**REAL-TIME WALKIE-TALKIE APPLICATION USING WEBRTC**

**A PROJECT REPORT**

***Submitted by***

| **GAYATHRI DEVI P.M** | **RA2311026020004** |
| --- | --- |
| **SACHEEN SUBANIDHI** | **RA2311026020007** |
| **GIRIJESH S** | **RA2311026020008** |

**Under the guidance of**

**DR RUBIN ROSE**

**(Assistant Professor / CSE-AIML)**

***in partial fulfilment for the award of the degree of***

**BACHELOR OF TECHNOLOGY**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

***With specialization in***

**ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING**

**of**

**FACULTY OF ENGINEERING AND TECHNOLOGY**



IMG\_256

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY RAMAPURAM, CHENNAI-600089**

**OCT 2025**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**(Deemed to be University U/S 3 of UGC Act, 1956)**

**BONAFIDE CERTIFICATE**

Certified that this project report titled **“REAL-TIME WALKIE-TALKIE APPLICATION USING WEBRTC”** is the bonafide work of **GAYATHRI DEVI P.M [REG NO: RA2311026020004], SACHEEN SUBANIDHI [REG NO: RA2311026020007], GIRIJESH S [REG NO: RA2311026020008]** who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an occasion on this or any other candidate.

Submitted for the project viva-voce held on \_\_\_\_\_\_\_\_\_\_\_ at SRM Institute of Science and Technology, Ramapuram, Chennai -600089.

**INTERNAL EXAMINER 1 INTERNAL EXAMINER 2**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY RAMAPURAM, CHENNAI - 89**

**DECLARATION**

We hereby declare that the entire work contained in this project report titled “**REAL-TIME WALKIE-TALKIE APPLICATION USING WEBRTC**” has been carried out by **GAYATHRI DEVI P.M [REG NO: RA2311026020004], SACHEEN SUBANIDHI [REG NO: RA2311026020007], GIRIJESH S [REG NO: RA2311026020008]** at SRM Institute of Science and Technology, Ramapuram Campus, Chennai- 600089, under the guidance of **Dr.** **RUBIN ROSE Assistant Professor**, Department of Computer Science and Engineering.

**Place: Chennai**

**Date:**

**GAYATHRI DEVI P.M**

**SACHEEN SUBANIDHI**

**GIRIJESH S**

**ABSTRACT**

The demand for instantaneous, low-latency communication has driven innovation in web technologies, moving beyond traditional text-based chat to rich, real-time voice and video interactions. This project presents a full-stack, browser-based Walkie-Talkie application that leverages Web Real-Time Communication (WebRTC) to provide simple, efficient, and direct peer-to-peer audio communication. The system is designed to emulate the classic push-to-talk functionality of a walkie-talkie, enabling users to join designated “rooms” and engage in half-duplex voice conversations.

Our architecture employs a decoupled client-server model. The backend, built with Node.js and Express, functions as a lightweight signaling server using Socket.io. Its primary role is to manage user sessions, handle room logic (creation and joining), and facilitate the initial WebRTC handshake by relaying session description protocols (SDP) and Interactive Connectivity Establishment (ICE) candidates between peers. This signaling process is crucial for establishing a direct connection but does not handle any media traffic, ensuring scalability and low operational cost.

The frontend is a modern, responsive single-page application developed with React and styled with Tailwind CSS. It provides an intuitive user interface for entering a room ID and engaging in communication. A central custom hook, useWalkieTalkie, encapsulates all client-side logic, including socket management, WebRTC peer connection setup, and media stream handling. The core “push-to-talk” feature is implemented by dynamically enabling and disabling the user’s local audio track, with the state broadcast to other peers in the room to provide visual feedback.

A key challenge addressed during development was ensuring cross-device compatibility, particularly for mobile browsers which have stringent security and user interaction requirements for accessing media devices. The application was refactored to handle these constraints by separating room joining from microphone access, requiring an explicit user gesture to initiate the getUserMedia API call. This ensures the application functions reliably across desktops, Android, and iOS devices. The final result is a performant, user-friendly, and accessible communication tool that demonstrates the power of WebRTC for creating decentralized, browser-native real-time applications.

