WebRTC

Amir Fakharzadeh
99521487

Network Engineering Course Project
Fall 2024

Table of Contents

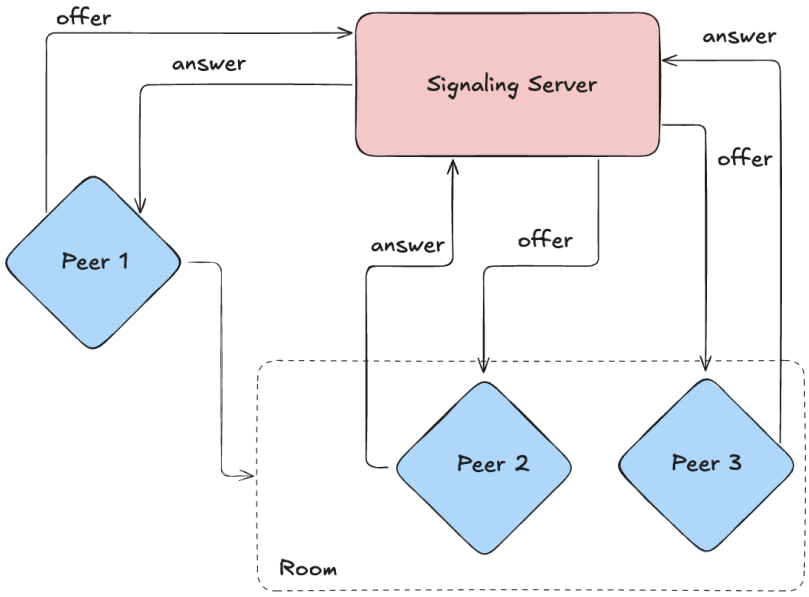
Introduction

The purpose of this project is to develop a real-time communication application using WebRTC technology, enabling users to engage in seamless video calls and text messaging. This platform will allow multiple users to participate in video conversations while exchanging text messages simultaneously, similar to services like Google Meet.

Project Overview

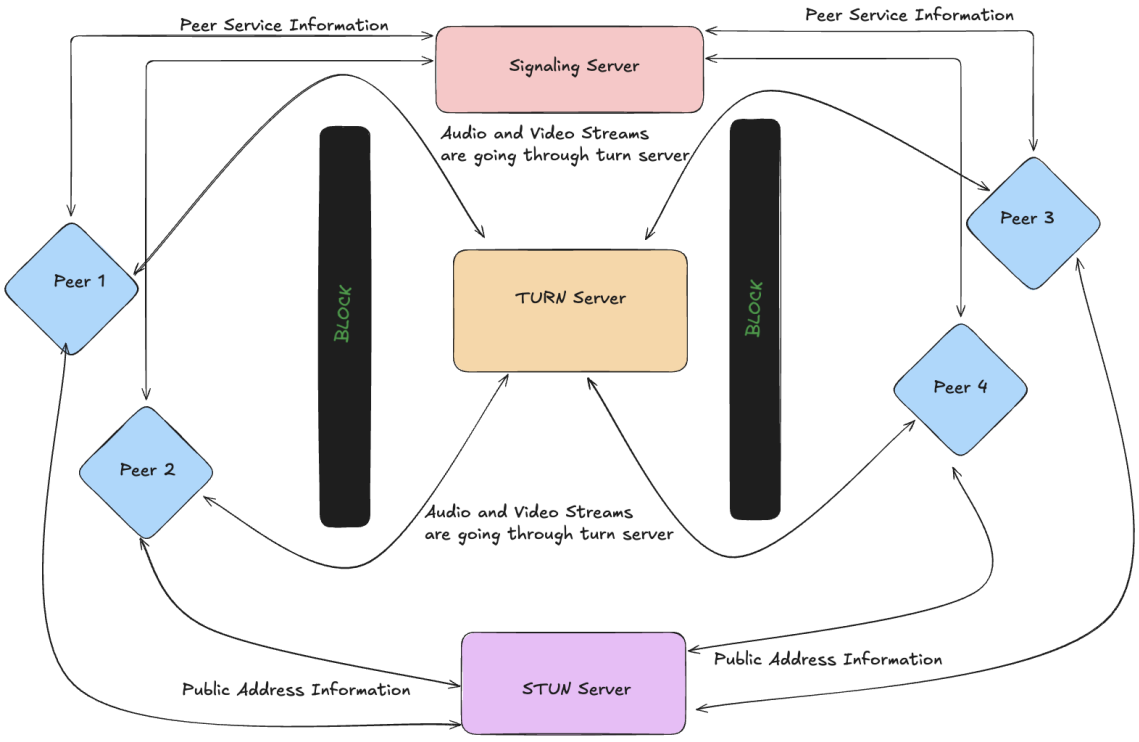
The application will support three users—H1, H2, and H3—allowing them to connect and interact simultaneously in a dynamic environment. It will be tested in two scenarios: first, when users are on the same internal network without knowledge of each other's IP addresses, and second, when they are on separate networks behind NAT.

