Università Degli Studi di Roma *Tor Vergata*

Macroarea di Scienze Matematiche, Fisiche e Naturali

Corso Di Laurea Triennale In Informatica

Studio e sperimentazione della tecnologia WebRTC

Relatore: Pietro Frasca

Candidato: Orlando Parente

Anno Accademico 2020/2021

Indice

Indice delle figure

1. INTRODUZIONE

WebRTC (Web Real-Time Communication) è una tecnologia introdotta da Google il 1° Giugno 2011 [10], per lo scambio di audio, video e messaggi (dati) tramite una connessione peer-to-peer in real time.
Essendo stata rilasciata con la *licenza BSD modificata* (3 clausole)[1] [8, 26] è una tecnologia open source [22].
Inoltre, il 26 Gennaio 2021 World Wide Web Consortium (W3C) e Internet Engineering Task Force (IETF) annunciano che webRTC è ora uno standard [29,10]. In altre parole, webRTC è sostanzialmente un insieme di API (Application Programming Interface) standard per javascript, supportata da Google, Microsoft, Mozilla, Apple ed Opera [28]: è largamente compatibile senza la necessità di installare ulteriori componenti (*plugin-free*) e permette ai programmatori di sviluppare facilmente e velocemente applicativi in grado di scambiarsi dati attraverso una efficiente connessione peer-to-peer. In definitiva, rappresenta un nuovo strumento in mano ai programmatori che permette loro di soddisfare la crescente domanda di applicativi in grado di effettuare videochiamate.

Infatti, lentamente, applicativi come MS Teams o Google Duo stanno sempre più prendendo il posto delle semplici chiamate vocali. Lo scenario delle telecomunicazioni si sta spostando completamente sul web e per questo motivo è nata l'esigenza di uno standard che permette di soddisfare le nuove esigenze. WebRTC risponde proprio questa necessità e allo stesso tempo accelera questo processo, semplificando la creazione, appunto, di questi applicativi [13].

Nei prossimi capitoli si analizzeranno, dapprima il funzionamento di questa tecnologia ed in seguito come utilizzarla con un esempio pratico.

[1] Esistono diverse versioni della licenza BSD. Partendo dalla versione originale a 4 clausole, sono state successivamente introdotte le versioni a 3, 2 e 0 clausole [27] . WebRTC è stato rilasciato sotto la licenza BSD a 3 clausole, anche conosciuta con altri nomi come "licenza BSD modificata" o "nuova licenza BSD". Tale versione della licenza BSD è stata riconosciuta compatibile con la filosofia open source [22].

2. CONTESTUALIZZAZIONE DELLA TECNOLOGIA WEBRTC

La tecnologia webRTC è usata per la creazione di applicativi soft *real-time* in grado di stabilire una *connessione peer-to-peer* per lo scambio di dati.

2.1 Applicativi Soft Real-Time con webRTC

Le videochiamate costituiscono la principale funzionalità che si può sviluppare con questa tecnologia. Esse sono l'esempio tipico di un *applicativo soft real-time,* in cui si necessita che i dati vengano trasmessi e ricevuti, appunto, in "tempo reale": ogni pacchetto è utile solo se arriva al destinatario entro un certo tempo (deadline).

In sostanza, gli stream di video ed audio (per esempio, di una videochiamata) sono apprezzabili/accettabili solo se tutti i pacchetti contenenti ognuno una parte dello stream arrivano con una breve distanza temporale tra l'uno e l'altro, anche a costo di perderne qualcuno o di riceverli in un ordine diverso rispetto a quello in cui sono stati inviati.

Si preferisce, quindi, usare per applicativi soft real-time una *connessione non affidabile,* ma veloce, piuttosto di una *connessione affidabile,* ma molto più lenta della prima. Nella pila protocollare del modello TCP/IP, questa scelta si traduce scegliendo di usare al *Livello Di Trasporto* il protocollo UDP (User Datagram Protocol) o TCP (Transmission Control Protocol).

In particolare il protocollo UDP permette un rapido scambio di pacchetti (segmenti) ma, proprio per ciò, non si occupa di controllare né che i pacchetti arrivino tutti a destinazione, né che arrivino nel giusto ordine. Non implementa nemmeno un controllo sulla congestione della rete (cioè, offre una connessione *non affidabile*)[5].

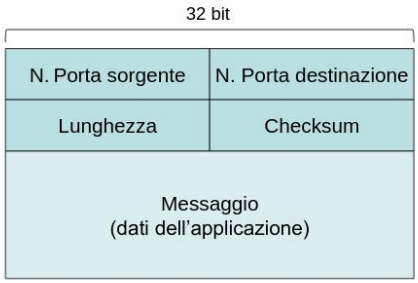

Figura 1 - Struttura segmento UDP [5].

Al contrario, il protocollo TCP pone le sue priorità sull'arrivo del pacchetto e nell'ordine prestabilito, piuttosto che sul tempo di arrivo (cioè, offre una connessione *affidabile*). Per questi motivi, il TCP comprende:

- una prima fase (*fase di handshake*) in cui stabilisce una connessione *unicast punto-punto* in grado di trasmettere dati in *full duplex*, ovvero una connessione con un solo destinatario ed un solo mittente, in cui i dati possono essere trasmessi in entrambe le direzioni contemporaneamente[6];

un controllo sulla sequenza di arrivo dei pacchetti, per verificare che sia corretta, attraverso i campi "Numero Di Sequenza" (SEQ) e "Numero Di Riscontro" (RIS) del segmento TCP, i quali si riferiscono rispettivamente al numero d'ordine del primo byte nel segmento all'interno del flusso di trasmissione e al byte sullo stream di ricezione corrispondente alla sequenza più il numero dei dati trasmessi [6];

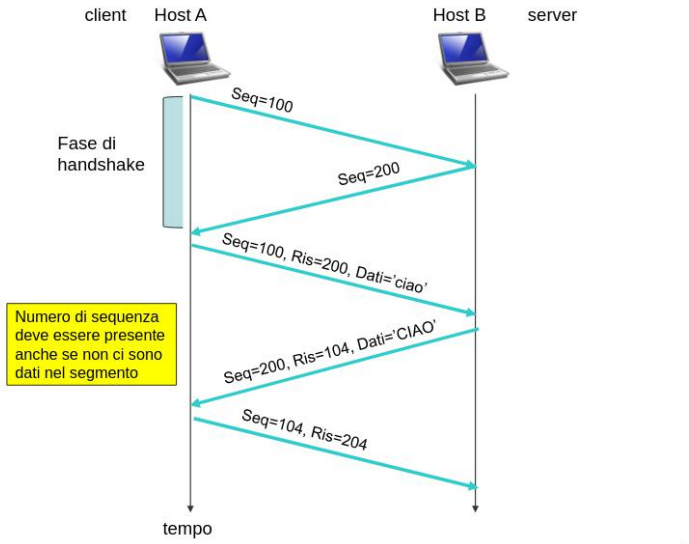

Figura 2 - Scambio di messaggi tramite connessione TCP [6]

un controllo sul tempo che impiega il pacchetto ad arrivare a destinazione e ritornare l'ACK; per garantire l'arrivo del pacchetto al destinatario calcola un *intervallo di timeout*, che è dato dalla stima dell'RTT (Round Trip Time, tempo che passa dall'invio del segmento alla recezione dell'ACK) più quattro volte la variazione dello stesso RTT stimato, passato il quale, se non ha riscontrato nessun pacchetto di ACK, ritrasmette il segmento [6];

infine, un controllo sulla congestione del traffico in grado di modulare la velocità della trasmissione del pacchetto[6] .

