



UNIVERSITÀ⁹ DEGLI STUDI DI
NAPOLI FEDERICO II

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

Elaborato per l'esame di Web and Real Time
Communication Systems

Uniegle

Anno Accademico 2024/2025

Docente
Simon Pietro Romano

Autori
Francesco Scognamiglio M63001364
Felice Micillo M63001377

Abstract

Il presente documento descrive lo sviluppo di un Web and Real-Time Communication System, un'applicazione web progettata per mettere in contatto utenti sconosciuti tramite videochiamate e chat in tempo reale. L'applicazione è stata sviluppata con un backend basato su Node.js e un frontend realizzato in React.js, due tecnologie moderne e scalabili ideali per lo sviluppo di applicazioni web dinamiche e interattive. Per la gestione della comunicazione in tempo reale, il sistema utilizza Socket.io, una libreria che permette la comunicazione bidirezionale tra client e server tramite WebSocket, garantendo una trasmissione efficiente e a bassa latenza. Inoltre, sfrutta le WebRTC (Web Real-Time Communication), una tecnologia che consente la trasmissione diretta di audio e video tra i peer, riducendo il carico sui server e migliorando la qualità della connessione.

L'esperienza utente è simile a quella di piattaforme come Omegle o Chatroulette: due utenti vengono connessi casualmente e, solo dopo l'accettazione reciproca di una richiesta, si avvia una connessione sicura per la comunicazione video, audio e testuale.

Indice

| | |
|--|-----------|
| Abstract | i |
| 1 Introduzione | 1 |
| 1.1 Obiettivo | 1 |
| 1.2 Tecnologie utilizzate | 2 |
| 2 WebRTC | 3 |
| 2.1 Standardizzazione | 4 |
| 2.2 Architettura | 5 |
| 2.3 Signalling | 6 |
| 2.3.1 Fasi del Signalling | 6 |
| 3 Architettura e Flussi | 8 |
| 3.1 Architettura | 8 |
| 3.2 Sequenza di Negoziazione | 10 |
| 3.3 Instant Messaging | 13 |
| 4 Implementazione | 17 |
| 4.1 Server con Node.js | 18 |
| 4.1.1 Librerie utilizzate | 19 |

| | | |
|----------|---|-----------|
| 4.1.2 | Descrizione del codice | 20 |
| 4.2 | Client con React.js | 28 |
| 4.2.1 | Librerie utilizzate | 28 |
| 4.2.2 | Descrizione del codice | 29 |
| 5 | Guida all'utilizzo | 38 |
| 5.1 | Configurazione e Avvio del Frontend | 38 |
| 5.2 | Configurazione e Avvio del Backend | 39 |
| 5.3 | Configurazione | 40 |
| 5.4 | Presentazione | 40 |
| 5.4.1 | Pagina di Preview | 41 |
| 5.4.2 | Pagina User Profile | 42 |
| 5.4.3 | Pagina Chat | 44 |

Capitolo 1

Introduzione

Nel panorama delle applicazioni web moderne, la comunicazione in tempo reale ha assunto un ruolo sempre più centrale, permettendo interazioni fluide e immediate tra utenti sparsi in tutto il mondo. Il nostro progetto, un Web and Real-Time Communication System, si inserisce in questo contesto offrendo una piattaforma che connette due utenti casuali per interazioni audio-video e chat testuale in tempo reale.

1.1 Obiettivo

L'applicazione è stata sviluppata con l'obiettivo di fornire un'esperienza di connessione immediata tra sconosciuti, simile a piattaforme come Omegle o Chatroulette, garantendo stabilità, qualità di trasmissione e un'interfaccia utente moderna e intuitiva. Gli utenti possono stabilire una comunicazione video e testuale solo dopo aver accettato recipro-

camente una richiesta di connessione, migliorando così il controllo e la sicurezza delle interazioni.

1.2 Tecnologie utilizzate

Il sistema è costruito sfruttando tecnologie avanzate per garantire un'esperienza fluida e reattiva:

- **Backend:** sviluppato con Node.js, un runtime JavaScript lato server che garantisce elevate prestazioni e scalabilità, ideale per applicazioni real-time.
- **Frontend:** realizzato con React.js, una libreria JavaScript per la costruzione di interfacce utente dinamiche e interattive.
- **Socket.io:** una libreria che gestisce la comunicazione in tempo reale tra client e server, permettendo lo scambio di messaggi, eventi e dati con bassa latenza attraverso WebSocket.
- **WebRTC (Web Real-Time Communication):** una tecnologia che consente la trasmissione diretta di audio e video tra gli utenti senza la necessità di passare per un server intermedio, migliorando l'efficienza e riducendo il carico sui server.

Capitolo 2

WebRTC

La tecnologia Web Real-Time Communication (WebRTC) è un sistema open-source creato per permettere la comunicazione in tempo reale direttamente nei browser, senza bisogno di installare software o plugin esterni. WebRTC è stato sviluppato per rendere più semplice l'integrazione di funzionalità audio, video e di scambio dati tra utenti connessi su una rete IP, consentendo così applicazioni come videoconferenze, chat vocali e streaming di contenuti interattivi in modo efficiente e sicuro. Un elemento chiave che ha influenzato lo sviluppo di WebRTC è la sicurezza. A differenza di molti protocolli del passato, WebRTC è stato concepito con meccanismi di protezione integrati, come la crittografia end-to-end e la gestione sicura delle connessioni peer-to-peer.

2.1 Standardizzazione

Prima dell'avvento di WebRTC, la comunicazione in tempo reale sui browser era limitata a soluzioni proprietarie come Google Hangout, Facebook+Skype e Java Applet, ognuna delle quali operava in un ecosistema chiuso e incompatibile con le altre piattaforme. Questo costringeva gli utenti a utilizzare software specifici o a installare estensioni per comunicare con altri utenti sulla stessa rete.

Per affrontare questa frammentazione, è stato avviato un processo di standardizzazione che ha coinvolto due enti principali:

- **IETF (Internet Engineering Task Force):** responsabile della definizione dei protocolli di comunicazione per WebRTC.
- **W3C (World Wide Web Consortium):** responsabile della definizione delle API JavaScript, che consentono agli sviluppatori di integrare WebRTC direttamente nei browser.

Grazie a questa collaborazione, WebRTC è oggi uno standard supportato dai principali browser moderni, tra cui Google Chrome, Mozilla Firefox, Safari e Microsoft Edge, ed è diventato un punto di riferimento per lo sviluppo di applicazioni di comunicazione in tempo reale.

2.2 Architettura

L'architettura WebRTC è composta da tre elementi principali:

- **End-point (Browser/Web App)**

Sono i dispositivi (browser) che partecipano alla comunicazione. Utilizzano le API WebRTC per acquisire media (audio/video), negoziare la connessione e trasmettere dati in tempo reale.

- **Application Provider (Server di Signaling)**

Serve a scambiare le informazioni necessarie per impostare correttamente la sessione multimediale tra i peer. Non è un protocollo standardizzato e deve essere implementato dallo sviluppatore. Può utilizzare WebSocket, HTTP, MQTT o altre tecnologie di trasporto per lo scambio di messaggi.

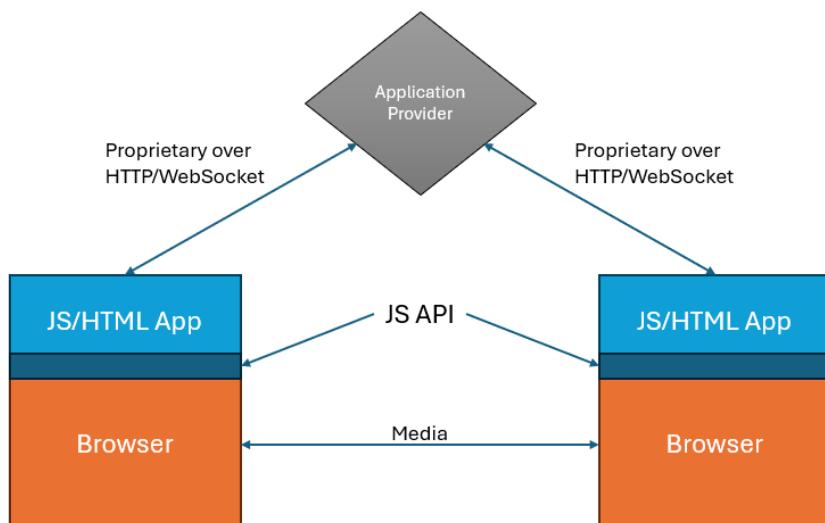


Figura 2.1: Architettura WebRTC

2.3 Signalling

Il signalling è il processo attraverso il quale vengono scambiate le informazioni necessarie per stabilire una connessione WebRTC tra due end-point. Poiché WebRTC non specifica un protocollo di signalling, è compito dello sviluppatore implementarlo in modo autonomo.