Architecture



This diagram illustrates the signaling process for establishing WebRTC connections in a chat or video call application. The process involves a **signaling server** that facilitates the exchange of connection setup messages (offer and answer) between peers in a room.

In the following picture we're going to discuss a little bit about the whole architecture of WebRTC connection including signaling server, STUN server, and TURN Server.



This diagram illustrates the architecture of a WebRTC connection, showcasing the roles of the **Signaling Server**, **STUN Server**, and **TURN Server** in establishing peer-to-peer communication. Initially, peers exchange **service information** (e.g., session descriptions and ICE candidates) through the signaling server, which helps them discover each other and set up the connection. For NAT traversal, the **STUN server** provides the public-facing IP addresses of the peers to attempt direct communication.

If direct communication fails due to firewalls or restrictive NAT configurations, media streams (audio and video) are relayed through the **TURN server**, which acts as an intermediary. The TURN server ensures that communication remains functional even

when direct peer-to-peer connections are blocked, providing a reliable fallback mechanism for seamless real-time communication.

Concepts

- **Signaling Server:** Facilitates the exchange of connection setup information (SDP, ICE candidates) between peers.
- **SDP (Session Description Protocol):** Describes the media session.Peers send **offer** and **answer** SDP during signaling.
- **STUN:** Helps peers discover their public IP and port when behind NAT.
- **TURN:** Relays media traffic when direct P2P communication fails due to restrictive NAT or firewalls.
- **ICE (Interactive Connectivity Establishment):** Finds the optimal connection path between peers using gathered ICE candidates. Performs connectivity checks and selects the best candidate pair.
- **ICE Candidates:** Represents possible network paths for communication between peers.
- **RTCPeerConnection:** A JavaScript API in WebRTC that enables peer-to-peer connections for real-time audio, video, and data communication, simplifying network negotiation and traversal.
- **getUserMedia:** API allows web applications to access a user's camera, microphone, or both, enabling the capture of audio and video streams for real-time communication or recording.

Technical Requirements

- **Node.js Server**:
    - Acts as the signaling server to exchange session descriptions (SDP) and ICE candidates between peers.
    - Manages WebSocket connections for real-time communication.

- **HTML/CSS/JavaScript Client**:
    - Frontend application for users to initiate and manage WebRTC connections.
    - Handles media streams, user interface, and peer connection setup via **RTCPeerConnection**.
    - Communicates with the Node.js signaling server using WebSockets.

- **Coturn for Configuring TURN Server**:
    - Provides a TURN (Traversal Using Relays around NAT) server for relaying media when direct peer-to-peer connections are not possible.
    - Configured to work alongside STUN servers for NAT and firewall traversal.

- **Google STUN Server**:
    - Utilized as a free STUN (Session Traversal Utilities for NAT) server to discover public-facing IP addresses of peers.
    - Ensures peers can attempt direct connections by resolving NAT issues.

- **A VPS for Deploying Both Client and Server**:
    - A Virtual Private Server (VPS) hosts the Node.js signaling server and the HTML/CSS/JavaScript client.
    - Provides a reliable, always-online environment for WebRTC applications.
    - Configured with Coturn and accessible for clients worldwide.

Implementation Details

- **Server Side**
    - **Server Setup:** This part initializes the Node.js server using **express**, **http**, and **socket.io**. The server listens on port 8000 and allows cross-origin requests, enabling communication with clients.

```
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 port = 8000;
httpServer.listen(port, () => {
    console.log(`server started on port ${port}`);
});
```

- **Handling User Connections:** This section listens for incoming WebSocket connections. It logs when a user connects or disconnects and notifies other peers about disconnected users.

```
const users = new Map();

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

- **Chat Messaging:** Handles real-time chat functionality by broadcasting chat messages from one client to all other connected clients.

