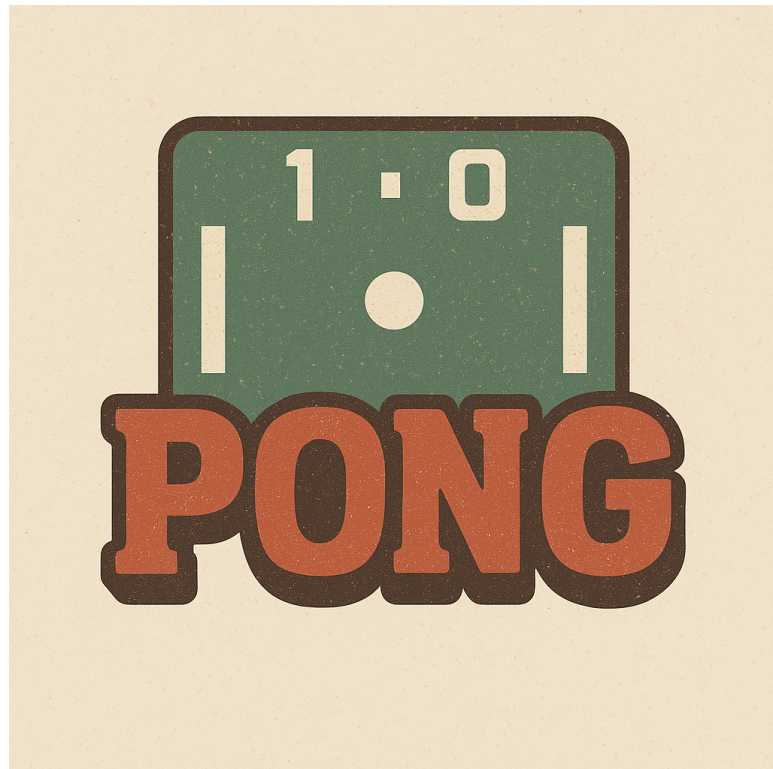


Elaborato di WebRTC

WebRTC_PONG



Giovanni Liguori

Simone Rinaldi

April, 2025

INDICE

INDICE	
CHAPTER 1 Introduzione	1
1.1 Breve descrizione del progetto	1
1.2 Obiettivo del progetto	1
CHAPTER 2 Tecnologie Utilizzate	2
2.1 Node.js	2
2.2 Express	2
2.3 Socket.io	3
2.4 WebRTC	3
2.5 WebAssembly (C++ e compilazione Emscripten)	3
2.6 HTML e CSS	4
CHAPTER 3 WebAssembly e Integrazione del Codice C++	5
3.1 Struttura del Codice C++	5
3.2 Compilazione in WebAssembly	5
3.3 Interfaccia tra JavaScript e WASM	6
CHAPTER 4 Architettura WebRTC: Connessione, Dati, Video	9
4.1 Instaurazione della Connessione	9
4.2 DataChannel: Scambio di Input di Gioco	12
4.3 Streaming Video: Webcam tramite WebRTC	16
CHAPTER 5 Installazione ed Esecuzione	19
5.1 Avvio del Server di Signaling	19
5.2 Deploy tramite Docker	20
5.3 Deploy su un Dominio pubblico	21

CHAPTER 1

Introduzione

WebRTC-Pong è un gioco multiplayer in tempo reale realizzato con WebAssembly, WebRTC per la comunicazione audio/video e data channel, e Socket.io per la gestione delle stanze e dei messaggi tra server e client. Il progetto mira a dimostrare come integrare componenti di streaming video e sincronizzazione del gioco in uno scenario multiutente, con logica di gioco sviluppata in C++ e compilata in WebAssembly.

1.1 Breve descrizione del progetto

L'applicazione offre una versione multiplayer del classico gioco "Pong", dove due giocatori si sfidano in tempo reale su una piattaforma web. Entrambi i giocatori possono vedersi tramite la webcam (streaming video bidirezionale) e interagire con la palla e le racchette attraverso un data channel WebRTC, rendendo l'esperienza dinamica e immediata. Tutta la logica di interazione client-server si basa su Socket.io, mentre la fisica del gioco è realizzata in C++ e compilata in WebAssembly.

1.2 Obiettivo del progetto

Lo scopo principale è mostrare come combinare diverse tecnologie moderne per creare un'esperienza di videogioco multiplayer interattivo direttamente all'interno del browser, senza bisogno di plugin aggiuntivi. In particolare:

- Dimostrare l'uso di WebRTC per la trasmissione in tempo reale di audio e video fra due o più browser.
- Integrare un data channel WebRTC per sincronizzare lo stato del gioco (posizione della palla, movimenti delle racchette, punteggi ecc.) con latenza estremamente bassa.
- Implementare un server Node.js con Socket.io per gestire la creazione e la join di stanze, nonché lo scambio di messaggi di segnalazione (signaling) necessari a instaurare la connessione WebRTC.
- Mostrare come un modulo C++ compilato in WebAssembly (con funzioni esposte a JavaScript) possa gestire la logica di gioco in modo performante e rendere l'esperienza fluida.