Le informazioni scambiate durante il signalling comprendono:

- **SDP (Session Description Protocol)**

Fornisce una descrizione dei parametri della sessione, come i codec utilizzati e i tipi di media supportati.

- **ICE Candidates (Interactive Connectivity Establishment)**

Include gli indirizzi IP e le porte disponibili per la connessione P2P.

Il signalling si basa su un canale di signaling che può essere realizzato tramite: WebSocket, HTTP/REST API oppure tramite sistemi di messaggistica personalizzati.

2.3.1 Fasi del Signalling

Di seguito la descrizione delle fasi del Signalling:

1. Creazione dell'Offerta SDP:

- Il primo peer genera un SDP Offer contenente i dettagli della sessione, come indirizzo IP, porte e codec audio/video.
- Questo SDP viene poi inviato all'altro peer tramite il canale di signalling.

2. Scambio dell'Offerta e Risposta SDP:

- Il secondo peer riceve l'SDP Offer e risponde con un SDP Answer, indicando i parametri che accetta.

3. Scambio degli ICE Candidates:

- Entrambi i peer raccolgono i candidati ICE per identificare il percorso migliore per la connessione.
- I candidati vengono scambiati attraverso il canale di signalling fino a trovare una connessione compatibile.

4. Connessione Diretta P2P:

- Quando gli ICE candidates risultano compatibili, i due peer stabiliscono una connessione diretta, evitando il server di signalling.
- La comunicazione avviene tramite SRTP (Secure Real-time Transport Protocol) per garantire sicurezza e crittografia.

Capitolo 3

Architettura e Flussi

L'applicazione è stata progettata per facilitare la comunicazione in tempo reale tra due utenti, utilizzando WebRTC per lo scambio di flussi multimediali e Socket.io per il signaling e la gestione della connessione tra i peer.

L'architettura si basa su due componenti principali:

- Backend (Server Node.js con Express e Socket.io)
- Frontend (React.js con socket.io-client e WebRTC API)

L'unico metodo di comunicazione tra client e server avviene tramite Socket.io, eliminando la necessità di richieste HTTP tradizionali.

3.1 Architettura

Di seguito una rappresentazione dell'architettura implementata:



Figura 3.1: Architettura dell’Applicazione

Come si può osservare nella figura, sono descritti a un livello generale i componenti Client e Server.

Il componente Client comprende:

- **WebRTC API**, che gestisce le connessioni peer-to-peer e lo streaming audio/video.
- **socket.io-client**, che consente la comunicazione con il server tramite WebSocket.
- **UI (components)**, che offre l’interfaccia utente per l’interazione tra gli utenti.
- **RTCPeerConnection**, responsabile della negoziazione della connessione WebRTC e dello scambio dei flussi multimediali.

Il componente Server include:

- **Express.js**, utilizzato per creare il server Node.js.
- **Socket.io**, che gestisce il signaling WebRTC e la comunicazione in tempo reale tra i client.

- **Gestione del Signalling**, che coordina l'invio e la ricezione dei messaggi di signaling tra i peer.
- **Gestione del matchmaking**, che abbina gli utenti disponibili per una conversazione.

La comunicazione avviene interamente tramite Socket.io, che consente al client e al server di scambiarsi messaggi di signalling WebRTC come SDP (Session Description Protocol) e ICE Candidates. Dopo la fase di signalling, se i peer trovano un percorso di connessione valido, viene stabilita una connessione diretta WebRTC peer-to-peer per lo scambio dei flussi audio/video.

3.2 Sequenza di Negoziazione

Questo sequence diagram illustra il processo di connessione tra due utenti in Uniegle. Il flusso descrive le interazioni tra i client (A e B) e il server, evidenziando il meccanismo di abbinamento e la successiva negoziazione WebRTC per la comunicazione peer-to-peer.

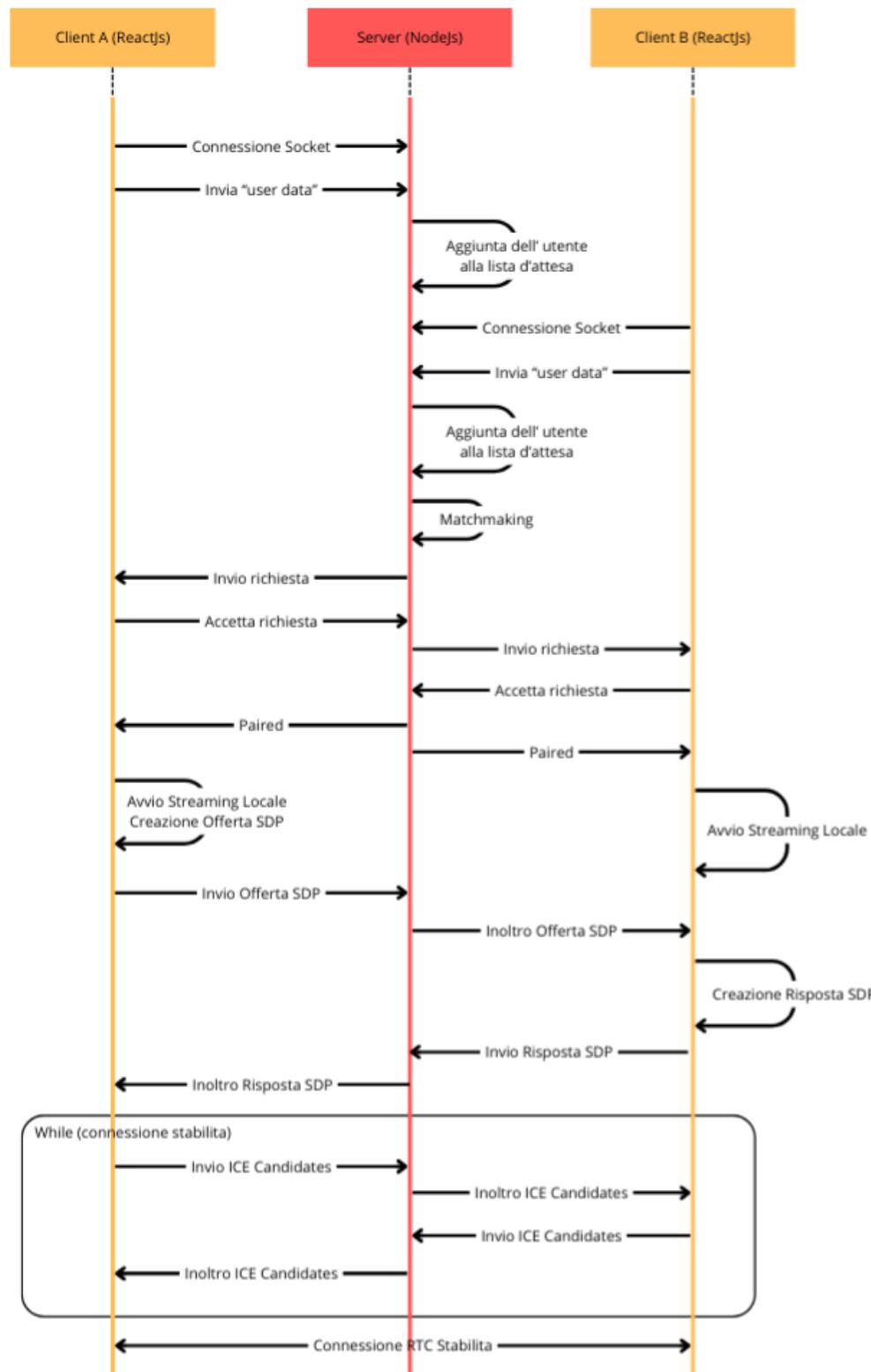


Figura 3.2: Sequenza di negoziazione

Di seguito la spiegazione dettagliata delle fasi di negoziazione:

1. Connessione iniziale

- Entrambi i client stabiliscono una connessione WebSocket con il server.
- Invia i dati utente al server per essere inseriti nella lista d'attesa.

