Alma Mater Studiorum · Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

WebRTC: studio della tecnologia e casi d'uso

Elaborato in:
Programmazione di reti

Relatore:
Prof. Andrea Piroddi

Presentata da:
Giovanni Maffi

Anno Accademico 2022/2023

Abstract

In questo documento viene analizzato WebRTC, una nuova tecnologia che rinnova le comunicazioni real-time tra più utenti tramite il solo utilizzo di un browser, permettendo di effettuare videochiamate senza dover scaricare programmi terzi. Il funzionamento di WebRTC è basato su API Javascript, tra cui RTCPeerConnectio, RTCDataChannel e MediaStream. Il suo utilizzo è stato di fondamentale importanza durante la pandemia COVID-19, permettendo a studenti e lavoratori di continuare le proprie attività da remoto. oltre a semplici videochiamate, può essere implementato anche in altri ambiti, come quello del controllo remoto di dispositivi, utile per esempio per monitorare il funzionamento di macchinari, oppure anche per la telemedicina, permettendo a un dottore di monitorare le condizioni dei propri pazienti a distanza. Riguardo a questo ultimo punto è stato sviluppato un progetto che permette di effettuare una videochiamata tra un dottore e un paziente.

"Sic Mundus Creatus Est"

Contents

Chapter 1

Introduzione

Negli ultimi 150 anni l'uomo ha fatto passi da gigante nel settore tecnologico/informatico con alcune invenzioni così importanti da cambiare il suo modo di vivere: basti pensare alla televisione presente in ogni casa, all'evoluzione dei computer e degli smartphone e, soprattutto, ad Internet che ha permesso all'uomo di informarsi su un qualsiasi argomento e di interagire con tutti in tempo reale. A tal proposito, la possibilità di comunicare in tempo reale con altri utenti è possibile grazie all'utilizzo di vari protocolli e tecnologie che, però, richiedono sempre l'utilizzo di plugin o software esterni. Nonostante ciò, negli ultimi anni è stata creata e sviluppata una nuova tecnologia che permette una comunicazione in tempo reale senza l'utilizzo di elementi terzi ed è sempre più discussa ed utilizzata: si tratta di Web Real-Time Communication (WebRTC).
Questo documento analizza questa tecnologia in tre capitoli: il primo analizza la storia delle comunicazioni real-time, dalla loro nascita fino alla creazione di WebRTC. Il secondo approfondisce sia la struttura di WebRTC, spiegando quali protocolli vengono utilizzati, come funzionano le API principali e il suo supporto su browser e smartphone, sia il suo funzionamento. Il terzo illustra le sue principali applicazioni in più campi, come quello della didattica a distanza o della monitorazione da remoto.
Infine è presente un quarto capitolo che propone un'applicazione WebRTC, che permette di effettuare una chiamata tra medico e paziente.

Chapter 2

Storia

2.1 La nascita della comunicazione in tempo reale

Nonostante WebRTC esista da poco più di 10 anni, la comunicazione a distanza nasce più di un secolo fa, con l'invenzione della telefonia. La nascita del telefono risale alla seconda metà del XIX secolo, anche se la paternità di tale invenzione è ancora oggi contestata (tra i tanti possibili inventori, i più accreditati sono Antonio Meucci, Innocenzo Manzetti, Johann Philipp Reis, Alexander Graham Bell e Charles Bourseul) [1]. Il suo utilizzo, però, si diffuse più tardi, intorno agli anni '20, ed è proprio in questo periodo che nascono i primi servizi di telecomunicazione. Col passare degli anni, i telefoni sono diventati sempre più sofisticati e diversificati. Verso la fine degli anni '80 si assiste ad una progressiva comparsa dei telefoni portatili, che porteranno sempre di più al disuso dei telefoni fissi [2].

Figure 2.1: Evoluzione della telefonia [55].

2.2 VoIP

La nascita di Internet ha cambiato notevolmente la comunicazione in tempo reale, permettendo lo streaming non solo di audio, ma anche di contenuto multimediale. La tecnologia più importante tra tutte è sicuramente VoIP.
VoIP (Voice over Internet Protocol) è un insieme di tecnologie che permette la comunicazione tra due o più utenti, sfruttando il protocollo IP per il trasporto dati.
Una tecnologia VoIP opera trasferendo segnali vocali tra indirizzi IP, il che significa che questi segnali devono essere codificati in pezzi di dati abbastanza piccoli da poter essere trasmessi. I campioni vocali del mittente vengono suddivisi in "pacchetti" vocali a cui sono fornite informazioni di instradamento e inviati al destinatario. I pacchetti vengono trasmessi uno ad uno, quindi si riformano il più vicino possibile allo stato originale, creando un'unica voce. Questo processo comprime il segnale vocale e quindi decomprime il segnale per il ricevitore [3].
Il primo utilizzo di VoIP risale al 1995 quando VocalTec Communication, un'azienda israeliana, brevetta il primo trasmettitore Voice over IP. Le basse velocità di connessione, però, significavano scarsa qualità audio, chiamate frequentemente interrotte e grave latenza e perdita di pacchetti. Questi problemi hanno impedito a VoIP di diffondersi in quegli anni; infatti la sua notorietà è avvenuta negli ultimi 20 anni, grazie alla creazione di tante applicazioni che utilizzano queste tecnologie per la comunicazione in tempo reale. Prima fra tutte è Skype: nato nel 2003 e pensato inizialmente come software per effettuare semplici chiamate, è stata la prima applicazione che, tramite l'utilizzo di voip, ha permesso la possibilità di attuare delle videochiamate; successivamente sono nate molte altre applicazioni simili e molte di queste hanno guadagnato una grande fama durante gli anni della pandemia di Covid [4].

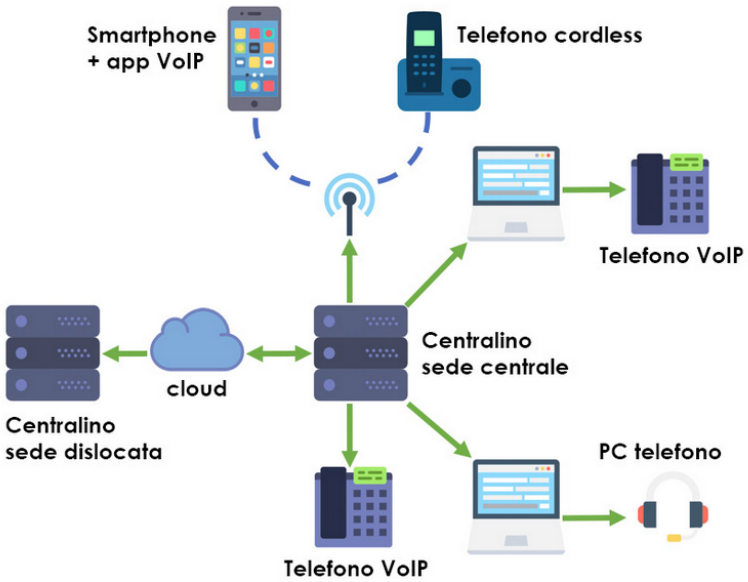

Figure 2.2: VoIP [56].

2.3 RTMP

Real-Time Messaging Protocol (RTMP) è un protocollo di comunicazione creato da Adobe per lo streaming di audio, video e dati su Internet. Creato da Macromedia (acquistato da Adobe), Il suo compito iniziale era quella di trasmettere i dati tra un server e Flash Player (un software per la visualizzazione di contenuti multimediali e l'esecuzione di applicazioni Internet avanzate, utilizzato per creare giochi o interi siti web). Nel corso degli anni, quest'ultimo è diventato sempre più conosciuto ed utilizzato, quindi anche RTMP aveva trovato largo impiego. Nonostante nel dicembre 2020 Adobe Flash Player sia stato dismesso, RTMP viene utilizzato ancora oggi come protocollo per lo streming [5].
La versione standard di RTMP prevede l'uso di TCP come protocollo di trasporto. Per prima cosa, client e server effettuano un handshake, scambiandosi tutte le informazioni riferenti alle regole della connessione (sistemi di criptazione, velocità, controllo errori ed altro). Una volta che il server conferma di aver ricevuto i blocchi di dati, può essere stabilita la connessione. A tale scopo, il client invia al server una richiesta di connessione sotto forma di action message (formato utilizzato per serializzare oggetti grafici come xml), dopodiché attende una conferma da parte del server. Quando l'ha ricevuta, il client può avviare lo streaming in tempo reale. Si stabilisce così una connessione stabile grazie alla quale i dati possono essere trasmessi in tempo reale. Per trasmetterli senza intoppi, RTMP suddivide lo stream in frammenti di dimensioni diverse. Queste possono essere "negoziate" tra client e server, altrimenti le dimensioni predefinite dei frammenti sono 64 byte per i dati audio e 128 byte per i dati video e la maggior parte degli altri tipi di dati [5].
Un programma che supporta questo protocollo è OBS, un software open source utilizzato per la registrazione di video, ma anche per lo streaming tramite RTMP [6].

2.4 Il problema

Come già detto, nel corso degli anni 2000 queste tecnologie (in particolare quelle VoIP) diventavano sempre più note ed utilizzate, ma il loro impiego non era possibile tramite web, quindi era necessario l'installazione di plugin (per esempio le prime versioni di Google Hangouts) o software (Skype). Tutto ciò fù risolto nel 2011, con la creazione di WebRTC.

2.5 WebRTC

Nel maggio 2010, Google acquistò GIPS (Global IP Solutions), una società di software VoIP e videoconferenza che aveva sviluppato molti componenti necessari per RTC, come vari codec (un software o un dispositivo che si occupa della codifica e/o decodifica digitale) e tecniche di cancellazione dell'eco. Successivamente, Google rese open source le tecnologie sviluppate da GIPS e si impegnò con gli organismi di standardizzazione pertinenti dell'IETF e del W3C per garantire il consenso del settore. Nel maggio 2011, Google rilasciò un progetto open source per la comunicazione in tempo reale basata su browser noto come WebRTC. Questo fu seguito da un lavoro per standardizzare i protocolli pertinenti nell'IETF e le API del browser nel W3C [7].
La prima implementazione avvenne sempre nel maggio del 2011, quando Ericsson Labs (multinazionale svedese operante nella fornitura di tecnologie e servizi di comunicazione, software e infrastrutture) ha creato la prima implementazione di WebRTC utilizzando una libreria WebKit modificata.
Nel corso degli anni i vari browser hanno iniziato a supportare l'utilizzo di WebRTC, ma l'impiego di questa tecnologia tra browser diversi è stato un problema difficile da risolvere, in particolare con tutti i browser compatibili con iOS, infatti il primo a supportare WebRTC è stato Safari nel 2017 [8].
Nell'ottobre 2011, il W3C ha pubblicato la sua prima bozza per la specifica. Le pietre miliari di WebRTC includono la prima videochiamata cross-browser (febbraio 2013), i primi trasferimenti di dati cross-browser (febbraio 2014) e, a partire da luglio 2014, Google Hangouts inizia ad utilizzare WebRTC [7].

Figure 2.3: Logo di WebRTC [57].

2.6 WebSocket

WebSocket è una tecnologia web nata nel 2008 che fornisce canali di comunicazione full-duplex attraverso una singola connessione TCP.
Il protocollo WebSocket consente l'interazione tra un browser Web (o un'altra applicazione client) e un server Web con un sovraccarico inferiore rispetto alle alternative half-duplex, facilitando il trasferimento dei dati in tempo reale da e verso il server. WebSocket, inoltre, permette un'interazione maggiore tra browser e server rispetto ad altri protocolli, facilitando la realizzazione di applicazione che forniscono contenuti in tempo reale. Tutto questo è possibile perché il server può spedire informazioni al browser senza aver ricevuto prima una richiesta dal client e consentendo il passaggio dei messaggi avanti e indietro mantenendo la connessione aperta [9].

2.6.1 WebRTC e WebSocket

Solitamente si può fare confusione tra queste due tecnologie. Entrambe possono essere utilizzate per lo stesso scopo, ma tra di loro ci sono varie differenze, tra le quali le principali sono [10]:

- WebSocket fornisce un protocollo di comunicazione computer client-server, mentre WebRTC offre un protocollo peer-to-peer e funzionalità di comunicazione per browser e app mobili.
- WebSocket è una scelta migliore quando l'integrità dei dati è fondamentale, in quanto si beneficia dell'affidabilità sottostante del TCP. D'altra parte, se la velocità è più importante e la perdita di alcuni pacchetti è accettabile, WebRTC su UDP è una scelta migliore.

Nonostante ciò, vedremo come WebSocket venga utilizzato per il funzionamento di WebRTC nella fase di signaling.

Chapter 3

Struttura e Funzionamento di WebRTC

3.1 Architettura

Come già detto, WebRTC offre agli sviluppatori di applicazioni Web la possibilità di scrivere app multimediali e in tempo reale sul Web, senza richiedere plug-in, download o installazioni. L'architettura complessiva la seguente [11]:

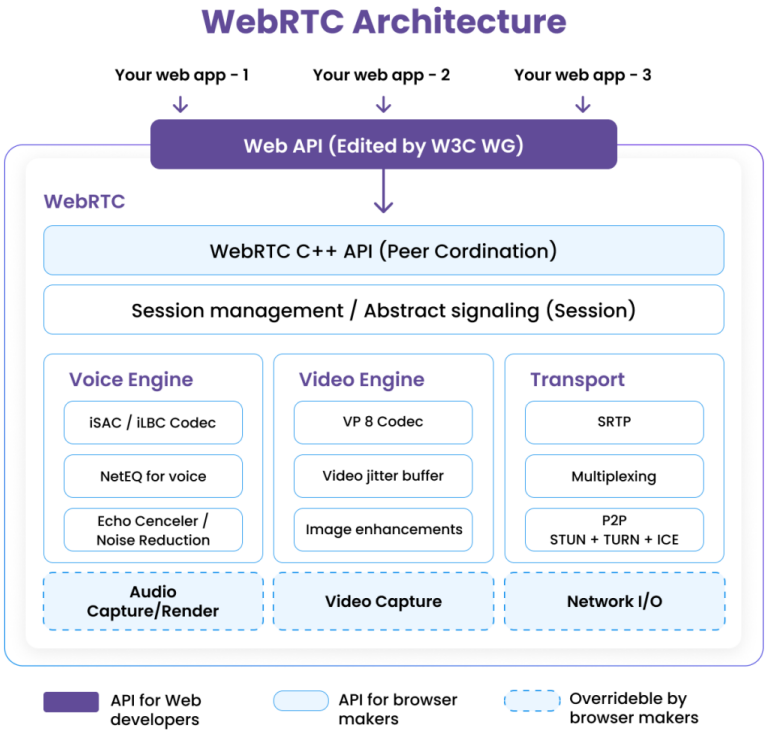

Figure 3.1: Architettura di WebRTC [58].