CHAPTER 2

Tecnologie Utilizzate

2.1 Node.js

Node.js è un *runtime* JavaScript basato sul motore V8 di Google Chrome, progettato per eseguire il codice JS al di fuori del browser. Le sue principali caratteristiche includono:

- *Event-driven architecture*: la gestione delle connessioni avviene in modo asincrono, ideale per applicazioni in tempo reale (come giochi multiplayer).
- *Non-blocking I/O*: le operazioni di input/output non bloccano il thread principale, consentendo a Node.js di gestire simultaneamente migliaia di connessioni.
- *Package manager (npm)*: la ricca collezione di pacchetti, tra cui `express` e `socket.io`, semplifica l'integrazione di funzionalità complesse.

In questo progetto, Node.js costituisce il server che:

1. Esegue il `server.js`, il quale crea un server HTTP e stabilisce la comunicazione tramite `Socket.io`.
2. Serve i file statici (HTML, CSS, JavaScript compilato in WebAssembly, ecc.) e si occupa di gestire le richieste dei client.

2.2 Express

Express è un framework Node.js minimalista e flessibile per creare server web. Offre:

- Un sistema di routing per definire le risposte alle varie route e gestire eventuali middleware.
- La capacità di servire file statici in maniera semplice, come avviene nel progetto tramite `app.use(express.static(...))`.
- Un'architettura modulare che permette di integrare facilmente `Socket.io` e altri servizi.

2.3 Socket.io

Socket.io è una libreria JavaScript progettata per rendere più semplice e affidabile la comunicazione in tempo reale tra client e server. Si basa principalmente sul protocollo WebSocket (o fallback quando non disponibile). Nel contesto di un videogioco multiplayer:

- Gestisce il signaling per WebRTC: ovvero, inoltra offerte, risposte e candidati *ICE* tra i due peer che devono comunicare.
- Consente la creazione di *stanze* o canali di comunicazione isolati, in cui i giocatori di una singola partita possono interagire senza interferire con altri utenti.
- Permette di inviare notifiche e aggiornamenti di stato (ad esempio, l'avvio della partita, l'ingresso di un nuovo giocatore, la partenza di un giocatore, ecc.) in modo immediato e sincrono con la logica di gioco.

2.4 WebRTC

WebRTC (Web Real-Time Communication) è un insieme di API che fornisce funzionalità di streaming audio e video (e trasmissione dati) tra browser, senza l'uso di plugin. Le sue componenti chiave includono:

- *RTCPeerConnection*: l'oggetto centrale che stabilisce la connessione P2P tra due client, gestendo la configurazione dei canali audio/video e la negoziazione con i server STUN/TURN.
- *MediaStream*: rappresenta le tracce video e audio. Ogni client cattura il proprio flusso (ad es. webcam), lo invia all'altro peer, e al contempo riceve il flusso remoto da mostrare nella pagina.
- *RTCDataChannel*: canale di dati a bassa latenza che permette di scambiare messaggi arbitrari (come lo stato del gioco) tra i peer, ideale per applicazioni di gaming in tempo reale.
- *Signaling*: i peer condividono le *offer*, le *answer* e i candidati *ICE* utilizzando un meccanismo esterno (in questo caso, *Socket.io*).

2.5 WebAssembly (C++ e compilazione Emscripten)

WebAssembly (Wasm) è un formato binario eseguibile dai browser moderni, che offre performance molto vicine al codice nativo. Nel progetto, si utilizza:

- **C++** per la logica di gioco (Pong): calcolo della fisica, gestione delle collisioni e dei punteggi, controllo della tastiera. In questo modo, operazioni matematiche e cicli di aggiornamento sono eseguiti con buone prestazioni.
- **Emscripten** come toolchain per compilare il codice sorgente in C++ in un modulo WebAssembly. Emscripten crea anche un “*glue code*” JavaScript che funge da interfaccia tra il modulo Wasm e il browser.
- **Interfacce JavaScript** per invocare le funzioni C++ dalla pagina web. Per esempio, funzioni come `start_game()` iniziano il ciclo principale del gioco, e `update_remote_state()` aggiorna le variabili condivise (posizione della pallina e delle racchette) sulla base dei dati ricevuti tramite il *DataChannel* WebRTC.

2.6 HTML e CSS

HTML (HyperText Markup Language) è il linguaggio di marcatura standard per la creazione di pagine web. Il suo ruolo principale è quello di definire la struttura del contenuto, specificando quali elementi appaiono nella pagina (*canvas* di gioco, pulsanti di navigazione, contenitori video, ecc.). In particolare, per il presente progetto:

- Viene gestito un layout a più sezioni: una schermata iniziale con i pulsanti per creare o unirsi a una stanza, una sezione *lobby* per mostrare l’elenco dei giocatori e il pulsante “Pronto”, e una sezione di gioco (*game screen*) contenente il *canvas* e due riquadri video (locale e remoto).
- Le aree interattive (pulsanti, liste delle stanze, informazioni di stato) sono definite tramite opportuni tag HTML, facilitando la scrittura di codice JavaScript che aggiunge, rimuove o aggiorna tali componenti.

