



UNIVERSITA' DEGLI STUDI DI  
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base  
Corso di Laurea Magistrale in Ingegneria Informatica

Documentazione progetto in Web and Real Time  
Communication Systems

## ***Purple Leopard: WebRTC VideoCall and Minigames***

Anno Accademico 2024/2025

Candidati

**Erika Morelli M63001616**

**Luca Pisani M63001627**

# Contents

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Tecnologie Utilizzate</b>	<b>3</b>
2.1	React & ReactDOM . . . . .	3
2.2	Socket.IO . . . . .	4
2.3	Node.js & Express . . . . .	4
2.4	Vite . . . . .	4
2.5	Altre librerie JavaScript . . . . .	4
2.6	HTML & CSS . . . . .	5
2.7	JavaScript . . . . .	5
<b>3</b>	<b>Frontend</b>	<b>6</b>
3.1	Struttura del frontend . . . . .	6
3.2	Accesso alla webcam (MediaStream) . . . . .	7
3.3	Gestione dell'ID utente . . . . .	7
3.4	Ricezione di una chiamata . . . . .	8
3.5	Gestione degli ICE Candidate . . . . .	9
3.6	Gestione dei minigiochi . . . . .	10
3.7	Gestione dello stato con React Hooks . . . . .	11
3.8	Riferimenti video . . . . .	11

3.9	Conclusione . . . . .	12
<b>4</b>	<b>Backend</b>	<b>13</b>
4.1	Struttura generale . . . . .	13
4.2	Gestione della connessione WebRTC . . . . .	14
4.3	Gestione dei Minigiochi . . . . .	17
4.4	Eventi dedicati ai giochi . . . . .	18
4.5	Gestione della disconnessione . . . . .	19
4.6	Considerazioni finali . . . . .	20
<b>5</b>	<b>Tecnologia WebRTC</b>	<b>21</b>
5.1	Componenti WebRTC . . . . .	21
5.2	Flusso di connessione WebRTC nel progetto . . . . .	22
5.3	Gestione ICE Candidate . . . . .	26
5.4	Ruolo del Signaling Server . . . . .	26
5.5	Sicurezza e scalabilità . . . . .	27
5.6	Conclusioni . . . . .	27
<b>6</b>	<b>Conclusioni Generali</b>	<b>29</b>

# Chapter 1

## Introduzione

Il progetto di cui segue la documentazione consiste in una applicazione web interattiva realizzata con l'obiettivo di esplorare e mettere in pratica le tecnologie di comunicazione real-time browser-to-browser. L'app si compone di un'interfaccia frontend sviluppata adoperando il framework React, affiancata da un backend Node.js che gestisce la logica di segnalazione attraverso Socket.IO.

Il cuore del progetto è rappresentato dall'utilizzo di **WebRTC (Web Real-Time Communication)**, una tecnologia che consente la comunicazione audio/video diretta tra utenti senza necessità di server intermedi per il flusso dei media. Grazie a questo, l'applicazione offre una piattaforma di videochiamata peer-to-peer dinamica, alla quale sono state integrate funzionalità ludiche attraverso una sezione di *minigame*, pensata per arricchire l'esperienza utente durante la connessione.

L'interfaccia utente è stata progettata per essere semplice e intuitiva, mentre la comunicazione client-server è affidata a Socket.IO per garantire uno scambio efficiente dei messaggi di segnalazione necessari alla connessione WebRTC. Il progetto è costruito con una struttura modulare, con particolare attenzione all'organizzazione del codice e alla chiarezza dell'architettura complessiva.

## Chapter 2

# Tecnologie Utilizzate

Il progetto è stato sviluppato utilizzando un insieme moderno di tecnologie per il web development, con un'attenzione particolare alla comunicazione in tempo reale tramite WebRTC. Di seguito sono elencate le principali tecnologie impiegate, accompagnate dalle rispettive versioni.

### 2.1 React & ReactDOM

React v19.0.0 è stato utilizzato per costruire l'interfaccia utente in modo dichiarativo e componibile. Insieme a `react-dom`, gestisce il rendering sul DOM del browser.

## 2.2 Socket.IO

Socket.IO v4.8.1 (sia lato client che server) è utilizzato per il signaling WebRTC e la comunicazione in tempo reale tra i client e il server.