2. Pairing e richiesta di connessione

- Il server avvia il matchmaking e abbina due utenti disponibili.
- Invia una richiesta di connessione a entrambi i client con i dettagli del partner.

3. Accettazione e fase di pairing

- I client accettano la richiesta di connessione.
- Una volta che entrambi confermano, il server notifica l'avvenuto pairing.
- Il server determina quale client avvierà la negoziazione WebRTC come initiator.

4. Scambio di SDP e ICE Candidates

- Il client initiator crea un'offerta SDP e la invia al partner tramite il server.
- Il client responder accetta l'offerta e genera una risposta SDP, che viene inoltrata all'iniziatore.
- I client scambiano i candidati ICE per attraversare NAT e firewall, stabilendo la connessione diretta.

5. Connessione WebRTC stabilità

- Una volta completato lo scambio dei candidati ICE, la connessione WebRTC è attiva e la comunicazione avviene direttamente tra i client.

3.3 Instant Messaging

Oltre alla connessione audio/video, l'app Uniegle offre un sistema di messaggistica in tempo reale, che permette agli utenti di scambiarsi messaggi di testo e di vedere quando l'altro sta scrivendo.

La chat è gestita in modo bidirezionale tra i due client, con il server che funge da intermediario. Il flusso può essere diviso in due funzionalità principali:

- Invio e ricezione di messaggi
- Indicatore di digitazione (Typing Status)

Di seguito è mostrato un sequence diagram che rappresenta l'interazione tra due client (Client A e Client B) e il server per la gestione della chat istantanea.

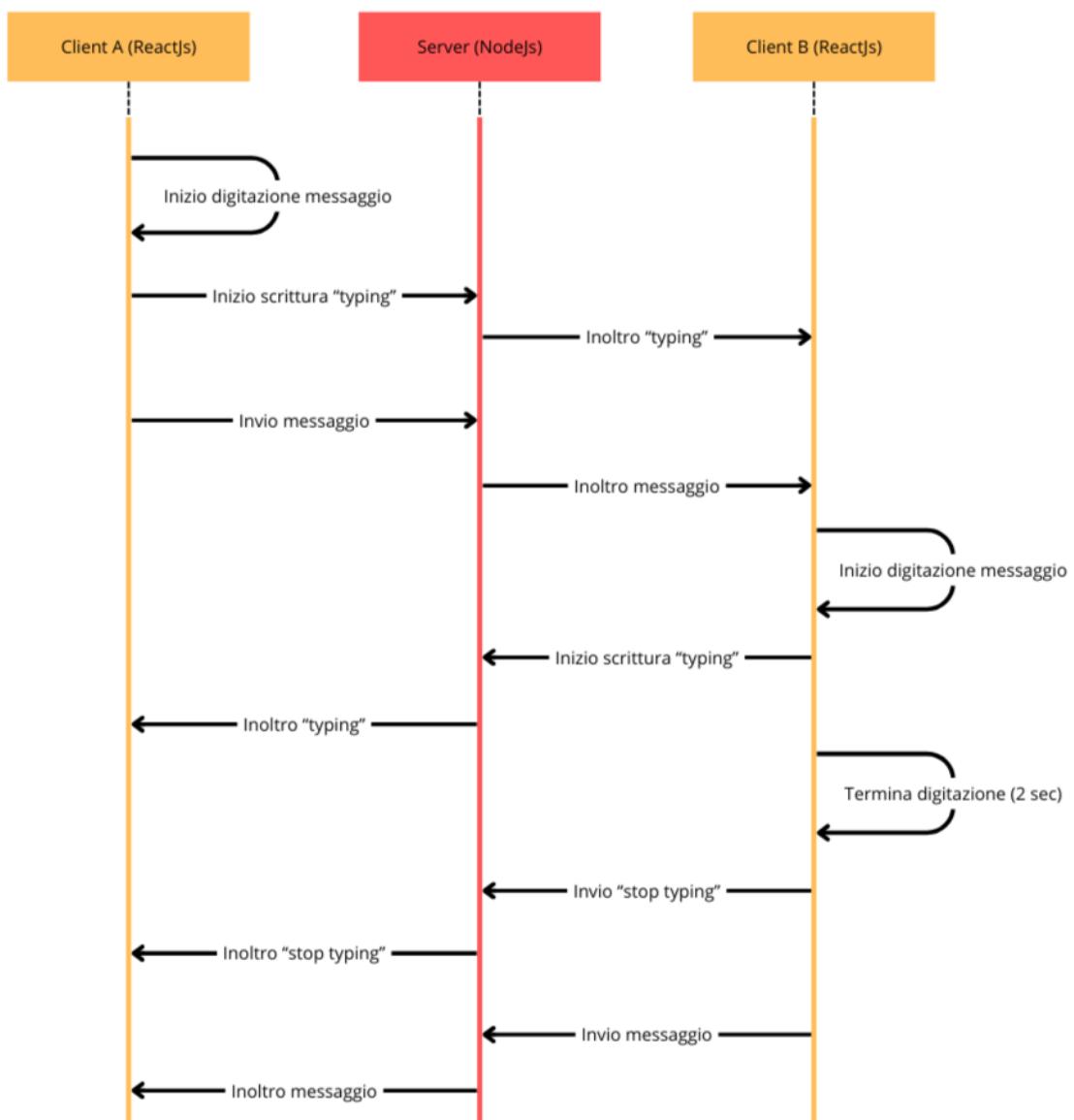


Figura 3.3: Sequenza della messaggistica istantanea

Di seguito la spiegazione degli step dell'instant messaging:

1. Inizio della digitazione (Typing)

- Client A inizia a digitare un messaggio:
 - Il client rileva l'input e invia un evento "typing" al server.
 - Il server inoltra il segnale di "typing" a Client B.
 - Client B riceve il segnale e mostra l'indicatore di digitazione.

2. Invio di un messaggio

- Client A invia un messaggio al server:
 - Il server lo riceve e lo inoltra immediatamente a Client B.
 - Client B riceve il messaggio e lo visualizza nella chat.

3. Client B inizia a digitare

- Client B inizia a digitare un messaggio di risposta:
 - Invia il segnale "typing" al server.
 - Il server inoltra il segnale "typing" a Client A.
 - Client A ora vede che Client B sta scrivendo.

4. Stop Typing dopo 2 secondi di inattività

- Se Client B smette di digitare per 2 secondi, viene inviato "stop typing":

- Client B non ha digitato per 2 secondi.
- Il client invia "stop typing" al server.
- Il server inoltra "stop typing" a Client A.
- Client A rimuove l'indicatore di digitazione.

5. Client B invia il messaggio

- Client B conferma il messaggio e lo invia al server:
 - Il server inoltra il messaggio a Client A.
 - Client A riceve il messaggio e lo visualizza.

Per evitare un eccessivo carico di richieste al server, l'invio del segnale "typing" e "stop typing" è ottimizzato tramite un delay di 2 secondi. Questo meccanismo è gestito interamente lato client e funziona come segue:

- Quando un utente inizia a digitare, il client attiva un timer di 2 secondi prima di inviare il segnale "typing" al server.
- Se l'utente continua a digitare prima della scadenza del timer, il conteggio viene resettato, impedendo l'invio ripetitivo del segnale.
- Quando l'utente smette di scrivere, viene attivato un altro timer di 2 secondi. Solo se non vengono rilevati nuovi input durante questo intervallo, il client invia il segnale "stop typing".

Capitolo 4

Implementazione

Per sviluppare l'app Uniegle, abbiamo scelto un'architettura client-server ben strutturata, separando chiaramente backend e frontend. Il backend gira su Node.js con Express.js per gestire il server e Socket.io per la comunicazione in tempo reale tra gli utenti. Per il frontend, invece, abbiamo puntato su React.js, sfruttando il paradigma dei componenti per una UI fluida e reattiva. La gestione dello stato è affidata agli hooks di React, mentre il dialogo con il server avviene tramite la Socket.io Client API. Per lo sviluppo, ci siamo affidati a Visual Studio Code e, per il versionamento del codice, a GitHub, così da garantire un lavoro di squadra efficiente e organizzato. Nei prossimi passaggi entremo più nel dettaglio: prima daremo un'occhiata al cuore del backend e alle sue funzionalità principali, poi ci concentreremo sul frontend e su come i vari componenti interagiscono con il server.