CSS fornisce gli stili e la formattazione visiva delle pagine HTML. È stato utilizzato per stabilire il posizionamento e il dimensionamento di elementi come il *canvas* di gioco e i riquadri video della webcam, impostare colori di sfondo, bordi, spaziature e font per rendere l’interfaccia più gradevole e intuitiva e creare layout reattivi (responsive).

Grazie a questa combinazione di tecnologie, l’applicazione può gestire in modo efficace e sincronizzato il flusso audio/video, il loop di gioco, i dati di input e l’aggiornamento dei punteggi in tempo reale direttamente all’interno del browser.

CHAPTER 3

WebAssembly e Integrazione del Codice C++

3.1 Struttura del Codice C++

Il file `pong.cpp` rappresenta il cuore della logica di gioco di Pong: qui troviamo il modello fisico (movimento della pallina, collisioni, punteggi) e la gestione della tastiera. All'inizio del file vengono definiti:

- **Costanti dimensionali** (`WIDTH`, `HEIGHT`, `PADDLE_WIDTH`, `BALL_SIZE`, ...) per specificare misure e posizioni iniziali di pallina e racchette.
- **Variabili di stato** (`ballX`, `ballY`, `leftPaddleY`, `rightPaddleY`, `scoreLeft`, `scoreRight`, ...) utili a tracciare il progresso della partita.
- **Funzioni EM_JS** come `clearCanvas`, `drawRect` e `drawText`, che consentono di eseguire codice JavaScript (per il rendering sul *canvas* HTML) direttamente all'interno del sorgente C++.

Lo scopo di `pong.cpp` è quello di gestire il *game loop* (calcolo fisico, collisioni, punteggi) e di esporre funzioni *export* a JavaScript (tramite Emscripten) per avviare e interrompere l'aggiornamento continuo.

3.2 Compilazione in WebAssembly

Per trasformare il codice C++ in un modulo WebAssembly (`.wasm`), si utilizza **Emscripten**, un toolkit che include il compilatore `emcc` e un set di *API* di interoperabilità.

- **Struttura di base:** `emcc` compila `pong.cpp` in WebAssembly generando sia un file `.wasm` sia un `.js` (detto *glue code*) contenente funzioni e wrapper.
- **Struttura del comando di compilazione utilizzato:**

```

1  emcc pong.cpp
2  -O2
3  -s WASM=1
4  -s "EXPORTED_FUNCTIONS=['_start_game','_set_role','_update_remote_state','_set_remote_paddle','_get_remote_paddle']"
5  -s "EXPORTED_RUNTIME_METHODS=['ccall','cwrap']"
6  -o pong.js
7

```

L'opzione `-o` specifica l'output, `-s WASM=1` abilita la compilazione WebAssembly, mentre `-s EXPORTED_FUNCTIONS` elenca le funzioni C++ da esportare in JavaScript (ad esempio `_start_game`, `_set_role`, ecc.).

È importante notare come la corretta dichiarazione delle funzioni da esportare consenta al codice JavaScript di *chiamarle* in fase di esecuzione (vedi sezione successiva sull'interfaccia *JS-WASM*).

3.3 Interfaccia tra JavaScript e WASM

L'integrazione tra WebAssembly e JavaScript avviene principalmente in due modalità:

1. **Chiamate dal C++ verso JavaScript:** si utilizzano macro come `EM_JS` o `EM_ASM` (riferite in `pong.cpp`) per eseguire codice JS quando serve, ad esempio per il rendering sul `<canvas>` o per inviare messaggi di gioco via *DataChannel* WebRTC. Un esempio è visibile è il seguente:

```

1  EM_JS(void, drawText, (const char* text, int x, int y), {
2      const canvas = document.getElementById('pongCanvas');
3      if (!canvas) return;
4      const ctx = canvas.getContext('2d');
5      ctx.font = '30px Arial';
6      ctx.fillStyle = 'black';
7      ctx.fillText(UTF8ToString(text), x, y);
8  });

```

2. **Chiamate da JavaScript verso C++:** grazie a Emscripten, le funzioni contrassegnate come `EXPORTED_FUNCTIONS` possono essere richiamate dall'ambiente

JavaScript. Nel progetto, `start_game()`, `update_remote_state()` e altre procedure *host/client* vengono invocate tramite funzioni come:

```
Module.ccall('start_game', null, [], []);
```

oppure con la sintassi `cwrap`, se si desidera un wrapper più comodo. In questo modo, l'applicazione JavaScript (ad esempio nel file `webrtc.js` o `main.js`) avvia il *ciclo di gioco* o aggiorna parametri come la posizione delle racchette.



Grazie a questi meccanismi, la logica C++ (fisica di gioco) e il frontend JavaScript (interfaccia, WebRTC) collaborano in maniera integrata.