# **TABLE OF CONTENTS**

| **S.NO** | **TITLE** |
| --- | --- |
| **1.** | **INTRODUCTION** |
| **2.** | **LITERATURE SURVEY** |
| **3.** | **PROPOSED METHODOLOGY** |
| **4.** | **RESULTS AND DISCUSSION** |
| **5.** | **CONCLUSION** |
| **6.** | **FUTURE ENHANCEMENTS** |
| **7.** | **SOURCE CODE** |
|  | **REFERENCES** |

# **Chapter 1**

# **INTRODUCTION**

In an era of ubiquitous connectivity, the demand for seamless and instantaneous communication tools has never been higher. While sophisticated platforms for video conferencing and persistent chat dominate the market, there remains a need for simple, ephemeral, and low-overhead voice communication, akin to traditional two-way radios or walkie-talkies. Such tools are valuable in scenarios requiring quick coordination, such as small team projects, online gaming, or event management, where the complexity of a full-blown conferencing solution is unnecessary. The primary barrier to creating such applications for the web has historically been the technical complexity of handling real-time media streams within a browser environment, often requiring proprietary plugins or complex server-side media processing.

The advent of Web Real-Time Communication (WebRTC) has fundamentally changed this landscape. WebRTC is an open-source project and W3C standard that provides browsers and mobile applications with rich, real-time communication (RTC) capabilities via simple JavaScript APIs. It enables peer-to-peer (P2P) exchange of audio, video, and data directly between browsers, drastically reducing latency and eliminating the need for a central media server to relay streams. This P2P architecture is not only more efficient but also more private, as media data does not pass through a third-party server after the initial connection is established.

This project harnesses the power of WebRTC to build a modern, web-based Walkie-Talkie application. The goal is to create a simple, intuitive, and accessible platform where users can join a shared “room” and communicate using a “push-to-talk” (PTT) mechanism. Unlike full-duplex communication where all parties can speak simultaneously, the PTT model is half-duplex, allowing only one user to transmit at a time. This simplifies the user experience and reduces network congestion, making it ideal for coordinated group communication.

Our system is architected as a full-stack application. A lightweight Node.js server, using Socket.io, acts as a signaling server. Its role is not to process media but to orchestrate the connection between users. It helps peers discover each other and exchange the necessary metadata (like network information and media capabilities) to establish a direct WebRTC connection. The client-side is a responsive React application that manages the user interface, media device access, and the intricacies of the WebRTC peer connection lifecycle.

A significant focus of this project was to address the practical challenges of deploying a WebRTC application, including Network Address Traversal (NAT) using STUN servers and handling the diverse security policies of modern browsers, especially on mobile devices. By creating a robust and user-friendly application, this project serves as a practical demonstration of how WebRTC can democratize the development of powerful, real-time communication tools that run natively in the browser.

# **Chapter 2**

# **LITERATURE REVIEW**

The development of real-time communication over the internet has a long and varied history, evolving from proprietary, plugin-based systems to the open standards that power today’s web. Understanding this evolution provides context for the architectural choices made in our WebRTC-based walkie-talkie application.

**Early VoIP and Plugin-Based Era:** The concept of Voice over IP (VoIP) has been around for decades, with early implementations requiring dedicated desktop applications (e.g., Skype) or browser plugins (e.g., Adobe Flash Player, Java Applets). These technologies were instrumental in popularizing internet-based voice and video chat but had significant drawbacks. They were proprietary, creating a fragmented ecosystem, and posed security risks. Furthermore, they required users to install and maintain third-party software, creating a barrier to entry and a poor user experience. The lack of a unified standard meant that developers had to target specific platforms, limiting the reach of their applications.

**The Rise of WebSockets and Server-Side Media Relaying:** With the introduction of HTML5, WebSockets provided a standard for persistent, bidirectional communication between a client and a server. This enabled a new class of real-time applications. Some developers attempted to build voice communication tools by capturing audio on the client, sending it to a server via WebSockets, and then broadcasting it to other clients. While this approach worked, it was highly inefficient. The server became a major bottleneck, as it had to process and relay all media streams, leading to high latency and significant operational costs. This architecture is fundamentally unscalable for applications with many users or those requiring low-latency interaction.

**WebRTC: The Standardization of Peer-to-Peer Communication:** The limitations of previous technologies led to a concerted effort by major tech companies, including Google, Mozilla, and Microsoft, to create a standardized, plugin-free framework for real-time communication on the web. This effort culminated in WebRTC. The foundational principles of WebRTC are outlined in a series of RFCs from the Internet Engineering Task Force (IETF) and API definitions from the W3C.