4.1 Server con Node.js

Node.js è un ambiente di runtime JavaScript lato server, costruito sul motore V8 di Chrome. In pratica, permette di eseguire codice JavaScript fuori dal browser, rendendolo perfetto per applicazioni veloci e scalabili. Il suo punto di forza è l'utilizzo di un modello event-driven e non bloccante, che gli consente di gestire molte richieste contemporaneamente senza rallentamenti. Tra le caratteristiche principali di Node.js vi sono:

- **Event-driven:** tutto si basa su un loop di eventi, che gestisce le richieste in modo efficiente senza spreco di risorse.
- **Asynchronous I/O:** le operazioni di input/output avvengono in modo non bloccante, migliorando nettamente le prestazioni.
- **Single-threaded (intelligente):** anche se lavora su un solo thread, sfrutta un sistema multi-thread sotto il cofano per gestire le operazioni più complesse.
- **Ecosistema enorme:** grazie a npm (Node Package Manager), hai a disposizione migliaia di librerie per integrare tutto ciò di cui hai bisogno.

4.1.1 Librerie utilizzate

Il codice del server Node.js è contenuto all'interno di un unico file, *server/server.js*, e si basa sull'utilizzo di tre pacchetti fondamentali:

- **Express:** Framework web minimalista per Node.js, utilizzato per creare e gestire il server HTTP in modo semplice ed efficiente.
- **http:** Modulo nativo di Node.js, impiegato per creare il server HTTP e integrarlo con WebSockets tramite Socket.io.
- **Socket.io:** Libreria che consente una comunicazione bidirezionale e in tempo reale tra client e server, gestendo le connessioni WebSocket per la video chat e la messaggistica istantanea tra gli utenti.

4.1.2 Descrizione del codice

Di seguito la descrizione del codice presente nel file server.js

Configurazione del server

```
const app = express();
const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: "http://localhost:5173",
    methods: ["GET", "POST"]
  },
  maxHttpBufferSize: 1e7
});

const PORT = 4000;

server.listen(PORT, () => {
  console.log(`Server in ascolto sulla porta ${PORT}`);
});
```

Figura 4.1: Configurazione del server

Il server viene avviato creando un'applicazione Express per gestire le richieste HTTP. Successivamente, tramite il modulo nativo http, viene istanziato un server HTTP, necessario per supportare WebSockets.

Per abilitare la comunicazione in tempo reale, viene integrato Socket.io, che permette di gestire connessioni WebSocket tra client e server. Poiché il frontend è sviluppato in React, il supporto per CORS viene configurato per consentire la comunicazione senza restrizioni di origine.

Matchmaking tra Utenti

```

function checkAndPairUsers() {
    cleanWaitingUsers();
    printLogs("checkandpairusers");

    while (waitingUsers.length >= 2) {
        const index1 = Math.floor(Math.random() * waitingUsers.length);
        let index2;
        do {
            index2 = Math.floor(Math.random() * waitingUsers.length);
        } while (index2 === index1);

        const user1 = waitingUsers.splice(index1, 1)[0];
        const user2 = waitingUsers.splice(index2 > index1 ? index2 - 1 : index2, 1)[0];

        const socket1 = io.sockets.sockets.get(user1);
        const socket2 = io.sockets.sockets.get(user2);

        if (!socket1?.userData) {
            io.to(user1).emit('missing userdata');
            printLogs('missing-userdata', { user: user1 });
            waitingUsers.push(user2);
            continue;
        }

        if (!socket2?.userData) {
            io.to(user2).emit('missing userdata');
            printLogs('missinguserdata', { user: user2 });
            waitingUsers.push(user1);
            continue;
        }

        activePairs[user1] = { id: user2.toString(), accepted: false };
        activePairs[user2] = { id: user1.toString(), accepted: false };

        io.to(user1).emit('request', { partner: user2, userData: socket2.userData });
        io.to(user2).emit('request', { partner: user1, userData: socket1.userData });
        printLogs('checkandpairusersfound', { user1, user2 });
    }
}

```

Figura 4.2: Matchmaking

Questa funzione controlla periodicamente se nella lista di attesa ci sono almeno due utenti disponibili per il pairing. Una volta trovata una coppia in maniera casuale, assegna a ciascun utente un partner e invia loro una richiesta di connessione. Se uno dei due utenti non

ha dati utente validi, il pairing viene annullato e l'altro utente viene rimesso in attesa.

Gestione della Disconnessione

```
function exitOrDisconnect(socket){
    cleanWaitingUsers();
    const partner = activePairs[socket.id];

    if(partner){
        if (partner.id) {
            io.to(partner.id).emit('partner disconnected');
        }
        waitingUsers.push(partner.id);
        delete activePairs[partner.id];
    }

    waitingUsers = waitingUsers.filter((id) => id !== socket.id);

    delete activePairs[socket.id];
    printLogs('exit', socket.id);
    checkAndPairUsers();
}
```

Figura 4.3: Disconnessione

Questa funzione controlla periodicamente se nella lista di attesa ci sono almeno due utenti disponibili per il pairing. Una volta trovata una coppia, assegna a ciascun utente un partner e invia loro una richiesta di connessione. Se uno dei due utenti non ha dati utente validi, il pairing viene annullato e l'altro utente viene rimesso in attesa.

Gestione dei Dati Utente e Matchmaking

```
socket.on('user data', (userData) => {
    printLogs('user data', {id: socket.id, data: userData})
    socket.userData = userData;
    waitingUsers.push(socket.id);
    checkAndPairUsers();
});
```

Figura 4.4: User Data

Quando un client invia i propri dati utente, il server lo aggiunge alla lista di attesa e avvia il processo di matchmaking.

Accettazione e rifiuto della richiesta di connessione

Gli utenti possono accettare o rifiutare la connessione con un partner suggerito.

```
socket.on('reject', () => {
    cleanWaitingUsers();
    const partner = activePairs[socket.id];

    if (partner) {
        io.to(partner.id).emit('rejected');
        io.to(socket.id).emit('rejected');
    }

    waitingUsers.push(partner.id);
    waitingUsers.push(socket.id);
    delete activePairs[socket.id];
    delete activePairs[partner.id];

    printLogs("requestrejected", socket.id)
    checkAndPairUsers();
});
```

Figura 4.5: Rifiuto della richiesta

Quando un utente rifiuta la richiesta di connessione, il server comunica l'operazione ad entrambi gli utenti e reinserisce gli utenti nella lista di attesa. Infine, esegue l'operazione di matchmaking.

```
socket.on('accept', () => {
  cleanWaitingUsers();
  You, 4 weeks ago • Primo push ...
  const partner = activePairs[socket.id];
  activePairs[partner.id] = {...activePairs[partner.id], accepted: true};

  if (partner) {
    printLogs("requestaccepted", {user1: socket.id, user2: partner.id});
    if(partner.accepted){
      io.to(socket.id).emit('paired', { partner: partner.id, initiator: false });
      io.to(partner.id).emit('paired', { partner: socket.id, initiator: true});
      printLogs("successconnection", {user1: socket.id, user2: partner.id});
    }else{
      printLogs("waitingrequestaccepted", partner.id);
    }
  }
});

socket.on('signal', (data) => {
  const partner = activePairs[socket.id];
  if (partner.id) {
    io.to(partner.id).emit('signal', data);
  }
});
```

Figura 4.6: Accettazione della richiesta

Quando entrambi un client accetta la richiesta di connessione, il server verifica se l'altro client ha accettato. Se l'altro client ha accettato la connessione allora il server invia un segnale di "pair" ad entrambi i client, in caso contrario, il client rimane in attesa di accettazione o rifiuto della richiesta.

Gestione della Negoziazione WebRTC

```
socket.on('signal', (data) => {
  const partner = activePairs[socket.id];
  if (partner.id) {
    io.to(partner.id).emit('signal', data);
  }
});
```

Figura 4.7: Gestione della Negoziazione

Quando un client vuole stabilire una connessione peer-to-peer con un altro client, invia un segnale al server contenente le informazioni necessarie (come l'offerta SDP o i candidati ICE). Il server riceve il segnale e lo inoltra al partner associato al client, consentendo così lo scambio di dati richiesto per completare la connessione WebRTC.