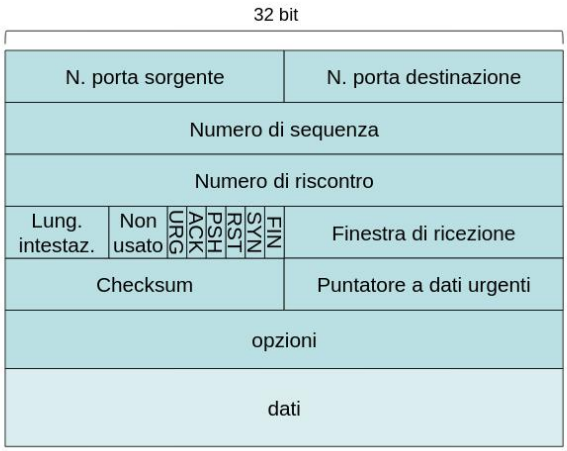

Figura 3 - Struttura segmento TCP [6].

La tecnologia webRTC è in grado di trasmettere sia tramite il protocollo TCP che tramite il protocollo UDP. A seconda della situazione sceglie automaticamente il protocollo migliore [2].

2.2 Connessione peer-to-peer

WebRTC crea un collegamento con un'architettura *peer-to-peer*, ovvero ogni host di tale architettura è chiamato *peer* (*pari*) e, a differenza dell'architettura client/server, può sia richiedere che offrire un servizio ad un altro peer. La tecnologia webRTC, ci consente quindi di stabilire una connessione che non necessita di un server che faccia da intermediario tra i due host che vogliono comunicare. Ciò si traduce in termini di maggiore efficienza e velocità.

3. ANALISI DEI PROTOCOLLI E DELLE TECNOLOGIE PER WEBRTC

3.1 Overview Connessione peer-to-peer con webRTC

Prima di descrivere i vari componenti necessari per la connessione peer-to-peer, si ritiene opportuno delineare il suo funzionamento.

L'obiettivo è quello di instaurare una connessione peer-to-peer tra due peer, in modo da non avere la necessità di servirsi di un server intermediario. Ma, per fare ciò, ognuno dei due peer deve essere a conoscenza di alcune informazioni sull'altro.

Nello specifico, supponiamo di voler instaurare una connessione peer-to-peer tra i peer A e B. A vuole creare una connessione con B, quindi crea una sdp offer, cioè una stringa contenente le informazioni necessarie per creare la

connessione. In seguito B crea dalla sdp offer generata da A la sdp answer. A questo punto, il peer A può finalmente creare la connessione con la sdp answer generata da B.

La fase in cui avviene lo scambio di queste stringhe si chiama *fase di Signaling*. Essa deve avvenire *prima* della stabilizzazione della connessione peer-to-peer. [12, 21]

Tuttavia, nelle specifiche webRTC non c'è nessuna tecnologia o tecnica indicata o consigliata per implementare questa fase. L'importante è che i due peer abbiano accesso alle stringhe sdp offer e sdp answer, il "come" è lasciato alla decisione del programmatore.

Per esempio, in un applicativo javascript ,che utilizza delle API webRTC, la fase di Signaling può essere implementata attraverso un server php, che si occupa di memorizzare le sdp offer e le sdp answer in un oggetto json: in questo modo, entrambi i peer hanno a disposizione in qualsiasi momento le stringhe generate. Si può osservare come questa fase viene risolta attraverso una architettura client/server. Tuttavia, terminato lo scambio, il server non interviene più in alcun modo sulla connessione peer-to-peer generata.

Per via della mancanza di indirizzi ipv4, la maggior parte degli host sono connessi ad Internet tramite router NAT (Network Address Translation); ciò significa che gli host dispongono di un ip privato ed in rete sono visibili tramite l'ip pubblico del router (*che non conoscono*). Ciò si rivela un problema per la tecnologia webRTC, perché il peer necessita di conoscere l'ip pubblico con il quale è presentato in rete per connettersi con l'altro peer. Tale problema si risolve grazie ai server STUN (Session Traversal Utilities for NAT), i quali, appunto, non fanno altro che restituire l'indirizzo pubblico con il quale l'host è visibile in rete.

Quando non si può creare una connessione peer-to-peer, i server TURN svolgono lo stesso ruolo dei server STUN e, in più, svolgono un servizio di ritrasmissione dei messaggi. In altre parole, se non si riesce a creare una efficiente connessione peer-to-peer, almeno si cerca di instaurare una connessione.

A questo punto, essendoci più server TURN e più server STUN, si hanno un sacco di opzioni per creare una connessione peer-to-peer. Ognuna di queste è chiamata *ice condidate*. Per utilizzare quella più efficace, ci viene in aiuto *ICE* (*Interactivity Connectivity Establishment*), il quale si occupa di collezionare tutti gli ice candidates disponibili, fra i quali scegliere il migliore per la creazione della connessione. [12, 18]

3.2 NAT

NAT è l'acronimo di *Network Address Translation* ed è un tipo di router che collega tanti nodi alla rete con il proprio ip pubblico. Il router, quindi, si occupa di instradare i messaggi che riceve dalla rete esterna al giusto host e di inviare i messaggi degli host fornendo il proprio ip pubblico.

In particolare, il router NAT assegna ad ogni host collegato ad esso un indirizzo privato (attraverso un server DHCP implementato su di esso). Gli insiemi di indirizzi ipv4 riservati a questa operazione sono 10.0.0.0/8, 172.16.0.0/12 e 192.168.0.0/16[2]. Quando un host vuole inviare un pacchetto ad un altro host che si trova al di fuori della rete locale, tale pacchetto avrà come ip del mittente quello privato assegnato all'host ed una porta LAN. Quando il router NAT riceve tale pacchetto (datagram), sostituisce l'indirizzo ip privato del mittente con il proprio ip pubblico e la porta LAN con una porta Internet. Oltre ad inoltrare il pacchetto, se non già presente *nella tabella delle traduzioni*, il router NAT vi aggiunge una riga contenente, appunto, l'ip pubblico, la porta Internet, l'ip privato e la porta LAN associati all'host. Tale riga permette di

[2] I blocchi di indirizzi 10.0.0.0/8; 172.16.0.0/12 e 192.168.0.0/16 sono degli indirizzamenti CIDR (Classless Inter-Domain Routing). Cioè sono del tipo **a.b.c.d/n** dove **n** è il numero di bit dell'indirizzo (partendo da sinistra) che rappresenta la rete [7].

associare ad un host della rete LAN identificato con "*ip privato e porta LAN*" con i corrispettivi "*ip pubblico e porta Internet*", con i quali l'host è identificato su Internet.