Rendering e Ciclo di Gioco nel Canvas

Il ciclo di gioco è fondato su due diverse modalità:

- **Host (isHost = true):** gestisce il vero e proprio *game loop* con `emscripten.set_main_loop(0, 1)`, calcola il movimento della pallina, verifica le collisioni e aggiorna i punteggi. Ogni *frame*, l'host effettua chiamate a:
 - `clearCanvas()` e `drawRect()` per disegnare la pallina e le racchette.
 - `drawText()` per i punteggi.
 - `sendGameState(...)` per inviare lo stato corrente via *DataChannel* al giocatore client.
- **Client (isHost = false):** non calcola la fisica di gioco, ma riceve dallo *host* i valori di posizione e punteggio e li passa a `update_remote_state()` (che sincronizza le variabili C++). Il suo *main loop* (registrato con `emscripten.set_main_loop(0, 1)`) serve a rilevare gli input locali (tasti su/giù) e inviarli all'host tramite funzioni JavaScript come `sendPaddleInput()`.

Nel caso `pong.cpp`, ogni disegno sul *canvas* HTML avviene chiamando le macro `EM_JS` (`clearCanvas`, `drawRect`, `drawText`), mentre la logica di collisione (controllo se la pallina urta il margine o la racchetta) è puramente C++. Quando un giocatore raggiunge il *MAX_SCORE*, l'host invia un segnale di *game over* sia in locale sia al client, e la funzione JavaScript `showWinner()` gestisce l'overlay di vittoria.

In conclusione, il sistema risulta **altamente modulare**: la fisica del gioco e il rendering principale “*di base*” sono in C++/WebAssembly, mentre tutto il networking, la parte multimediale (webcam), l'interfaccia grafica e il coordinamento delle stanze di gioco avvengono lato JavaScript tramite *Socket.io* e *WebRTC*.

CHAPTER 4

Architettura WebRTC: Connessione, Dati, Video

- Come viene instaurata la connessione:
 - Creazione della `PeerConnection`
 - Scambio di SDP via server signalling (`WebSocket` + `Node.js`)
 - ICE Candidate e completamento del setup
- `DataChannel`:
 - Chi lo crea (host)
 - Cosa viene inviato (input remoto)
 - Chi aggiorna lo stato di gioco (solo host → logica centralizzata)
- Streaming Video:
 - Acquisizione `MediaStream` locale
 - Invio tramite WebRTC ai peer
 - Rendering nel DOM

4.1 Instaurazione della Connessione

L'instaurazione della connessione tra due peer avviene attraverso l'uso dell'oggetto `RTCPeerConnection`. La logica di negoziazione è implementata in `webrtc.js`, con il supporto di un server di signalling via `WebSocket` gestito tramite `server.js` (`Node.js`).

- **Creazione della `PeerConnection`:** ogni client crea un'istanza di `RTCPeerConnection`, configurata con i server STUN per la risoluzione dei peer attraverso NAT.



```
1  const config = {  
2    iceServers: [{urls: 'stun:stun.l.google.com:19302'}]  
3  };  
4  
5  ...  
6  
7  this.peerConnection = new RTCPeerConnection(config);
```

- **Creazione e invio dell'offerta SDP (host):** il peer che crea la stanza (host) genera un'offerta SDP e la invia al server di signalling.



```
1  const offer = await this.peerConnection.createOffer();  
2    await this.peerConnection.setLocalDescription(offer);  
3    this.socket.emit('webrtc_offer', offer, this.gameState.roomId);
```

- **Ricezione dell'offerta e invio della risposta (client):** il peer ricevente imposta la descrizione remota, genera una risposta (SDP answer) e la invia al server.



```
1  async handleOffer(offer) {  
2    await this.createPeerConnection();  
3    await this.peerConnection.setRemoteDescription(  
4      new RTCSessionDescription(offer));  
5    const answer = await this.peerConnection.createAnswer();  
6    await this.peerConnection.setLocalDescription(answer);  
7    this.socket.emit('webrtc_answer', answer, this.gameState.roomId);  
8  }
```

- **Ricezione della risposta SDP (host):** l'host riceve la risposta e completa la configurazione impostandola come descrizione remota.

```

1  async handleAnswer(answer) {
2    await this.peerConnection.setRemoteDescription(
3      new RTCSessionDescription(answer));
4  }
5  async addIceCandidate(candidate) {
6    try {
7      await this.peerConnection.addIceCandidate(candidate);
8      addLog('Candidate ICE aggiunto.');
```

- **Scambio di ICE Candidates:** entrambi i peer, tramite l'evento `onicecandidate`, inviano i propri ICE candidates al server di signalling, che li inoltra all'altro peer. Ogni ICE ricevuto viene aggiunto alla rispettiva connessione.

```

1  this.peerConnection.onicecandidate = (event) => {
2    if (event.candidate) {
3      this.socket.emit(
4        'webrtc_ice_candidate', event.candidate, this.gameState.roomId);
5    }
6  };
```

```

1  async addIceCandidate(candidate) {
2    try {
3      await this.peerConnection.addIceCandidate(candidate);
4      addLog('Candidate ICE aggiunto.');
```



Figure 4.1: Diagramma di sequenza: processo di negoziazione WebRTC tramite signalling server.