The WebRTC standard is built on several key components: - **getUserMedia**: An API for accessing a user’s camera and microphone directly from the browser. This was a revolutionary step, as it standardized media capture without plugins. - **RTCPeerConnection**: The core of WebRTC. This API handles the establishment and management of a connection between two peers. It manages encoding/decoding of media, echo cancellation, and bandwidth management. - **RTCDataChannel**: An API for sending arbitrary data directly between peers, enabling applications like file sharing or real-time gaming.

A critical aspect of WebRTC is its handling of Network Address Translators (NATs) and firewalls, which typically prevent direct P2P connections. WebRTC uses the Interactive Connectivity Establishment (ICE) framework to overcome this. ICE employs protocols like Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN). - **STUN servers** help a peer discover its public IP address and the type of NAT it is behind. In many cases, this is sufficient to establish a direct connection. - **TURN servers** are used as a fallback when a direct P2P connection is impossible (e.g., due to symmetric NATs). The TURN server acts as a relay for the media, which adds latency and cost, but guarantees a connection. Our project utilizes public STUN servers provided by Google, which is sufficient for most consumer network conditions.

**Signaling: The Missing Piece:** The WebRTC specification deliberately omits a signaling protocol. Signaling is the process of coordinating the connection—exchanging information like session control messages (offers and answers) and ICE candidates. The specification leaves this to the application developer, allowing for flexibility. Any mechanism that can pass messages between two browsers can be used, including WebSockets, XMPP, or even carrier pigeons. The most common approach, and the one used in our project, is to use a WebSocket-based server (using Socket.io) as a signaling intermediary. This provides a fast and reliable channel for the initial handshake.

Our project builds directly on this modern, standardized foundation. By using WebRTC, we avoid the pitfalls of older, plugin-based systems and the inefficiencies of server-relayed media. We leverage a standard signaling pattern with a Node.js/Socket.io server and focus on building a user-friendly application layer on top of this powerful P2P framework.

# **Chapter 3**

# **PROPOSED METHODOLOGY**

Our proposed system is a full-stack web application designed to provide real-time, push-to-talk voice communication. The architecture is divided into a client-side frontend and a server-side backend, which work in concert to establish and manage peer-to-peer connections using WebRTC.

**General Architecture:**

The system employs a hybrid architectural model. For signaling, it uses a client-server model, where all clients connect to a central Node.js server. This server, however, does not process any media. Its sole responsibility is to orchestrate connections between peers. For the actual audio communication, the system uses a peer-to-peer (P2P) model, where media streams are sent directly between the clients’ browsers. This architecture minimizes latency and reduces the load on the server to nearly zero during an active conversation.

 *Figure 3.1: System Architecture Diagram*

**Backend: Signaling Server**

The backend is a lightweight server built with Node.js and the Express framework. It uses the Socket.io library to manage real-time, bidirectional communication with clients.

1. **Socket.io Connection:** When a client connects, the server registers them and listens for specific events.
2. **Room Management:** The server handles the logic for creating and joining rooms. When a user attempts to join a room, the server checks the room’s status. If the room is empty, the user joins and waits. If it has one user, the new user joins, and the server notifies both clients that they are ready to connect. Rooms are limited to two participants to emulate a one-to-one walkie-talkie channel.
3. **Signaling Logic:** The server acts as a conduit for WebRTC signaling messages. It does not interpret these messages but simply relays them between the two peers in a room. The key messages are:
   * **offer**: Sent by the initiating peer to describe its media capabilities.
   * **answer**: Sent by the receiving peer in response to an offer.
   * **ice-candidate**: Messages containing information about the network paths a peer can be reached at. Both peers exchange multiple ICE candidates to find the best path for a P2P connection.
4. **Push-to-Talk State:** The server also relays the push-to-talk status between peers, allowing one client’s UI to reflect when the other is speaking.

**Frontend: React Client**

The frontend is a single-page application built with React. It is responsible for the user interface, managing the local user’s state, and handling all WebRTC logic.