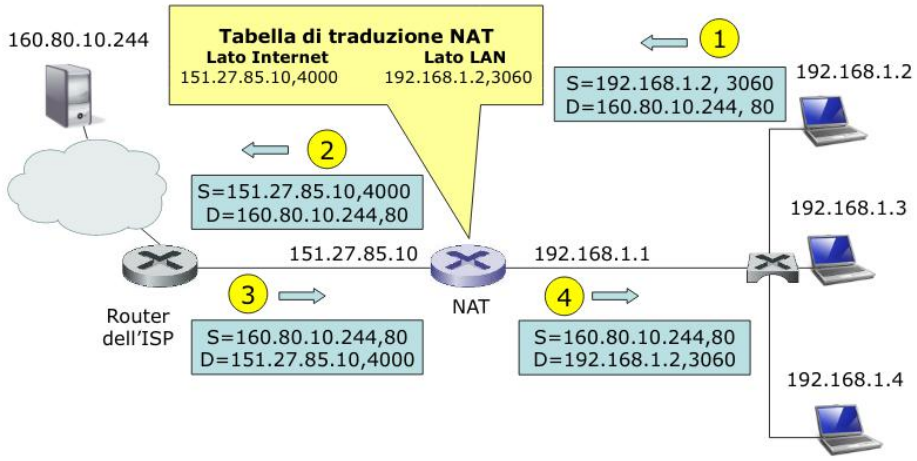

Figura 4 - Esempio di invio richiesta e ricezione risposta di un nodo collegato ad Internet tramite router NAT [7].

Viceversa, se il router riceve un pacchetto dalla rete che ha come destinatario un certo ip pubblico con la relativa porta internet, dalla tabella delle traduzioni riesce a ricavarne il corrispettivo ip privato e la porta LAN associata e, quindi, ad instradare il pacchetto al giusto nodo. Se non c'è tale corrispondenza (match), nella tabella delle traduzioni il pacchetto viene scartato.

IP Internet	Porta Internet	IP LAN	Porta LAN
151.27.85.10	4000	192.168.1.2	3060
151.27.85.10	4001	192.168.1.3	2050
151.27.85.10	5000	192.168.1.3	6700

Figura 5 - Esempio tabella di traduzione di un router NAT [7].

I NAT esistono per risolvere il grande problema dell'ipv4, che ha un numero limitato di indirizzi che non sono sufficienti per coprire tutti gli host. Infatti,

attraverso un router NAT si possono collegare fino a 65 536 nodi[3] con *un solo* ip pubblico[7].

Il router NAT può essere, sebbene non sia stato creato per tale motivo, un modo per garantire una certa sicurezza, limitando gli accessi provenienti dall'esterno: di fatto, il NAT si pone fra la rete privata ed Internet. Come descritto di seguito, ci sono diversi tipi di NAT e ognuno di essi decide secondo un certo *metodo di traduzione* quali richieste provenienti dall'esterno accettare e quali scartare.

3.2.1 One-To-One NAT (o Full-cone NAT)

I router One-to-one NAT non prevedono alcun tipo di restrizione: una volta che a un *ip privato e relativa porta LAN* vengono associati rispettivamente un *ip pubblico e una porta Internet* nella tabella di traduzione del router NAT, l'host a cui sono associati può ricevere un pacchetto da qualsiasi mittente [12].

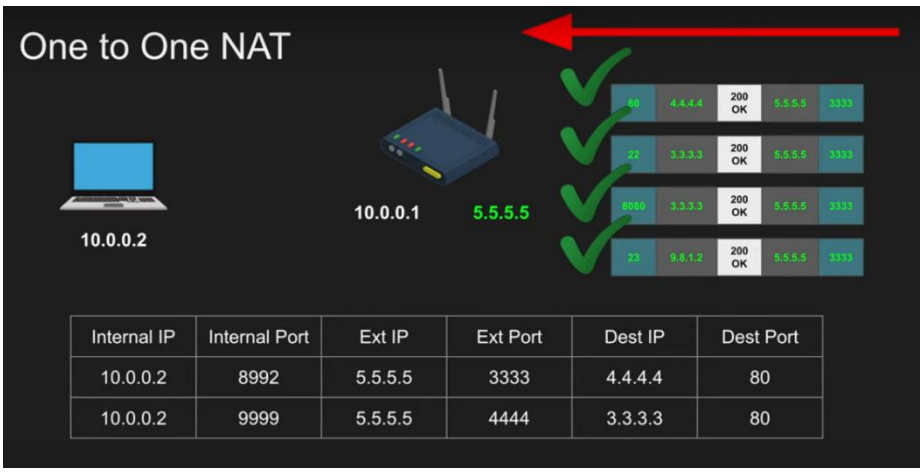

Figura 6 - Esempio di pacchetti provenienti da host esterni "accettati" da router One to One NAT [12].

[3] Dato che il numero di porta è composto di 16 bit, si hanno a disposizione 2^{16}= 65 536 porte a cui collegare un host.

3.2.2 Address Restricted-cone NAT

I router Address Restricted-cone NAT inoltrano un pacchetto proveniente da un host esterno nella rete LAN *solo se* l'indirizzo del mittente è gia presente nella tabella di traduzione del router. Ciò significa che un host esterno alla rete può comunicare con un host interno a condizione che abbia precedentemente ricevuto un pacchetto da un host della rete LAN, non ha importanza se da un host diverso da quello con cui vuole comunicare. In questa selezione non viene presa in considerazione la porta [12].

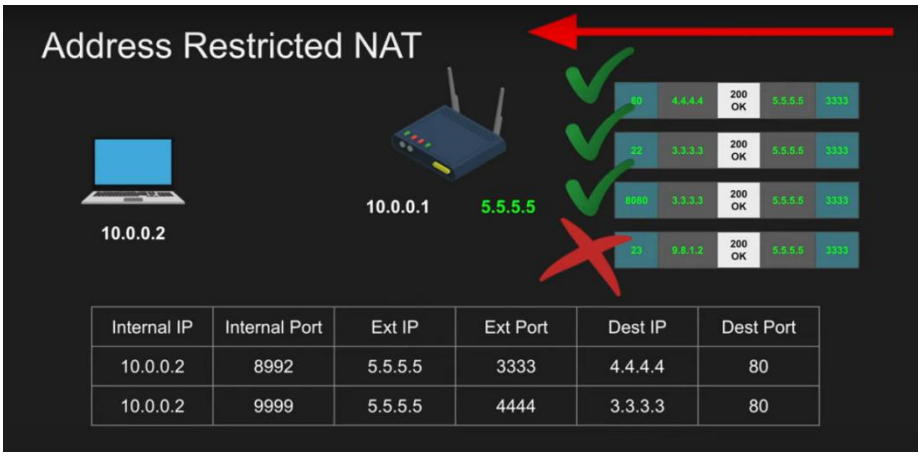

Figura 7 - Esempio di pacchetti "accettati" da router Address Restricted NAT [12].