Gestione della Messaggistica Istantanea

```
socket.on('chat message', (message) => {
  const partner = activePairs[socket.id];
  if (partner && partner.id) {
    const accept1 = activePairs[partner.id].accepted;
    const accept2 = activePairs[socket.id].accepted;
    if(accept1 && accept2){
      io.to(partner.id).emit('chat message', message);
    }
  }
});
```

Figura 4.8: Messaggistica Istantanea

Questa funzione si occupa della gestione della chat tra due client connessi. Quando un client invia un messaggio al server, quest'ultimo verifica se entrambi i client hanno accettato la connessione. Solo se

entrambi hanno confermato la connessione, il messaggio viene inoltrato al destinatario. Questo controllo evita che vengano inviati messaggi a client che non hanno ancora stabilito una connessione, garantendo che la comunicazione avvenga solo tra utenti effettivamente abbinati.

Indicazione dello Stato di Digitazione

```
socket.on('typing', () => {
  const partner = activePairs[socket.id];
  if (partner && partner.id) {
    io.to(partner.id).emit('typing', socket.id);
  }
});

socket.on('stop typing', () => {
  const partner = activePairs[socket.id];
  if (partner && partner.id) {
    io.to(partner.id).emit('stop typing', socket.id);
  }
});
```

Figura 4.9: Stato di Digitazione

La funzione "typing" è responsabile della gestione dello stato di digitazione all'interno della chat. Quando un client inizia a scrivere un messaggio, invia un segnale "typing" al server.

Quando un client smette di digitare, invia un segnale "stop typing" al server, che lo inoltra al client partner.

Gestione dell'evento Skip

```
socket.on('skip', () => {
    cleanWaitingUsers();
    const partner = activePairs[socket.id];

    if(partner){
        if (partner.id) {
            io.to(partner.id).emit('partner disconnected');
        }
        waitingUsers.push(partner.id);
        delete activePairs[partner.id];
    }

    waitingUsers = waitingUsers.filter((id) => id !== socket.id);
    waitingUsers.push(socket.id)

    delete activePairs[socket.id];
    if(partner){
        printLogs('skip', {user1: socket.id, user2: partner.id});
    }
    checkAndPairUsers();
});
```

Figura 4.10: Evento Skip

Questa funzione gestisce la funzionalità di "skip", che consente a un client di terminare la conversazione attuale e cercare un nuovo partner. Quando un client decide di saltare la connessione, l'evento viene attivato e il server invia una segnalazione di disconnessione al partner e reinserisce il client nella waiting list.

4.2 Client con React.js

React.js è una delle librerie JavaScript più utilizzate per lo sviluppo di interfacce utente moderne, interattive e performanti. Creata da Facebook, è progettata per semplificare la creazione di applicazioni web grazie a un'architettura basata su componenti riutilizzabili.

Uno dei suoi punti di forza è il Virtual DOM, un sistema che permette di aggiornare l'interfaccia in modo reattivo e veloce, evitando il ricaricamento completo della pagina. Questo rende le applicazioni più fluide, scalabili e modulari.

Caratteristiche principali:

- **Component-based:** l'interfaccia è suddivisa in componenti riutilizzabili, rendendo il codice più organizzato e facile da mantenere.
- **Virtual DOM:** gli aggiornamenti vengono applicati solo sugli elementi modificati, migliorando nettamente le prestazioni.
- **Hooks:** funzioni come useState e useEffect semplificano la gestione dello stato e degli effetti collaterali, senza bisogno di classi.

4.2.1 Librerie utilizzate

Per lo sviluppo del client in React.js, è stato adottato un insieme di librerie che hanno contribuito a migliorare l'esperienza utente, ottimizzando la navigazione, la comunicazione in tempo reale e l'aspetto

grafico dell'applicazione.

Tra le librerie utilizzate, tre risultano particolarmente rilevanti:

- **react-router-dom:** libreria fondamentale per la gestione della navigazione tra le diverse pagine dell'applicazione. Permette di definire route dinamiche e di aggiornare l'interfaccia utente in base alla posizione dell'utente, evitando ricaricamenti della pagina e migliorando la fluidità dell'esperienza di navigazione.
- **socket.io-client:** consente la comunicazione bidirezionale in tempo reale tra il frontend e il server attraverso WebSockets. È essenziale per gestire le connessioni, abilitare la messaggistica istantanea e facilitare la negoziazione WebRTC tra gli utenti.
- **@mui/material:** un set di componenti UI pre-progettati e personalizzabili, basati sulle linee guida di Material Design di Google. Viene utilizzato per arricchire l'interfaccia grafica.

4.2.2 Descrizione del codice

Nel frontend dell'applicazione, la comunicazione con il server è gestita attraverso la libreria socket.io-client, che permette di ricevere e processare in tempo reale gli eventi WebSocket inviati dal server.

Per organizzare questa interazione, il codice utilizza un useEffect, che entra in azione non appena il componente "Chat" viene montato. All'interno di questa funzione, vengono impostati i listener per diversi

eventi WebSocket, consentendo al client di rispondere dinamicamente a cambiamenti di stato, connessioni WebRTC e nuovi messaggi in chat.

Gestione della connessione WebSocket

```
socket.on('connect', () => {
  disconnectChat();
  navigate("/", {replace: true});
});
```

Figura 4.11: Evento Connect

Quando il client stabilisce una connessione con il server WebSocket, viene eseguita la funzione `disconnectChat()`, il cui compito è chiudere eventuali sessioni attive prima di avviare una nuova connessione. Subito dopo, l'utente viene reindirizzato alla pagina principale dell'applicazione. Questo meccanismo è fondamentale per garantire che ogni connessione parta da uno stato pulito, evitando conflitti o problemi legati a sessioni precedenti ancora aperte.

Gestione dello Scambio di Segnali WebRTC

```
socket.on('signal', async (data) => {
  if (data.candidate) {
    await peerConnection.current.addIceCandidate(data.candidate);
    setDisabledChat(false);
    setOpenRequest(false);
    setMessages([]);
    setStatus(`connected`);
  } else if (data.sdp) {
    await peerConnection.current.setRemoteDescription(new RTCSessionDescription(data.sdp));
    if (data.sdp.type === 'offer') {
      await startStream(selectedVideoDevice, selectedAudioDevice);
      const answer = await peerConnection.current.createAnswer();
      await peerConnection.current.setLocalDescription(answer);
      socket.emit('signal', { sdp: answer });
    }
  }
});
```

Figura 4.12: Evento Signal

Questo evento gestisce lo scambio dei segnali WebRTC tra i due client, facilitando la comunicazione peer-to-peer.

- Se il messaggio ricevuto contiene un ICE candidate, viene aggiunto alla connessione WebRTC per ottimizzare il percorso di comunicazione.
- Se invece il messaggio include un SDP (Session Description Protocol), viene impostato come descrizione remota per stabilire la connessione tra i peer.
- Nel caso in cui il segnale ricevuto sia un'offerta SDP, il client avvia lo streaming della propria videocamera e genera una risposta SDP, che viene inviata al server per completare il processo di negoziazione WebRTC.

Gestione dell'Abbinamento tra Client

```
socket.on('paired', async (data) => {
    setStatus("paired");
    const configuration = {
        iceServers: [
            { urls: 'stun:stun.l.google.com:19302' },
        ],
    };

    peerConnection.current = new RTCPeerConnection(configuration);

    peerConnection.current.ontrack = (event) => {
        remoteVideoRef.current.srcObject = event.streams[0];
    };

    peerConnection.current.onicecandidate = (event) => {
        if (event.candidate) {
            socket.emit('signal', { candidate: event.candidate });
        }
    };
    await startStream(selectedVideoDevice, selectedAudioDevice);

    if(data.initiator){
        const offer = await peerConnection.current.createOffer();
        await peerConnection.current.setLocalDescription(offer);
        socket.emit('signal', { sdp: offer });
    }
});
```

Figura 4.13: Evento Paired

Per stabilire la connessione WebRTC, viene creata una nuova istanza di RTCPeerConnection, che gestisce la comunicazione tra i due client. Quando il client riceve tracce audio/video, queste vengono assegnate al video player remoto, permettendo la visualizzazione del flusso multimediale. Se il client genera un ICE candidate, lo invia al server, che si occupa di inoltrarlo al partner per ottimizzare la connessione. Se il client è l'initiator, crea un'offerta SDP e la invia all'altro client, avviando così la negoziazione WebRTC.