Come si nota dalla figura 3.1, l'architettura si può dividere in due livelli:

- il livello per gli sviluppatori di applicazioni Web, il Web API, l'insieme delle API che viene utilizzato da sviluppatori per la creazione di applicazioni simili a videochat basate sul Web. Alcuni esempi di API sono RTCPeerConnection, RTCDataChannel, e MediaStream.
- Il livello API per gli sviluppatori di browser, in cui è presente l'API WebRTC C++, consente ai produttori di browser di implementare facilmente la proposta di Web API. Qui sono presenti anche delle API sovrascrivibili.

Il livello transport gestisce l'ordine e la perdita dei pacchetti e si occupa di stabilire e mantenere connessioni tra reti di diverso tipo; per farlo, vengono impiegate le tecnologie STUN, TURN e ICE (di queste se ne parlerà in seguito).
I componenti di voce e video sono strutture responsabili del trasferimento di stream audio e video tramite algoritmi codec. Esiste un'enorme quantità di codec diversi; alcuni di questi sono già presenti all'interno di WebRTC, come per esempio iSAC (un codec vocale a banda larga) per l'audio e VP8 (appendice A) per il video. Quando due browser si connettono insieme, scelgono il codec supportato più ottimale tra due utenti. Fortunatamente, WebRTC esegue la maggior parte della codifica dietro le quinte.
L'architettura di WebRTC dal lato client server ha 2 modelli: il primo, il più utilizzato, è quello a triangolo, dove i due dispositivi presenti eseguono la stessa applicazione web. L'API RTCPeerConnection permette di creare e configurare una connessione peer-to-peer tra i due dispositivi. La segnalazione viene eseguita solitamente tramite HTTP o WebSocket [12].

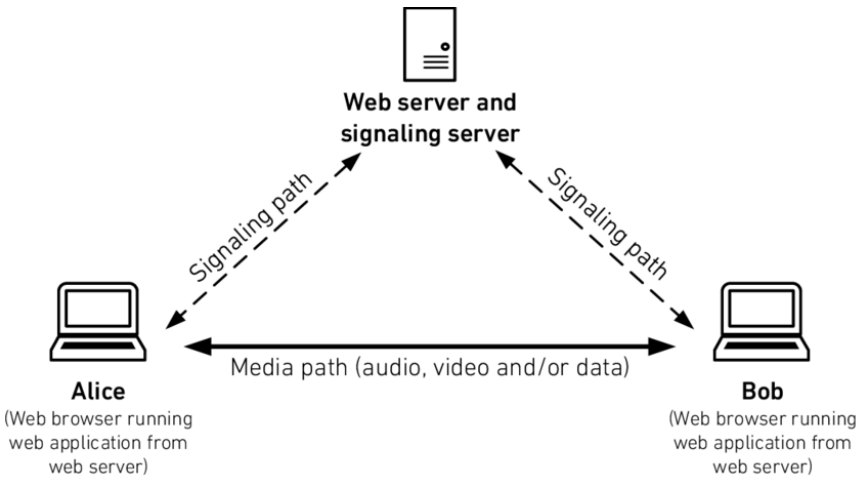

Figure 3.2: Modello triangolare [59].

Esiste anche un altro modello, ispirato a quello trapezoidale di SIP (Session Initial Protocol), dove i client eseguono un'applicazione web da server diversi. Questo modello viene utilizzato meno perché rispetto a quello a triangolo consente agli sviluppatori Web una minore flessibilità nella gestione delle connessioni degli utenti [12].

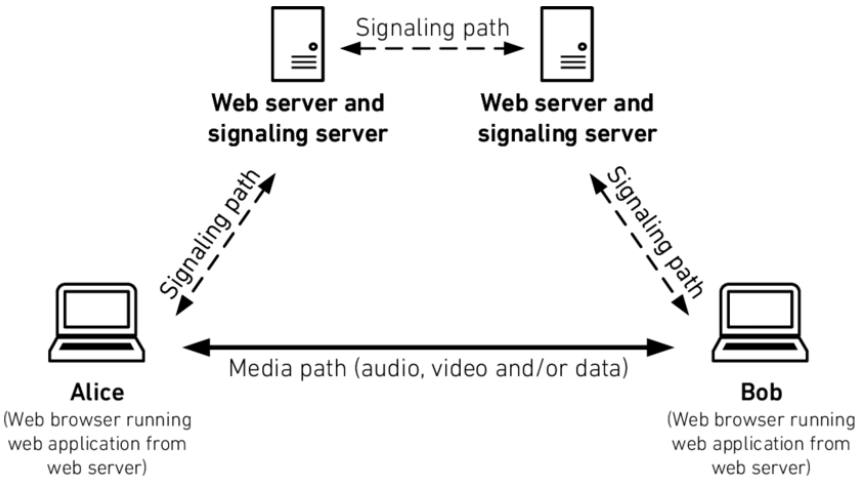

Figure 3.3: Modello trapezoidale [60].

3.2 API

Nel contesto di WebRTC, le API rappresentano la componente principale per il funzionamento di questa tecnologia, sia per quanto riguarda la connessione tra utenti, sia per quanto riguarda il riconoscimento di dispositivi video ed audio utilizzati.
Ma cos'è un API? Un Application Programming Interface è un'interfaccia tra due o più software che permette di farli comunicare tra loro. Un documento o uno standard che descrive come creare o utilizzare tale connessione o interfaccia è chiamato API specification. Si dice che un sistema informatico che soddisfa questo standard implementi un'API. Il termine API può riferirsi alla specifica o all'implementazione. Un'API è simile a una GUI (Graphical User Interface) con una grande differenza: un'API aiuta gli sviluppatori di software ad accedere agli strumenti web mentre una GUI aiuta a rendere un programma più facile da capire per gli utenti [13].
Un'API è spesso composta da diverse parti che fungono da strumenti o servizi disponibili per il programmatore. Si dice che un programma o un programmatore che utilizza una di queste parti chiami quella parte dell'API. Le chiamate che costituiscono l'API sono anche note come subroutine, metodi, richieste o endpoint. Un'API può essere creata su misura per una particolare coppia di sistemi o può essere uno standard condiviso che consente l'interoperabilità tra molti sistemi [14].
L'obbiettivo di un API è quello di semplificare la programmazione, nascondendo i dettagli interni di come funziona un sistema, esponendo solo quelle parti che un programmatore troverà utili e mantenendole coerenti anche se i dettagli interni in seguito cambiano; per esempio, un'API per l'input/output di file potrebbe fornire allo sviluppatore una funzione che copia un file da una posizione a un'altra senza richiedere che lo sviluppatore comprenda le operazioni del file system che si verificano dietro le quinte.
L'utilizzo di API può essere fatto in diversi scenari, come nei linguaggi di programmazione, librerie software, sistemi operativi e hardware per computer. Nonostante la loro nascita ed inizio utilizzo siano datati (la loro origine risale agli anni '40, ma inizieranno ad essere utilizzate solo negli anni '60), ancora oggi sono molto impiegate e di vitale importanza per il funzionamento di programmi che le utilizzano (come WebRTC) [14].

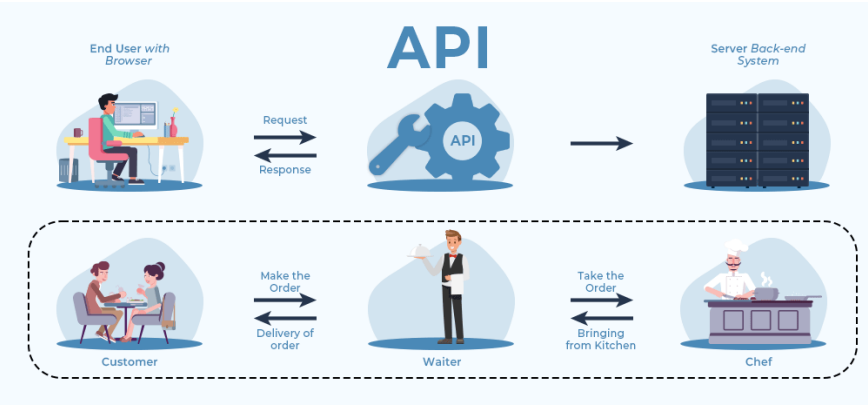

Figure 3.4: funzionamento di un API [61].

La quantità di API esistenti e il loro funzionamento diverso ha portato ad una suddivisione di queste in diverse categorie; La maggior parte delle API utilizzate per il funzionamento di WebRTC appartengono alla categoria delle Web API, che consente la comunicazione tra computer collegati ad Internet ed accessibili utilizzando il protocollo HTTP [15].
Come già detto, le API rappresentano la componente principale per il funzionamento di WebRTC, e questa tecnologia ne utilizza in gran numero, ma le fondamentali sono: RTCPeerConnection, RTCDataChannel e MediaStream.

3.2.1 RTCPeerConnection

L'interfaccia RTCPeerConnection rappresenta la componente fondamentale per impostare una connessione peer tra due o più utenti. L'attività principale dell'oggetto RTCPeerConnection è fornire metodi per connettersi ad altri utenti, mantenere e monitorare la connessione e chiuderla quando non è più necessaria. Gestisce anche una connessione UDP con un altro utente [16].
RTCpeerConnection è un oggetto Javascript, che può essere creato nel seguente modo:

```
var pc = new RTCPeerConnection(config);
```

il nuovo oggetto RTCPeerConnection rappresenta una connessione tra un dispositivo locale e un peer remoto. Al costruttore, come in questo caso, può essere passato il parametro config, un oggetto che fornisce opzioni per configurare la nuova connessione.
Tra i tanti metodi che RTCPeerConnection possiede ci sono [16]:

- *createOffer*: crea una offerta SDP con lo scopo di iniziare una nuova connessione WebRTC con un altro utente.
- *createAnswer*: crea una risposta SDP a un'offerta ricevuta da un peer remoto durante la negoziazione offerta/risposta di una connessione WebRTC.
- *addStream*: aggiunge un MediaStream come sorgente locale di audio o video. Questo metodo è deprecato e obsoleto, quindi potrebbe non funzionare su alcuni browser; per questo motivo è meglio utilizzare *addTrack()* una volta per ogni traccia che si desideri inviare al peer remoto.

3.2.2 RTCDataChannel

RTCDataChannel rappresenta un canale di rete che viene utilizzato per il trasferimento di dati tramite una connessione peer-to-peer bidirezionale. Per questo motivo, ogni canale dati è associato a un RTCPeerConnection e ogni connessione peer può avere fino a un massimo di 65.534 canali dati (il limite effettivo può variare da browser a browser). Come vedremo successivamente, i canali dati utilizzano il protocollo SCTP, che permette di stabilire una connessione TCP o UDP in base al tipo di traffico che si desidera spedire.
Per creare un canale dati e chiedere ad un utente di unirsi, si deve utilizzare il metodo *createDataChannel()* di RTCPeerConnection. Questo crea un nuovo canale connesso al peer remoto sul quale è possibile trasferire qualsiasi tipo di dati (a differenza di media stream, dove si può solo trasmettere video ed immagini). *createDataChannel()* ha due parametri, label (campo obbligatorio) che corrisponde al nome del canale dati, e options (campo opzionale), il parametro che fornisce delle opzioni di configurazione per il canale. l'utente invitato a scambiare dati riceve un evento datachannel per informarlo che il canale dati è stato aggiunto alla connessione [17].
Un esempio di codice di *createDataChannel()* potrebbe essere il seguente:

```
[code]
//establishing peer connection
var peerConn = new RTCPeerConnection();
//...
//end of establishing peer connection
//here we can start sending direct messages to another peer
var dataChannel = peerConnection.createDataChannel("myChannel",
dataChannelOptions);
[/code]
```

3.2.3 MediaStream

MediaStream è un API che viene messa a disposizione dai browser agli sviluppatori per accedere agli stream (insieme di tracce audio/video) tramite i dispositivi dell'utente. Questa API ha tre funzionalità principali [12]:

- Fornisce a uno sviluppatore l'accesso a un oggetto stream.
- Gestisce la selezione dei dispositivi input dell'utente nel caso in cui questo disponga di più telecamere o microfoni sul proprio dispositivo.
- Fornisce un livello di sicurezza attraverso richieste di autorizzazioni all'utente, effettuate prima che un'applicazione Web possa iniziare a recuperare uno stream.

Un oggetto MediaStream si ottiene in due modi: creandolo tramite il suo costruttore oppure richiamando un metodo che lo restituisce. Il costruttore crea un nuovo oggetto MediaStream che può essere uno stream vuoto(nessun parametro), uno stream basato su un altro stream esistente passato come parametro oppure uno stream che contiene un array specifico di tracce (di tipo MediaStreamTrack).
Il metodo più importante che crea un Mediastream è *getUserMedia()* di MediaDevices. Quello che fa questo metodo è richiedere l'autorizzazione all'utente per utilizzare i dispositivi di input, creando un MediaStream che contiene le tracce per i tipi di media richiesti. Un Altro metodo molto utilizzato che restituisce un MediaStream è *getDisplayMedia()*, che, come *getUserMedia()*, richiede all'utente l'autorizzazione per la condivisione del suo schermo [18].

3.2.4 altre API

Oltre alle API citate, ne esistono molte altre, ognuna con il proprio compito. Alcune di queste sono [19]:

- MediaDevices: un'API che fornisce l'accesso ai dispositivi di input multimediali collegati come fotocamere, microfoni e condivisione dello schermo.
- MediaStreamTrack: rappresenta una singola traccia multimediale all'interno di uno stream; in genere si tratta di tracce audio o video, ma possono esistere anche altri tipi di tracce.
- RTCDataChannelEvent: rappresenta un evento correlato ad un RTCDataChannel.
- RTCIceTransport: API che fornisce informazioni sul livello di trasporto ICE utilizzato per l'invio ed il ricevimento dei dati. Viene adottata se è necessario accedere alle informazioni sullo stato della connessione.
- RTCSessionDescription: Descrive un'estremità di una connessione e come è configurata.

3.3 Protocolli

La pila protocollare di WebRTC è la seguente:

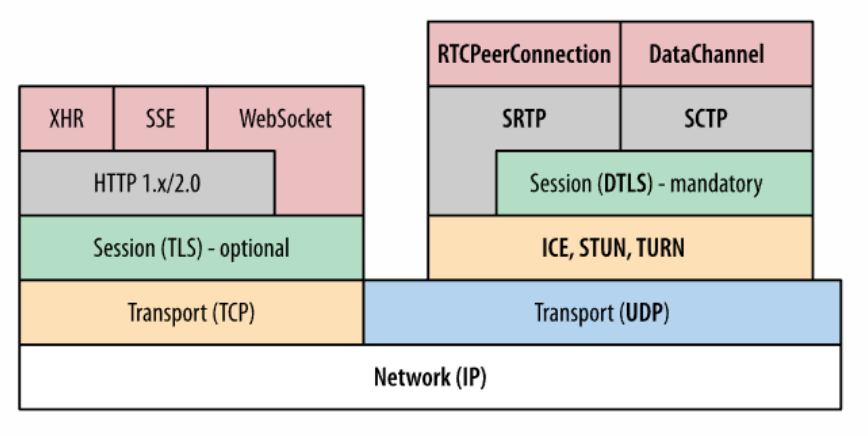

Figure 3.5: Pila protocollare di WebRTC [62].