4.2 DataChannel: Scambio di Input di Gioco

In questo progetto, il DataChannel WebRTC è utilizzato per trasmettere in tempo reale gli input di gioco dal client all'host. L'host è responsabile dell'intera logica del gioco e invia aggiornamenti allo stato tramite lo stesso canale.

- **Creazione (host):** quando il peer assume il ruolo di host, crea esplicitamente il DataChannel tramite `createDataChannel` e invia l'offerta SDP.

```

1  if (this.gameState.role === 'host') {
2    this.dataChannel = this.peerConnection.createDataChannel('game');
3    this.setupDataChannel();
4    const offer = await this.peerConnection.createOffer();
5    await this.peerConnection.setLocalDescription(offer);
6    this.socket.emit('webrtc_offer', offer, this.gameState.roomId);
7  }

```

- **Ricezione (client):** il client riceve automaticamente il DataChannel tramite l'evento `ondatachannel`.

```

1  this.peerConnection.ondatachannel = (event) => {
2    this.dataChannel = event.channel;
3    this.setupDataChannel();
4  };

```

- **Invio dell'input (client → host):** il client, attraverso il C++, rileva gli input da tastiera e li invia al JS tramite `EM_ASM`. Il JS usa `dataChannel.send` per inoltrare il comando all'host.

```

1 void update_client_input() {
2     if (clientUpPressed) {
3         EM_ASM({
4             if (typeof sendPaddleInput === 'function') {
5                 sendPaddleInput('up');
6             }
7         });
8     }
9     if (clientDownPressed) {
10        EM_ASM({
11            if (typeof sendPaddleInput === 'function') {
12                sendPaddleInput('down');
13            }
14        });
15    }
16 }

```

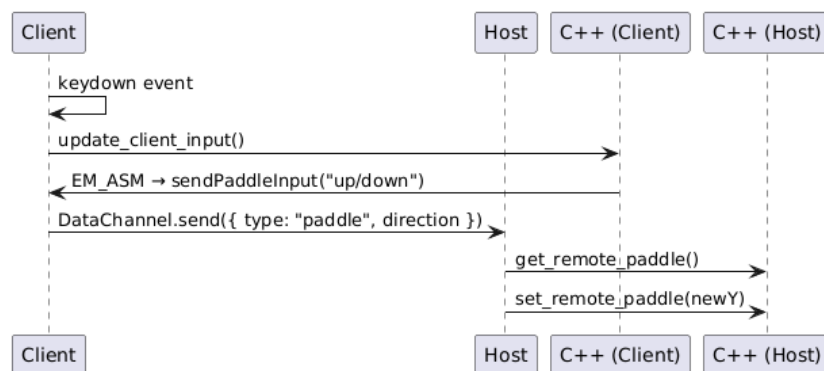


Figure 4.2: Diagramma di sequenza: invio degli input da parte del client e gestione da parte dell'host.

- **Gestione dell'input (host):** l'host riceve il comando (es. "up" o "down") via DataChannel, lo interpreta, e aggiorna la posizione della racchetta remota.


```

1  this.dataChannel.onmessage = (event) => {
2    addLog('Messaggio ricevuto: ' + event.data);
3    try {
4      const data = JSON.parse(event.data);
5      if (data.type === 'gameState') {
6        Module.ccall(
7          'update_remote_state', null,
8          ['number', 'number', 'number', 'number', 'number', 'number'], [
9            data.ballX, data.ballY, data.leftPaddleY, data.rightPaddleY,
10           data.scoreLeft, data.scoreRight
11         ]);
12      } else if (data.type === 'paddle') {
13        const current = Module.ccall('get_remote_paddle', 'number', [], []);
14        let newY = current;
15        const PADDLE_SPEED = 7;
16
17        if (data.direction === 'up') {
18          newY = current - PADDLE_SPEED;
19        } else if (data.direction === 'down') {
20          newY = current + PADDLE_SPEED;
21        }
22
23        if (newY < 0) newY = 0;
24        if (newY + 100 > 600) newY = 600 - 100;
25        Module.ccall('set_remote_paddle', null, ['number'], [newY]);
26      }
27    };

```

- **Invio dello stato di gioco (host → client):** ad ogni frame, l'host invia lo stato corrente (posizione pallina, racchette, punteggio) al client tramite `sendGameState()`.

```

1  EM_ASM({
2    if (typeof sendGameState === 'function') {
3      sendGameState($0, $1, $2, $3, $4, $5);
4    }
5  }, ballX, ballY, leftPaddleY, rightPaddleY, scoreLeft, scoreRight);

```

- **Aggiornamento grafico (client):** il client riceve i dati e li inoltra al modulo WebAssembly tramite `Module.ccall`, che aggiorna la visualizzazione sul canvas.

```

1  Module.ccall(
2    'update_remote_state', null,
3    ['number', 'number', 'number', 'number', 'number', 'number'], [
4      data.ballX, data.ballY, data.leftPaddleY, data.rightPaddleY,
5      data.scoreLeft, data.scoreRight
6    ]);

```

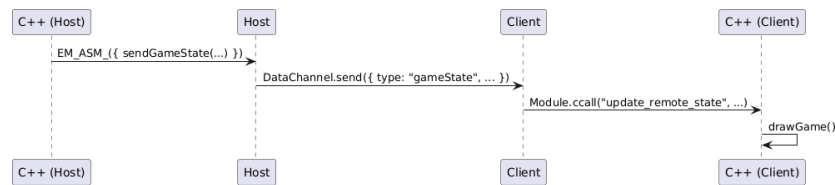


Figure 4.3: Diagramma di sequenza: aggiornamento dello stato di gioco dall'host al client.

