1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

CORSO DI LAUREA IN
INFORMATICA APPLICATA
SCUOLA DI
SCIENZE TECNOLOGIE E FILOSOFIA DELL'INFORMAZIONE

Corso di Reti di Calcolatori
Progetto per la sessione invernale 2020-2021

Studente:
Giacchetti Filippo - Matricola: 282558

Indice:

1. **Introduzione**

1.1 Specifica del progetto

Si vuole creare un'applicazione web di chat video, che permetta a più utenti connessi alla rete internet di comunicare tra di loro condividendo video e audio. Quando un utente apre per la prima volta l'app, sarà creata una stanza per lui con un identificatore univoco e quindi basterà condividere l'URL della stanza alla persona interessata per scambiare messaggi solo con quest'ultima.

1.2 Motivazioni

È passato poco più di un anno da quando è iniziata la pandemia di Coronavirus, a causa della quale siamo stati costretti a rimanere in casa per molto tempo e questo ha reso la video chat il principale metodo di comunicazione a cui affidarsi (per la scuola, il lavoro o semplicemente per parlare con amici e familiari). Tutto questo, insieme al fatto che un progetto di questo tipo ci permette di immergerci in una serie di tecnologie utili, mi ha portato a sceglierlo.

2. **Tecnologie utilizzate**

2.1 WebSocket

WebSocket è un protocollo di comunicazione, che fornisce canali di comunicazione full duplex su una singola connessione TCP. Il protocollo WebSocket è stato standardizzato dall'IETF come RFC 6455 nel 2011.
WebSocket è distinto da HTTP, ma entrambi i protocolli si trovano al livello 7 nel modello OSI e dipendono dal TCP al livello 4.
Secondo RFC 6455, WebSocket è progettato per funzionare su porte HTTP 80 e 443, nonché per supportare proxy e intermediari HTTP, quindi compatibile col protocollo HTTP.
L'handshake di WebSocket utilizza l'header Upgrade di HTTP per ottenere la compatibilità, inoltre il client invia Sec-WebSocket-Key, un header contenente byte casuali con codifica base64 e il server risponde con un HASH della chiave Sec-WebSocket-Accept nell'header.
Questo per evitare che un eventuale proxy invii nuovamente una precedente conversazione memorizzata in cache.
Questo protocollo consente l'interazione tra client e server con un overhead inferiore rispetto alle alternative half-duplex, facilitando la comunicazione in tempo reale.
L'utilizzo delle porte TCP 80 o 443, in caso di connessioni crittografate, è utile in tutti gli ambienti dove viene bloccato il traffico Internet non web tramite firewall.

2.2 WebRTC e PeerJS

WebRTC è una nuova tecnologia con cui è possibile aggiungere capacità di comunicazione in tempo reale alla propria applicazione che funziona su uno standard aperto.
Supporta video, voce e dati generici da inviare tra pari, consentendo agli sviluppatori di creare potenti soluzioni di comunicazione vocale e video.
La tecnologia è disponibile su tutti i browser moderni e sui client nativi per tutte le principali piattaforme.
Le tecnologie alla base di WebRTC sono implementate come standard web aperto e disponibili come API JavaScript regolari in tutti i principali browser. Il progetto WebRTC è open-source e supportato da Apple, Google, Microsoft e Mozilla, tra gli altri.
Poiché lavorare direttamente con WebRTC può essere un po' complesso, in nostro aiuto viene PeerJS.
Quest'ultimo prende l'implementazione di WebRTC nel browser e racchiude un'API coerente ed elegante attorno ad essa.
Inizialmente erano disponibili solo canali dati inaffidabili, poi l'evoluzione di PeerJS stesso (se configurato adeguatamente) ha portato all'utilizzato anche di canali dati affidabili.
Con PeerJS, identificare i peer è più semplice. Ogni peer viene identificato utilizzando nient'altro che un ID, ovvero una stringa che il peer può scegliere da solo o farne generare una da un server.

2.3 WebSocket vs WebRTC

Sebbene entrambi facciano parte della specifica HTML5, essi hanno ruoli differenti:

- i WebSocket → hanno lo scopo di abilitare la comunicazione bidirezionale tra un browser e un server Web;
- WebRTC → è pensato per offrire la comunicazione in tempo reale tra i browser (prevalentemente comunicazioni vocali e video).