3.3.1 Trasporto

Come si può notare, WebRTC può utilizzare sia UDP o TCP a livello di trasporto, ma, a differenza della maggior parte delle applicazioni web, WebRTC utilizza quasi sempre UDP per la spedizione di dati. Perché questo? Come già detto, TCP è più utilizzato perché è più affidabile rispetto a UDP. Questo perché nello spedire dati, TCP garantisce che [20]:

- Ogni dato spedito viene marcato come consegnato.
- Ogni dato che non arriva al mittente viene rispedito e la spedizione di messaggi viene temporaneamente disattivata.
- Ogni dato è unico senza duplicati dalla parte del mittente.
- Tutti i dati spediti arrivino in ordine.

per questi motivi TCP è adottato dalle maggior parte delle applicazioni Web. Nel richiedere una pagina HTLM, per esempio, tutti i dati devono arrivare nel corretto ordine. Ma questa applicazione non è adatta a tutti gli scenari; infatti le applicazioni per la trasmissione di video e audio richiedono una connessione più veloce. Ed è per questo che UDP è utilizzato. Questo protocollo, a differenza di TCP, non garantisce il corretto ordine dei dati spediti e non da informazione sullo stato di ogni pacchetto, ma la velocità di spedizione di dati è nettamente superiore. Inoltre, nelle applicazioni di streaming video o audio dove vengono spediti un determinato numero di frame al secondo, è più importante l'invio dell'ultimo frame rispetto all'invio di ogni singolo frame. Quindi è possibile perdere qualche frame senza che l'utente se ne accorga perché li rimpiazzerà con ciò che vede. Questo è un altro motivo per cui UDP è più utilizzato [12].
L'unico caso in cui viene utilizzato TCP nelle applicazioni WebRTC è quello in cui UDP non è disponibile, per esempio quando è bloccato dal firewall [21].

3.3.2 Instradamento

Quando si vuole effettuare una connessione tra due utenti è necessario creare una ”strada” tra questi due. Questo lavoro può essere complesso perché è probabile che le reti dei due utenti abbiano diversi livelli di controllo legati alla sicurezza. Esistono però tecnologie utilizzate per trovare un percorso chiaro verso un altro utente, tra cui [12]:

- STUN (Session Traversal Utilities for NAT).
- TURN (Traversal Using Relays around NAT).

STUN è un insieme standardizzato di metodi utilizzato per l'attraversamento di gateway NAT (Appendice B), permettendo agli host di conoscere i propri indirizzi IP che servono, nel contesto di WebRTC, per effettuare comunicazioni peer to peer attraverso Internet [22]. Per farlo, un host invia una richiesta al server STUN per ottenere l'IP pubblico e le informazioni sulle porte; di conseguenza il server STUN recupera tali informazioni:

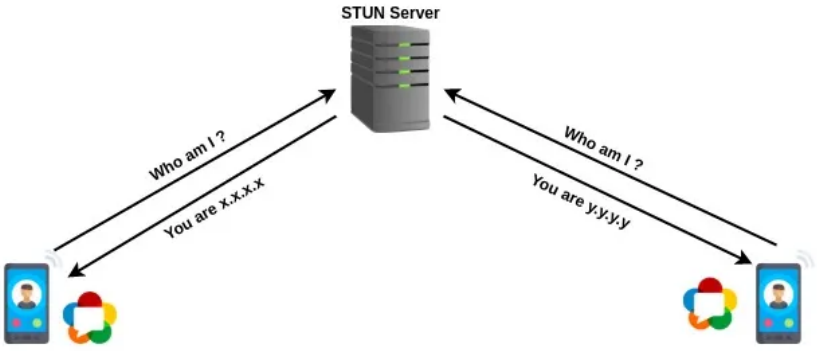

Figure 3.6: Funzionamento di STUN [63].

A volte, però, STUN potrebbe non essere utilizzabile, come nel caso di NAT simmetrici che, a causa delle loro restrizioni, non permettono a STUN di funzionare. In questi casi viene utilizzato TURN.
TURN consente a un server di inoltrare pacchetti di dati (media nel caso di WebRTC) tra peer attraverso un server TURN intermedio. I peer possono instradare i media al server TURN che li trasmetterà continuamente all'altro peer. Dal momento che il server inoltra tutti i media in continuazione, questa procedura richiede uno sforzo piuttosto costoso. Questo è il motivo per cui TURN viene utilizzato quando non sono possibili connessioni dirette e non ci sono altre alternative [23].

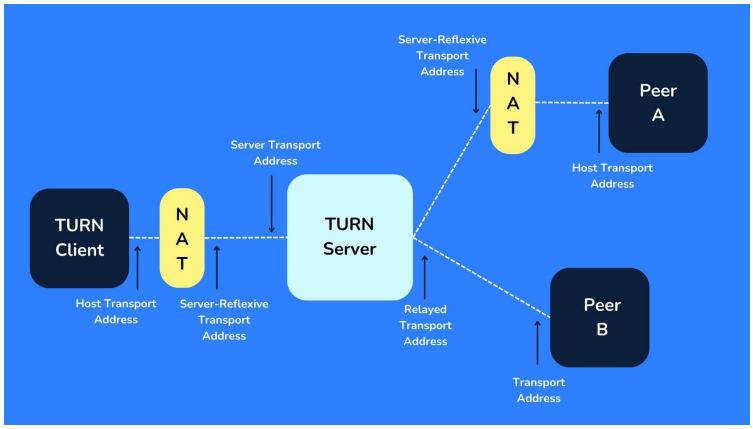

Figure 3.7: Funzionamento di TURN [64].

Per trovare il miglior percorso, WebRTC utilizza ICE (Interactive Connectivity Establishment), una tecnologia che permette di utilizzare STUN e TURN assieme. ICE utilizza queste due tecnologie per creare la migliore connessione peer to peer tra gli utenti, trovando e verificando in ordine una serie di indirizzi che funzionino per tutti. Al suo avvio, ICE non conosce la rete di ciascun utente, quindi la prima cosa che fa è attraversare una serie di fasi in modo incrementale per scoprire come è configurata la rete di ciascun client, utilizzando un diverso insieme di tecnologie.
STUN e TURN sono usati per trovare ogni candidato ICE, un elemento creato dai peer che descrive il routing e i protocolli necessari affinché WebRTC sia in grado di comunicare con un dispositivo remoto. ICE utilizzerà il server STUN per trovare un IP esterno. Se la connessione fallisce, tenterà di utilizzare il server TURN. Quando si avvia una connessione peer WebRTC, in genere viene proposto un numero di candidati da ciascuna estremità della connessione, fino a quando essi non concordano reciprocamente su quello che ritengono più efficiente. WebRTC utilizza quindi i dettagli di quel candidato per avviare la connessione [24].

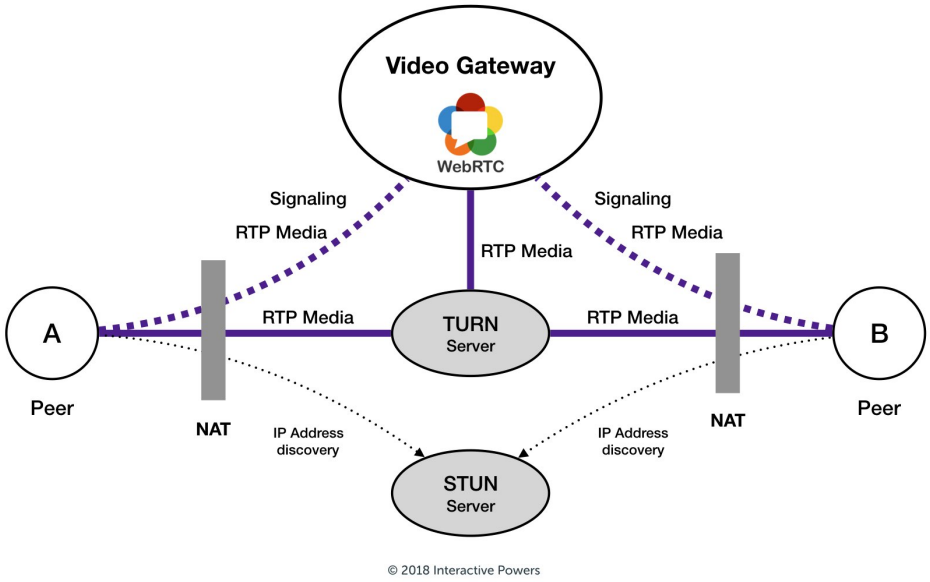

Figure 3.8: Funzionamento di ICE [62].

3.3.3 altri protocolli

SDP (Session Description Protocol) è uno standard per descrivere informazioni come il tipo di dati inviati, il protocollo di trasferimento utilizzato, l'indirizzo IP, la porta degli utenti e tutte le altre informazioni che permettono ai peer di capirsi una volta che i dati vengono trasferiti. Quindi SDP è un protocollo che descrive il formato di dati che i peer si scambiano per descrivere la connessione. WebRTC utilizza questo protocollo per negoziare i parametri della sessione durante la fase di signaling [25].
SCTP (Stream Control Transmission Protocol) è un protocollo di trasporto che viene utilizzato per inviare rapidamente dati attraverso i Data Channel. Quando un applicazione WebRTC utilizza un canale dati in base alle esigenze può utilizzare UDP o TCP. Per questo motivo SCTP viene utilizzato da WebRTC, perché ha caratteristiche che soddisfano sia UDP che TCP, come la protezione del livello di trasporto effettuata da DTLS, la possibilità, durante la spedizione, di poter scomporre e riassemblare i dati dalla parte del destinatario, poter disattivare l'affidabilità e la spedizione ordinata, garantendo la latenza necessaria per i sistemi in tempo reale [12].
WebRTC utilizza RTP (Real time protocol) per scambiare contenuti multimediali crittografati tramite SRTP. RTP è un protocollo che fornisce gli strumenti necessari per implementare lo streaming in tempo reale. La cosa più importante di questo protocollo è che offre flessibilità allo sviluppatore, consentendogli di gestire la latenza, la perdita di pacchetti e la congestione [26].

3.4 Funzionamento

3.4.1 Signaling

Prima di poter iniziare una comunicazione tramite WebRTC, gli utenti interessati devono effettuare uno scambio di informazioni; questo processo si chiama signaling. Durante questa fase, lo scambio di informazioni determina i protocolli di comunicazione, i canali, i codec, i formati multimediali e il metodo di trasferimento dei dati, nonché qualsiasi informazione di instradamento richiesta. Nelle specifiche di WebRTC non è descritto uno standard da seguire per effettuare il signaling e qualsiasi utente può effettuarlo in modo differente. Un esempio di signaling potrebbe avvenire in questo modo [12]:

- Viene creata una lista dei potenziali candidati alla connessione peer.
- un utente sceglie un'altra persona con cui stabilire una connessione.
- Il signaling layer avverte l'altro utente che qualcuno sta provando a connettersi a lui. A questo punto può accettare o rifiutare.
- Se accetta, l'utente chiamante viene notificato.
- Quest'ultimo inizializza una RTCPeerConnection con l'altro utente.
- Entrambi gli utenti si scambiano informazioni riguardanti software e hardware attraverso il signaling server.
- Entrambi si scambiano informazioni sulla posizione.
- Fatto tutto ciò la connessione può iniziare, a meno che non fallisca.

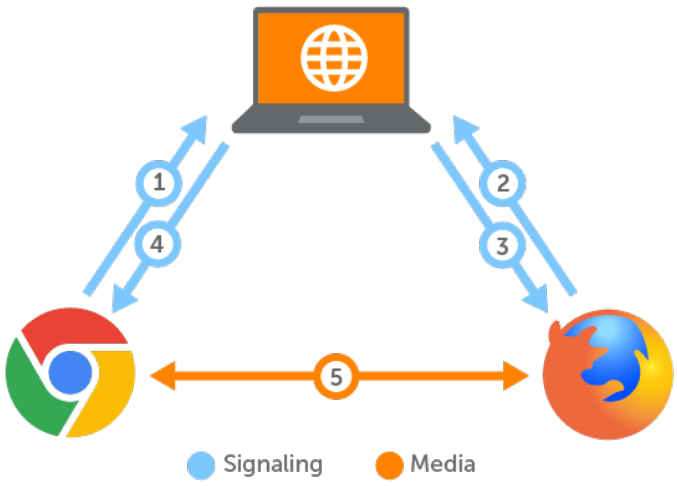

Figure 3.9: Esempio di Signaling [65].

3.4.2 Signaling con WebSocket

Un altro esempio di signaling può essere fatto con uno dei meccanismi più utilizzati per effettuare questo processo: il protocollo WebSocket, che consente ai peer di scambiare informazioni sulla connessione in tempo reale tramite un server WebSocket. Per realizzare un servizio di questo tipo è necessario stabilire una connessione bidirezionale WebSocket tra uno o più browser e un server. Successivamente si passa all'integrazione del servizio WebSocket nel server HTTP, aprendo la porta WebSocket e mettendolo in ascolto. Nel browser, è necessario creare un oggetto WebSocket in base all'indirizzo e alla porta del server. La connessione WebSocket verrà creata dopo l'handshake. Effettuati tutti questi passaggi, il browser e il server sono in grado di scambiarsi informazioni [27]:

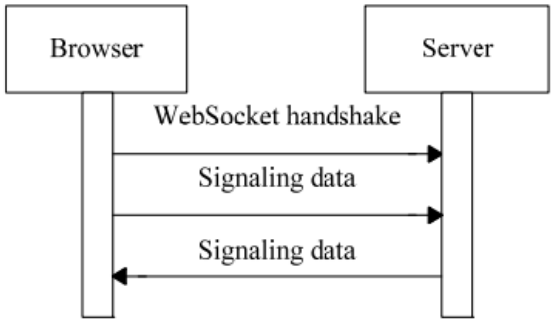

Figure 3.10: Esempio di Signaling con Websocket.