4.3 Streaming Video: Webcam tramite WebRTC

Oltre alla logica di gioco e allo scambio dati tramite DataChannel, il progetto implementa lo streaming video tra i due peer. Questo consente ai giocatori di vedersi in tempo reale, rendendo l'esperienza più immersiva e interattiva.

Accesso e cattura della webcam

Ogni peer, al momento della creazione della connessione, richiede l'accesso alla propria webcam e microfono tramite l'API `getUserMedia()` di WebRTC. Il flusso viene subito associato all'elemento video locale.

```

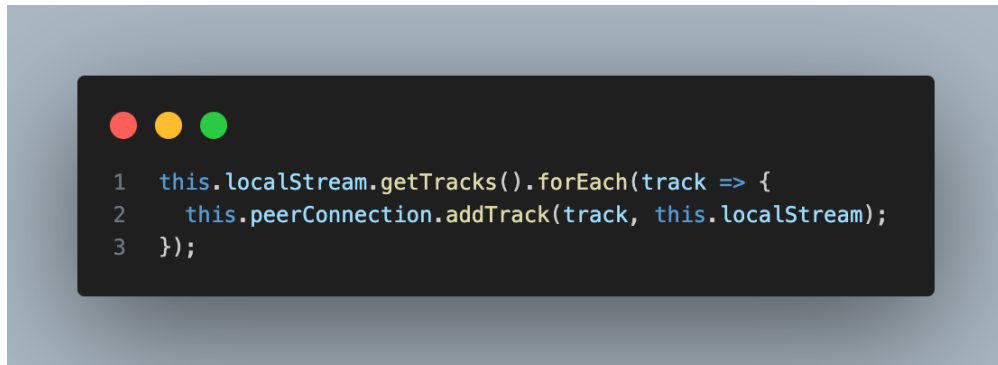
1  this.localStream =
2    await navigator.mediaDevices.getUserMedia({video: true, audio: true});
3  const localVideo = document.getElementById('localVideo');
4  localVideo.srcObject = this.localStream;

```

Figure 4.4: Richiesta accesso alla webcam e visualizzazione locale.

Invio dello stream all'altro peer

Una volta acquisito, lo stream locale viene inviato tramite WebRTC al peer remoto. Questo avviene aggiungendo le tracce della webcam alla `RTCPeerConnection` tramite il metodo `addTrack()`.

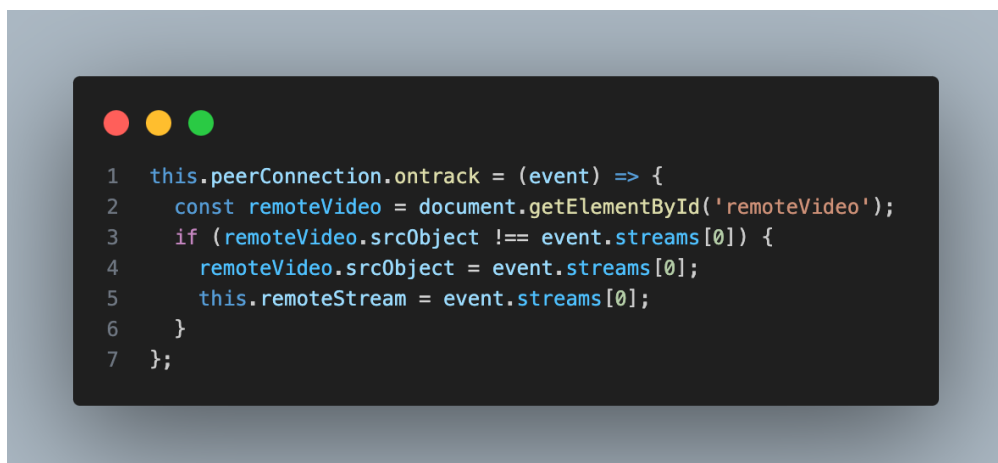
A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains three lines of JavaScript code:

```
1 this.localStream.getTracks().forEach(track => {  
2   this.peerConnection.addTrack(track, this.localStream);  
3 });
```

Figure 4.5: Aggiunta delle tracce media alla PeerConnection.

Ricezione e visualizzazione dello stream remoto

Il peer che riceve la traccia remota imposta un handler per l'evento `ontrack`, che aggiorna dinamicamente l'elemento video dedicato.

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

```
1 this.peerConnection.ontrack = (event) => {  
2   const remoteVideo = document.getElementById('remoteVideo');  
3   if (remoteVideo.srcObject !== event.streams[0]) {  
4     remoteVideo.srcObject = event.streams[0];  
5     this.remoteStream = event.streams[0];  
6   }  
7 };
```

Figure 4.6: Gestione della traccia video remota.