3.2.3 Port Restricted-cone NAT

Port Restricted-cone NAT è simile all'Address Restricted-cone NAT Router, ma prende in considerazione anche la porta del mittente. Quindi, un pacchetto arrivato al router NAT da parte di un host esterno alla rete LAN del router stesso, viene accettato *solo se* nella tabella di traduzione sono già presenti il suo indirizzo ip e la sua porta, ovvero se è presente una riga contenente l'ip pubblico e la porta dell'host esterno (mittente), il che implica che un host interno ha già inviato un pacchetto all'indirizzo (ip pubblico, porta) dell'host esterno [12].

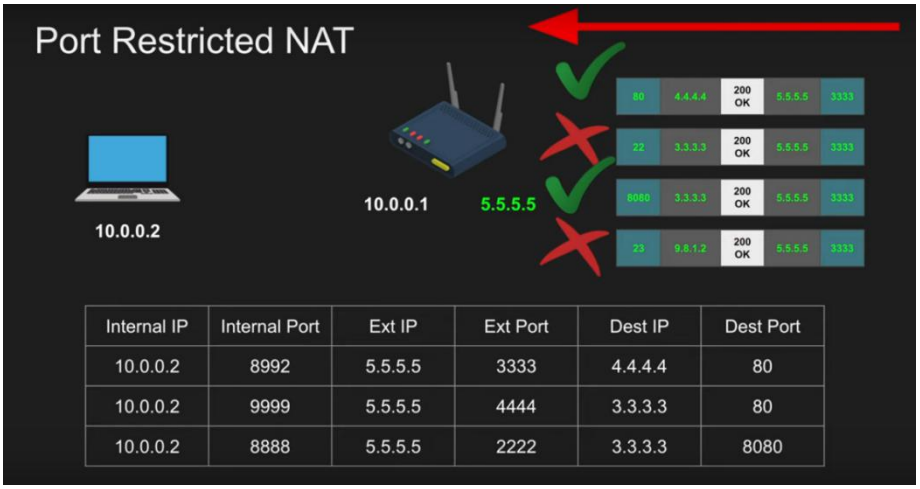

Figura 8 - Esempio di pacchetti accettati da router Port Restricted NAT [12].

3.2.4 Symmetric NAT

Tra tutti i metodi di traduzione, il Symmetric NAT è quello più restrittivo. Infatti, accetta solo messaggi da host esterni che inviano ad un indirizzo (ip, porta) di un host interno alla rete, il quale ha già inviato precedentemente un messaggio allo stesso indirizzo (ip_host_esterno, porta_host_esterno) dell'host esterno che tenta di contattarlo [12].

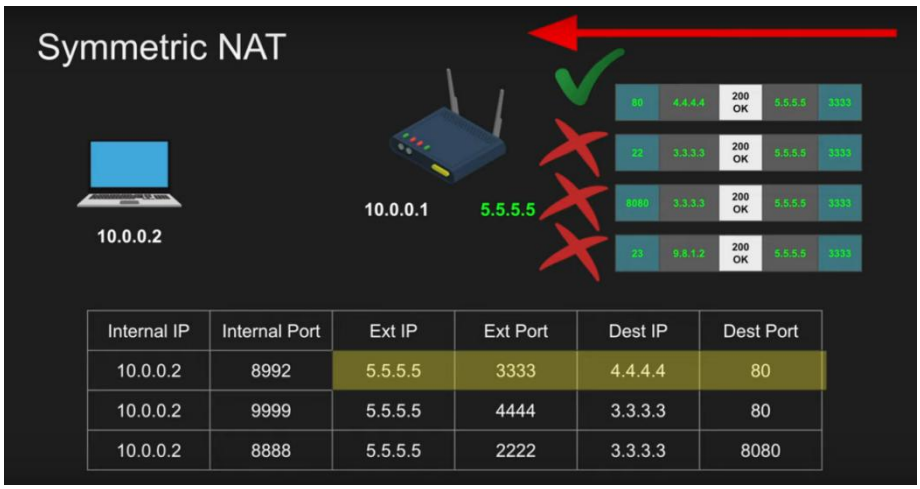

Figura 9 - Esempio di pacchetti accettati da router Symmetric NAT [12].

3.3 STUN

Lo STUN (Session Traversal Utilities for NAT) si riferisce a dei server che, ad una richiesta di un client, rispondono con l'ip pubblico e la porta Internet del client stesso. Il server STUN è fondamentale nel processo della connessione peer-to-peer con webRTC, perché permette anche ai peer collegati ad Internet

tramite un router NAT di conoscere l'ip e la porta con i quali sono visibili nella rete; ciò è necessario perché la connessione possa instaurarsi con successo. In particolare, un server STUN è utile se l'host è connesso ad Internet tramite un *full-cone NAT, un address-restricted-cone NAT o un port-restricted-cone NAT*. Per quanto riguarda i *symmetric NAT* è necessario mettere in collegamento i due peer tramite un server TURN [12].

I server STUN sono "economici" da mantenere, infatti, si può notare come svolgano solo una funzione estremamente semplice: restituire l'indirizzo pubblico del client della richiesta [12].

In generale, si usa il server TURN (al posto del server STUN) quando non è possibile creare una connessione diretta (peer-to-peer) tra i due peer, ovvero quando non si riesce a conoscere gli indirizzi pubblici che identificano in modo univoco i due peer nella rete, come avviene, ad esempio, quando le reti aziendali o governative usano firewall per proteggere l'accesso a queste informazioni [14].

Quindi, attraverso un server TURN *non* si instaura una connessione peer-to-peer.

3.3.1 Server STUN con Full-cone NAT

Il server STUN può essere utilizzato con il router Full-cone NAT.

Siano il peer A e il peer B due host collegati in due reti LAN differenti e siano entrambi collegati ad Internet tramite un router NAT, allora il collegamento peer-to-peer avviene in diversi step.

1. A invia una richiesta al server STUN, richiedendogli il proprio indirizzo ip e la porta con i quali è visibile in Internet. Si può osservare che tali dati

corrispondono all'ip pubblico del router NAT e alla porta Internet che il router stesso ha assegnato ad A.

2. Il server STUN invia un pacchetto in cui inserisce l'indirizzo ip e la porta internet del mittente della richiesta.

3. A riceve la risposta del server STUN ed entra a conoscenza del proprio indirizzo pubblico. Si può osservare che il router NAT, effettivamente, scambia l'ip pubblico e la porta Internet con l'ip privato e la porta LAN di A, ma tali dati ora sono presenti anche nel corpo del messaggio e, per questo motivo, A può entrarne in possesso,nonostante il router NAT.

4. In modo analogo, B richiede e riceve l'indirizzo ip e la porta con i quali è visibile in rete. È da notare come dopo questi scambi di messaggi con il server STUN, nelle tabelle di traduzione dei rispettivi peer esiste un match tra l'indirizzo pubblico e l'indirizzo privato dei peer. Tale match si è creato nella fase di invio della richiesta al server STUN.

5. Ora il peer A può inviare la sua richiesta di connessione, che comprende anche il proprio ip pubblico e la propria porta Internet.

6. Essendo il router NAT, al quale è collegato B, un full-cone NAT e, dato che esiste un match tra l'indirizzo pubblico e privato di B nella tabella di traduzione, il router instraderà la richiesta di connessione di A a B.

7. B risponde alla richiesta e, per motivi analoghi a quanto scritto nel punto 6, il router NAT al quale è collegato A gli instraderà il messaggio di B.

La connessione è ora instaurata e i due peer possono comunicare senza aver più bisogno dell'intervento del server STUN [12].

3.3.2 Server STUN con Address-restricted-cone NAT

L'utilizzo del server STUN con un Address-restricted-cone NAT per conoscere l'indirizzo pubblico dell'host avviene in modo analogo al precedente (par. 3.3.1) con la differenza che l'ip pubblico richiedente sia presente nella tabella di traduzione del router NAT a cui è collegato il ricevente. Siano A e B due peer, posto che A voglia creare una connessione con B, un nodo collegato al router NAT al quale è collegato anche B deve aver inviato un messaggio ad A (su una qualsiasi porta) prima che questi possa tentare di instaurare una connessione con B. Infatti, solo in questo modo per il metodo di traduzione Address-restricted-cone del router NAT, la richiesta di A verrà accettata dal router ed inoltrata a B [12] .

3.3.3 Server STUN con Port-restricted NAT

L'utilizzo del server STUN con un Port-restricted NAT avviene in modo analogo a quello con il router Address-restricted-cone NAT, ma con una condizione nella selezione dei pacchetti da parte del router più forte. Infatti, siano A e B due peer e posto che A voglia creare una connessione peer-to-peer con B, la richiesta di A verso B verrà fatta passare dal router NAT solo se l'indirizzo pubblico e la porta Internet di A sono presenti nella tabella di traduzione del router, ovvero, solo se prima della richiesta di A, un nodo collegato al router NAT a cui è collegato B ha inviato un pacchetto all'indirizzo pubblico e alla porta Internet con i quali A vuole fare la richiesta [12].

3.3.4 Server STUN con Symmetric NAT

Un procedimento differente va seguito quando almeno uno dei due peer è collegato ad Internet tramite router Symmetric NAT.

Siano A e B due nodi tali che appartengono a reti LAN differenti ed almeno uno dei due è collegato alla rete tramite Symmetric NAT, allora, tra i due non si può

creare una connessione peer-to-peer, per via delle restrizioni troppo forti del metodo di traduzione Symmetric NAT. Infatti, sono consentiti solo i pacchetti provenienti da host (ip_mittente, porta_mittente) che hanno precedentemente ricevuto un pacchetto dal nodo che vogliono raggiungere (ip_destinatario, porta_destinatario). Per tale motivo, in questo caso al posto del server STUN entra in gioco il server TURN, il quale fa da intermediario sicuro (fidato) tra i due host. [12]

3.4 TURN

Nella maggior parte dei casi, la connessione webRTC viene stabilita attraverso un server STUN e attraverso l'architettura peer-to-peer. Ma, quando ciò non è possibile, per esempio se si vogliono collegare due host connessi ad Internet attraverso il router Symmetric NAT oppure se si vogliono collegare due host il cui indirizzo pubblico è protetto da firewall, allora si riesce a creare una connessione attraverso un server TURN. Tuttavia, tale connessione non è peer-to-peer, cioè i due host non sono collegati direttamente l'uno con l'altro, ma per far in modo che il primo host possa inviare un messaggio al secondo, deve utilizzare il server TURN (e viceversa). In altre parole, invia il messaggio al server TURN che, a sua volta provvederà ad inoltrarlo al destinatario. I due host accetteranno i pacchetti da parte del server TURN perché è stato inserito tra le sorgenti sicure (fidate) dai rispettivi router NAT quando ognuno dei due host gli ha richiesto il proprio indirizzo pubblico.

TURN (Traversal Using Relay around NAT), quindi, si occupa sostanzialmente di ritrasmettere i messaggi da un host ad un altro. A differenza del server STUN, è molto costoso da mantenere [12, 14].

Di fatto, un server TURN implementa il server STUN più un servizio di inoltro dei pacchetti ricevuti verso il destinatario [3].

3.5 ICE

ICE (Interactive Connectivity Establishment) viene utilizzato per trovare la soluzione migliore per creare una connessione peer-to-peer che sia il più efficace possibile. Infatti, si possono combinare diverse scelte in gruppi, di modo che ogni gruppo di opzioni sia in grado di creare una connessione peer-to-peer tra i due host. Per esempio, si può scegliere di utilizzare UDP come protocollo al livello di trasporto ed un server STUN, piuttosto che scegliere di utilizzare il protocollo TCP ed un server TURN, oppure, ancora scegliere un server STUN piuttosto di un altro (ce ne può essere più di uno disponibile).

L'insieme di tutte le scelte necessarie per instaurare una connessione viene chiamato **ICE candidate: i**n altre parole, si tratta di una (lunga) stringa scritta seguendo il protocollo **SDP** (**Session Description Protocol**), che descrive un formato in cui scrivere tutte le scelte fatte sulle opzioni necessarie per instaurare la connessione.

Quindi, ICE colleziona tutti i possibili ICE candidates [18, 12].

Come abbiamo visto, in ogni situazione possono esserci più ICE candidate possibili. Tuttavia, qualora la connessione sia possibile, tra i diversi ICE candidate ce n'è sicuramente uno che garantisce di instaurare una connessione migliore di quella che si potrebbe creare con gli altri, cioè che ha il minor ritardo possibile. Quindi è necessaria un'operazione di ricerca per individuare ed usare l'ICE candidate migliore. Tale processo viene automatizzato da ICE che, nello specifico, tenta di instaurare una connessione con i seguenti gruppi di opzioni, provandole una alla volta nell'ordine in cui sono presentate di seguito:

1. Connessione UDP Diretta (cioè, peer-to-peer), con server STUN.

2. Connessione TCP Diretta (cioè, peer-to-peer), su porta HTTP, con server STUN.
3. Connessione TCP Diretta (cioè, peer-to-peer), su porta HTTPS, con server STUN.
4. Connessione Indiretta (cioè, non peer-to-peer) per ritrasmissione, ovvero connessione tramite server TURN.

Se la creazione della connessione con il primo gruppo di opzioni fallisce, prova con il successivo, finché non avviene con successo.[2, 18]

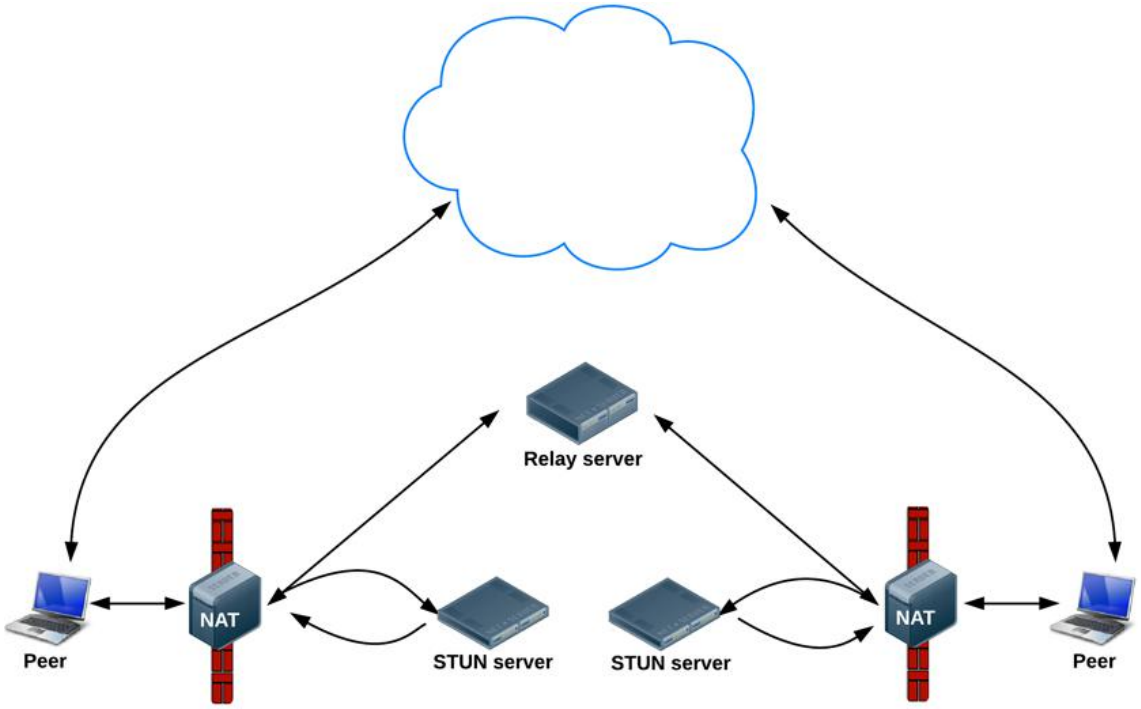

Figura 10 - Ricerca candidati per la connessione webRTC [4].

3.6 SDP

SDP (Session Description Protocol) è un protocollo per la descrizione di una connessione SDP, che contiene codec[4], indirizzi sorgenti e informazioni sul tempo di audio e video [20].

[4] codec deriva da coder-decoder ed è un programma, algoritmo o dispositivo che codifica o decodifica uno stream di dati [16].

Perché ci sia una connessione peer-to-peer fra due peer, è necessario che il peer che vuole instaurare la connessione generi l'offerta SDP (SDP offer) e da questa, l'altro peer generi la risposta SDP (SDP answer). Queste due stringhe contengono tutte le informazioni necessarie perché si possa instaurare la connessione peer-to-peer. Tali informazioni sono scritte seguendo il formato descritto da SDP. [12, 20]

```
v=0
o=alice 2890844526 2890844526 IN IP4 host.anywhere.com
s=
c=IN IP4 host.anywhere.com
t=0 0
m=audio 49170 RTP/AVP 0
a=rtpmap:0 PCMU/8000
m=video 51372 RTP/AVP 31
a=rtpmap:31 H261/90000
m=video 53000 RTP/AVP 32
a=rtpmap:32 MPV/90000
```

Figura 11 - Esempio di SDP [20].

3.7 Fase di Signaling

È necessario che i due peer si scambino la SDP Offer e l'SDP Answer affinché entrambi abbiano le informazioni necessarie sull'altro per instaurare con successo una connessione. Ciò viene effettuato nella *Fase di Signaling* la quale, però, non è supportata da nessuna API webRTC e nemmeno è indicata una soluzione da preferire. Tutto ciò che viene specificato è che lo scopo di questa fase consiste dapprima nel rendere accessibile l'SDP Offer[5] al peer al quale ci si vuole connettere, tale peer potrà in questo modo generare la SDP Answer da inviare al peer di partenza, che finalmente ha tutte le informazioni necessarie

[5] Contestualizzando, si sta parlando della prima fase per instaurare una connessione con webRTC. Il peer che vuole generare una connessione con un altro peer, deve generare una sdp offer da inviargliela. Nella frase si fa riferimento a tale sdp offer.

e può avviare la connessione. Il “come” fare ciò è lasciato da decidere al programmatore; si può scegliere un qualsiasi modo per inviare le due stringhe, come per esempio tramite WhatsApp, Facebook, Twitter, E-mail, etc ...

Nell’applicativo d’esempio riportato, si è deciso di implementare questa fase attraverso un server php, che salva e rende disponibile la domanda e l’offerta sdp in un oggetto json. In questo modo, entrambi i peer hanno accesso alle informazioni necessarie.

Per instaurare una connessione peer-to-peer con la tecnologia webRTC è necessario che i due peer abbiano già in precedenza, in qualche modo, comunicato (per scambiarsi i file sdp), ovvero sia stata già instaurata una connessione tra di essi. Quindi, viene naturale chiedersi quale sia l’utilità di questa tecnologia se funziona solo insieme ad un altro modo di creare una connessione. Il vantaggio è duplice:developer - MediaStream

1 - *Una connessione efficace,* infatti, la tecnologia webRTC si preoccupa di scegliere le configurazioni ottimali per instaurare una connessione con la più bassa *latency* possibile.

2 - La s*emplicità delle API webRTC, in quanto, c*reare un applicativo con webRTC significa avere a disposizione molte API che semplificano notevolmente la fase dello sviluppo.

In seguito vedremo le API principali [12, 21, 24].

4. API webRTC

WebRTC implementa le seguenti tre APIs:

1. MediaStream (getUserMedia);
2. RTCPeerConnection;
3. RTCDataChannel.

Queste 3 APIs sono supportate sul mobile e sul desktop:
sul desktop, in particolare da Safari, Firefox, Edge, Chrome e Opera [4].

4.1 MediaStream (getUserMedia)

MediaStream è una interfaccia che rappresenta uno stream di dati, dove uno stream è composto da tante track. Ogni track si traduce come una istanza di MediaStreamTrack e rappresenta una singola traccia, generalmente video o audio [19].

Quindi, MediaStream si occupa di catturare l'output del microfono e della camera del dispositivo (computer o mobile). Il modo più comune per fare ciò è attraverso l'oggetto *navigator.mediaDevices* che implementa, appunto MediaStream. In particolare si usa il metodo *getUserMedia()* che *prende in input* un oggetto json che descrive (secondo l'intefaccia MediaStreamConstraints) i requisiti da seguire (constraints) e *restituisce in output* una *promise,* che verrà risolta in un MediaStream il quale descrive, appunto, lo stream generato dai dispositivi individuati e che seguono i vincoli (constraints) passati in input. Il metodo *getUserMedia()* richiede il permesso all'utente per l'uso dei dispositivi. Se l'utente rigetta tale permesso l'acquisizione dello stream fallisce e viene lanciato il **PermissionDeniedError**. Tale processo fallisce anche se non viene trovato nessun dispositivo collegato

che rispecchia i vincoli necessari; in questo caso viene invece lanciato il **NotFoundError** [25].

```
const constraints = {
'video': true,
'audio': true
}

navigator.mediaDevices.getUserMedia(constraints)
    .then(stream => {
        console.log('Got MediaStream:', stream);
    })
    .catch(error => {
        console.error('Error accessing media devices.', error);
});
```

Figura 12 - Esempio di cattura dello stream di audio e video [25].

4.1.1 Vincoli e Stream

Come già anticipato, si possono definire dei vincoli (constraints) che andranno a vincolare lo stream (istanza di MediaStream) restituito da *getUserMedia(constraints)*. Tuttavia, questi vincoli possono essere ignorati, possono far fallire la creazione dello stream o non essere quelli effettivamente impostati. Di seguito, si andranno a chiarire questi aspetti.

Il processo di applicazione dei vincoli avviene in diversi step.

1. Se necessario, si richiama *MediaDevices.getSupportedConstraints()*. Tale funzione permette di avere la lista delle proprietà vincolabili conosciute dal browser. Essa è necessaria solo per essere certi che contenga una determinata proprietà; infatti, *le proprietà non presenti in tale lista vengono semplicemente ignorate dal browser*.

2. Dopo aver analizzato il supporto del browser, analizza le “capacità” (**capabilities**) delle API con il metodo *getCapabilities()* di una traccia che restituisce i vincoli e i valori supportati su tale traccia.

3. Infine, viene richiamato il metodo della traccia *applyConstraints()* per applicare i vincoli desiderati.

A questo punto è possibile vedere i vincoli passati nell'ultimo *applyConstraints()* con *getConstraints(). Questi ultimi possono non essere i vincoli effettivamente impostati,* infatti, una volta determinati i vincoli voluti può capitare che alcuni di questi vengano ignorati (si veda il punto 1 nel processo dell'applicazione dei vincoli) oppure che alcuni valori assegnati debbano essere modificati; inoltre, i valori di default della piattaforma non vengono mostrati. I vincoli effettivamente applicati si chiamano *settings* e si ottengono con *getSettings().*

L'ottenimento dello stream può fallire anche a causa dei valori assegnati ai vincoli. Un vincolo (constraint) è formato dal nome della proprietà che si vuole vincolare e dal valore che gli si desidera assegnare. Ci sono due modi per definire i valori dei vincoli: con un valore specifico e con un range di valori.

Quando si definisce un vincolo con un valore specifico, tale valore viene considerato *non necessario* (*not required*) e pertanto la richiesta *non fallisce,* anche se tale valore potrebbe essere modificato a seconda della necessità.

```
let constraints = {
width: 1920,
height: 1080,
aspectRatio: 1.777777778};
```

Figura 13 - Esempio di definizione di constraints con valori specifici [15].

Mentre, se si definisce un vincolo con un range di valori, allora si assegna al vincolo un oggetto json in cui si può definire un valore "min", "max" oppure "exact". In questo caso, tali valori vengono considerati *obbligatori* (*mandatory*), quindi *se tali vincoli non possono essere rispettati l'ottenimento dello stream fallisce [15].*

```
let constraints = {
    width: { min: 640, ideal: 1920, max: 1920 },
    height: { min: 400, ideal: 1080 },
    aspectRatio: 1.777777778,
    frameRate: { max: 30 },
    facingMode: { exact: "user" }
};
```

Figura 14 - Esempio di constraints con range di valori [15].

4.1.2 Stream Locale

Dopo aver creato lo stream con getUserMedia()[6] (si veda 4.1) è necessario inserirlo nel corpo del sito[7] di modo che sia visibile. Più in particolare bisogna associare lo stream ad un *oggetto html* **video** contenuto nel *body* della pagina. Per fare ciò, si inserisce, appunto, un *tag video* nel *body* come mostrato di seguito:

```
<html>
<head><title>Local video playback</video></head>
<body>
<video id="localVideo" autoplay playsinline controls="false"/>
</body>
</html>
```

Figura 15 - Esempio di definizione di oggetto HTML video [25].

Quindi, si associa a tale oggetto html lo stream, come segue:

```
document.querySelector('video#localVideo').srcObject = stream;
```

Figura 16 - Associazione dello stram all'oggetto video [25].

A questo punto lo stream sarà visibile tramite l'oggetto html video [25].

---

[6] Cioè dopo aver creato una costante nominata "stream" e dopo avergli assegnato lo stream dei dispositivi impostati con il seguente comando:

const stream = await navigator.mediaDevices.getUserMedia(constraints);

NOTA: tale comando può essere eseguito solo in una async function.

[7] Si intende un sito web che si vuole realizzare con l'appoggio della tecnologia webRTC

4.2 RTCPeerConnection

RTCPeerConnection è l'oggetto che si occupa di gestire la connessione. Il suo costruttore prende in input un oggetto RTCConfiguration, che si occupa di descrivere come è impostata la connessione e contiene informazioni sui server ICE da usare [25].

Quindi, per instaurare una connessione bisogna creare un oggetto RTCPeerConnection, passandogli nel costruttore le informazioni sui server ICE da usare. Ciò va fatto su entrambi i peer.

```
const configuration = {'iceServers': [{'urls': stun:stun.l.google.com:19302'}]}
const lc = new RTCPeerConnection(configuration);
```

Figura 17 - Esempio di creazione di un oggetto per la instaurazione e gestione di una connessione peer-to-peer

Dal peer che vuole aprire la connessione si deve creare la stringa sdp offer con *createOffer()*. Quindi, l'oggetto creato viene settato come descrizione locale della connessione con *setLocalDescription()*. Infine, è necessario inviare l'offer all'altro peer in modo che quest'ultimo possa generare da questa la sdp answer. Nell'esempio sottostante, la sdp offer viene inviata ad un server php dal quale sarà accessibile all'altro peer.

```
/*
* Quando viene creata una sdp offer, la invia al server
*/
lc.onicecandidate = e =>{
    if( e.candidate ){
        // se c'è un altro candidato, lo aggiunge alla lista dei candidati
    } else { // se non ci sono più candidati, invia la sdp offer generata al signaling server
        send_offer_to_sig_server( JSON.stringify( lc.localDescription ) );
    }
}

// crea l' offer; quindi la imposta come descrizione locale della connessione

lc.createOffer().then( o => lc.setLocalDescription( o ) );
```

Figura 18 - Nell'applicativo di esempio, l'offer_room invia la SDP Offer al Signaling Server

Quando il peer che ha generato la offer sdp riceve anche la answer sdp, allora gli basta settare l'answer come descrizione remota con *setRemoteDescription()* e la connessione è aperta.

Come anticipato, anche sul peer che riceve la richiesta di connessione va creato un oggetto RTCConnection. Dopo aver ricevuto la offer, si genera la answer sdp e si setta la offer come descrizione remota e la answer come descrizione locale.