Una volta stabilita la connessione tra browser e server segue l'identificazione tra utenti, necessaria per lo scambio di informazioni di negoziazione. Per farlo possono essere utilizzate le Rooms di socket.io, un meccanismo che in questo contesto ha lo scopo di ottenere l'identificazione tra gli utenti. Il processo di identificazione tra due peer è il seguente [27]:

- Un utente A visita l'indirizzo di una Room e stabilisce una connessione WebSocket con il server.
- Successivamente, il browser spedisce un messaggio (join_in) al server, il cui contenuto descrive l'identificazione della room.
- Una volta ricevuto il messaggio, il server verifica se esiste una room con lo stesso nome del messaggio di A. In caso negativo, ne crea una nuova.
- Un utente B che vuole comunicare con A si collega all'indirizzo della stessa room a cui si è collegato l'utente A ed effettua lo stesso procedimento visto al primo punto.
- ottenuto il messaggio di B, il server genera dei SocketID diversi per ogni utente e li spedisce in broadcast ad ogni peer.

Questi ID servono per identificare ogni utente. Se A desidera negoziare i parametri per una connessione WebRTC con B, deve solo inviare il messaggio di negoziazione e il socketID dell'utente B al server. Questo selezionerà l'utente B tramite socketID e inoltrerà il messaggio dall'utente A. La risposta dall'utente B all'utente A segue la stessa procedura.

3.4.3 Connessione

Una volta scambiati i parametri per stabilire la connessione, intervengono le API.
La prima che viene utilizzata è MediaStream che con *getUserMedia()* richiede l'autorizzazione per accedere agli stream audio e video dell'utente tramite microfono e webcam, permettendo di selezionare quali input utilizzare, nel caso l'utente ne abbia più di uno.
Dopo aver ottenuto un flusso multimediale tramite il metodo *getUserMedia()*, il passaggio successivo di un'app WebRTC è quello di trasmettere i dati in tempo reale a un altro browser. Prima però è necessario stabilire una connessione tra due (o più) utenti. Questo è il compito principale di RTCPeerConnection. Supponiamo che due utenti vogliano comunicare tra loro; per farlo, l'utente chiamante crea un oggetto RTCPeerConnection per stabilire la connessione e successivamente viene creata un'offerta ed estesa all'altro browser con il metodo *createOffer()*. Questa offerta enumera i potenziali codec, i metodi di crittografia, i possibili ICE e altre informazioni iniziali disponibili per una sessione WebRTC. Questo processo utilizza SDP. Dopo che questa descrizione è stata generata, viene inviata al peer con la quale si vuole comunicare che, a sua volta, genera una risposta SDP con il metodo *createAnswer()* (la risposta contiene informazioni su qualsiasi supporto già collegato alla sessione, codec e opzioni supportate dal browser e qualsiasi candidato ICE già raccolto) e la spedisce al chiamante. Quando la risposta arriva a quest'ultimo, la connessione viene stabilita [16].
L'ultima operazione da effettuare è trasmettere dati in tempo reale. Questo compito è effettuato da RTCDataChannel. Grazie a questa API è possibile creare canali dati associati a una RTCPeerConnection semplicemente richiamando il metodo *createDataChannel()*. Fatto ciò è possibile spedire messaggi tramite il protocollo SCTP che saranno crittografati con DTLS [28].

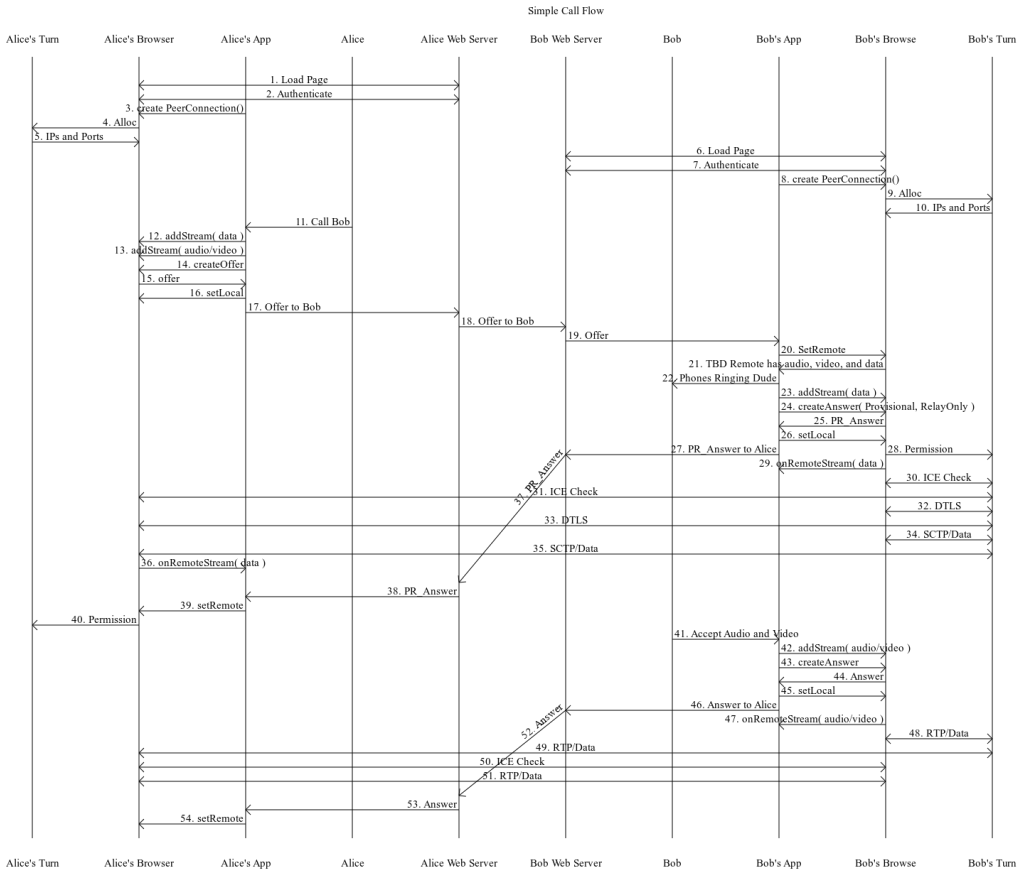

Figure 3.11: Un possibile esempio di chiamata tra due utenti [66].

3.5 Supporto

Alla fine del 2018 tutti i principali browser e sistemi operativi per dispositivi mobili hanno cominciato a supportare WebRTC. Nonostante ciò è possibile che alcune operazioni possano funzionare erroneamente o non funzionare completamente. Un esempio è l'utilizzo di Microsoft Teams su Firefox: l'app funziona, ma non completamente, perché su questo browser non sono disponibili alcune funzioni che su Chrome, per esempio, sono supportate [29]. Di seguito si analizzerà il supporto browser e mobile di WebRTC.

3.5.1 Browser

Il primo browser che ha supportato WebRTC è ovviamente Chrome (in quanto sono entrambi prodotti da Google) con la versione 23 rilasciata nel novembre del 2012. Solo su Chrome, inoltre, è possibile visualizzare alcune informazioni dei flussi audio e video, sia inviati che ricevuti, in esecuzione su una comunicazione, come i pacchetti persi, quanti bit e pacchetti vengono ricevuti o spediti al secondo [30].
Nel giro di un anno, anche Firefox e Opera hanno rilasciato le prime versioni con supporto WebRTC. Nello stesso periodo gli sviluppatori di Chrome e Firefox iniziarono a lavorare assieme per risolvere i possibili problemi che non permettevano la comunicazione tramite questi due browser [12].
Il supporto su Safari arrivò molto più tardi, con Safari 11 nel 2017. Questo ritardo è dato da due possibili motivazioni: la prima è la natura di Apple, che rende complesso lo sviluppo e l'uso di tecnologie non prodotte da Apple stessa; la stessa cosa si verifica su iOS. La seconda motivazione era la minaccia economica che costituiva WebRTC per le app di comunicazione real-time sull'Apple strore, come Facetime [31].
Come si nota in figura, l'unico browser che non supporta WebRTC è Internet Explorer in quanto nel corso degli anni è stato sempre meno utilizzato fino a non essere più supportato da Microsoft stessa nel febbraio del 2022, lasciando il posto ad Edge, che ha iniziato a supportate WebRTC nel 2020.

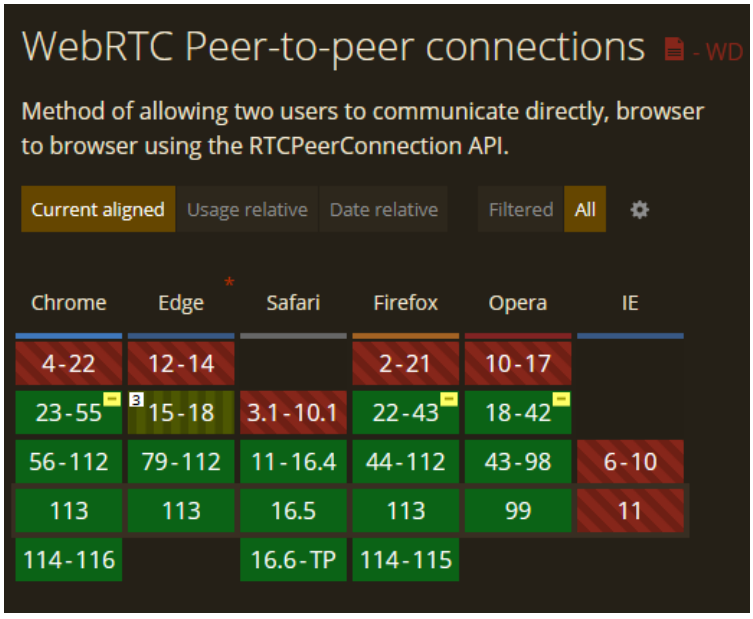

Figure 3.12: Supporto dell'API RTCPeerConnection su Browser [30].

3.5.2 Mobile

Quando si parla di smartphone il supporto per WebRTC è molto più complesso da realizzare rispetto a quello su computer. Per quest'ultimi abbiamo visto che per comunicare è necessario utilizzare un browser che supporti WebRTC. Su smartphone invece anche il sistema operativo deve supportare questa tecnologia, motivo per il quale l'utilizzo di WebRTC su mobile è tardato.
Nel 2014, Google ha rilasciato Android 5 (conosciuto anche come Lollipop), la prima versione che supporta le tre componenti principali di WebRTC: RTCPeerConnection, MediaStream e RTCDataChannel. I browser per mobile che supportano meglio WebRTC son Chrome, Firefox e Opera [30].
Il supporto su iOS è arrivato con iOS 11 nel 2017, assieme a safari, ma nonostante WebRTC sia supportato da diversi anni, è stato più volte segnalata la presenza di numerosi bug che rendono quasi inutilizzabile questa tecnologia su dispositivi iOS [32].

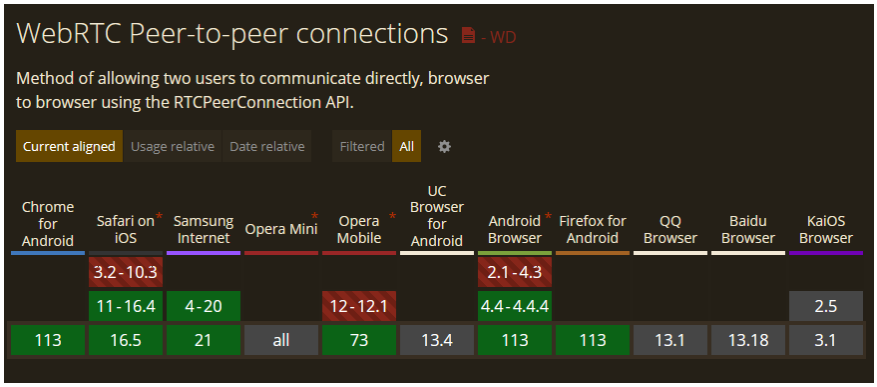

Figure 3.13: Supporto dell'API RTCPeerConnection su Mobile [30].

3.6 Sicurezza

WebRTC utilizza due protocolli per la sicurezza, DTLS (Datagram Transport Layer Security) e SRTP (Secure Real-time Transport Protocol).
DTLS è un protocollo di comunicazione che fornisce sicurezza alle applicazioni basate su datagrammi (come UDP e SCTP) consentendo agli utenti di comunicare tra loro prevenendo l'intercettazione, la manomissione o la contraffazione dei messaggi. L'obbiettivo di DTLS è quello di realizzare TLS (appendice E) per datagrammi. Il trasporto di datagrammi non richiede né fornisce una consegna affidabile o ordinata dei dati. Il protocollo DTLS conserva questa proprietà per i dati dell'applicazione. Per poter comunicare, Client e server DTLS devono concordare una cifratura e le chiavi. Questi valori vengono determinati eseguendo un handshake DTLS, che è per gran parte uguale a quello TLS. In questo processo, il client inizia la comunicazione con il server inviando un messaggio "Hello". A questo il server risponde inviando al client il proprio certificato SSL che contiene la sua chiave pubblica. Il client verifica il certificato e, una volta estratta la chiave pubblica, la utilizza per criptarerne una nuova, la chiave pre-master e la spedisce al server. Quando quest'ultimo la decritta, sia client che server la utilizzano per creare la chiave privata comune. Il client, a questo punto, spedisce un messaggio criptato con la chiave segreta appena creata al server. La stessa cosa fa ques'ultimo, determinando la fine dell'handshake DTLS [33].
SRTP è un protocollo utilizzato per i pacchetti RTP, destinato a fornire crittografia, autenticazione e integrità dei messaggi e protezione dagli attacchi ai dati RTP, in applicazioni sia unicast che multicast. Per avviare una sessione SRTP è necessario specificare le chiavi e la cifratura. Tutta la configurazione viene eseguita durante l'handshake DTLS poiché SRTP non ha una fase di handshake. La generazioni di chiavi viene fatta da una Key Derivation Function definita da SRTP. Quando viene creata una sessione SRTP, gli input vengono eseguiti attraverso questa funzione per generare le chiavi per il cifrario SRTP [33].

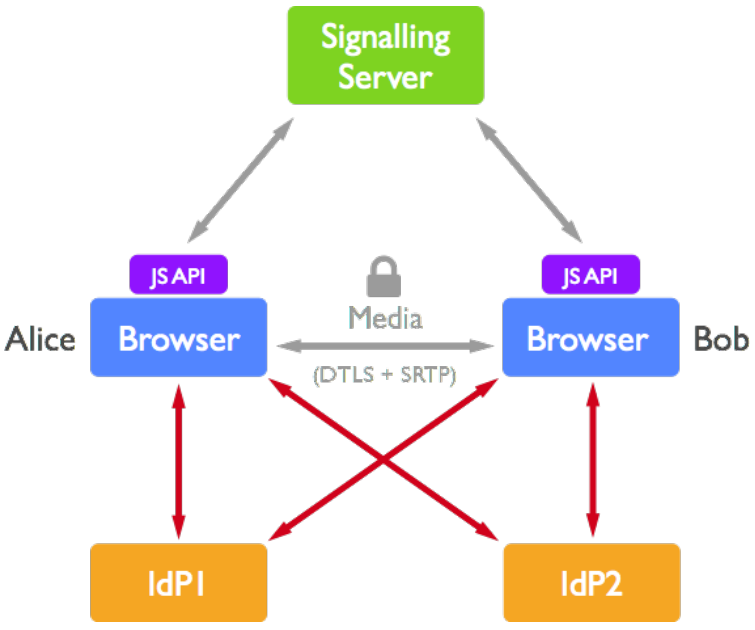

Figure 3.14: funzionamento DTLS e SRTP [67].

3.6.1 Un possibile attacco

Nonostante questi protocolli, WebRTC non è esente da problemi di sicurezza. Come descritto nei capitoli precedenti, WebRTC necessita degli indirizzi IP pubblici degli utenti per poter creare una connessione fra questi. Durante, per esempio, una videochiamata tra due utenti, si può verificare una perdita dati chiamata WebRTC leak, ovvero un browser Web espone inavvertitamente gli indirizzi IP degli utenti. L'esposizione di un indirizzo IP può causare danni gravi a un utente; un malintenzionato che riesce ad accedere alle informazioni di identificazione personale come indirizzi IP, richieste DNS e geolocalizzazione basata su IP può utilizzare questi dati per tracciare l'attività online di un individuo. Inoltre un attaccante esperto potrebbe effettuare un NAT Slipstreaming, un attacco che consente a un utente malintenzionato di accedere in remoto a qualsiasi servizio TCP/UDP associato a un sistema dietro il NAT di una vittima, aggirando il NAT/firewall di quest'ultima [34].
Tutti questi problemi possono essere però prevenuti tramite l'utilizzo di una VPN (Virtual Private Network), una tecnologia che consente di creare una rete privata tra due dispositivi su Internet. Tra le tante funzioni di una VPN c'è anche quella di "cambiare" l'indirizzo IP dell'utente con un indirizzo fittizio; in questo modo è possibile navigare in sicurezza, perché nel caso ci sia una WebRTC leak, l'attacante non potrà fare nulla con l'indirizzo IP della VPN [35].

Figure 3.15: Funzionamento di una VPN [68].

Chapter 4

Applicazioni

4.1 Costi e Implementazione

Il numero di applicazioni real-time che utilizzano WebRTC al giorno d'oggi sono numerosissime, questo perché realizzarne una è molto semplice e non implica costi importanti. Fino a tredici anni fa, però, le cose erano completamente diverse; infatti la creazione di un servizio di comunicazioni richiedeva, oltre alla realizzazione delle infrastrutture server, lo sviluppo di applicazioni per ogni piattaforma o sistema operativo che avesse un numero di utenti non trascurabile. Ignorandone uno di questi comportava il rischio di non riuscire a raggiungere un livello di adozione sufficiente ad alimentare un business sostenibile.
La struttura di WebRTC e l'evoluzione dei browser hanno cambiato tutto questo; infatti la possibilità di effettuare connessione real-time peer-to-peer ha permesso a qualunque sviluppatore con capacità anche elementari di programmazione Web di realizzare un servizio di comunicazione real-time, audio e video. L'investimento richiesto sarà notevolmente più basso, in quanto basterà acquistare un semplice servizio in Web-hosting [31].

Figure 4.1: utilizzo di WebRTC nei principali ambiti [69].

4.2 Casi d'Uso

4.2.1 Riunioni e Didattica a Distanza

Dopo la sua creazione, WebRTC ha sempre più acquisito fama grazie al suo facile impiego per effettuare chiamate standard o videochiamate. Il suo utilizzo è impennato, però, all'inizio del 2020, con l'arrivo della pandemia del Covid-19; infatti questa tecnologia ha permesso a studenti e lavoratori di continuare le proprie attività nonostante l'isolamento in casa.
Nel contesto scolastico WebRTC facilita la creazione di aule virtuali, offrendo lezioni coinvolgenti in tempo reale. Gli educatori possono utilizzare streaming video e audio, lavagne interattive e funzionalità di chat per condurre ambienti di apprendimento virtuale. Gli studenti possono partecipare attivamente, porre domande e collaborare con i compagni, favorendo un'esperienza educativa dinamica e interattiva. Inoltre è possibile effettuare sessioni individuali tra professore e studente; con gli strumenti appena citati, un insegnante può offrire una guida personalizzata ed esperienze di apprendimento interattive allo studente [36]. Durante la pandemia, inoltre, un

Figure 4.2: eLearning [70].

grande numero di aziende ha deciso di continuare ad operare optando per lo Smart Working, ovvero lavorare da casa senza doversi recare in ufficio. Anche in questo ambito WebRTC è una scelta più che azzeccata per effettuare chiamate/videochiamate che consentono ai dipendenti di lavorare tra loro o di svolgere riunioni.

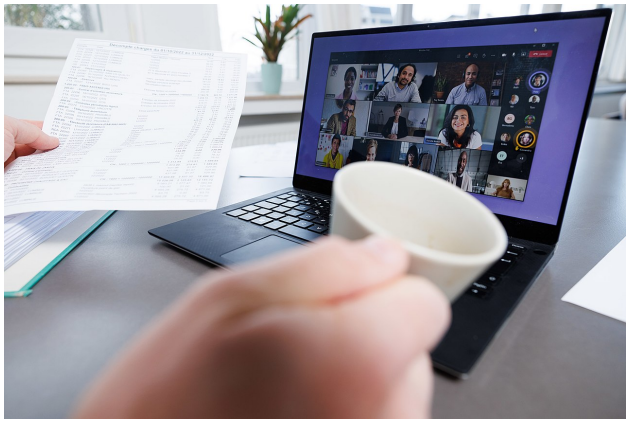

Figure 4.3: Una riunione effettuata in remoto [71].

4.2.2 IoT

Quando si parla di Internet of Things si intendono tutti quegli oggetti fisici dotati di sensori, capacità di elaborazione, software e altre tecnologie che si connettono e scambiano dati con altri dispositivi e sistemi tramite Internet o altre reti di comunicazione. Il campo si è evoluto grazie alla convergenza di più tecnologie, tra cui sensori di merci, sistemi embedded sempre più potenti e machine learning [37].
Tra tutte queste tecnologie ha trovato impiego anche WebRTC, la cui applicazione in IoT è divisa in due gruppi: Device-to-Person e Device-to-Cloud. Nel primo gruppo rientrano tutte quelle applicazioni che consentono agli utenti di vedere o ascoltare cosa sta succedendo dall'altra parte come, per esempio, telecamere di sicurezza, video citofoni e babymonitor.
In Device-to-Cloud rientrano i casi di automazione. Ad esempio, la combinazione di WebRTC e IoT è adatta a qualsiasi ambito in cui, ad esempio, i droni identificano malfunzionamenti e forniscono un controllo completo, garantendo il flusso del processo in tempo reale [38].
Gli use cases di WebRTC nell'Internet of Things sono vari, ma i due casi principali sono quello per Smart Home e quello in ambito industriale. Nell'ambito delle Smart Home, WebRTC può essere utilizzato per dispositivi citofonici o cassette postali intelligenti che utilizzano questa tecnologia per la comunicazione audio/video con applicazioni Web e mobile. Nel primo caso, i residenti delle smart home sono in grado di comunicare con i visitatori, assicurandosi che nessun caso di urgenza passi inosservato. Le caselle di posta intelligenti abilitate con la tecnologia WebRTC, invece, permettono di sbloccare le proprie caselle di posta da ufficio o anche quando si è in vacanza. Un altro esempio di IoT nelle smart home è Alexa Echo. Tramite dispositivo si può utilizzare l'API RTCSessionController permettendo di comunicare con tutti gli altri dispositivi Echo presenti in casa (Immagine) [39].
L'uso di WebRTC nel contesto industriale può essere molto utile per le operazioni di sorveglianza sui macchinari, rilevando tramite sensori possibili malfunzionamenti e avvisare gli operatori in tempo reale con WebRTC. Infatti l'applicazione più comune prevede il rilevamento di qualsiasi aumento insolito della temperatura delle macchine e ricercare le possibili cause come fumo dalla macchina, scintille che possono provocare incendi o sovraccarico. Con l'aiuto di WebRTC e sensori collegati alle macchine, gli avvisi possono essere inviati sotto forma di audio o video live, prevenendo così incidenti gravi. Inoltre, come già citato, il lavoro di rilevamento può essere migliorato tramite l'utilizzo di droni, che possono inviare riprese video in diretta che possono poi essere valutate e gestite di conseguenza (esempio) [39].

4.2.3 Assistenza sanitaria

Nell'ambito sanitario l'utilizzo di WebRTC è vario. Il campo più importante in cui può intervenire WebRTC è la telemedicina. La telemedicina è l'insieme di tecniche mediche e informatiche che consentono la cura di un paziente a distanza o più in generale di fornire servizi sanitari a distanza, permettendo di superare le barriere geografiche ed aiutare le comunità rurali svantaggiate, per la mancanza di accesso all'assistenza sanitaria. In questo ambito WebRTC viene utilizzato per creare applicazioni di comunicazione real-time come [38]:

- Teleconferenza appuntamenti medico-paziente.
- Sedute di terapia a distanza.
- Teleconferenze di terapia di gruppo.
- Incontri clinici.
- Osservazione a distanza delle sale operatorie.

La possibilità di utilizzare WebRTC in qualsiasi momento senza dover installare nessuna applicazione terza e la sicurezza crittografica che garantisce di salvaguardare i dati sanitari personali dei pazienti rappresentano per gli operatori sanitari dei punti a favore per l'utilizzo di WebRTC.

Figure 4.4: Telemedicina con WebRTC.

Un altro utilizzo è rivolto ai dispositivi indossabili come FitBit o Apple Watch; infatti questi salvano vite allertando, tramite WebRTC, il contatto di emergenza degli utenti e il pronto soccorso ogni volta che si verifica un picco o un calo improvviso della frequenza cardiaca [39].

4.2.4 Giochi Multiplayer

Il mondo videoludico è uno dei più importanti contesti applicativi di WebRTC, che viene utilizzato principalmente per creare una comunicazione real-time nei giochi multiplayer su un browser. Prima dell'invenzione di WebRTC, la tecnologia maggiormente utilizzata per creare una connessione su un browser game era WebSocket che, grazie alla sua struttura, permetteva a due o più utenti di giocare assieme. WebSocket però utilizza TCP come protocollo di trasporto e, come abbiamo già visto, questo protocollo rende la comunicazione affidabile; nel contesto dei giochi multiplayer viene preferita una connessione UDP perché il fattore più importante non è l'arrivo assicurato e ordinato dei dati, bensì che questi siano spediti velocemente. Nonostante ciò, WebSocket gioca um ruolo importante per la fase di signaling tra gli utenti [40].
Un altro importante utilizzo è quello riguardante il cloud gaming, un tipo di servizio online che esegue videogiochi su server remoti e li trasmette direttamente al dispositivo di un utente. I tre principali tipi di cloud gaming sono i giochi peer-to-peer, lo streaming dei giochi e il download progressivo [41]. In questo ambiente WebRTC è un'ottima scelta applicativa per la sua bassa latenza e per il fatto di essere provato e testato in molte grandi applicazioni in tempo reale con miliardi di utenti, come Google Meets, Discord, Facebook Manager e altre. Il funzionamento è molto simile ad una videochiamata, con la differenza che un utente si collegherà tramite una connessione peer-to-peer ad un'istanza di gioco [42]:

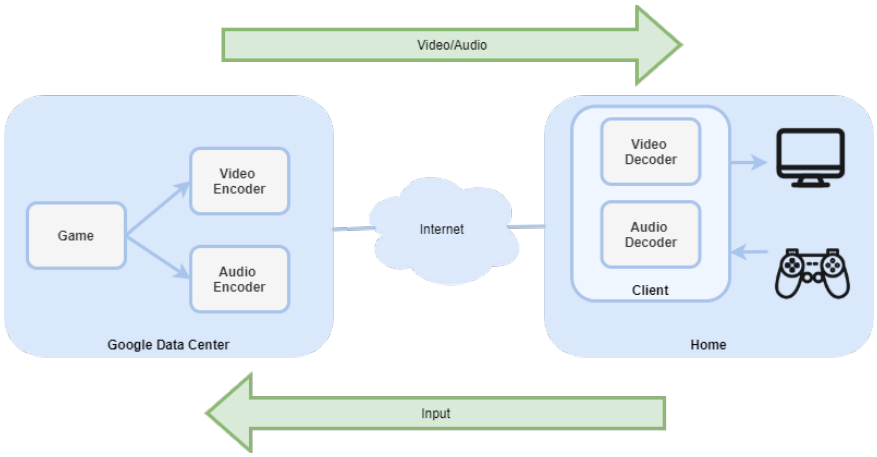

Figure 4.5: Architettura di Stadia, ex piattaforma di cloud gaming [72].

Nell'architettura di questa immagine si nota che quando un giocatore esegue un comando dal controller, questo viene spedito tramite i canali dati al servizio di cloud gaming, qui verrà elaborato e riprodotto. I segnali video ed audio vengono codificati e spediti tramite RTCPeerConnection; una volta arrivati sono decodificati dal dispositivo dell'utente [42].

4.3 WebRTC contro tutti

WebRTC non è l'unico strumento per creare applicazioni di comunicazioni real-time; infatti la concorrenza è grande e una delle tecnologie rivali più utilizzata è RTMP. Come abbiamo visto all'inizio, questo protocollo utilizzato sia da software (come OBS) che siti web (come YouTube) si basa su TCP. La scelta tra RTMP e WebRTC dipende dalle esigenze dello sviluppatore, considerando i punti di forza e limiti di entrambe le tecnologie: se da un lato WebRTC offre una migliore latenza, notevolmente più bassa rispetto a quella di RTMP, grazie all'utilizzo di UDP, dall'altro RTMP garantisce una maggiore scalabilità, migliorando l'esperienza di visualizzazione agli utenti e consentendo ai server locali di memorizzare nella cache i contenuti in streaming [43]. Entrambe le tecnologie hanno un alto livello di sicurezza: abbiamo già visto che WebRTC utilizza SRTP e DTLS per crittografare messaggi e per l'autenticazione. RTMP utilizza una sua variante, RTMPS, che permette di aggiungere un livello di sicurezza extra, che può essere TSL o SSL, per prevenire accessi non autorizzati. Infine è importate sottolineare che RTMP oltre al live streming viene utilizzato in pochissimi altri casi, a differenza di WebRTC che, come si è visto in precedenza, può essere implementato in più ambienti con successo [43].

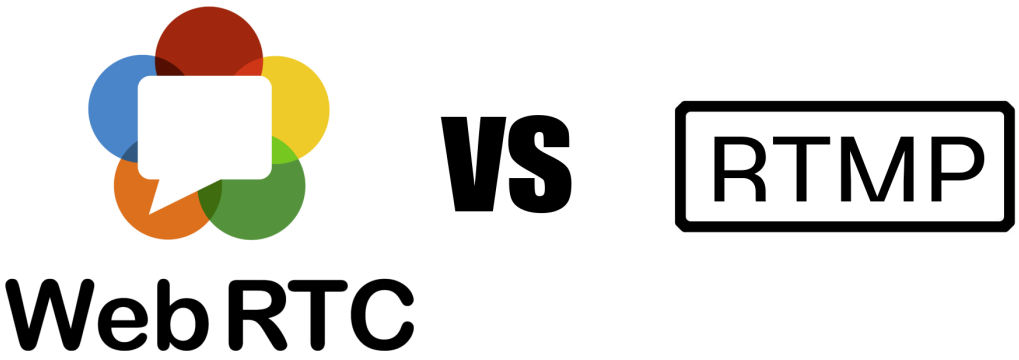

Figure 4.6: WebRTC vs RTMP.

Un'altra tecnologia di streaming audio e video concorrente di WebRTC è HLS, un protocollo di streaming a bitrate adattivo (una tecnica che funziona rilevando la larghezza di banda e la capacità della CPU di un utente in tempo reale, regolando di conseguenza la qualità del flusso multimediale) sviluppato da Apple, ampiamente utilizzato per lo streaming live su dispositivi mobili e browser desktop. HLS è progettato per funzionare con il protocollo HTTP e può essere facilmente integrato nell'infrastruttura basata su HTTP esistente. Le differenze tra HLS e WebRTC non sono molto diverse di quelle tra RTMP e WebRTC. Infatti HLS utilizza TCP come protocollo di trasporto, quindi ha una latenza più alta rispetto a WebRTC. A livello di sicurezza HLS utilizza HTTPS con TSL, garantendo un'autenticazione, protezione della privacy e integrità dei dati [44]. Per quanto riguarda la compatibilità, HLS risulta una migliore scelta di WebRTC solamente su dispositivi e browser Apple, mentre nei restanti casi WebRTC è più funzionale [45].

4.3.1 Teams vs Zoom

Teams e Zoom sono tra le applicazioni più utilizzate al mondo che forniscono servizi gratuiti o a pagamento per effettuare chiamate e videochiamate con un grande numero di utenti. La prima è un'applicazione creata da Microsoft nel 2017; la seconda è stata rilasciata dall'azienda Zoom Video Communications nel 2011. Le differenze principali tra queste due è l'utilizzo di WebRTC da parte di Teams; infatti questa applicazione può essere utilizzata anche via browser, mentre per utilizzare Zoom è necessario installare la corrispettiva applicazione. Altre differenze sono le seguenti [46]:

- l'abbonamento gratuito di Teams supporta riunioni fino a 30 ore tra due persone oppure di 1 ora con un massimo di 100 utenti, un tempo migliore del piano gratuito di Zoom, che consente riunioni tra 100 utenti per un massimo di 40 minuti [47]. Se si guardano tutti gli abbonamenti si nota come Teams sia una soluzione migliore per riunioni tra molte persone; infatti, a differenza di Zoom, permette di fare Webinar con 1000 utenti ed eventi in diretta con un massimo di 20000 partecipanti [48].
- Un difetto di Teams è il cospicuo utilizzo di RAM, che può rallentare in modo sensibile una macchina con poca memoria. Zoom invece richiede un utilizzo di RAM inferiore quindi risulta una scelta migliore su computer poco potenti.
- Su smartphone Zoom funziona in modo eccelso sia su Android che su iOS grazie all'aplicazione ben fatta e facile da utilizzare. Teams, invece, funziona senza problemi su Android, mentre il suo utilizzo su iOS non è possibile via Web; è necessario scaricare la relativa app [49].

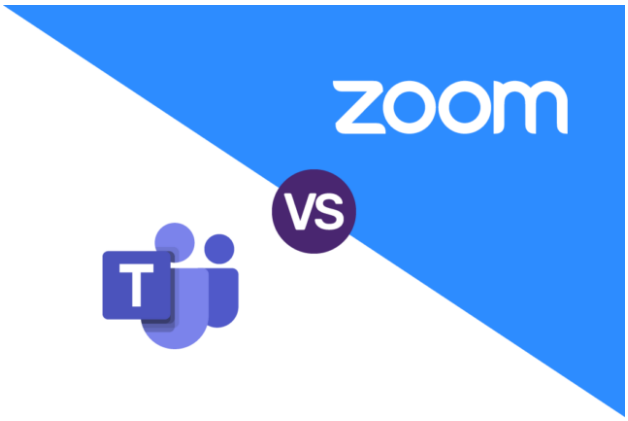

Figure 4.7: Teams vs Zoom [73].

Altre differenze, come la dimensione dei file da trasferire, gestione dei documenti e altro vengono riportate nella seguente immagine:

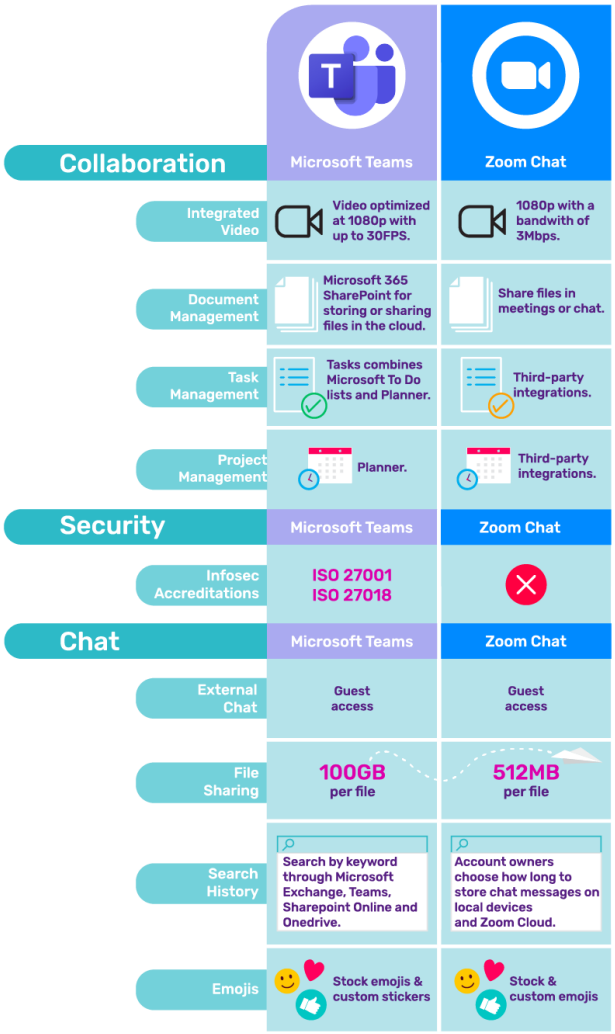

Figure 4.8: differenze tra Teams e Zoom [73].

Chapter 5

Progetto

5.1 Introduzione

Il progetto proposto è una applicazione di videochiamata WebRTC tra due utenti con la possibilità di spedire file in totale sicurezza. E' infatti garantita la privacy dei dati dal certificato SSL, indispensabile per l'attivazione della connessione WebRTC. In conseguenza di tale propensione di WebRTC alla protezione dei dati personali, una sua possibile applicazione è nel campo della telemedicina nel trasferimento dei dati tra un dottore ed un paziente. In questo caso WebRTC consente di effettuare una videochiamata e di spedire file testuali con relative informazioni mediche del paziente (come, per esempio, la temperatura corporea, le pulsazioni, l'elettrocardiogramma, l'ossimetria e altro) per far si che un dottore possa verificare lo stato del suo paziente.
E' possibile visionare il funzionamento del progetto cliccando qui.

5.2 Strumenti utilizzati

Per la realizzazione dell'applicazione sono stati utilizzati i due linguaggi Javascript (per il funzionamento) e HTML (per la pagina web). Il codice è stato scritto su Visual Studio Code, un editor di codice sorgente sviluppato da Microsoft. Il lato server è stato realizzato tramite Node.js, un runtime system open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript. Per la fase di signaling è stato utilizzato Firebase, una piattaforma per la creazione di applicazioni per dispositivi mobili e web sviluppata da Google [80]. Grazie a questa piattaforma è possibile creare un progetto con un database Firestore (un database flessibile e scalabile per lo sviluppo di dispositivi mobili, Web e server da Firebase) e collegarlo alla propria applicazione JS. Firebase è stato preferito a WebSocket perché a livello di codice è più semplice e veloce da implementare; inoltre ha funzioni di notifica utente ogni volta che avviene un cambiamento all'interno del DB Firestore.

5.3 Funzionamento

5.3.1 Creazione Certificato SSL

Prima di analizzare il funzionamento del progetto è necessario parlare dei certificati SSL. Il motivo per il quale un certificato diventa indispensabile in un'applicazione WebRTC è il metodo *getUserMedia()* dell'API MediaDevices perché, con l'uscita di Chrome 47 (2015), quest'API non è più supportata su pagine HTTP, ma solo su pagine HTTPS; infatti se utilizzata su HTTP, *getUserMedia()* restituirà un errore (NotAllowedError), cosa che non accade su HTTPS [77]. *getUserMedia()*, però, verifica solamente che la connessione tra client e server sia HTTPS, senza controllare se il certificato sia stato rilasciato da una Certification Authority (CA) nota, come invece fa il browser; per questo motivo ho deciso di creare un certificato firmato da una CA fittizia, resa nota al browser, per non avere alcun tipo di avvertenza.
Per la creazione dei certificati sono state utilizzate le seguenti funzioni di OpenSSL (uno strumento da riga di comando open source che consente agli utenti di eseguire varie attività relative a SSL) [78]:

- Le prime istruzioni riguardano la creazione del certificato della CA, che servirà per firmare il certificato del server. La prima permette di creare una chiave privata associandogli una password; la seconda consente di generare un certificato utilizzando la chiave appena generata.

```
openssl genrsa -out CA.key -des3 2048

openssl req -x509 -sha256 -new -nodes -days 3650 -key CA.key -out CA.pem
```

- Viene generata poi la chiave privata del server (domain.key) con lo stesso comando, al fine di creare una Certificate Signing Request (CSR) ovvero la specifica utilizzata dalle CA per la creazione e il rilascio di certificati da esse firmati [79]. La funzione di OpenSSL utilizzata è la seguente.

```
openssl req -key domain.key -new -out domain.csr
```

- Come specifiche aggiuntive per la CSR è stato creato un file di configurazione (localhost.exe) contenenti i dettagli tecnici relativi al certificato da creare e firmare. L'informazione fondamentale è la lista di domini o indirizzi IP per i quali il certificato deve essere rilasciato.

```
authorityKeyIdentifier = keyid,issuer
basicConstraints = CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment,
dataEncipherment
subjectAltName = @alt_names

DNS.1 = localhost
IP.1 = 127.0.0.1
IP.2 = 192.168.1.232
```

- Infine tramite il certificato della CA, la chiave privata della CA, il CSR e il file di configurazione, viene creato e firmato il certificato del server.

```
openssl x509 -req -in domain.csr -CA CA.pem -CAkey CA.key
-CAcreateserial -days 3650 -sha256 -extfile localhost.ext
-out domain.crt
```

L'ultima operazione da effettuare è inserire il certificato della CA nella lista di root CA attendibili del browser su cui viene testata l'applicazione.

5.3.2 Lato Server

Una volta creato il certificato SSL è possibile implementarlo nella pagina attraverso il file index.js che gestisce il lato server. Le principali dipendenze utilizzate sono express, fs (file system) e https. Express, un framework per applicazioni web per Node.js, progettato per creare web application e API, viene utilizzato per due motivi: con il metodo get dice al server cosa fare quando viene chiamata una richiesta get sulla route specificata [81]; i metodi use e static permettono di fornire al server i file nella cartella public (in questo caso style.css, client.js e main.js) [82]. Il framework fs ci permette di effettuare operazioni sui file presenti nel computer, come Leggere, Creare, Aggiornare, Cancellare e Rinominare i file. Il metodo readFileSync viene utilizzato per leggere il contenuto della chiave e del certificato per restituirlo nelle apposite variabili [83]. Infine https viene utilizzata per creare il server con chiave, certificato e funzione di callback, mettendolo in ascolto sulla porta specificata (in questo caso la 8081).

```
var express = require('express');
var app = express();
var https = require('https')
const fs = require('fs')

app.get('/', function (req, res) {
        fs.createReadStream('index.html').pipe(res)
})

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

var privKey = fs.readFileSync('certificates/domain.key')
var cert = fs.readFileSync('certificates/domain.crt')


https.createServer({
        key: privKey,
        cert: cert,
        passphrase: 'qwerty'
}, app).listen(8081);
```

5.3.3 Lato Client

Client.js è il file che fornisce il funzionamento dell'applicativo. Questo è suddiviso in due parti: la prima riguarda la dichiarazione e creazioni di variabili; le prime riguardano il collegamento del nostro progetto a Firebase, inserendo nella variabile firebaseConfig le informazioni che si trovano nella applicazione web creata sul sito di Firebase e richiamando il metodo initializeApp, e al database Firestore. Successivamente ci sono le variabili per la connessione peer (servers contiene i server STUN che verranno utilizzati durante la fase ICE) e quelle per accedere agli stream audio e video degli utenti; infine vengono dichiarate le variabili per gestire gli elementi HTML.

```
const firebaseConfig = {
        apiKey: "GbRTScPBmV5IWaTt-kqxCV561bWJ4z7TRIlKpEu0Mo",
        authDomain: "videocall-150f7.firebaseapp.com",
        databaseURL: "https://videocall-150f7-default-rtdb.
                                                    firebaseio.com",
        projectId: "videocall-150f7",
        storageBucket: "videocall-150f7.appspot.com",
        messagingSenderId: "543562675407",
        appId: "1:157235812407:web:6OP11aa5ab27vPaLae8b708"
};

firebase.initializeApp(firebaseConfig)

const firestore = firebase.firestore();

const servers = {
        iceServers: [
        {
                urls: ['stun:stun1.l.google.com:19302',
                        'stun:stun2.l.google.com:19302'],
        },
        ],
        iceCandidatePoolSize: 10,
};

let pc;
let localStream = null;
let remoteStream = null;

//HTML variables
...
```