```
socket.on('chat-message', (message) => {
    socket.broadcast.emit('chat-message', message, socket.id);
});
```

- **WebRTC Signaling:** Implements WebRTC signaling by exchanging SDP offers, answers, and ICE candidates between peers via the signaling server. It also broadcasts when a new user joins a call.

```
socket.on('offer', (offer, targetSocketId) => {
    io.to(targetSocketId).emit('offer', offer, socket.id);
});

socket.on('answer', (answer, targetSocketId) => {
    io.to(targetSocketId).emit('answer', answer, socket.id);
});

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

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

- **Client Side**
    - **Connecting to the Server:** Establishes a WebSocket connection between the client and the signaling server for real-time communication.

```
const socket = io('http://188.213.199.201:8000');
```

    - **User Media and Local Stream Setup:** Uses the **getUserMedia** API to access the user's camera and microphone and streams the captured media to the local video element. It also notifies the server that the user has joined the call.

```
async function getLocalStream() {
    try {
        localStream = await navigator.mediaDevices.getUserMedia({ video: true,
audio: true });
        localVideo.srcObject = localStream;
        localVideo.play();
        socket.emit('join-call');
    } catch (error) {
        console.error('error media devices:', error);
    }
}
getLocalStream();
```

- **Adding Media Tracks:** Creates an **RTCPeerConnection** object with a STUN server for NAT traversal. It attaches local audio and video tracks to the peer connection.

```
function createPeerConnection(peerSocketId) {
    const peerConnection = new RTCPeerConnection({
        iceServers: [{ urls: 'stun:stun.l.google.com:19302' }],
    });

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

- **Handling Remote Streams:** Receives remote media tracks from other peers and displays them in dynamically created video elements.

```
peerConnection.ontrack = (event) => {
    const remoteStream = event.streams[0];
    const remoteVideo = document.createElement('video');
    remoteVideo.srcObject = remoteStream;
    remoteVideo.autoplay = true;
    videoGrid.appendChild(remoteVideo);
};
```

- **WebRTC Signaling:** Handles ICE candidate generation and signaling. It exchanges SDP offers, answers, and ICE candidates between peers to establish a WebRTC connection.

```
peerConnection.onicecandidate = (event) => {
    if (event.candidate) {
        socket.emit('ice-candidate', event.candidate, peerSocketId);
    }
};
socket.on('offer', async (offer, peerSocketId) => {
    const peerConnection = createPeerConnection(peerSocketId);
    await peerConnection.setRemoteDescription(new RTCSessionDescription(offer));
    const answer = await peerConnection.createAnswer();
    await peerConnection.setLocalDescription(answer);
    socket.emit('answer', answer, peerSocketId);
```

```
});
```

- **Chat Functionality:** Implements real-time chat by sending and receiving messages through the signaling server. Messages are displayed in the chat container.

```
function sendMessage() {
    const message = chatInput.value.trim();
    if (message) {
        addMessageToChat(`You: ${message}`, 'self');
        socket.emit('chat-message', message);
        chatInput.value = '';
    }
}
socket.on('chat-message', (message, senderId) => {
    addMessageToChat(`${senderId.substring(0, 3)}...: ${message}`);
});
```

- **Mute/Unmute Audio and Video:** Allows users to toggle their audio and video streams on or off during the call.