Detto questo si potrebbe dire che WebRTC sostituisca i WebSocket, ma questa cosa non è possibile. Queste due tecnologie in realtà sono a stretto contatto tra di loro, perché WebRTC non ha segnali propri, cosa necessaria per aprire una connessione peer WebRTC e quindi fa affidamento a WebSocket sicuri.
Quindi ricapitolando, per connettere un canale dati WebRTC è necessario prima segnalare la connessione tra i due browser e per fare ciò abbiamo bisogno che comunichino in qualche modo attraverso un server web. Ciò si ottiene utilizzando un WebSocket sicuro o HTTPS. Quindi WebRTC non può davvero sostituire i WebSocket o agire in maniera autonoma.

3. **Tecnologie per lo sviluppo**

Per lo sviluppo del server è stato utilizzato “Node.js” con i seguenti moduli:

- Express.js: framework server per Node.js, usato per applicazioni web;
- Socket.io: libreria Javascript per applicazioni web in tempo reale;
- EJS: framework che permetterà di rendere e servire file HTML da server Express;
- UUID: protocollo per la generazione di identificatori univoci casuali;
- PeerJS: libreria che ci consente di utilizzare WebRTC (ne abbiamo parlato precedentemente nel capitolo 2.3).

3.1 Node.js

Node.js è un runtime di JavaScript costruito sul motore JavaScript V8 di Google Chrome, open source ed orientato agli eventi.
Node.js consente di utilizzare JavaScript per scrivere codice eseguibile lato server, cosa prima non possibile in quanto JavaScript nasce come linguaggio client interpretato da un browser.
L'architettura orientata agli eventi rende possibile l’I/O asincrono.
Questo design punta ad ottimizzare il throughput e la scalabilità nelle applicazioni web con molte operazioni I/O intensive, è inoltre ottimo per applicazioni web real-time.
Il modello di networking ad eventi è ritenuto più efficiente in situazioni critiche con un elevato traffico di rete, in quanto il programma può rimanere in uno stato di sleep fino alla notifica del sistema operativo del verificarsi di determinati eventi e solo dopo può eseguire delle funzioni di callback.

3.2 Express

Express è un framework per applicazioni web Node.js e viene installato come un qualunque altro modulo. È stato progettato per creare web application e API ed ormai definito il server framework standard per Node.js. Express.js fondamentalmente aiuta a gestire il back-end, dai percorsi, alla gestione delle richieste e delle viste.

3.3 Socket.io

Socket.io è una libreria Javascript per applicazioni web in tempo reale. Comprende una comunicazione bidirezionale real-time tra i web client e i server.
È formata da due parti:

- una libreria lato client che gira sul browser;
- una libreria lato server per Node.js.

Entrambi i componenti hanno la stessa struttura API. Come Node.js, è orientata agli eventi.
A differenza della libreria ufficiale WebSocket permette di inviare messaggi specificando un nome di un evento, aiuta la gestione dei client connessi senza mantenere una struttura apposita per tutte le connessioni.
Socket.io non è basato su WebSocket, ma utilizza questa tecnologia solo quando è disponibile.
Anche se, ormai, tutti i browser più popolari possiedono un supporto al protocollo WebSocket, grazie a questa libreria anche i browser non aggiornati o più datati, rendono possibile l’utilizzo della piattaforma.

3.4 EJS

EJS sta per “Embedded JavaScript”, ed è uno tra i motori di visualizzazione dei modelli più popolari per Node.js ed Express.js.
È uno strumento che ci consente di scrivere markup HTML, condito con i suoi tag o sintassi definiti che inseriranno variabili nell'output finale del modello o eseguiranno una logica di programmazione in fase di esecuzione prima di inviare l'HTML finale a il browser per la visualizzazione.
Perché EJS?

- è utile soprattutto ogni volta che si deve generare HTML con molto JavaScript.
- Se si ha a che fare con la generazione di contenuti dinamici o con qualcosa che ha a che fare con aggiornamenti in tempo reale, può ridurre notevolmente il caricamento del codice.

3.5 UUID

Gli UUID (Universally Unique IDentifier), conosciuti anche come GUID (Globally Unique IDentifier), secondo RFC 4122, sono identificatori progettati per fornire certe garanzie di unicità.
Gli UUID sono generalmente utilizzati per identificare informazioni che devono essere uniche all'interno di un sistema o di una rete. La loro unicità e la bassa probabilità di essere ripetuti li rende utili per essere chiavi associative nei database e identificatori in diversi ambiti.
Lo scopo di un UUID è quello di avere un identificatore universalmente unico. Ci sono generalmente due ragioni per usare gli UUID:

- Non si vuole che un database (o qualche altra autorità) controlli centralmente l'identità dei record;
- C'è la possibilità che più componenti possano generare indipendentemente un identificatore non unico.