La seconda parte riguarda il funzionamento dei vari pulsanti. Il primo gestisce l'accesso ai dispositivi dell'utente; prima di tutto viene creata la connessione peer tramite RTCPeerConnection, necessario per aggiungere le tracce audio e video alla variabile pc. Seguono poi le operazioni sugli stream locale e remoto. Sul primo richiamiamo il metodo getUserMedia per chiedere all'utente di concedere l'autorizzazione ad accedere al suo microfono e webcam; se concessa, tutte le tracce ottenute su localStream vengono aggiunte sulla variabile pc. Lo stream remoto invece viene creato e riempito con tracce vuote tramite l'EventHandler ontrack di RTCPeerConnection (ogni volta che una traccia viene aggiunta alla variabile pc, ontrack viene lanciato eseguendo la funzione specificata). Infine aggiungiamo i due stream alle variabili che gestiscono gli elementi video HTML.

```
webcamButton.onclick = async () => {
        pc = new RTCPeerConnection(servers);

        localStream = await navigator.mediaDevices.getUserMedia({
                video: true, audio: true })
```

```
        remoteStream = new MediaStream()

        localStream.getTracks().forEach((track) => {
                pc.addTrack(track, localStream)
        });

        pc.ontrack = (event) => {
                event.streams[0].getTracks().forEach((track) => {
                        remoteStream.addTrack(track)
                });
        };

        webcamVideo.srcObject = localStream;
        remoteVideo.srcObject = remoteStream;
};
```

Una volta date le autorizzazione per accedere ai propri dispositivi, l'utente può creare una chiamata oppure partecipare ad una già esistente. Nel primo caso deve premere il pulsante "Crea chiamata"; il processo per creare un'offerta è il seguente: una volta premuto il bottone, vengono istanziate tre variabili per collegarsi alle collezioni del database Firestore che riguardano le chiamate e i candidati ICE dei due utenti (il database è strutturato in modo che per ogni chiamata corrisponda un identificativo al quale sono associati offerta, risposta e candidati ICE). Successivamente viene generato un id casuale tramite la proprietà .id di firestore: questo verrà utilizzato come id della chiamata. La prossima istruzione serve per ottenere e scrivere sul database i candidati ICE del chiamante. Una volta fatto ciò viene creata l'offerta tramite il metodo createOffer (di RTCPeerConnection) e impostata come descrizione locale; le sue informazioni vengono inserite nella variabile offer che successivamente verrà scritta sul database. Infine sono presenti due listener onSnapshot: il primo fa si che se un utente risponde alla chiamata, la sua risposta venga impostata come descrizione remota se quest'ultima è vuota; il secondo invece aggiunge un nuovo candidato ICE del utente chiamato ogni volta che ne viene aggiunto uno sul database.

```
callButton.onclick = async () => {
        const callDoc = firestore.collection('calls').doc();

        const offerCandidates = callDoc.
                collection('offerCandidates');
        const answerCandidates = callDoc.
                collection('answerCandidates');

        callInput.value = callDoc.id;

        pc.onicecandidate = (event) => {
                event.candidate && offerCandidates.add(event.
                                                candidate.toJSON());
        };

        const offerDescription = await pc.createOffer();
        await pc.setLocalDescription(offerDescription);

        const offer = {
                sdp: offerDescription.sdp,
                type: offerDescription.type,
        };

        await callDoc.set({ offer });
}
```

Il codice del pulsante per unirsi ad una chiamata è per la maggior parte identico al pulsante per creare una chiamata, con l'unica differenza che non si gestisce l'offerta, ma la risposta RTCPeerConnection. Dopo aver creato le variabili per il database e aggiunto i candidati ICE alla collezione answerCandidates, si prende dal database l'offerta e la si imposta come descrizione remota; successivamente si crea la risposta, la si imposta come descrizione locale e viene aggiunta al database. Anche qui è presente un listener per far si che ogni ICE del chiamante aggiunto al database venga aggiunto anche alla variabile RTCPeerConnection.

```
answerButton.onclick = async () => {
        const callId = callInput.value;
        const callDoc = firestore.collection('calls').
          doc(callId);
        const answerCandidates = callDoc.
          collection('answerCandidates');
        const offerCandidates = callDoc.
          collection('offerCandidates');

        pc.onicecandidate = (event) => {
          event.candidate && answerCandidates.add(event.
            candidate.toJSON());
        };

        const callData = (await callDoc.get()).data();
        const offerDescription = callData.offer;
        await pc.setRemoteDescription(new
          RTCSessionDescription(offerDescription));

        const answerDescription = await pc.createAnswer();
        await pc.setLocalDescription(answerDescription);

        const answer = {
          type: answerDescription.type,
          sdp: answerDescription.sdp,
        };

        await callDoc.update({ answer });

        offerCandidates.onSnapshot((snapshot) => {
          snapshot.docChanges().forEach((change) => {
            console.log(change);
            if (change.type === 'added') {
              let data = change.doc.data();
              pc.addIceCandidate(new
                RTCIceCandidate(data));
            }
          });
        });
};
```

Il quarto pulsante consente agli utenti di spedire i file con i dati diagnostici del paziente. Questo trasferimento è possibile grazie ad un Data Channel, impostato dal metodo *createDataChannel()* di RTCPeerConnection. Una volta creato si gestisce l'evento onmessage, ovvero cosa deve fare il canale dati quando un messaggio viene spedito. Nel codice seguente osserviamo in particolare la funzione onMessageCallback nella quale, una volta che il file è stato spedito, viene creato nel browser di destinazione un nuovo Blob nel quale si caricano tutte le informazione del file ricevuto e successivamente si crea un link che, se cliccato, permette all'utente di scaricare il Blob appena creato.

```
reciveBuffer = []
pc = new RTCPeerConnection(servers);
dataChannel = pc.createDataChannel('fileChannel',
                { negotiated: true, id: 0 });
dataChannel.onmessage = onMessageCallback;

function onMessageCallback(event) {
        receiveBuffer.push(event.data);

        const receivedFile = new Blob(receiveBuffer);
        receiveBuffer = [];

        downloadAnchor.href = URL.createObjectURL(receivedFile);
        downloadAnchor.download = "temperatura.txt"
        downloadAnchor.textContent =
        `Click to download the file`;
        downloadAnchor.style.display = 'block';
        console.log("file ricevuto");
}
```

Nel pulsante di spedizione viene selezionato il file dall'elemento input HTML e tramite una variabile FileReader il contenuto del file viene letto, salvato su una variabile (data) e spedito tramite il metodo send di RTCDataChannel.

```
sendButton.onclick = async () => {
        const file = fileInput.files[0];
        const reader = new FileReader();

        reader.onload = (event) => {
                const data = event.target.result;
                dataChannel.send(data);
                console.log("file spedito");
        };

        reader.readAsArrayBuffer(file);
}
```

Infine, l'ultimo pulsante permette all'utente di riagganciare. Al suo interno viene semplicemente chiusa la connessione peer e il canale dati tramite il metodo *close()*, rimuove tutte le tracce dallo stream remoto e locale e si cancella l'ID della chiamata.

```
hangupButton.onclick = async () => {
        pc.close();
        dataChannel.close();

        remoteStream.getTracks().forEach(function(track){
                track.stop();
        });
        remoteVideo.srcObject = remoteStream;

        localStream.getTracks().forEach(function(track) {
                track.stop();
        });
        webcamVideo.srcObject = localStream;

        callInput.value = "";
}
```

5.3.4 Webpack

Il file client.js contiene al suo interno dipendenze ad altri file js, presenti sul server tra le dipendenze terze: non essendo questi disponibili lato client, l'esecuzione di client.js non può andare a buon fine. Per ovviare a questo problema è stato utilizzato Webpack, un module bundler, ovvero uno strumento per organizzare e combinare un file e le sue dipendenze in un unico file; viene solitamente utilizzato quando un progetto diventa troppo grande o quando lavoriamo con librerie che hanno dipendenze a più livelli. Dato un file js in input, Webpack crea un grafico delle sue dipendenze, includendo sia quelle presenti tra i sorgenti del progetto, sia quelle presenti in librerie terze. Questo grafico è finalizzato alla creazione di un bundle, ovvero un file js indipendente. per utilizzare Webpack si utilizza il comando sottostante; questo creerà un nuovo file (main.js) che, una volta inserito nel tag script di html, farà funzionare correttamente il programma sul browser senza alcun tipo di errore. Ovviamente questo operazione è da effettuare ogni volta che si fa una modifica sul file client.js.

```
webpack ./public/client.js
```

Chapter 6

Conclusioni

Questo documento ha analizzato la tecnologia WebRTC, dimostrando che cos'è, come funziona e quali sono le sue applicazioni. Infatti come abbiamo visto questa tecnologia nata 10 anni fa ha risolto un problema alla quale da tempo si cercava una soluzione: creare applicazioni di comunicazione in tempo reale utilizzandole via browser, quindi senza installare plugin o software terzi.
Il suo funzionamento si basa sull'utilizzo di pagine HTML, ma soprattutto di API Javascript, che permettono di utilizzare i dispositivi input audio e video dell'utente (MediaStream), creare una connessione con uno o più peer (RTCPeerConnection) e scambiare messaggi (RTCDataChannel). Utilizzando il protocollo UDP per il trasporto e la tecnologia ICE per trovare il percorso tra i due utenti, WebRTC consente una connessione veloce. La sicurezza è garantita grazie ai protocolli SRTP e DTLS, che crittano ed evitano l'intercettazione dei messaggi. Questo comunque non rende WebRTC completamente sicuro; infatti possono verificarsi delle perdite riguardanti gli IP degli utenti; l'utilizzo di una VPN, però, riduce sensibilmente questa potenziale falla nel sistema di sicurezza.
Dopo la sua creazione, tutti i browser e sistemi mobile si sono impegnati per renderlo utilizzabile tra browser differenti; infatti nel 2013 avviene la prima chiamata cross-browser. E' vero anche che i dispositivi Apple hanno iniziato a supportare questa tecnologia nel 2017 e ancora oggi risulta più problematica rispetto a quella su Android (per quanto riguarda iPhone) o su Chrome/FireFox (per quanto riguarda Safari).
Negli anni WebRTC ha trovato impiego in molti campi, primo fra tutti la didattica/lavoro a distanza. Ci sono due campi in particolare dove WebRTC viene già utilizzato, ma ha ancora molte potenzialità per il futuro; si parla dell'IoT e della telemedicina. Oggi, grazie a questa tecnologia, è possibile effettuare operazioni di automazione industriale e di monitoraggio remoto tramite dispositivi IoT oppure sedute mediche a distanza tra paziente e dottore con la telemedicina. In questo ambiente WebRTC potrebbe un giorno essere utilizzato non solo per effettuare chiamate di controllo, ma anche per eseguire operazioni mediche di tipo chirurgico oppure di trapianto d'organi effettuate tramite, per esempio, l'accesso remoto a macchinari appositi.
Il progetto sviluppato in questa tesi ha mostrato le potenzialità di questo strumento di comunicazione in un ambiente controllato come quello tra medico e paziente. Come sviluppo futuro si potrebbe pensare di automatizzare il riconoscimento dei dispositivi di diagnostica collegati al paziente e fare in modo che in chiamata il medico possa vedere il paziente e contemporaneamente in una finestra laterale leggere i dati istantanei delle misure restituite dai vari dispositivi, quali termometro, pulsossimetro, elettrocardiografo ed altre informazioni.

Appendix A

VP8

Il VP8 (Video Compression Format o Video Compression Specification) è una specifica per la codifica e la decodifica di video ad alta definizione come file o bitstream per la visualizzazione rilasciata nel 2008. A differenza del suo omologo codec H.264, il codec VP8 è gratuito. Ciò è dovuto al fatto che Google ha rilasciato tutti i brevetti che possiede sotto una licenza pubblica royalty-free (un tipo di licenza che permette l'utilizzo di una risorsa con limitate restrizioni sul suo utilizzo e pagando una cifra iniziale estremamente contenuta.). H.264, tuttavia, contiene una tecnologia brevettata e richiede licenze da parte dei detentori di brevetti e royalty limitate per l'hardware. Opera, Firefox, Chrome e Chromium supportano la riproduzione di video VP8 nel tag video HTML5. Secondo Google, VP8 è utilizzato principalmente in connessione con WebRTC e come formato per brevi animazioni in loop, in sostituzione del Formato di interscambio grafico [50].

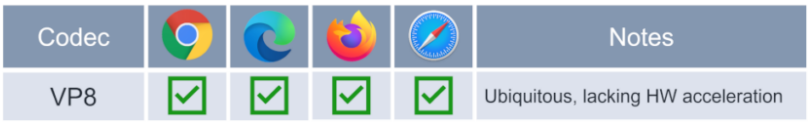

Figure A.1: Supporto di VP8 sui browser principali [74].

Appendix B

NAT

Con NAT (Network Address Translation) si intendono tutti quei processi di modifica di uno o più indirizzi IP privati locali in un indirizzo IP pubblico globale tramite apparati di rete come router o firewall. Questo offre la possibilità di accedere a Internet con maggiore sicurezza e privacy nascondendo l'indirizzo IP del dispositivo dalla rete pubblica, anche durante l'invio e la ricezione di traffico [51]. Lo scopo principale del NAT è quello di ovviare alla scarsità di indirizzi IP pubblici disponibili; le tecniche utilizzate per effettuare queste operazioni rendono i dispositivi non direttamente raggiungibili da Internet, garantendo la sicurezza e privacy citata prima. Esistono sostanzialmente due tipi di NAT: il source NAT e il destination NAT; l'unica differenza tra questi due è la modifica dell'indirizzo del pacchetto che inizia una nuova connessione: nel primo caso viene modificato quello di sorgente, nel secondo quello di destinazione. Il tipo di NAT più utilizzato è PAT(Port Address Translation), un tipo di source NAT in cui gli indirizzi dei dispositivi di una rete privata vengono rappresentati verso l'esterno come un solo indirizzo IP. In questo tipo di NAT vengono modificate anche le porte TCP e UDP delle connessioni in transito (per questo il nome Port Address Translation) [52].

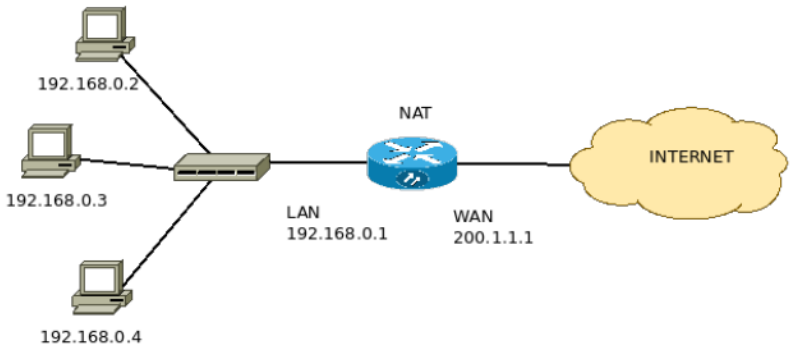

Figure B.1: Funzionamento del NAT [75].