```
muteAudioButton.addEventListener('click', () => {
    const audioTrack = localStream.getAudioTracks()[0];
    if (audioTrack) {
        audioTrack.enabled = !audioTrack.enabled;
        muteAudioButton.textContent = audioTrack.enabled ? 'Mute Audio' : 'Unmute
Audio';
    }
});
muteVideoButton.addEventListener('click', () => {
    const videoTrack = localStream.getVideoTracks()[0];
    if (videoTrack) {
        videoTrack.enabled = !videoTrack.enabled;
        muteVideoButton.textContent = videoTrack.enabled ? 'Mute Video' : 'Unmute
Video';
    }
});
```

Additional Features

- **Mute and Unmute Audio/Video:** Users can easily toggle their audio and video streams on or off during the call. The "Mute Audio" and "Mute Video" buttons

allow users to maintain privacy or eliminate background noise when needed. These buttons dynamically update their labels and styles to indicate the current state (e.g., "Unmute Audio" when muted).

- **Speaking Indicator:** The application visually highlights the video division of a user who is speaking, based on their voice strength. When the volume of the user's audio exceeds a certain threshold, the border and shadow of their video element change dynamically, drawing attention to the active speaker. This feature uses an audio analyzer to monitor the audio levels in real-time and provides a seamless way to identify the speaker during group calls.

Scenarios

In a scenario with three peers on the same local network but unaware of each other's IP addresses, a **STUN server** is used to facilitate the discovery of local IPs and establish direct peer-to-peer connections. Initially, all three peers connect to a **signaling server** to exchange session description information (SDP) and ICE candidates. Each peer uses the STUN server to discover its private IP address and share it with the other peers during the signaling process. This ensures that each peer can identify the network routes necessary for direct communication without needing a TURN server.

Once the signaling is complete and ICE candidates are exchanged, each peer establishes a direct connection with the others using their private local IPs. The audio and video streams flow directly between the peers, minimizing latency since the communication remains within the local network. This setup is efficient and lightweight, as the STUN server is only used during the initial connection

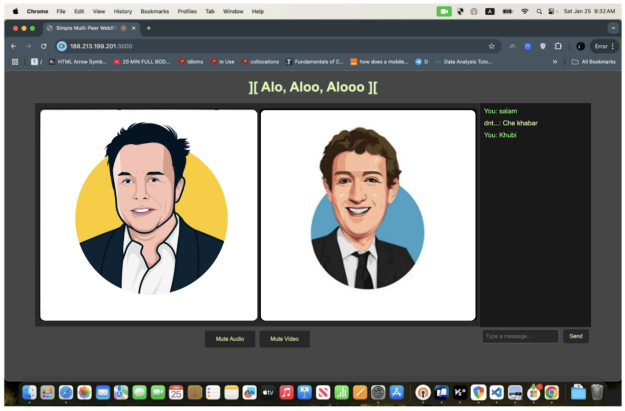
phase, and the actual media streams do not leave the local network. Such a scenario is ideal for group calls or collaborations within a shared network environment, providing fast and reliable communication.

In the second scenario, the peers are located in different networks (e.g., behind separate NATs or firewalls) and are unable to establish a direct peer-to-peer connection. This occurs because their private IP addresses are not directly accessible from outside their respective networks. While a **STUN server** helps peers discover their public-facing IP addresses, certain NAT configurations (e.g., symmetric NAT) or restrictive firewalls prevent a direct connection from being established. As a result, audio and video streams cannot flow between the peers, necessitating the use of a **TURN server**.

A **TURN (Traversal Using Relays around NAT) server** acts as a relay for media streams when direct peer-to-peer communication fails. During the signaling process, ICE candidates generated by the TURN server are exchanged between peers. If no direct route is found, the peers send their audio and video streams to the TURN server, which then relays the streams to the other peers. This ensures reliable communication even in challenging network conditions, albeit with slightly higher latency due to the relaying

process. By using a TURN server, peers in different networks can seamlessly share audio and video streams, enabling real-time communication regardless of network restrictions.

Without TURN (using STUN)

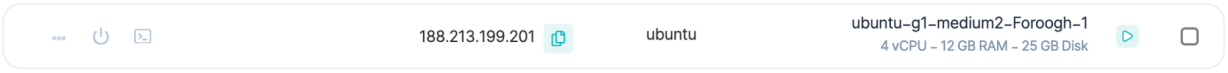


With TURN

Challenges and Solutions

- **Peer-to-Peer Structure and Connection Mapping:** One of the primary challenges was managing the peer-to-peer (P2P) connection structure as more users joined the call. A P2P architecture requires creating individual connections for each peer, which means that with every new participant, the number of connections grows exponentially. To address this, a map of connections was implemented to dynamically manage these relationships. This approach allowed the server to track active peers and their associated connections efficiently, ensuring smooth communication without overwhelming the system.
- **Configuring a VPS on Arvan Cloud:** Setting up a Virtual Private Server (VPS) on Arvan Cloud presented initial challenges, particularly in configuring the environment to host the signaling server, STUN/TURN server, and client application. Ensuring compatibility between all components required proper setup of network rules and server dependencies. After thorough testing and troubleshooting, the VPS was successfully configured to provide a stable

environment for the WebRTC application, facilitating global accessibility for all peers. http://188.213.199.201:3000/



- **Addressing Insecure Origin Issues:** When connecting to the signaling server, browsers like Chrome flagged the application as an insecure origin because the protocol was HTTP instead of HTTPS. This blocked connections due to security restrictions. As an interim solution during development, Chrome's settings were modified using the chrome://flags interface to temporarily treat the origin as secure. This allowed testing to proceed, but the long-term solution involved migrating the application to a secure HTTPS protocol for production environments.
- **STUN Server Limitations:** After deploying the STUN server on the VPS, it became apparent that clients could only connect with each other when they were on the same network. This limitation was due to the STUN server's inability to handle complex NAT configurations or restrictive firewalls. This challenge highlighted the need for a TURN server, which could relay media streams when direct connections failed. Consequently, the Coturn software was configured as a TURN server to resolve this issue and enable seamless communication across different networks.
- **Configuring and Deploying a TURN Server:** The failure of the STUN-only solution led to the decision to configure a TURN server using Coturn. This was a critical step in addressing connectivity issues for peers on different networks. Setting up Coturn required careful configuration of network ports, security credentials, and integration with the WebRTC application. After implementation, the TURN server successfully relayed audio and video streams, ensuring reliable communication even in restrictive network environments.

```
sudo apt update
sudo apt install coturn
sudo nano /etc/turnserver.conf

# add this config to the end of the config file
listening-port=3478
fingerprint
use-auth-secret
static-auth-secret=596a61dc4165c2a71ba497f4570a8929
realm=5.34.195.146
no-stdout-log
user=amiroxEsmeMan:salamEsmeManAmire

sudo systemctl enable coturn
sudo systemctl start coturn

sudo ufw allow 3478/tcp
sudo ufw allow 3478/udp
sudo ufw allow 8000
```

Then we should add our configured TURN server to list of ICE Servers in our client code.

```
const peerConnection = new RTCPeerConnection({
       iceServers: [
           { urls: 'stun:stun.l.google.com:19302' },
           {
               urls: "turn:188.213.199.201:3478",
              username: "amiroxEsmeMan",
              credential: "salamEsmeManAmire",
          },
       ],
   });
```

- **CSS and JavaScript Challenges:** As peers joined the call sequentially, there were challenges in maintaining a consistent user interface (UI) for all participants. The dynamic nature of adding and removing video elements required precise CSS and JavaScript adjustments to ensure the layout remained visually appealing and functional. These issues were resolved through iterative testing and debugging,

resulting in a polished client view that adapted seamlessly to the changing number of participants in the call.

Questions

1) **Difference Between STUN and TURN**

STUN (Session Traversal Utilities for NAT) is a server that helps discover the public IP address of a user and shares it with the other peer to establish a direct Peer-to-Peer (P2P) connection. When users are behind NAT or firewalls, and direct communication is not possible due to restrictive NAT configurations, **TURN** (Traversal Using Relays around NAT) is used. The TURN server acts as a relay to transmit data (audio and video) between peers. STUN is used for networks with simple NAT, while TURN is necessary for more complex or restrictive network environments.

$$Bandwidth = Resoloution\ Bitrate * Frame\ Rate$$

$$Resolution\ Bitrate = Resolution\ Factor * Frame\ Rate * Codec\ Efficiency\ Factor$$

2) **Bandwidth Required for a 720p Video Call**

For a 720p video call, the typical bitrate is 1,500 kbps (kilobits per second 1280x720). Assuming a frame rate of 30 frames per second (fps).

Resolution Bitrate: 1,500 kbps
Frame Rate: 30 fps

Since the bitrate already factors in the frame rate, the required bandwidth is approximately 1.5 Mbps (megabits per second) per user for smooth 720p video streaming.

3) **Bandwidth Required for a 1080p Video Call**

For a 1080p video call, the typical bitrate is 3,000 kbps (1920x1080). Using the same frame rate of 30 fps, the bandwidth requirement can be:
Resolution Bitrate: 3,000 kbps
Frame Rate: 30 fps

Again, as the bitrate accounts for the frame rate, the required bandwidth is approximately **3 Mbps (megabits per second)** per user for high-quality 1080p video streaming.
Also I must add that both text chat and audio needs an additional bandwidth.
Text chat uses negligible bandwidth, as messages are typically a few kilobytes at most.
Audio streams typically require much less bandwidth compared to video streams.
Opus Codec: Requires **20-40 kbps** for narrowband (voice-only) and **50-100 kbps** for wideband (high-quality audio).

Resources

- WebRTC Slides
- Diagarm
- NodeJS Documentation
- WebRTC Documentation
- Learn STUN & TURN Servers on WebRTC
- CSS Tutorial w3schools
- Environment: signaling, STUN and TURN servers