Gli UUID sono abbastanza sicuri per quasi tutti gli scopi pratici.

4. **Ambiente di sviluppo**

Il progetto è stato sviluppato tramite Visual Studio Code, il quale dà la possibilità di collegarsi al proprio account GitHub in modo da tenere traccia delle varie versioni del programma.
Il progetto può essere trovato al seguente link:

https://github.com/Giacche18/MyVideoChatApp

Come prima cosa è stata creata una nuova repository pubblica e poi è stata clonata in locale, nella nuova directory generata è stato inizializzato il programma Node.js con il seguente comando:

```
npm init
```

Eseguendo questo comando, il gestore di pacchetti per Node.js (npm), chiederà informazioni riguardanti il nuovo progetto: *nome, descrizione, versione, autore, ecc...*
Con i dati ricevuti in input e letti dalla cartella .git , il programma crea automaticamente il file package.json, un file in formato JSON contenente vari metadati rilevanti per il progetto. Questo file viene utilizzato per fornire informazioni a npm che gli consentono di identificare il progetto e gestire le sue dipendenze.
Per installare i vari moduli è stato eseguito il comando:

```
npm i express ejs socket.io uuid peer
```

Così facendo npm, aggiorna il file package.json aggiungendo le dipendenze per il funzionamento del progetto, crea una cartella chiamata node-modules, dove sono presenti i sorgenti dei moduli installati e crea il file package-lock.json che descrive l'albero esatto che è stato generato, in modo che le installazioni successive siano in grado di generare alberi identici, indipendentemente dagli aggiornamenti di dipendenza intermedi.
Per una comodità di sviluppo è stato installato anche nodemon, uno strumento che aiuta a sviluppare applicazioni basate su node.js riavviando automaticamente l'applicazione quando vengono rilevate modifiche ai file nella directory e viene installata con il seguente comando:

```
npm i --save-dev nodemon
```

Infine, nel nostro package.json, aggiungeremo uno script per avviare il nostro server:

```
"script": {
     "start:dev": "nodemon server.js",
     "start": "node server.js"
},
```

In questo modo sarà più semplice avviare il programma normalmente o in modalità sviluppo con uno dei seguenti comandi:

```
npm i run start
npm i run dev:start
```

5. **Implementazione del programma**

5.1 Server Web

Il server è scritto in Node.js e sfrutta i moduli precedentemente descritti. Per prima cosa sono stati importati i moduli e funzioni sia quelli installati esternamente tramite npm, sia quelli scritti ad hoc per l'implementazione del progetto.
In seguito è stato creato:

- un oggetto *server*, utilizzando Express;
- un oggetto *PeerServer* (a cui si connette PeerJS) per mediare le connessioni di nuovi utenti. Quest'ultimo agisce solo come broker di connessione, ovvero funge da collegamento tra un client e un server o tra due o più client peer.

Dopo di che il nostro server express viene avviato in ascolto su una determinata porta. La scelta della porta avviene tramite un meccanismo che determina la presenza o meno di una variabile d'ambiente con l'apposito valore da attribuire alla porta, in caso di mancanza il server viene messo in ascolto sulla porta 3030.
Per utilizzare EJS in Express, dobbiamo configurare il nostro motore di modelli e possiamo farlo aggiungendo la seguente riga di codice nel file server.js:

```
app.set('view engine', 'ejs');
```

Una volta fatto questo creeremo la cartella *views* nella nostra directory (EJS è accessibile per impostazione predefinita nella cartella views). All'interno di questa cartella, aggiungiamo il nostro file EJS.
Per definire la cartella che contiene i nostri file statici, quali immagini, file CSS e file JavaScript è stata utilizzata la funzione middleware integrata in Express, il tutto aggiungendo la seguente riga di codice:

```
app.use(express.static('public'));
```

In questo modo Express ricerca i file relativi alla cartella statica.
Per la gestione delle connessioni è stato creato un oggetto di Socket.io, chiamato "*io*" e viene messo in ascolto per una nuova connessione.
Su un evento di connessione ("*connection*"), viene eseguita una funzione di callback, che ha come parametro un oggetto socket, l'uso principale di questo oggetto è quello di metterlo in ascolto per altri eventi. Ogni connessione ha un socket in ascolto differente, in modo tale da poter distinguere le comunicazioni, i possibili eventi sono:

- "*join-room*": generato da un nuovo client che si connette ad una stanza, il client invierà un oggetto con il suo id univoco (creato e impostato da PeerJS per poterlo identificare) e la stanza scelta (che se non specificata sarà una sequenza univoca di numeri e lettere creata dal modulo UUID"). La funzione di callback permette all'utente connesso di entrare nella stanza e notifica a tutti i socket collegati tranne quello su cui viene chiamato che qualcuno è entrato.
- "*message*": generato dall'invio di un nuovo messaggio da parte di un utente, il client invierà il messaggio in formato stringa, la funzione di callback manderà il messaggio a tutti gli utenti connessi alla medesima stanza tramite l'id della stanza.
- "*disconnect*": generato dalla disconnessione volontaria o meno di un client. La funzione di callback notifica a tutti i socket collegati tranne quello su cui viene chiamato che qualcuno si è disconnesso.

5.2 Client Web

La parte web comprende sostanzialmente una pagina web, ovvero la nostra video chat vera e propria. È stata sviluppata prevalentemente mediante l'uso del framework Bootstrap, che raccoglie strumenti per la creazione di siti e applicazioni web, quali CSS e JavaScript.
La pagina della video chat comprende due sezioni distinte, a sinistra è presente:

- una navbar nella parte alta della pagina, che ha funzione principalmente estetica e dove andiamo a riportare il nome del progetto e un logo;
- una barra di comandi nella parte bassa della pagina, al cui interno troviamo due bottoni: uno per mutare e smutare l'audio e uno per bloccare e sbloccare la riproduzione del video;
- infine nella parte compresa tra i due elementi che abbiamo appena citato abbiamo una griglia video all'interno della quale saranno visualizzati i video degli utenti collegati.

Nella parte destra, invece, è presente la chat con in alto i messaggi e in basso un campo input per l'invio di nuovi messaggi.
La pagina esegue il file *script.js* presente nella cartella js, questo file contiene le istruzioni e le funzioni per rendere la pagina dinamica.
Per prima cosa viene creata una costante socket, presa dall'API di Socket.io tramite il *path/socket.io/socket.io.js* richiesto dal file EJS. Con questa costante è possibile utilizzare la libreria per rimanere in ascolto o inviare eventi.
Successivamente è stato creato un nuovo elemento *Peer* che rappresenta l'utente e che ci permetterà, anche in questo caso, di rimanere in ascolto o inviare eventi con il nostro *PeerServer* (che abbiamo creato lato server).
A questo punto ci dedichiamo alla stesura del codice che ci permetterà di creare l'oggetto che contiene tutte le informazioni video e audio del client. Quest'ultimo è chiamato di default "*stream*" e lo si ottiene grazie alla funzione *navigator.mediaDevices.getUserMedia*, mentre per poter visualizzare queste informazioni sulla nostra pagina web viene eseguita la funzione *AddUserVideoStream* Quest'ultima non fa altro che prelevare le informazioni video dall'elemento *stream* creato precedentemente e salvarle all'interno di un elemento *video* che abbiamo creato in precedenza. Dopo di che la funzione rimane in attesa di un evento "*loadmetadata*", che rappresenta l'esatto momento in cui le informazioni video sono state caricate nella pagina web e quindi possiamo eseguire il video (con *video.play*()) e visualizzarlo nell'elemento HTML "*video-grid*".
L'esecuzione del codice lato client inizia nell'esatto momento in cui il client esegue per la prima volta l'applicazione e entra all'interno di una video chat room, scatenando l'evento "*open*", ciò su cui il nostro *Peer* rimane in ascolto, e quello che succederà è che il *Peer* raccoglierà tutte le informazioni WebRTC del client appena connesso e le riassumerà in un ID univoco, più semplice da utilizzare, che lo rappresenta. Questo ID verrà utilizzato per eseguire delle "*call*" ai vari utenti presenti all'interno della stessa room e metterli in contatto tramite una comunicazione *peer-to-peer* che permetterà all'utente di scambiare informazioni in tempo reale. Su questo evento viene eseguita una funzione di callback che non fa altro che comunicare al server che un nuovo utente si è collegato inviandogli il suo id e la stanza scelta.
In seguito viene messo in ascolto socket sui seguenti eventi:

- "*user-connected*": generato dal server per inviare le informazioni inerenti all'utente che si è collegato a quella determinata stanza. La funzione di callback esegue la funzione *connectToNewUser* (userId, stream) spiegata in seguito;
- "*createMessage*": generato dal server per notificare l'arrivo di un nuovo messaggio, da parte di un utente. La funzione di callback non fa altro che stampare il messaggio all'interno della nostra chat e esegue la funzione scrolltoBottom() spiegata in seguito;

- "*disconnect*": generato dal server per notificare che un utente si è disconnesso. La funzione di callback non fa altro che chiudere la *peer-call* tra gli utenti che erano collegati alla stessa room per poter successivamente rimuovere il video dell'utente che si è disconnesso.