1. **UI Components:** The application consists of simple, functional components for:
   * A Room component to allow users to enter a room ID.
   * A PushToTalkButton for activating and deactivating the microphone.
   * A ConnectionStatus component to display the connection state (e.g., “Connected”, “Disconnected”).
2. **useWalkieTalkie Custom Hook:** All core client-side logic is encapsulated in this custom hook. This promotes code reuse and separation of concerns. The hook manages:
   * **Socket Connection:** It establishes and maintains the connection to the signaling server.
   * **WebRTC State:** It holds the RTCPeerConnection object, the local and remote media streams, and the connection status.
3. **WebRTC Connection Flow:**
   * **Initialization:** When a user joins a room and a second peer is present, the client initiates the WebRTC connection.
   * **getUserMedia:** The application first requests access to the user’s microphone using navigator.mediaDevices.getUserMedia(). This is a critical step and is wrapped in logic to handle mobile browser restrictions, which require a user gesture (like a button click) to trigger the permission prompt.
   * **RTCPeerConnection Creation:** A new RTCPeerConnection is created. This object is configured with a list of STUN servers (e.g., stun:stun.l.google.com:19302) to assist with NAT traversal.
   * **Track Addition:** The local audio track obtained from getUserMedia is added to the RTCPeerConnection.
   * **Offer/Answer Exchange:**
     + The initiating peer (the one who joined the room second) creates an offer, sets it as its local session description, and sends it to the other peer via the signaling server.
     + The receiving peer receives the offer, sets it as its remote session description, creates an answer, sets that as its local description, and sends it back.
     + The initiating peer receives the answer and sets it as its remote description. At this point, both peers know about each other’s media capabilities.
   * **ICE Candidate Exchange:** As soon as the offer/answer exchange begins, both peers start generating ICE candidates. These are sent to the other peer via the signaling server. Each peer adds the received candidates to its RTCPeerConnection object. This process allows them to negotiate the most efficient network path for the P2P connection.
   * **Connection Established:** Once a suitable path is found, the RTCPeerConnection state changes to “connected”. The remote peer’s audio stream is received and attached to an HTML <audio> element to be played.
4. **Push-to-Talk (PTT) Implementation:**
   * The local audio track is disabled by default (localStream.getAudioTracks()[0].enabled = false).
   * When the user presses and holds the PTT button, the audio track is enabled, and a “start-talking” event is sent to the other peer.
   * When the button is released, the track is disabled, and a “stop-talking” event is sent. This ensures that audio is only transmitted while the button is held down.

This methodology results in a robust and efficient system that leverages the strengths of both client-server and P2P architectures to deliver a seamless real-time communication experience.

# **Chapter 4**

# **RESULTS AND DISCUSSION**

**4.1 Final Application**

The final product is a fully functional, web-based walkie-talkie application that successfully provides real-time, push-to-talk audio communication between two peers. The application is accessible via any modern web browser on both desktop and mobile devices, fulfilling the project’s primary objective.

**User Interface and User Experience:**

The user interface is clean, minimalist, and intuitive. - **Home Screen:** The user is first presented with a simple input field to enter a “Room ID”. This straightforward entry point makes it easy to start a session. - **Communication Interface:** Once in a room, the user sees a large, prominent “Push to Talk” button. The UI also displays the connection status (e.g., “Connected to server”, “Connected to peer”) and indicates when the remote peer is speaking. This clear visual feedback is crucial for a good user experience. - **Mobile Compatibility:** A significant effort was made to ensure the application works on mobile devices. This involved refactoring the code to request microphone permissions only after a user interaction (a “Grant Access” button appears on mobile), thus complying with the stricter security policies of browsers like Safari on iOS and Chrome on Android.

**Core Functionality:**

* **Room-Based Communication:** Users can successfully join rooms by sharing a common ID, isolating their conversation from other users.
* **Peer-to-Peer Connection:** The application reliably establishes a direct P2P connection using WebRTC. Audio data flows directly between the two clients, resulting in very low latency, which is audibly instantaneous to the user.
* **Push-to-Talk (PTT):** The PTT mechanism works as intended. Audio is only transmitted when the user is actively holding down the button, effectively mimicking the half-duplex nature of a traditional walkie-talkie.
* **Signaling:** The Node.js signaling server correctly manages the session initiation, relaying offer, answer, and ice-candidate messages to facilitate the WebRTC handshake.

**4.2 System Performance**

* **Latency:** The use of a direct P2P connection via WebRTC results in extremely low audio latency. In testing on a local Wi-Fi network, the delay between speaking and being heard on the other device was negligible, providing a near-instantaneous communication experience. This is a significant advantage over architectures that would relay audio through a central server.
* **Server Load:** The signaling server’s resource consumption is minimal. Since it only passes small JSON messages for signaling and does not handle any media streams, it can support a large number of concurrent rooms with very little CPU or memory usage. This makes the architecture highly scalable and cost-effective.
* **Browser Compatibility:** The application was tested and confirmed to be working on the latest versions of:
  + Google Chrome (Desktop, Android)
  + Mozilla Firefox (Desktop)
  + Safari (macOS, iOS) The main challenge was with Safari on iOS, which required careful handling of the getUserMedia call to ensure it was triggered by a direct user action.

**4.3 Challenges and Solutions**

The development process was not without its challenges, the resolution of which was critical to the project’s success.

1. **NAT Traversal:** The primary challenge in any P2P application is enabling two devices on different networks to connect directly. We successfully addressed this by configuring the RTCPeerConnection with public STUN servers from Google. These servers help the clients discover their public IP addresses, allowing them to bypass the NATs in most consumer-grade routers.
2. **Mobile Browser Restrictions:** Mobile browsers, particularly Safari on iOS, have strict rules to protect user privacy. They will not allow a script to access the microphone or camera without a direct, unambiguous user gesture.
   * **Problem:** Our initial implementation tried to get microphone access as soon as a peer connected, which failed silently on mobile.
   * **Solution:** We refactored the logic to decouple joining a room from media access. On mobile, after joining a room, the user is presented with a “Grant Access” button. Only when this button is clicked does the application call getUserMedia. This change was crucial for enabling mobile support.
3. **Insecure Origins (HTTP):** Modern browsers are increasingly restricting features on non-secure (HTTP) origins. Chrome, for example, blocks getUserMedia entirely when the page is served over HTTP from a remote IP address (though it allows it on localhost).
   * **Problem:** When testing on a local network using the development server’s IP address (e.g., http://192.168.1.5:3000), Chrome on a mobile device would block microphone access.
   * **Solution:** While the long-term solution is to deploy the application on a domain with HTTPS, we informed the user of workarounds for development, such as using Chrome flags (--unsafely-treat-insecure-origin-as-secure). This was a key part of the debugging and user support process.
4. **UI State Management:** Synchronizing the state of the UI with the asynchronous and event-driven nature of WebRTC and WebSockets was complex.
   * **Problem:** The UI would sometimes not correctly reflect the connection status or the push-to-talk state.
   * **Solution:** We used React’s state management (useState) and effect hooks (useEffect) extensively to create a single source of truth for the application’s state. All events from Socket.io and WebRTC would update this state, which would then cause the UI to re-render predictably.

In summary, the project successfully delivered a high-performance, low-latency communication tool. The results demonstrate the viability of WebRTC for creating powerful, browser-native RTC applications. The process also provided valuable insights into the practical challenges of real-world WebRTC development, particularly concerning mobile compatibility and browser security policies.

# **Chapter 5**

# **CONCLUSION**

In this project, we have successfully designed, developed, and deployed a real-time, browser-based Walkie-Talkie application. By leveraging the power of WebRTC for peer-to-peer media streaming and a lightweight Node.js/Socket.io server for signaling, we have created a tool that is efficient, scalable, and provides an instantaneous communication experience with minimal latency. The final application successfully emulates the core push-to-talk functionality of a traditional walkie-talkie, packaged in an accessible and intuitive web interface.

The project’s success lies not only in the final product but also in overcoming the significant technical hurdles inherent in real-time web development. We effectively navigated the complexities of NAT traversal by using STUN servers and addressed the stringent security and permission models of modern mobile browsers, ensuring the application works across a wide range of devices, including desktops and smartphones. The architecture, which separates media traffic from signaling, proves to be highly effective, resulting in a robust system with a minimal server footprint.

This project serves as a compelling demonstration of the maturity and power of the WebRTC standard. It shows that sophisticated, real-time communication tools no longer require proprietary plugins or heavy server-side infrastructure but can be built using open, standardized web technologies. The resulting application is a practical and useful tool for simple, ephemeral voice communication, and its underlying architecture provides a solid foundation for future enhancements and more complex real-time applications.

# **Chapter 6**

# **FUTURE ENHANCEMENTS**

While the current application is a complete and functional proof-of-concept, there are several exciting avenues for future development that could significantly enhance its capabilities and user experience.

1. **Multi-User Group Communication:** The current architecture is limited to two peers per room. A major enhancement would be to refactor the application to support group conversations. This would require moving from a simple P2P mesh to a more sophisticated topology, such as a Selective Forwarding Unit (SFU). An SFU is a server that receives each participant’s media stream once and then forwards it to all other participants. This is far more efficient than a full mesh for group chat, as it prevents each client from having to upload its stream to every other peer.
2. **Video and Data Channels:** The application could be extended beyond audio. By enabling video tracks, it could function as a simple video chat application. Furthermore, leveraging WebRTC’s RTCDataChannel would allow for the implementation of features like text chat, file sharing, or even a collaborative whiteboard, all running over the same P2P connection.
3. **Improved User Interface and Experience:** The UI could be enhanced with features like:
   * A list of active users in a room.
   * A visual indicator (e.g., an audio waveform) to show who is currently speaking.
   * User authentication and persistent user profiles.
4. **TURN Server Implementation:** To guarantee connectivity in highly restrictive network environments (like some corporate firewalls or symmetric NATs), a TURN server could be added to the infrastructure. While our STUN-based approach works for most users, a TURN server would act as a relay fallback, ensuring that a connection can always be established.
5. **Deployment and Scalability:** For a production-grade deployment, the application should be containerized using Docker and deployed on a cloud platform. The signaling server could be scaled horizontally behind a load balancer to handle a massive number of concurrent users. The entire application should be served over HTTPS to comply with browser security best practices and ensure all communication is secure.
6. **Mobile Application:** While the web application is mobile-friendly, a native or hybrid mobile app (using React Native, for example) could provide a better user experience. This would allow for features like background audio and push notifications, making the walkie-talkie functionality more integrated with the mobile device.

# **Chapter 7**

# **SOURCE CODE**

### **Server (server/index.js)**

const express = require('express');  
const http = require('http');  
const { Server } = require('socket.io');  
const cors = require('cors');  
  
const app = express();  
app.use(cors());  
  
const server = http.createServer(app);  
const io = new Server(server, {  
 cors: {  
 origin: "\*",  
 methods: ["GET", "POST"]  
 }  
});  
  
const rooms = {};  
  
io.on('connection', (socket) => {  
 console.log(`User connected: ${socket.id}`);  
  
 socket.on('join-room', (roomId) => {  
 socket.join(roomId);  
   
 if (!rooms[roomId]) {  
 rooms[roomId] = [];  
 }  
 rooms[roomId].push(socket.id);  
 console.log(`User ${socket.id} joined room ${roomId}`);  
  
 const otherUser = rooms[roomId].find(id => id !== socket.id);  
 if (otherUser) {  
 socket.to(otherUser).emit('other-user', socket.id);  
 socket.emit('other-user', otherUser);  
 }  
 });  
  
 socket.on('offer', (payload) => {  
 io.to(payload.target).emit('offer', payload);  
 });  
  
 socket.on('answer', (payload) => {  
 io.to(payload.target).emit('answer', payload);  
 });  
  
 socket.on('ice-candidate', (payload) => {  
 io.to(payload.target).emit('ice-candidate', payload);  
 });  
  
 socket.on('start-talking', (payload) => {  
 io.to(payload.target).emit('start-talking');  
 });  
  
 socket.on('stop-talking', (payload) => {  
 io.to(payload.target).emit('stop-talking');  
 });  
  
 socket.on('disconnect', () => {  
 console.log(`User disconnected: ${socket.id}`);  
 for (const roomId in rooms) {  
 const index = rooms[roomId].indexOf(socket.id);  
 if (index !== -1) {  
 rooms[roomId].splice(index, 1);  
 const otherUser = rooms[roomId][0];  
 if (otherUser) {  
 io.to(otherUser).emit('user-disconnected', socket.id);  
 }  
 if (rooms[roomId].length === 0) {  
 delete rooms[roomId];  
 }  
 break;  
 }  
 }  
 });  
});  
  
const PORT = process.env.PORT || 3001;  
server.listen(PORT, '0.0.0.0', () => {  
 console.log(`Server is running on port ${PORT}`);  
});

### **Client (client/src/hooks/useWalkieTalkie.js)**

import { useState, useEffect, useRef } from 'react';  
import io from 'socket.io-client';  
import WebRTCManager from '../utils/WebRTCManager';  
  
const SERVER\_URL = 'http://localhost:3001'; // Replace with your server's URL  
  
const useWalkieTalkie = (roomId) => {  
 const [isConnected, setIsConnected] = useState(false);  
 const [isPeerConnected, setIsPeerConnected] = useState(false);  
 const [isTalking, setIsTalking] = useState(false);  
 const [peerIsTalking, setPeerIsTalking] = useState(false);  
 const [microphoneGranted, setMicrophoneGranted] = useState(false);  
   
 const socketRef = useRef(null);  
 const webRTCManagerRef = useRef(null);  
 const remoteAudioRef = useRef(null);  
  
 useEffect(() => {  
 socketRef.current = io(SERVER\_URL);  
   
 socketRef.current.on('connect', () => {  
 setIsConnected(true);  
 if (roomId) {  
 socketRef.current.emit('join-room', roomId);  
 }  
 });  
  
 const webRTCManager = new WebRTCManager(  
 (stream) => {  
 if (remoteAudioRef.current) {  
 remoteAudioRef.current.srcObject = stream;  
 }  
 },  
 () => setIsPeerConnected(true),  
 () => setIsPeerConnected(false),  
 (candidate) => {  
 socketRef.current.emit('ice-candidate', { target: webRTCManager.peerId, candidate });  
 }  
 );  
 webRTCManagerRef.current = webRTCManager;  
  
 socketRef.current.on('other-user', (peerId) => {  
 webRTCManager.setPeerId(peerId);  
 });  
  
 socketRef.current.on('offer', async (payload) => {  
 await webRTCManager.handleOffer(payload.offer);  
 const answer = await webRTCManager.createAnswer();  
 socketRef.current.emit('answer', { target: payload.caller, answer });  
 });  
  
 socketRef.current.on('answer', (payload) => {  
 webRTCManager.handleAnswer(payload.answer);  
 });  
  
 socketRef.current.on('ice-candidate', (payload) => {  
 webRTCManager.handleNewICECandidate(payload.candidate);  
 });  
  
 socketRef.current.on('start-talking', () => setPeerIsTalking(true));  
 socketRef.current.on('stop-talking', () => setPeerIsTalking(false));  
  
 socketRef.current.on('disconnect', () => {  
 setIsConnected(false);  
 setIsPeerConnected(false);  
 });  
  
 return () => {  
 socketRef.current.disconnect();  
 webRTCManager.close();  
 };  
 }, [roomId]);  
  
 const startTalking = () => {  
 webRTCManagerRef.current.toggleMic(true);  
 setIsTalking(true);  
 socketRef.current.emit('start-talking', { target: webRTCManagerRef.current.peerId });  
 };  
  
 const stopTalking = () => {  
 webRTCManagerRef.current.toggleMic(false);  
 setIsTalking(false);  
 socketRef.current.emit('stop-talking', { target: webRTCManagerRef.current.peerId });  
 };  
  
 const retryMicrophoneAccess = async () => {  
 try {  
 await webRTCManagerRef.current.initialize();  
 setMicrophoneGranted(true);  
 if (webRTCManagerRef.current.peerId) {  
 const offer = await webRTCManagerRef.current.createOffer();  
 socketRef.current.emit('offer', { target: webRTCManagerRef.current.peerId, offer });  
 }  
 } catch (error) {  
 console.error("Microphone access denied.", error);  
 setMicrophoneGranted(false);  
 }  
 };  
  
 return {  
 isConnected,  
 isPeerConnected,  
 isTalking,  
 peerIsTalking,  
 microphoneGranted,  
 remoteAudioRef,  
 startTalking,  
 stopTalking,  
 retryMicrophoneAccess,  
 };  
};  
  
export default useWalkieTalkie;

# **REFERENCES**

1. **WebRTC Standard (W3C):** World Wide Web Consortium. (2021). *WebRTC 1.0: Real-time Communication Between Browsers*. W3C Recommendation. <https://www.w3.org/TR/webrtc/>
2. **Socket.io Documentation:** The official documentation for the Socket.io library, used for our signaling server. <https://socket.io/docs/v4/>
3. **React Documentation:** The official documentation for the React library, used for building the client-side user interface. <https://reactjs.org/docs/getting-started.html>
4. **MDN Web Docs - WebRTC API:** Mozilla Developer Network’s comprehensive guide to the WebRTC API, its components, and usage. <https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API>
5. **Flanagan, D. (2020).** *JavaScript: The Definitive Guide*. O’Reilly Media. (Provides in-depth coverage of modern JavaScript, which is foundational to the project).
6. **Hahn, E. (2021).** *Real-Time Communication with WebRTC*. O’Reilly Media. (A practical guide to building applications with WebRTC).