Gestione della Disconnessione del Partner

```
socket.on('partner disconnected', () => {
    setRequest(null);
    setOpenRequest(false);
    setDisabledChat(true);
    setNewMessagesCount(0);
    setMessages([]);
    setStatus("disconnected");
    remoteVideoRef.current.srcObject = null;
});
```

Figura 4.14: Evento Partner Disconnected

Quando il partner si disconnette, l’interfaccia utente si aggiorna per segnalare la fine della connessione.

- L’oggetto richiesta viene resettato per evitare dati obsoleti.
- La chat viene disabilitata e tutti i messaggi vengono cancellati, impedendo ulteriori interazioni.
- Lo stato dell’utente cambia in "disconnected", e lo streaming video viene interrotto, liberando le risorse del sistema.

Gestione della Messaggistica Istantanea

```
socket.on('chat message', (msg) => {
    setMessages((prevMessages) => [...prevMessages, { sender: 'partner', text: msg }]);
    setIsTyping(false);
    if(!openDrawerRef.current){
        setNewMessagesCount((prevCount) => prevCount + 1);
    }
    if(!muteRef.current){
        const audio = new Audio(notificationSound);
        audio.play().catch((err) => console.error("Errore nella riproduzione del suono:", err));
    }
});
```

Figura 4.15: Evento Chat Message

Quando il client riceve un nuovo messaggio, questo viene aggiunto alla lista dei messaggi.

- Se la chat è chiusa, viene mostrata una notifica con il numero di nuovi messaggi.
- Se le notifiche audio non sono disattivate, viene riprodotto un suono per avvisare l'utente.
- L'indicatore di digitazione viene disattivato una volta ricevuto un messaggio.

Gestione dell'Indicatore di Digitazione

```
socket.on('typing', () => {
|   setIsTyping(true);
|});
socket.on('stop typing', () => {
|   setIsTyping(false);
|});
```

Figura 4.16: Evento Typing/Stop Typing

Questi eventi vengono attivati quando il partner sta digitando un messaggio.

- Se il client riceve "typing", l'indicatore di digitazione viene mostrato.
- Quando riceve "stop typing", l'indicatore scompare.

Gestione delle Richieste di Connessione

```
socket.on('request', (data) => {
    setRequest(data);
    setOpenRequest(true);
    setStatus("waitrequest")
})

socket.on('rejected', () => {
    setRequest(null);
    setOpenRequest(false);
    setStatus("rejectrequest");
})
```

Figura 4.17: Evento Request/Rejected

Quando un client riceve una richiesta di connessione da un altro utente "request", i dati della richiesta vengono salvati nello stato dell'applicazione e viene aggiornato lo stato in "waitrequest", in attesa di risposta dal partner. Se la richiesta di connessione viene rifiutata, l'interfaccia utente viene aggiornata per notificare l'evento e lo stato passa a "rejectrequest".

Gestione dell'Errore "Missing User Data"

```
socket.on('missing userdata', () => {
    disconnectChat();
    navigate("/user-profile", {replace: true});
})
```

Figura 4.18: Evento Missing Userdata

Se il server rileva che il client non ha fornito i dati utente necessari, la connessione viene interrotta e l'utente viene reindirizzato alla pagina del profilo per completare le informazioni mancanti.

Chiusura dei Listener WebSocket

```
return () => {
    socket.off('signal');
    socket.off('paired');
    socket.off('partner disconnected');
    socket.off('chat message');
    socket.off('user data');
    socket.off('request');
    socket.off('reject');
    socket.off('typing');
    socket.off('stop typing');
};
```

Figura 4.19: Chiusura Listner

Questa funzione si occupa di acquisire e gestire lo stream audio-video dell'utente:

- Il primo step prevede l'uso di navigator.mediaDevices-.getUserMedia(), che richiede i permessi per accedere alla video-camera e al microfono. Se l'utente ha specificato dispositivi preferiti tramite videoDeviceId e audioDeviceId, verranno utilizzati quelli; altrimenti, il sistema selezionerà i dispositivi predefiniti.
- Una volta ottenuto lo stream, il riferimento localVideoRef.current-

.srcObject viene aggiornato per mostrare in tempo reale l’anteprima del video, permettendo all’utente di vedere la propria immagine prima di avviare la chiamata.

- Le tracce multimediali vengono collegate alla connessione WebRTC (peerConnection). Il metodo getSenders() recupera gli oggetti RTCRtpSender, che rappresentano i flussi multimediali attualmente in uso.
- Per le tracce video, se esiste già un sender video attivo, la traccia viene aggiornata con replaceTrack() per cambiare fotocamera senza interrompere la chiamata. Se non esiste, viene aggiunta con addTrack(). Per le tracce audio, si segue lo stesso processo, garantendo che il microfono funzioni correttamente durante la conversazione.

Capitolo 5

Guida all'utilizzo

La repository GitHub dell'applicazione è organizzata in due cartelle principali all'interno del branch main:

- **client/** che contiene il codice del frontend sviluppato con React.js.
- **server/** che contiene il codice del backend sviluppato con Node.js e Socket.io.

5.1 Configurazione e Avvio del Frontend

Per eseguire il client React, seguire questi passaggi:

1. Accedere alla cartella *client/*: **cd client**
2. Installare le dipendenze necessarie utilizzando npm:
npm install

Questo comando installerà automaticamente tutti i pacchetti definiti nel file package.json.

3. Avviare l'applicazione in modalità di sviluppo: **npm run dev**

Dopo l'avvio, verranno mostrati in console gli host disponibili per accedere all'applicazione tramite browser.

4. (*Opzionale*) Avviare l'applicazione in modalità di produzione:

- Eseguire il comando per generare la build ottimizzata:

npm run build

- Una volta completata la build, eseguire l'applicazione con:

npm run preview

- Verrà mostrato in console l'host di riferimento per accedere all'applicazione in modalità di produzione.

5.2 Configurazione e Avvio del Backend

Per eseguire il server Node.js, seguire questi passaggi:

1. Accedere alla cartella *server/*: **cd server**

2. Installare le dipendenze necessarie utilizzando npm:

npm install

Questo comando installerà automaticamente tutti i pacchetti definiti nel file package.json.

3. Avviare il server: **npm run server**

Il server sarà quindi in esecuzione e pronto a gestire le connessioni WebSocket.

5.3 Configurazione

Per garantire una corretta comunicazione tra il client e il server, è fondamentale configurare correttamente gli URL nei file di configurazione:

- **Nel server (server.js)**

Modificare la riga 11 per impostare correttamente l'URL del client nel parametro CORS di Socket.io.

- **Nel client (router.jsx)**

Modificare la riga 8 per impostare correttamente l'URL del server all'interno della connessione WebSocket.

5.4 Presentazione

L'applicazione offre un'esperienza di video chat anonima e in tempo reale, strutturata in tre sezioni principali: Preview, User Profile e Chat.

Grazie a un design responsive, l'interfaccia si adatta perfettamente sia ai dispositivi desktop che mobile, offrendo un'esperienza fluida e coerente su qualsiasi schermo. Questo risultato è stato ottenuto com-

binando CSS con i componenti altamente personalizzabili di MUI Material, che permettono di mantenere un'estetica moderna, intuitiva e facilmente accessibile.

5.4.1 Pagina di Preview



Figura 5.1: Preview

La pagina Preview funge da punto di ingresso all'applicazione. Al centro della schermata, l'utente viene accolto dal logo dell'applicazione e da uno slogan, che sintetizza l'essenza del servizio in modo chiaro e immediato. Un bottone ben visibile invita a proseguire verso la pagina User Profile, dando il via all'esperienza di chat. Nella parte inferiore della pagina, un footer riporta i nomi degli autori dell'applicazione.