Diagramma di sequenza dello streaming video

La seguente figura illustra il processo completo di streaming: acquisizione, invio tramite WebRTC e rendering nel DOM remoto.

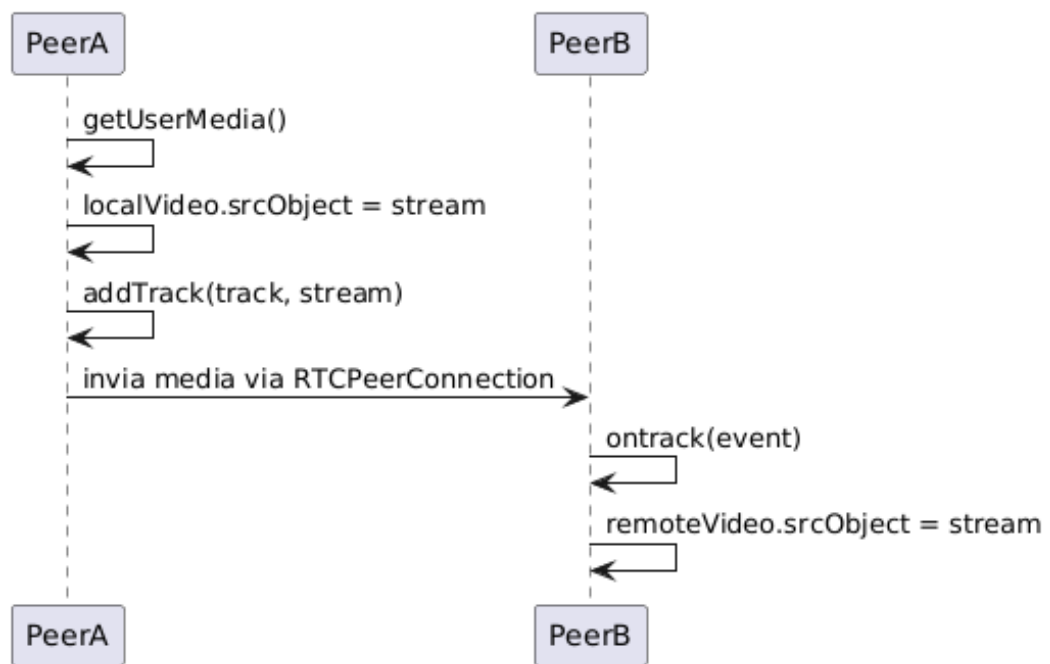


Figure 4.7: Diagramma di sequenza: streaming webcam tra peer.

CHAPTER 5

Installazione ed Esecuzione

In questa sezione concludiamo la documentazione descrivendo il processo di **compilazione** del codice C++ in WebAssembly, l'**avvio del server di signaling** e infine la **procedura di deploy**. Vengono anche illustrati i file Docker utilizzati per semplificare il rilascio in ambienti di produzione.

Compilazione del C++ in WebAssembly

La prima fase consiste nella *build* del codice C++ (`pong.cpp`) tramite il **toolchain Emscripten**. Ricordiamo brevemente i passaggi chiave:

1. Assicurarsi di avere Emscripten installato, con il comando:

```
emcc --version
```

2. Posizionarsi nella directory contenente `pong.cpp`.
3. Eseguire il comando di compilazione:

```
emcc pong.cpp -o pong.js -s WASM=1 \
-s EXPORTED_FUNCTIONS='["_start_game", "_set_role", ...]' \
-s EXPORTED_RUNTIME_METHODS='["cwrap", "ccall"]'
```

4. Al termine, si otterranno due file: `pong.js` (glue code) e `pong.wasm`, i quali vengono poi importati e utilizzati dal codice JavaScript dell'applicazione.

A questo punto, la logica di gioco scritta in C++ è disponibile in forma *WebAssembly*, pronta per essere *richiamata* dalle funzioni JavaScript che si occupano del networking e del rendering.

5.1 Avvio del Server di Signaling

Il cuore dell'architettura in tempo reale è il *server di signaling*, che coordina i due (o più) *peer* nella fase di negoziazione WebRTC. Per avviare il server in locale:

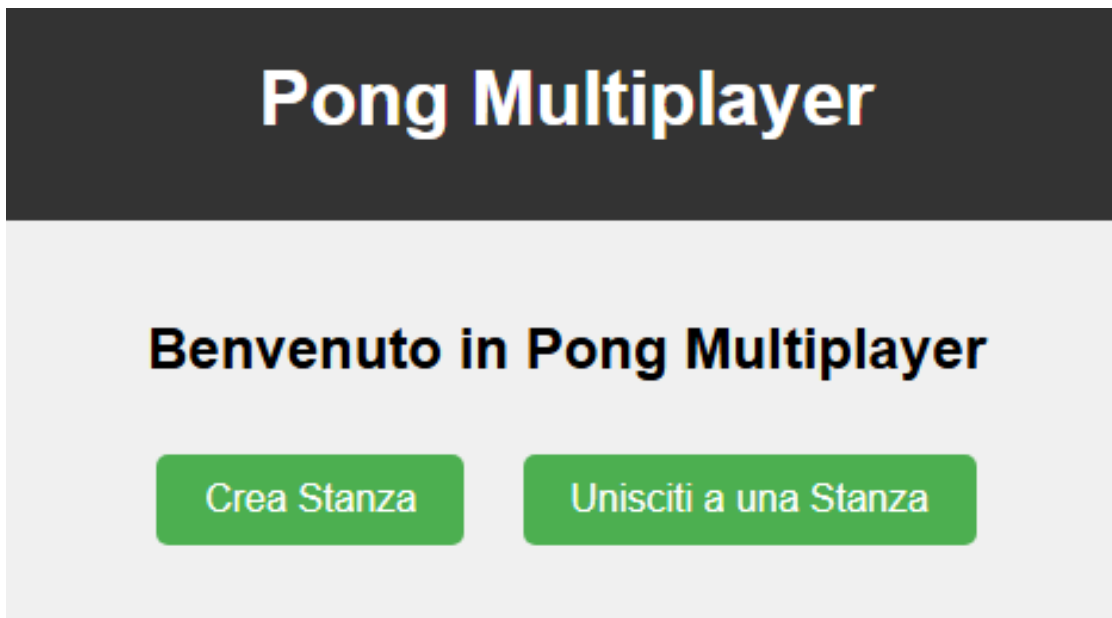
1. Installare le dipendenze (quali `express` e `socket.io`):

```
npm install
```

2. Avviare il server:

```
node server.js
```

3. Collegarsi all'indirizzo `http://localhost:3000` (o la porta configurata) per verificare che la pagina principale sia raggiungibile.



5.2 Deploy tramite Docker

Per semplificare il processo di distribuzione (deployment) in ambiente di produzione, è stato creato un **Dockerfile** e un **docker-compose.yml**. Tale approccio consente di racchiudere l'intera applicazione in un container eseguibile ovunque sia disponibile Docker. È sufficiente utilizzare il seguente comando:

```
docker-compose up -d --build
```

per eseguire il container in *detached mode*. L'applicazione risulterà così raggiungibile sulla porta 3000 della macchina host.

5.3 Deploy su un Dominio pubblico

Per rendere l'applicazione **Pong WebRTC** accessibile via Internet in maniera sicura (con HTTPS) e far funzionare correttamente le componenti webcam e audio, si è ricorsi a:

- Un **dominio pubblico** (`https://simone-rinaldi.me`).
- La creazione di un **tunnel Cloudflare** per instradare il traffico verso il server Node.js in esecuzione.
- L'installazione e la configurazione di **certificati SSL/TLS**, in modo che il server potesse rispondere via HTTPS.

Questo perchè i principali browser moderni (Chrome, Firefox, Edge) impongono che le funzionalità `navigator.mediaDevices.getUserMedia` (e dunque webcam, microfono) siano disponibili **solo** se:

- L'origine della pagina è in `https://` (ovvero un contesto sicuro).
- Oppure la pagina è `http://localhost` (caso speciale, consentito per lo sviluppo locale).

Cloudflare offre la possibilità di creare un **tunnel** che collega la propria infrastruttura (o container Docker, o server locale) con la rete Cloudflare. Configurando opportunamente un *tunnel* si può stabilire una connessione sicura tra il container che esegue Node.js e la piattaforma Cloudflare.

Il traffico in ingresso al dominio (es. `simone-rinaldi.me`) attraversa i sistemi Cloudflare, che provvedono a reindirizzarlo verso l'*origin server* in esecuzione. Quando si desidera gestire *in prima persona* la terminazione SSL/TLS direttamente sul server Node.js, occorre disporre di:

- **Certificato pubblico** (file `.crt`).
- **Chiave privata** (file `.key`).

Questi file possono vanno poi inclusi all'interno del container Docker o montati come *volume* (ad es. in `/etc/ssl/cloudflare`). Sul codice Node.js, in particolare nel file `server.js`, si sostituisce la creazione del server da:

```
const server = http.createServer(app);
```

a qualcosa di simile (utilizzando il modulo `https`):

```
const fs = require('fs');
const https = require('https');

const options = {
  key: fs.readFileSync('/etc/ssl/cloudflare/example.key'),
  cert: fs.readFileSync('/etc/ssl/cloudflare/example.crt')
};

const server = https.createServer(options, app);
```

in modo che esso risponda direttamente in HTTPS sulla porta desiderata (di default, 443). In questo modo, chiunque visiti `https://simone-rinaldi.me` può:

- *Creare o unirsi* a una stanza di gioco.
- Avviare la *connessione WebRTC*, trasmettere webcam e comandi di gioco.
- Visualizzare eventuali *screen* di vittoria, punteggi e log in tempo reale.

Nella figura seguente si può vedere la schermata di gioco con la webcam di sinistra (*host*) e la webcam di destra (*client*), oltre al *canvas* centrale in cui la pallina e le racchette sono disegnate in tempo reale:

