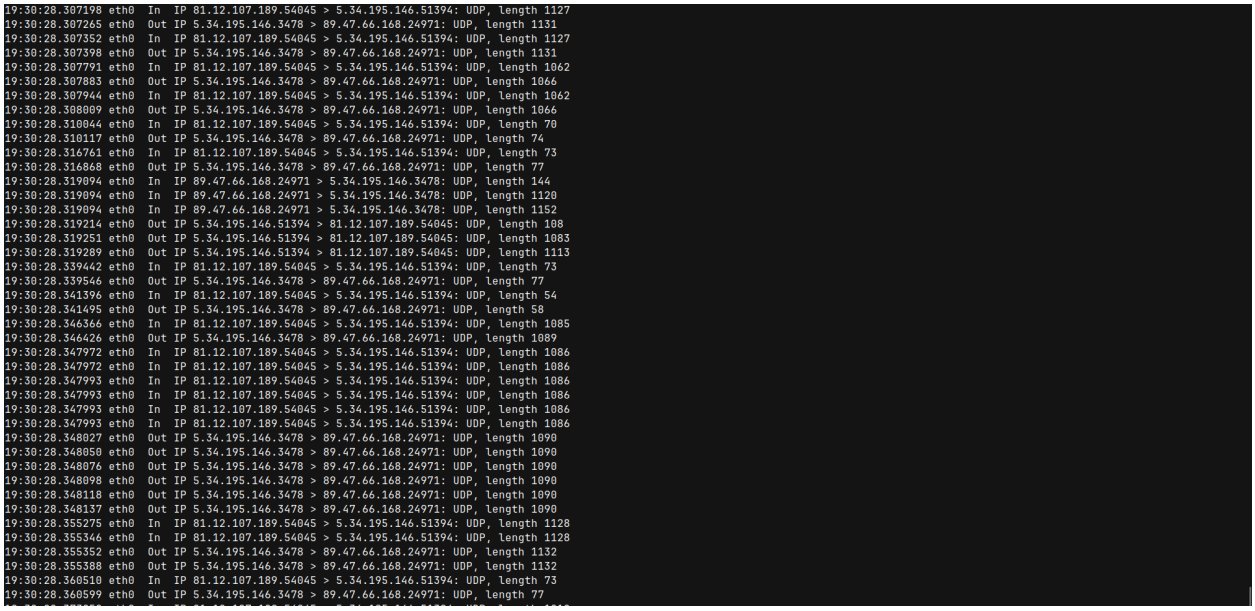
First, let's check that with the current configuration and using only stun server, we doesn't have each other's stream:



and Now, let’s modify the ice server configurations and add my turn server to it:



and here is the tcpdump on my turn server in arvancloud server:



that shows it relays the traffic between these two hosts.