5.4.2 Pagina User Profile

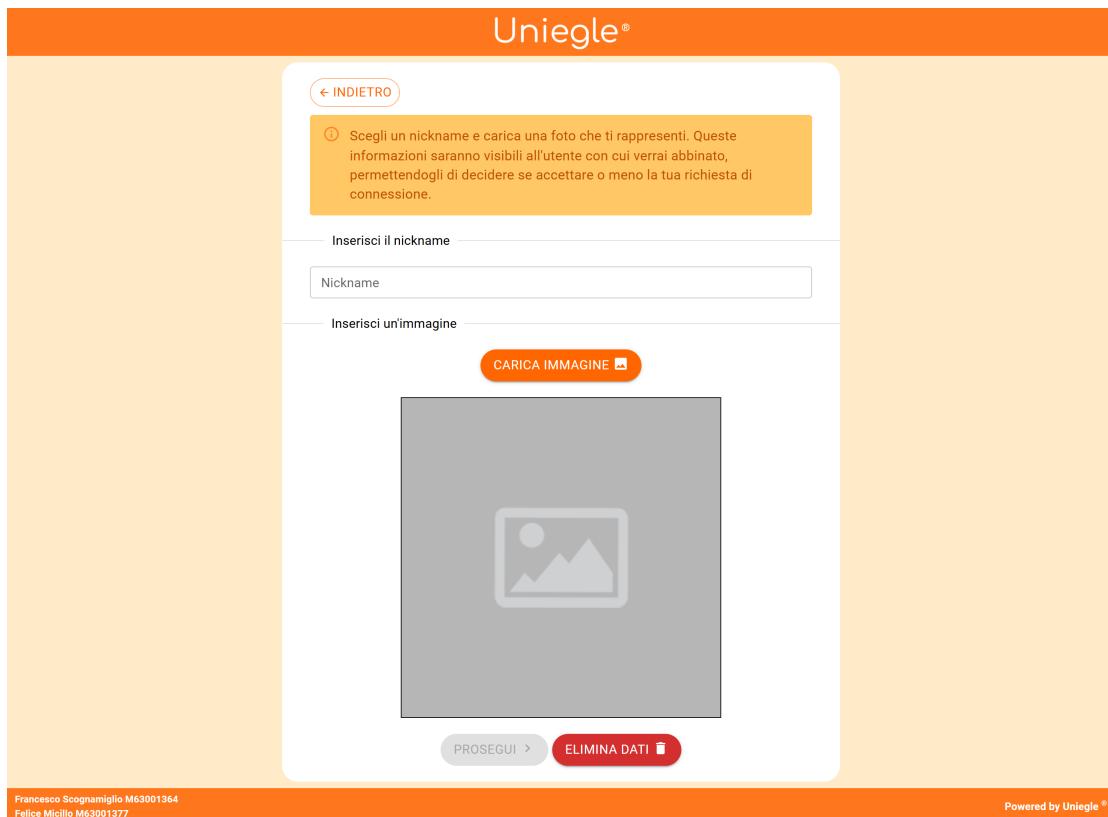


Figura 5.2: User Profile

Dopo aver superato la pagina di Preview, l'utente viene indirizzato alla pagina User Profile, dove deve inserire alcune informazioni essenziali prima di poter iniziare una chat. L'utente deve scegliere un nickname, che verrà visualizzato dal partner durante la conversazione, e un'immagine che lo rappresenti, che può essere caricata in diversi modi.

CAPITOLO 5. GUIDA ALL'UTILIZZO

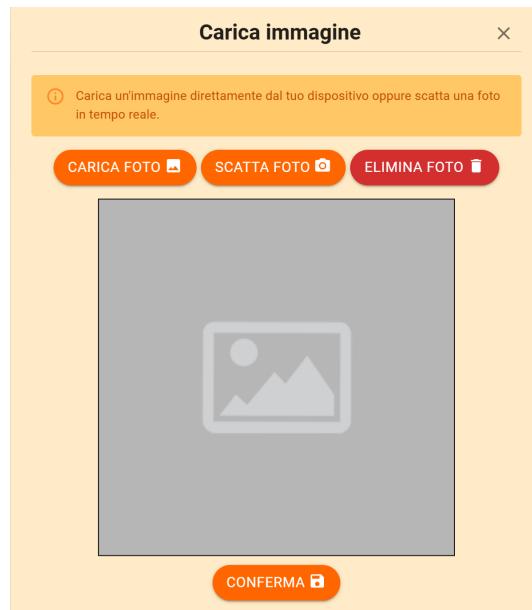


Figura 5.3: Edit Image

Oltre alla classica opzione di upload da file, è stata implementata una funzionalità che consente di scattare una foto direttamente dalla webcam o dalla fotocamera del telefono.

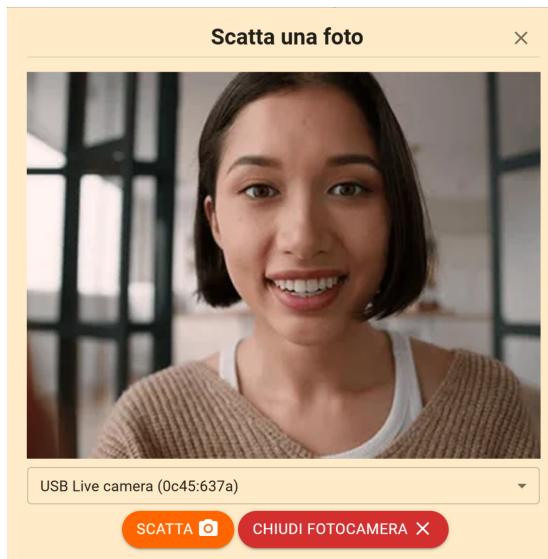


Figura 5.4: Scatta Foto

Inoltre, è possibile rimuovere l'immagine attuale per poterne caricare un'altra. Una volta selezionata l'immagine desiderata, è necessario

confermare la scelta per confermare l'immagine caricata. Una volta inseriti nickname e immagine rappresentativa si può procedere alla chat.

5.4.3 Pagina Chat

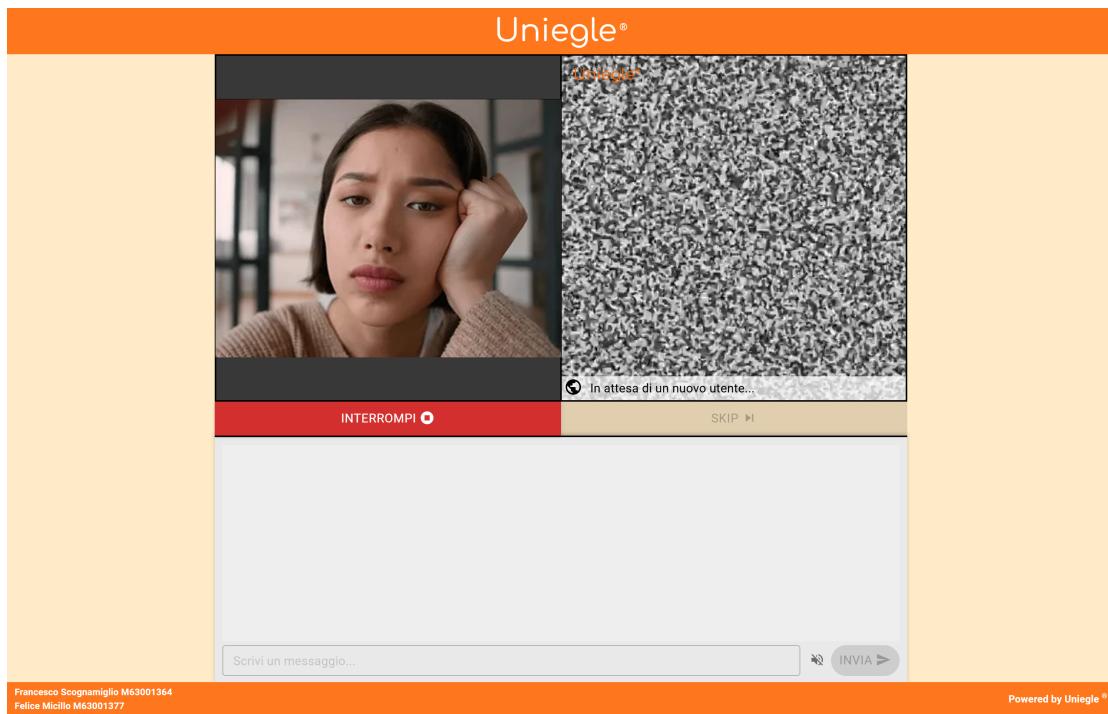


Figura 5.5: Pagina Chat

La schermata presenta due riquadri: da un lato, viene mostrato il proprio video, mentre dall'altro, inizialmente vuoto, apparirà il video del partner una volta avvenuto il pairing. La pagina chat include anche due pulsanti fondamentali: Interrompi e Skip. Il pulsante Interrompi permette all'utente di terminare la sessione e tornare alla pagina precedente, mentre il pulsante Skip consente di saltare il partner attuale e cercare un nuovo utente con cui connettersi.