## 2.3 Node.js & Express

Express v4.21.2 è il framework backend scelto, eseguito in ambiente Node.js. È responsabile della gestione del server, delle connessioni socket e del routing.

## 2.4 Vite

Per il frontend, si è utilizzato Vite v5.4.15 come bundler, in combinazione con il plugin @vitejs/plugin-react v4.0.0.

## 2.5 Altre librerie JavaScript

- `socket.io-client` v4.7.5 – usato per connettere il frontend al server WebSocket.
- `react-icons` v5.5.0 – per le icone SVG.
- `cors` v2.8.5 – gestione CORS sul backend.

- `buffer v6.0.3`, `events v3.3.0`, `util v0.12.5` – per compatibilità con Node.js API.

## 2.6 HTML & CSS

La struttura visiva è stata realizzata con HTML5 e stile CSS custom, definito nel file `App.css`, con supporto a componenti React.

## 2.7 JavaScript

Tanto il frontend quanto il backend sono stati sviluppati in JavaScript moderno (ES6+), garantendo coerenza tra client e server.



# Chapter 3

## Frontend

Il frontend dell'applicazione è stato sviluppato utilizzando **React**, con una struttura component-based che consente una gestione dinamica ed efficiente dell'interfaccia utente. L'applicazione è in grado di gestire chiamate WebRTC, streaming video, eventi socket in tempo reale e minigiochi interattivi. Di seguito vengono analizzate le principali funzionalità e componenti.

### 3.1 Struttura del frontend

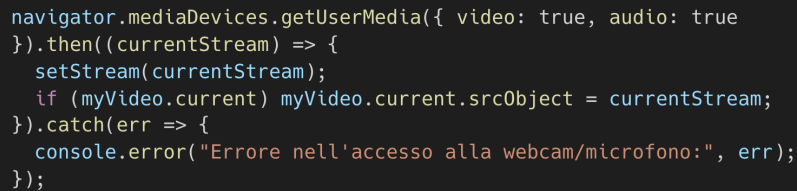
La cartella `frontend/` include i seguenti elementi principali:

- `index.html`: punto di ingresso HTML dell'app.
- `src/App.jsx`: componente React principale dove risiede la logica.
- `src/App.css`: stili applicati all'interfaccia.

- games/: directory con i minigiochi modulari.

## 3.2 Accesso alla webcam (MediaStream)

L'app accede al flusso audio/video dell'utente tramite l'API `getUserMedia`, e assegna il flusso al video locale.

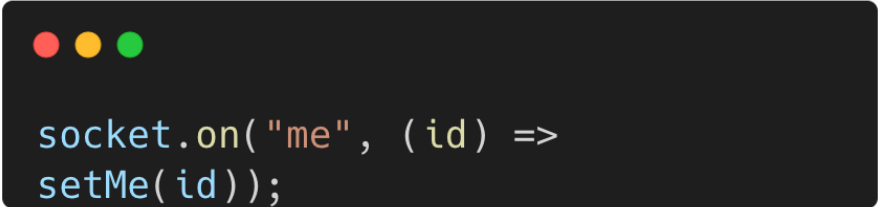
A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following JavaScript code:

```
navigator.mediaDevices.getUserMedia({ video: true, audio: true
}).then((currentStream) => {
  setStream(currentStream);
  if (myVideo.current) myVideo.current.srcObject = currentStream;
}).catch(err => {
  console.error("Errore nell'accesso alla webcam/microfono:", err);
});
```

Figure 3.1: Accesso alla camera e al microfono

## 3.3 Gestione dell'ID utente

Appena il client si connette al server Socket.IO, riceve un identificatore univoco che viene memorizzato per la sessione corrente.

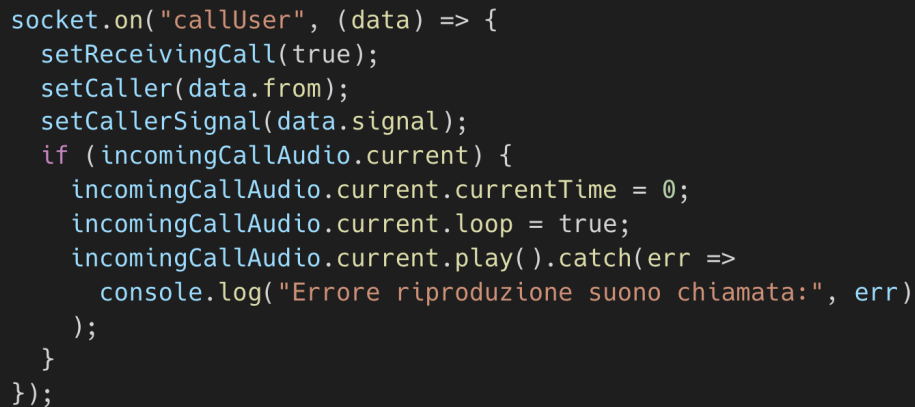


```
socket.on("me", (id) =>
  setMe(id));
```

Figure 3.2: Ricezione dell' ID utente dal server

### 3.4 Ricezione di una chiamata

Quando un altro utente invia una richiesta di chiamata, il sistema aggiorna lo stato e riproduce un suono di notifica.

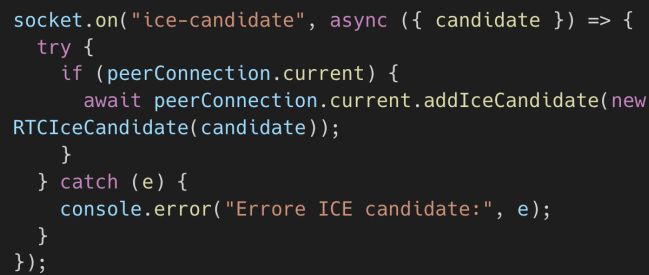


```
socket.on("callUser", (data) => {
  setReceivingCall(true);
  setCaller(data.from);
  setCallerSignal(data.signal);
  if (incomingCallAudio.current) {
    incomingCallAudio.current.currentTime = 0;
    incomingCallAudio.current.loop = true;
    incomingCallAudio.current.play().catch(err =>
      console.log("Errore riproduzione suono chiamata:", err)
    );
  }
});
```

Figure 3.3: Gestione ricezione chiamata

## 3.5 Gestione degli ICE Candidate

WebRTC utilizza ICE (Interactive Connectivity Establishment) per trovare il percorso ottimale tra peer. Le candidate vengono gestite come segue:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following JavaScript code:

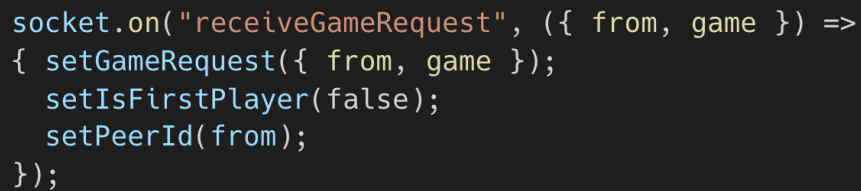
```
socket.on("ice-candidate", async ({ candidate }) => {
  try {
    if (peerConnection.current) {
      await peerConnection.current.addIceCandidate(new
RTCIceCandidate(candidate));
    }
  } catch (e) {
    console.error("Errore ICE candidate:", e);
  }
});
```

Figure 3.4: Aggiunta di ICE Candidates

## 3.6 Gestione dei minigiochi

Durante la chiamata, è possibile avviare minigiochi interattivi. La ricezione e l'avvio sono gestiti tramite eventi socket.


### Ricezione richiesta di gioco

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript code snippet for handling a 'receiveGameRequest' event on a socket.

```
socket.on("receiveGameRequest", ({ from, game }) =>
{ setGameRequest({ from, game });
  setIsFirstPlayer(false);
  setPeerId(from);
});
```

Figure 3.5: Ricezione richiesta minigioco

### Avvio e chiusura del gioco

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript code snippet for handling 'gameStartConfirmed' and 'gameEnded' events on a socket.


```
socket.on("gameStartConfirmed", ({ game }) =>
setActiveGame(game));
socket.on("gameEnded", () => setActiveGame(null));
```

Figure 3.6: Gestione avvio e fine del gioco

## 3.7 Gestione dello stato con React Hooks

React viene utilizzato per mantenere aggiornato lo stato dell'interfaccia.

Un esempio è la gestione dello stream remoto:

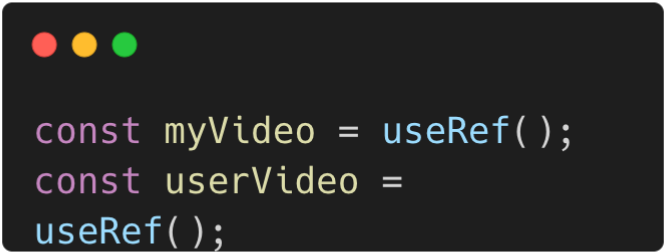
A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of JavaScript code: `const [stream, setStream] =` on the first line and `useState(null);` on the second line.

```
const [stream, setStream] =  
useState(null);
```

Figure 3.7: Stato per lo stream remoto

## 3.8 Riferimenti video

I video locali e remoti sono associati tramite 'ref', per un accesso diretto agli elementi DOM:

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains three lines of JavaScript code: `const myVideo = useRef();` on the first line, `const userVideo =` on the second line, and `useRef();` on the third line.

```
const myVideo = useRef();  
const userVideo =  
useRef();
```

Figure 3.8: Riferimenti ai video

## 3.9 Conclusione

Il frontend integra in modo efficace WebRTC, React e Socket.IO, permettendo un'esperienza utente fluida e interattiva. L'architettura a componenti rende il progetto facilmente estendibile, in particolare per l'aggiunta di nuove funzionalità come ulteriori minigiochi o strumenti collaborativi.

# Chapter 4

## Backend

Il backend dell'applicazione è implementato in Node.js utilizzando **Express** per la gestione del server HTTP e **Socket.IO** per la comunicazione real-time. La sua funzione principale è quella di fungere da *signaling server* per la connessione WebRTC e da coordinatore degli eventi relativi ai minigiochi.

### 4.1 Struttura generale

Il file principale del backend è `server.js`, che svolge le seguenti funzioni:

- Serve i file statici del frontend già compilato
- Inizializza il server HTTP e il server WebSocket
- Gestisce tutti gli eventi socket necessari a WebRTC e ai minigiochi





```
const express = require("express");
const http = require("http");
const { Server } = require("socket.io");
const cors = require("cors");
const path = require("path");


const app = express();
app.use(cors());
app.use(express.static(path.join(__dirname, "..", "frontend",
"dist")));
const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: "*",
    methods: ["GET", "POST"]
  }
});
```

Figure 4.1: Inizializzazione del server

## 4.2 Gestione della connessione WebRTC

Il backend non gestisce direttamente le connessioni peer-to-peer WebRTC, ma si occupa del *signaling*, cioè lo scambio di messaggi tra utenti per avviare la connessione.


### Invio ID utente



```
socket.emit("me",  
socket.id);
```

Figure 4.2: Invio ID utente

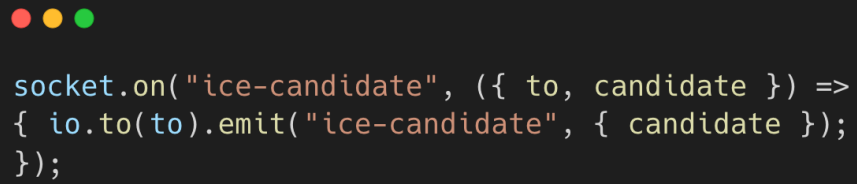
## Chiamata tra utenti



```
socket.on("callUser", ({ userToCall, signalData, from }) => {  
  io.to(userToCall).emit("callUser", { signal: signalData, from  
});  
  
socket.on("answerCall", ({ signal, to }) => {  
  io.to(to).emit("callAccepted", signal);  
});
```

Figure 4.3: Chiamata tra utenti

## Scambio di ICE candidate



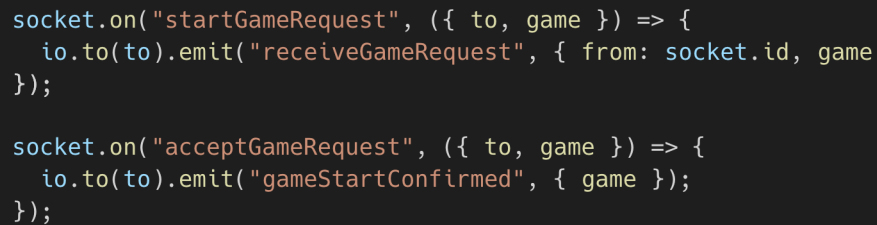
```
socket.on("ice-candidate", ({ to, candidate }) =>
{ io.to(to).emit("ice-candidate", { candidate });
});
```

Figure 4.4: Scambio ICE Candidates

## 4.3 Gestione dei Minigiochi

Il backend riceve le richieste di gioco e le sincronizza tra i due partecipanti.


### Richiesta e accettazione

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of JavaScript code. The first line is a socket listener for 'startGameRequest' that emits 'receiveGameRequest' to the other client. The second line is a socket listener for 'acceptGameRequest' that emits 'gameStartConfirmed' to the other client.

```
socket.on("startGameRequest", ({ to, game }) => {  
  io.to(to).emit("receiveGameRequest", { from: socket.id, game  
});  
  
socket.on("acceptGameRequest", ({ to, game }) => {  
  io.to(to).emit("gameStartConfirmed", { game });  
});
```

Figure 4.5: Richiesta e accettazione

### Chiusura del gioco

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains one line of JavaScript code: a socket listener for 'leaveGame' that emits 'gameEnded' to the other client.

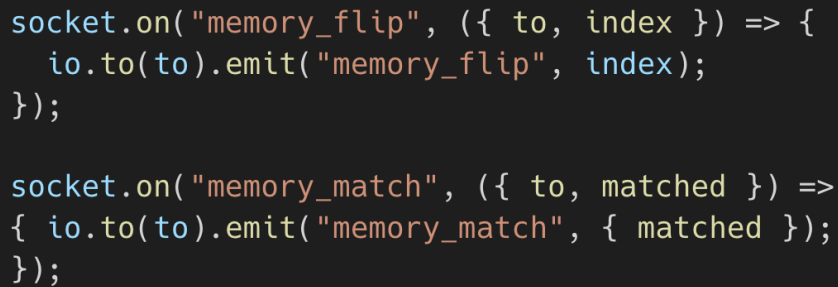
```
socket.on("leaveGame", ({ to }) =>  
{ io.to(to).emit("gameEnded");  
});
```

Figure 4.6: Chiusura del gioco

## 4.4 Eventi dedicati ai giochi

Ogni minigioco ha una serie di eventi specifici. Ad esempio:

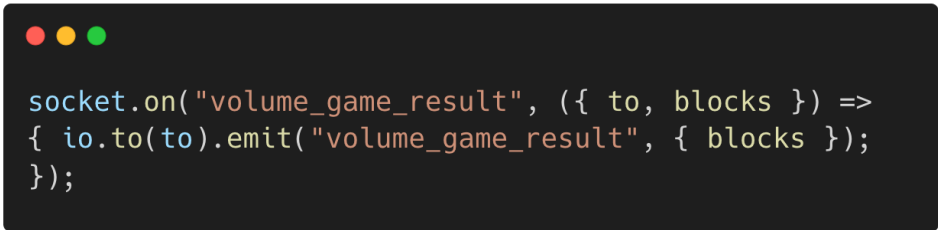
### Memory Game



```
socket.on("memory_flip", ({ to, index }) => {  
  io.to(to).emit("memory_flip", index);  
});  
  
socket.on("memory_match", ({ to, matched }) =>  
{ io.to(to).emit("memory_match", { matched });  
});
```

Figure 4.7: Minigioco A

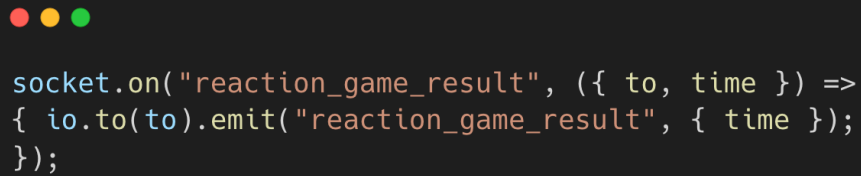
### Volume Game



```
socket.on("volume_game_result", ({ to, blocks }) =>  
{ io.to(to).emit("volume_game_result", { blocks });  
});
```

Figure 4.8: Minigioco B

## Reaction Game




```
socket.on("reaction_game_result", ({ to, time }) =>
{ io.to(to).emit("reaction_game_result", { time });
});
```

Figure 4.9: Minigioco C

## 4.5 Gestione della disconnessione

Il backend rileva la disconnessione degli utenti per eventuali cleanup o notifiche.



```
socket.on("disconnect", () => {
  console.log("Utente disconnesso:",
    socket.id);
});
```

Figure 4.10: Disconnessione

## 4.6 Considerazioni finali

Il backend funge da hub centrale per la comunicazione tra utenti e per il coordinamento delle attività di gioco. Pur non gestendo direttamente lo streaming video, la sua funzione di signaling e orchestrazione è essenziale per il corretto funzionamento del sistema.

## Chapter 5

# Tecnologia WebRTC

WebRTC (Web Real-Time Communication) è una tecnologia open-source che consente la comunicazione audio, video e di dati in tempo reale tra browser, senza la necessità di plugin o software di terze parti. È pensata per creare connessioni peer-to-peer sicure e ad alte prestazioni.

Nel presente progetto, WebRTC viene utilizzata per stabilire chiamate video tra due utenti, coordinata da un server di signaling realizzato con Socket.IO.

### 5.1 Componenti WebRTC

Una connessione WebRTC completa coinvolge tre componenti principali:

- **MediaStream**: il flusso audio/video catturato dal browser con



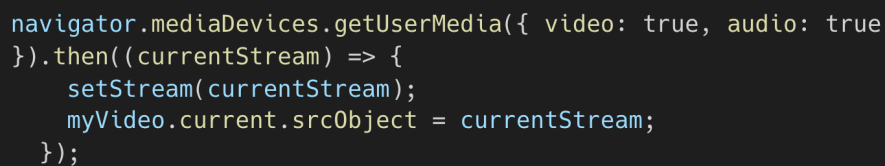
`getUserMedia`. Viene ottenuto lato client con `getUserMedia`.

- **RTCPeerConnection**: la connessione tra due peer, responsabile dello scambio di dati.
- **Signaling**: il canale esterno (non definito da WebRTC) usato per scambiare informazioni di connessione tra peer. E' realizzato tramite eventi `socket.io`.

## 5.2 Flusso di connessione WebRTC nel progetto

### 1. Ottenimento del flusso locale

Il primo passo è catturare il flusso audio/video dell'utente:



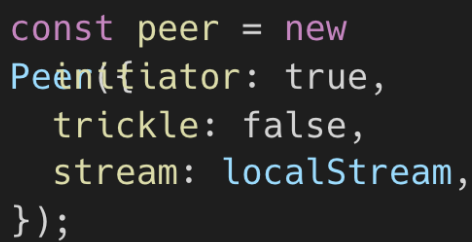
```
navigator.mediaDevices.getUserMedia({ video: true, audio: true
}).then((currentStream) => {
    setStream(currentStream);
    myVideo.current.srcObject = currentStream;
});
```

Figure 5.1: Acquisizione flusso locale

## 2. Inizio della chiamata (peer "chiamante")

Quando l'utente A vuole avviare la chiamata, viene creato un oggetto


Peer con il flag `initiator: true`:

A code snippet displayed in a dark-themed editor window with three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in a syntax-highlighted style: 

```
const peer = new
Peer({initiator: true,
      trickle: false,
      stream: localStream,
});
```

Figure 5.2: Inizio chiamata primo peer

Alla creazione, il peer genera un `signal` (descrizione SDP e ICE candidates), che viene inviato via socket:



```
peer.on("signal", (data) =>
{ socket.emit("callUser", {
  userToCall: idToCall,
  signalData: data,
  from: me,
});
});
```

Figure 5.3: Signaling

### 3. Ricezione della chiamata (peer "ricevente")

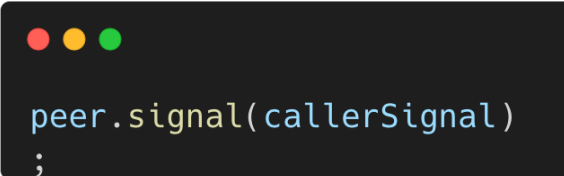
L'utente B riceve l'evento e crea a sua volta un oggetto Peer:

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript code snippet for creating a new Peer object.

```
const peer = new  
Peer({  
  initiator: false,  
  trickle: false,  
  stream: localStream,  
});
```

Figure 5.4: Ricezione chiamata secondo peer

Dopo aver ricevuto il segnale di A, lo passa al peer locale:

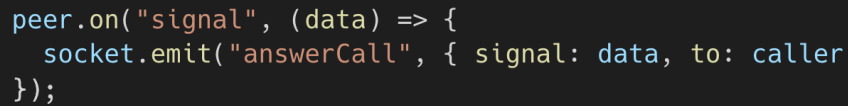
A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript code snippet for signaling a peer.

```
peer.signal(callerSignal)  
;
```

Figure 5.5: Signaling

## 4. Accettazione della chiamata

Quando B è pronto, genera a sua volta un `signal` che viene ritrasmesso ad A:

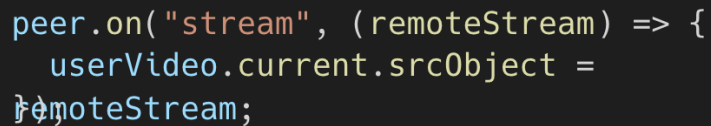
A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript code snippet for handling a 'signal' event on a 'peer' object.

```
peer.on("signal", (data) => {  
  socket.emit("answerCall", { signal: data, to: caller  
});
```

Figure 5.6: Accettazione chiamata

## 5. Streaming remoto

Quando la connessione è stabilita, ogni peer riceve il flusso dell'altro:

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript code snippet for handling a 'stream' event on a 'peer' object.

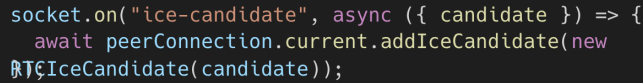
```
peer.on("stream", (remoteStream) => {  
  userVideo.current.srcObject =  
  remoteStream;
```

Figure 5.7: Acquisizione stream remoto

### 5.3 Gestione ICE Candidate

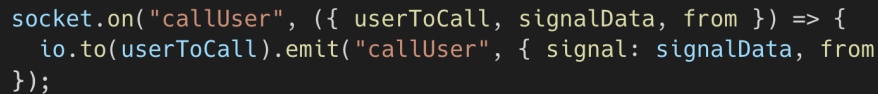
### 5.4 Ruolo del Signaling Server

Il server backend si occupa di smistare i messaggi di signaling tra i peer:

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript code snippet for handling ICE candidates.

```
socket.on("ice-candidate", async ({ candidate }) => {  
  await peerConnection.current.addIceCandidate(new  
  RTCIceCandidate(candidate));  
});
```

Figure 5.8: Gestione degli ICE Candidates

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript code snippet for signaling.

```
socket.on("callUser", ({ userToCall, signalData, from }) => {  
  io.to(userToCall).emit("callUser", { signal: signalData, from  
  });  
});
```

Figure 5.9: Signaling

## 5.5 Sicurezza e scalabilità

WebRTC fornisce crittografia end-to-end nativa (DTLS + SRTP). Tuttavia, l'uso di un signaling server non protetto potrebbe esporre a vulnerabilità. Per migliorare la sicurezza:

- È consigliato utilizzare HTTPS/WSS in produzione.
- È possibile integrare autenticazione e autorizzazione nel signaling.

## 5.6 Conclusioni

L'implementazione WebRTC nel progetto risulta ben strutturata e conforme ai principi fondamentali di questa tecnologia. La comuni-

cazione peer-to-peer risulta stabile e reattiva, rendendo l'esperienza utente fluida anche durante l'esecuzione dei minigiochi.

## Chapter 6

# Conclusioni Generali

Il progetto presentato dimostra come sia possibile, utilizzando tecnologie moderne come **WebRTC**, **React** e **Socket.IO**, realizzare un'applicazione web interattiva e completa, capace di offrire comunicazioni audio-video in tempo reale tra utenti, arricchite da elementi ludici come i minigiochi integrati.

Durante lo sviluppo sono stati affrontati vari aspetti tecnici rilevanti:

- La **gestione delle connessioni peer-to-peer**, con una particolare attenzione alla sincronizzazione tra utenti e alla fluidità dei flussi video.
- L'uso di **React** ha facilitato l'organizzazione del frontend, permettendo una gestione reattiva dell'interfaccia utente e una struttura modulare.



- Il backend, seppur leggero, ha svolto un ruolo essenziale nel *signaling WebRTC* e nella gestione degli eventi di gioco, mantenendo la comunicazione tra peer stabile e performante.

L'integrazione dei minigiochi rappresenta un valore aggiunto, rendendo l'esperienza più interattiva e coinvolgente. Questo ha richiesto una buona coordinazione tra frontend e backend, nonché una gestione precisa degli stati utente e delle interazioni real-time.