Le funzioni elencate in precedenza sono:

- "*connectToNewUser* (userId, stream) → come parametro richiede l'id del client collegato alla nostra stessa room e l'oggetto stream (come spiegato precedentemente). A questo punto verrà eseguita la *peer-call*, tramite la quale inviamo all'altro client il nostro stream in modo che lui possa vederci e sentirci. Quindi creiamo un nuovo elemento "video" e rimaniamo in attesa dell'evento "*stream*" (ovvero aspettiamo che ci arrivino le informazioni dello stream dell'utente che abbiamo chiamato) e successivamente eseguiremo la funzione addVideoStream per visualizzare sul nostro browser l'altro utente connesso.
- "*scrolltoBottom*()" → non ha parametri di ingresso e quello che fa è semplicemente darci la possibilità di scorrere la chat messaggistica verso l'alto o verso il basso.

A questo punto abbiamo eseguito la call, tramite *connectToNewUser*, verso gli altri utenti ma ancora non saremo in grado di vederci a vicenda sul rispettivo browser (vedremo solo noi stessi). Questo perché dobbiamo rispondere alla call e questo viene fatto sempre dal Peer. Infatti quest'ultimo, dal momento in cui l'utente si è collegato all'applicazione per la prima volta, rimane in attesa dell'evento "*call*" e permette all'utente di rispondere e condividere il proprio stream.

La disconnessione invia automaticamente un evento disconnect al server.

6. Progetto online

Per mettere online la piattaforma è stato utilizzato il servizio di continuous delivery messo a disposizione gratuitamente da Heroku. Viene collegata alla propria app di Heroku un repository pubblico, l'app online viene automaticamente aggiornata ogni qualvolta venga effettuata una modifica sul branch master del repository collegato.
La piattaforma di Heroku automaticamente installa le dipendenze descritte nei file *package.json* e *package-lock.json* , così facendo non c'è bisogno di caricare nella repository i moduli, che potrebbero appesantire l'upload e il download.
Inoltre sempre nel file *package.json* sono descritti gli script di esecuzione ed Heroku può così eseguire il server senza problemi.
Come descritto in precedenza per la scelta della porta è stato implementato un meccanismo che rileva la presenza o meno di una variabile d'ambiente per la porta, Heroku dispone già di questa variabile "PORT" e non può essere scelta arbitrariamente.

L'idea di mettere il progetto online è nata grazie ad una limitazione della funzione *navigator.mediaDevices.getUserMedia* che, dopo l'aggiornamento di Chrome alla versione 47, richiede necessariamente una connessione sicura HTTPS per poter essere eseguita e permettere di divulgare informazioni video e audio in tempo reale attraverso la rete.
In Heroku, tutte le richieste arrivano all'applicazione come semplice http, ma lui le esegue con https (in quanto implementa questo protocollo) e quindi pur essendo gratuita offre un buon servizio in ambito sicurezza.

Il progetto è stato chiamato **keep-in-touch-02** ed è online sull'indirizzo:

https://keep-in-touch-02.herokuapp.com/

7. Conclusioni

Gli obiettivi predisposti sono stati raggiunti, l'uso di Node.js ha favorito lo studio sul paradigma di programmazione ad eventi, la libreria Socket.io ha facilitato l'utilizzo e la comprensione del protocollo WebSocket e infine il framework PeerJS ha permesso di utilizzare in modo più semplice la tecnologia WebRTC.
Grazie a Node.js e il framework Express è stato semplice ed intuitivo lo sviluppo del server web. Socket.io è un'ottima libreria per interfacciarsi alle comunicazioni real-time semplice e intuitiva, con un carico di studio non troppo elevato è stato possibile sviluppare un software in grado di gestire più connessioni.
L'introduzione di WebRTC ha aperto le porte a molti tipi di nuove applicazioni Web che possono essere eseguiti nel browser senza la necessità di plug-in aggiuntivi. Tuttavia, essendo una tecnologia relativamente nuova, pone ancora alcune sfide uniche e complesse.
L'utilizzo di PeerJS permette di affrontare queste sfide fornendo un'API elegante.
La piattaforma Heroku pur essendo gratuita offre un buon servizio, soprattutto per le connessioni sicure, in quanto implementa il protocollo HTTPS.