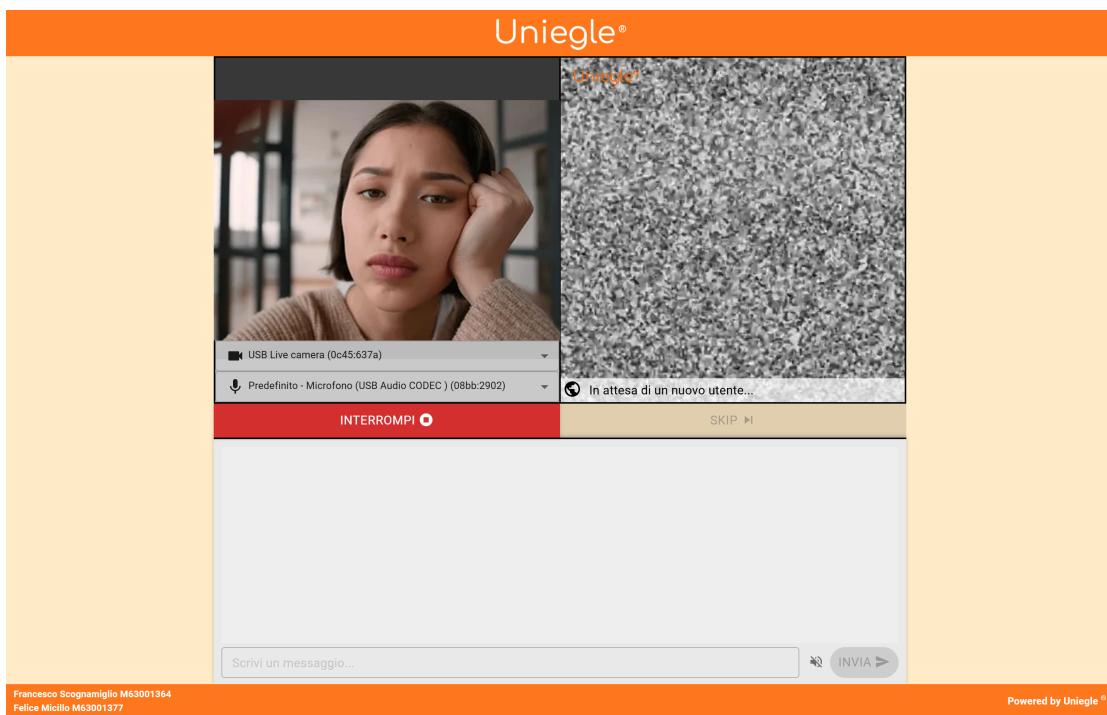


Figura 5.6: Cambio Dispositivo Video/Audio

Durante la sessione di chat, l'utente ha la possibilità di modificare le impostazioni audio e video. Passando il mouse sopra il proprio video, compaiono delle opzioni per cambiare la webcam o il microfono utilizzato. Questa funzionalità è particolarmente utile per chi dispone di più dispositivi di acquisizione e desidera passare rapidamente da una periferica all'altra senza dover interrompere la connessione. Oltre alla comunicazione video, l'applicazione offre anche una chat testuale, che diventa attiva non appena l'accoppiamento con un partner viene confermato.

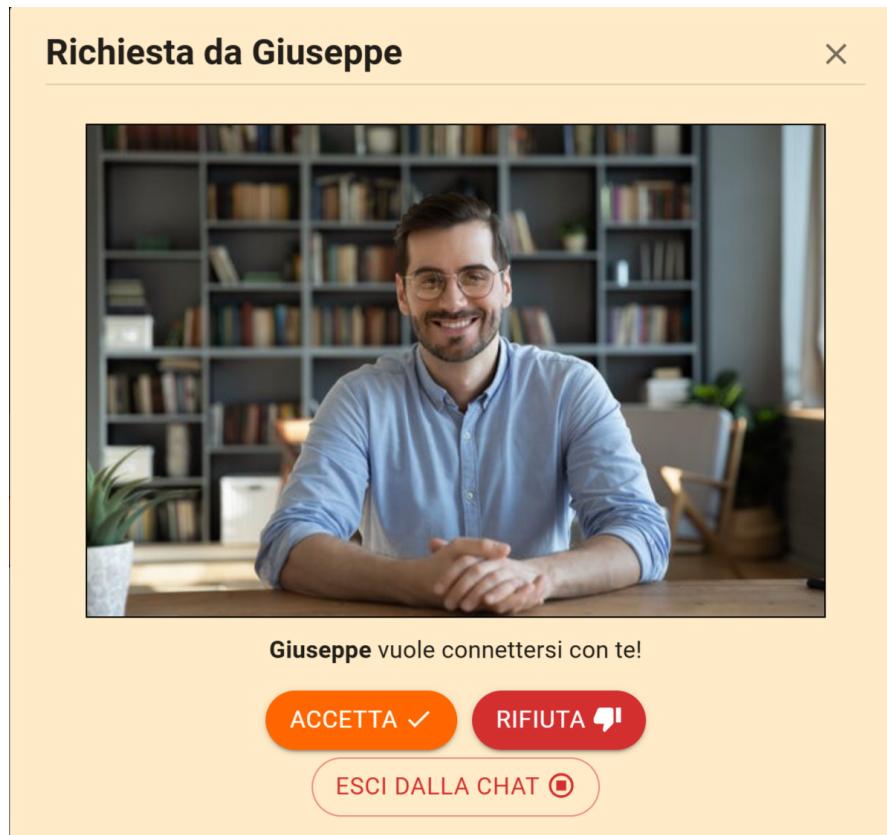


Figura 5.7: Richiesta di Connessione

Il processo di abbinamento tra utenti è gestito tramite un sistema di richiesta di pairing. Quando viene trovata una corrispondenza, compare un pop-up che permette di accettare o rifiutare la connessione. Se entrambi gli utenti accettano, la chat si attiva immediatamente e i due possono iniziare a comunicare. In caso di rifiuto, il sistema cercherà automaticamente un nuovo partner disponibile.

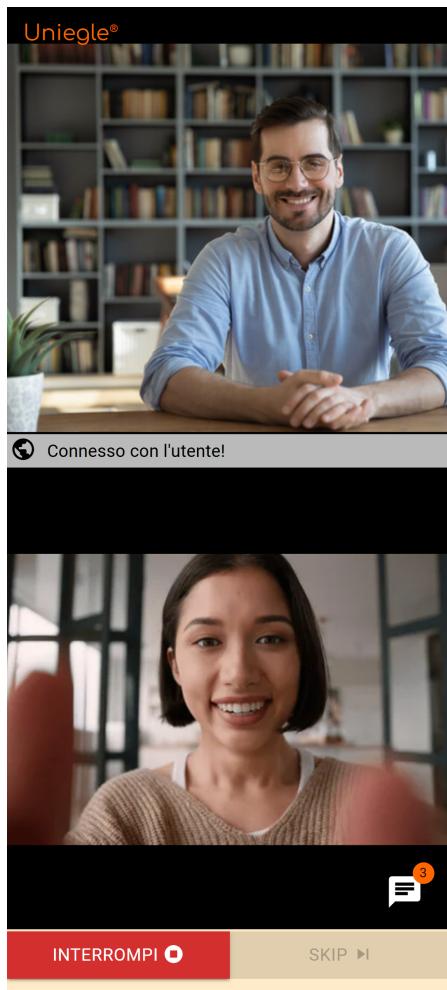


Figura 5.8: Chat Versione Mobile

Nella versione per dispositivi mobili vengono mantenute tutte le funzionalità della versione desktop. Su smartphone, l'utente può cambiare la fotocamera tra quella frontale e quella posteriore, tramite un pulsante posto in alto a sinistra in corrispondenza del proprio video. Inoltre, la chat testuale è accessibile attraverso un'icona dedicata, che può essere aperta e chiusa in qualsiasi momento. Se la finestra della chat testuale è chiusa, un'icona posizionata nell'angolo in basso a destra notifica la presenza di nuovi messaggi, mostrando il numero di quelli ancora non letti.