Appendix C

TLS

Transport Layer Security (TLS) è un protocollo crittografico progettato per fornire la sicurezza delle comunicazioni su una rete di computer. Il protocollo è ampiamente utilizzato in applicazioni come e-mail, messaggistica istantanea e voice over IP. Il protocollo TLS mira principalmente a fornire sicurezza, l'integrità e l'autenticità attraverso l'uso della crittografia, come l'uso di certificati, tra due o più applicazioni informatiche comunicanti.Il Datagram Transport Layer Security (DTLS) strettamente correlato è un protocollo di comunicazione che fornisce sicurezza alle applicazioni basate su datagrammi [53].
TLS può funzionare sia con crittografia simmetrica che asimmetrica. Nel primo caso i messaggi vengono crittati con una chiave che conoscono solamente il mittente e il destinatario lunga 128 o 256 bit. Questo tipo di crittografia è efficiente a livello computazionale, ma lo scambio della chiave comune deve essere fatto in modo sicuro. Nella crittografia asimmetrica, mittente e destinatario hanno entrambi una coppia di chiavi: una pubblica ed una privata. Le chiavi pubbliche dei destinatari vengono utilizzate per crittografare i messaggi dai mittenti, mentre le chiavi private vengono utilizzate per decodificare. Con TLS asimmetrico non è presente lo scambio di chiavi tra i due utenti. Inoltre la chiave private è matematicamente correlata alla chiave pubblica, ma utilizzando chiavi sufficientemente lunghe sarà molto difficile dedurre la chiave privata da quella pubblica. Per questi motivi TLS asimmetrico è più sicuro di TLS simmetrico e anche più utilizzato [54].

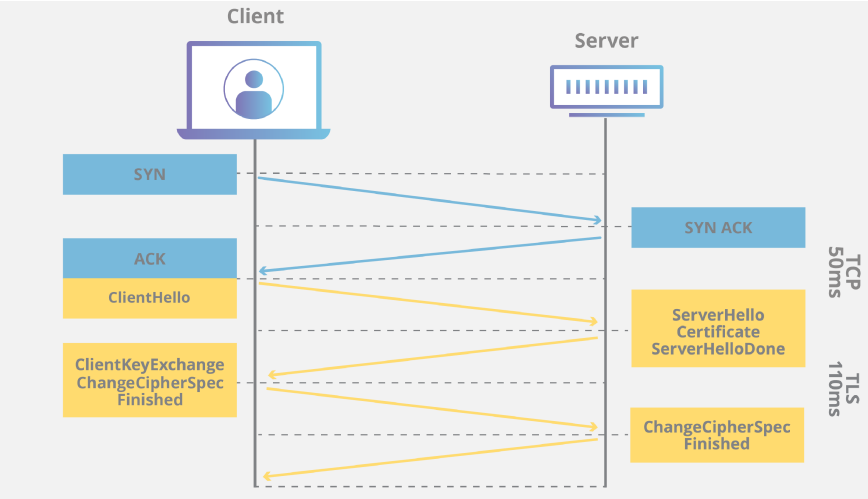

Figure C.1: Handshake TLS [76].

Bibliografia e Immagini

[1] Wikipedia. *Telefono*: https://it.wikipedia.org/wiki/Telefono

[2] *L'invenzione del telefono: più di 150 anni di storia*: https://www.tecnotrade.it/news/meucci-e-telefono-150-anni-di-storia/

[3] Fotini-Niovi Pavlidou, Stylianos Karapantazis (2009). *VoIP: A comprehensive survey on a promising technology*: https://personal.utdallas.edu/ kxs028100/Papers/VoIP.Survey.pdf

[4] *The History of VoIP and Internet Telephony*: https://getvoip.com/blog/history-of-voip/

[5] *RTMP: Real Time Messaging Protocol*: https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/rtmp/

[6] *Introducing OBS and RTMP support for Sendbird Live*: https://sendbird.com/blog/live-video-streaming-obs-rtmp-support

[7] Wikipedia. *WebRTC*: https://en.wikipedia.org/wiki/WebRTC

[8] Olivier Anguenot. *The WebRTC Story*: https://www.webrtc-developers.com/the-webrtc-story-part-ii/

[9] Wikipedia. *WebSocket*: https://en.wikipedia.org/wiki/WebSocket

[10] *WebRTC vs. WebSocket: Key differences and which to use*: https://ably.com/topic/webrtc-vs-websocket

[11] *WebRTC Architecture*: https://webrtc.github.io/webrtc-org/architecture/

[12] TutorialsPoint. *Webrtc Tutorial*: https://www.tutorialspoint.com/webrtc/webrtc_tutorial.pdf

[13] Donika Mehmeti, Linus Palmblad . *Application Programming Interfaces, an exploration of their properties and what to consider during implementation*: https://www.diva-portal.org/smash/get/diva2:1685846/FULLTEXT01.pdf

[14] Wikipedia. *API*: https://en.wikipedia.org/wiki/API

[15] *What is an API (Application Programming Interface)?* : https://www.geeksforgeeks.org/what-is-an-api/

[16] *RTCPeerConnection*: https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection

[17] *RTCDataChannel*: https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel

[18] *MediaStream*: https://developer.mozilla.org/en-US/docs/Web/API/MediaStream

[19] *WebRTC API*: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API

[20] *Differences between TCP and UDP*: https://www.geeksforgeeks.org/differences-between-tcp-and-udp/

[21] *Why you should prefer UDP over TCP for your WebRTC sessions*: https://bloggeek.me/why-you-should-prefer-udp-over-tcp-for-your-webrtc-sessions/

[22] Wikipedia. *STUN*: https://en.wikipedia.org/wiki/STUN

[23] *WebRTC TURN server: Everything you need to know*: https://www.100ms.live/blog/webrtc-turn-server

[24] Redouane Meddane (2020). *STUN TURN and ICE for NAT Traversal*

[25] *Introduction to WebRTC protocols*: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols

[26] *What, Why and How (WebRTC)*: https://webrtcforthecurious.com/docs/01-what-why-and-how/

[27] Cui Jian, Zhuying Lin (2015). *Research and Implementation of WebRTC Signaling via WebSocket-based for Real-time Multimedia Communications*

[28] *RTCDataChannel - WebRTC Explained*: https://www.onsip.com/voip-resources/voip-fundamentals/rtcdatachannels

[29] *Microsoft Teams meetings on unsupported browsers*: https://learn.microsoft.com/en-us/microsoftteams/unsupported-browsers

[30] *RTCPeerConnection Support*: https://caniuse.com/?search=RTCPeerConnection

[31] Alberto Baravaglio, Alberto Cuda, Enrico Marocco (2013). *WebRTC: La nuova sfida nelle comunicazioni real-time audio/video*

[32] *WebRTC browser support on desktop and mobile*: https://bloggeek.me/webrtc-browser-support/

[33] *WebRTC Securing*: https://webrtcforthecurious.com/docs/04-securing/

[34] Gordon H. (2022). *WebRTC IP Leaks: Should You Still Be Worried?*: https://getstream.io/blog/webrtc-ip-leaks/

[35] *Verifica fughe di WebRTC*: https://surfshark.com/it/webrtc-leak-test

[36] *Benefits of using WebRTC for Online Education and eLearning*: https://www.digitalsamba.com/blog/benefits-of-using-webrtc-for-online-education-and-elearning

[37] Wikipedia. *IoT*: https://en.wikipedia.org/wiki/Internet_of_things

[38] *Applications of WebRTC in IoT*: https://www.iotforall.com/use-cases-for-webrtc-in-iot-applications

[39] *Powering IoT with secure real time communication*: https://rtcweb.in/powering-iot-with-secure-real-time-communication/

[40] Antonio Preucil (2021). *Using WebRTC for a browser multiplayer game*: https://dev.to/bornfightcompany/using-webtrc-for-a-browser-multiplayer-game-in-theory-59dk

[41] Wikipedia. *Cloud gaming*: https://it.wikipedia.org/wiki/Cloud_gaming

[42] Alberto Gonzalez (2021). *WebRTC Cloud Gaming: Unboxing Stadia.*: https://webrtc.ventures/2021/02/webrtc-cloud-gaming-unboxing-stadia/

[43] Traci Ruether (2022). *RTMP vs WebRTC*: https://www.wowza.com/blog/rtmp-streaming-real-time-messaging-protocol

[44] Wikipedia. *HTTPS*: https://en.wikipedia.org/wiki/HTTPS

[45] *HLS Support*: https://caniuse.com/?search=hls

[46] *Microsoft Teams vs. Zoom* (2022): https://www.crazyegg.com/blog/microsoft-teams-vs-zoom/

[47] *Zoom piani e licenze*: https://zoom.us/it/pricing

[48] *Teams: meetings, webinars, and live events*: https://learn.microsoft.com/en-us/microsoftteams/quick-start-meetings-live-events

[49] *Microsoft forum*: https://answers.microsoft.com/en-us/msteams/forum/all/teams-is-not-supported-on-any-ios-browser/14ce2d1f-bfbe-4fde-8fde-cd0a4df1e2cb

[50] Wikipedia. *VP8*: https://en.wikipedia.org/wiki/VP8

[51] P. Srisuresh, Jasmine Networks, K. Egevang (2001). *Traditional IP Network Address Translator (Traditional NAT)*: https://www.rfc-editor.org/rfc/rfc3022.html

[52] Wikipedia. *NAT*: https://en.wikipedia.org/wiki/Network_Address_Translation

[53] Wikipedia. *TLS*: https://en.wikipedia.org/wiki/Transport_Layer_Security

[54] *TLS Basics*: https://www.internetsociety.org/deploy360/tls/basics/

[55] Jay Hilotin: *A brief history of the telephone*: https://gulfnews.com/world/a-brief-history-of-the-telephone-1.1603713110135

[56] *Telefono e centralino VoIP per il business, tutto quello che bisogna sapere*: https://www.iotatau.com/telefono-centralino-voip-business-hotel/

[57] Emiliano Tomasoni: *Che cos'è WebRTC ed è sicuro?*: https://blog.wildix.com/it/cose-webrtc-ed-sicuro/

[58] Maulik Shah: *WebRTC Architecture – A Layman's Guide*: https://tragofone.com/webrtc-architecture-types-guide/

[59] Martin Meszaros: *WebRTC triangle architecture*: https://www.researchgate.net/figure/WebRTC-triangle-architecture-27_fig3_328334940

[60] Martin Meszaros: *WebRTC trapezoid architecture*: https://www.researchgate.net/figure/WebRTC-trapezoid-architecture-according-to-Maruschke-et-al-27-and-Alvestrand-49_fig4_328334940

[61] Akshita Kumawat: *What is an API (Application Programming Interface)?*: https://www.geeksforgeeks.org/what-is-an-api/

[62] Omer Sayem: *STUN,TURN and ICE servers - NAT traversal for WebRTC*: https://dev.to/sayem_omer/stun-turn-and-ice-servers-nat-traversal-for-webrtc-5e29

[63] Kavirajan ST: *What is WebRTC and How to Setup STUN/TURN Server for WebRTC Communication?*: https://medium.com/av-transcode/what-is-webrtc-and-how-to-setup-stun-turn-server-for-webrtc-communication-63314728b9d0

[64] Karthikeyan S: *WebRTC TURN Server: Internals and Beyond*: https://webdev.100ms.live/blog/webrtc-turn-server

[65] *Webrtc-sandbox-flutter*: https://github.com/lambiengcode/webrtc-sandbox-flutter?ref=flutterawesome.com

[66] W3C: *WebRTC 1.0: Real-time Communication Between Browsers*: https://dev.w3.org/2011/webrtc/editor/archives/20121019/webrtc.html

[67] *A Study of WebRTC Security*: https://webrtc-security.github.io/

[68] *Cosa è un router VPN è perché ne hai bisogno?* : https://community.fs.com/it/blog/what-is-vpn-router-why-you-need-it.html

[69] Dalibor Kofjač: *WebRTC – the technology connecting the modern world*: https://ekobit.com/blog/webrtc-the-technology-connecting-the-modern-world/

[70] Corriere Quotidiano: *Il 90% degli italiani è favorevole all'introduzione dell'educazione finanziaria nelle scuole e l'80% sul posto di lavoro*: https://corrierequotidiano.it/scuola/il-90-degli-italiani-e-favorevole-allintroduzione-delleducazione-finanziaria-nelle-scuole-e-l80-sul-posto-di-lavoro/

[71] Wikipedia: *Telelavoro*: https://it.wikipedia.org/wiki/Telelavoro

[72] Alberto Gonzalez: *WebRTC Cloud Gaming: Unboxing Stadia*: https://webrtc.ventures/2021/02/webrtc-cloud-gaming-unboxing-stadia/

[73] Dominic Kent: *Microsoft Teams vs Zoom: Which Is Best For Your Business?*: https://dispatch.m.io/microsoft-teams-vs-zoom/

[74] *VP8*: https://bloggeek.me/webrtcglossary/vp8/

[75] *Cosa è NAT e quali sono i vantaggi dei firewall NAT?*: https://community.fs.com/it/blog/what-is-nat-and-what-are-the-benefits-of-nat-firewalls.html

[76] *Cosa succede in un handshake TLS? | Handshake SSL*: https://www.cloudflare.com/it-it/learning/ssl/what-happens-in-a-tls-handshake/

[77] *MediaDevices: getUserMedia() method*: https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia

[78] *How to Get SSL HTTPS for Localhost*: https://www.section.io/engineering-education/how-to-get-ssl-https-for-localhost/

[79] Wikipedia: *Certificate Signing Request*: https://it.wikipedia.org/wiki/Certificate_Signing_Request

[80] Wikipedia: *Firebase*: https://en.wikipedia.org/wiki/Firebase

[81] *Routing HTML*: https://expressjs.com/it/guide/routing.html

[82] *Serving static files in Express*: https://expressjs.com/en/starter/static-files.htm

[83] *Node.js fsreadFileSync Method*: https://www.geeksforgeeks.org/node-js-fs-readfilesync-method/

Ringraziamenti

Infine, volevo dedicare questa ultima pagina per ringraziare tutte le persone che mi hanno sostenuto in questi anni di università, a partire dal mio relatore di tesi Andrea Piroddi che, oltre ad avermi proposto l'argomento WebRTC, è sempre stato disponibile sia ad aiutarmi nella scrittura della tesi, sia a visionare il progresso del lavoro con riunioni settimanali.
Ringrazio la mia famiglia per avermi sostenuto moralmente ed economicamente e mi abbia spronato a raggiungere questo risultato.
In conclusione, ringrazio i miei amici per aver reso questa esperienza universitaria più piacevole.