4.3 RTCDataChannel

RTCDataChannel è un'API che permette a due peer di scambiarsi qualsiasi tipo di dato su una connessione creata con RTCConnection. Per creare un DataChannel basta eseguire il metodo *createDataChannel* sulla RTCConnection del peer locale che ha richiesto la connessione. Sul peer remoto invece, *non* va creato un nuovo DataChannel, ma va recuperato quello creato su tale connessione. Quindi, si mette in ascolto sull'evento "datachannel" che restituirà RTCDataChannelEvent. Quest'ultimo ha una proprietà chiamata "channel" che contiene proprio il DataChannel che è stato creato sulla comunicazione dei due peer. In questo modo entrambi i peer hanno accesso al DataChannel.

PEER LOCALE

```
const peerConnection = new RTCPeerConnection(configuration);
const dataChannel = peerConnection.createDataChannel();
```

Figura 19 - Esempio di creazione di DataChannel

PEER REMOTO

```
const peerConnection = new RTCPeerConnection(configuration);
peerConnection.addEventListener('datachannel', event => {
    const dataChannel = event.channel;
});
```

Figura 20 - Esempio di recupero di DataChannel da un connessione RTCPeerConnection

Una volta creato un DataChannel, si possono inviare dei messaggi con il metodo *send(),* passandogli come parametri i dati da inviare, che possono essere di tipo String, Blob, ArrayBuffer o/e ArrayBufferView. È possibile, invece, gestire l'arrivo di messaggi mettendo un listener sull'evento "message".

Infine, si può gestire l'apertura e la chiusura del DataChannel mettendo un listener in ascolto sugli eventi "open" e "close" [25].

5. L'USO DELLE API WebRTC IN UN APPLICATIVO D'ESEMPIO

Dopo aver trattato la complessità che c'è sotto questa tecnologia, si mostra di seguito un esempio pratico, descrivendo un applicativo per effettuare le video chiamate tramite una connessione peer-to-peer.

5.1 Introduzione Esempio

L'obiettivo di questo esempio è quello di mostrare i vantaggi e gli svantaggi individuati nella sperimentazione della tecnologia webRTC. Nella fattispecie, si è usata questa tecnologia per lo sviluppo di un applicativo in grado di mettere in videochiamata due peer. Questo perché si è scelto di sperimentare questa tecnologia sviluppando funzionalità in cui le API webRTC riescono a dare un contributo significativo. Quindi, siano A e B due computer situati in due reti LAN differenti e posto che A voglia effettuare una videochiamata con B, A accede all'index della cartella local_peer_RTC. Da qui può impostare un commento ed una password (opzionali) e poi creare una "room" cliccando sul bottone "create room". Il commento/descrizione della room serve a dare una (breve) indicazione su "con chi" si vuole effettuare la videochiamata o di cosa si vuole parlare, per esempio "Room per fan di FRIENDS". In questo caso, potrebbe essere "Room riservata a B". Invece, la password serve per consentire l'accesso alla "room" solo ai suoi possessori. Per esempio, il peer A può inserire una password "psw123" concordata con B, così da essere sicuro di avviare la videochiamata con quest'ultimo.

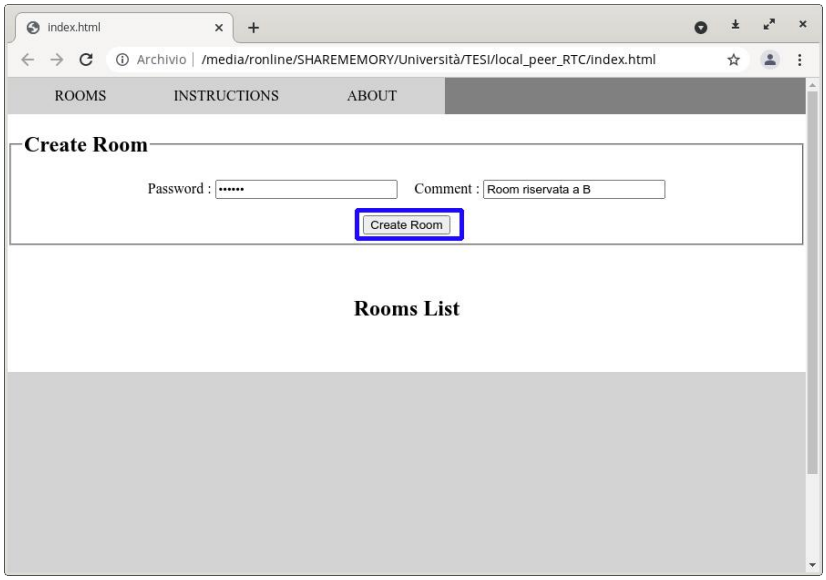

Figura 21 - Creazione di una room.

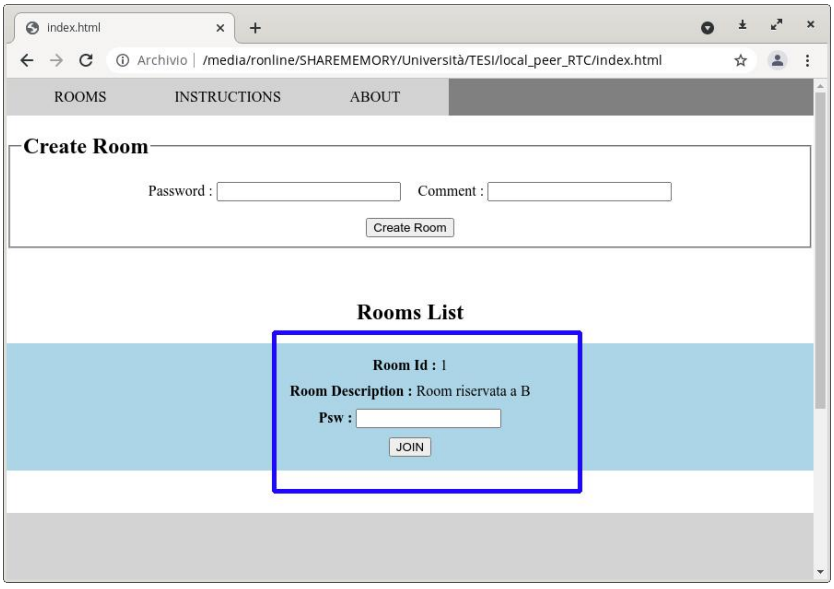

Figura 22 - Dopo che una room è stata creata, compare nella Rooms List.

Prima di procedere è necessario chiarire che cosa si intende con "room"[8] ("stanza"). Sulla pagina di index[9] vi possono accedere più di due host, ma l'obiettivo dell'applicativo è quello di sperimentare una connessione

[8] Non si tratta di un termine tecnico, ma di un termine fittizio per rappresentare un componente dell'applicativo come descritto.

[9] Cioè la pagina principale del sito.

peer-to-peer limitata a due peer[10]. Si può quindi immaginare di costruire una stanza("room") vuota, a cui possono accedere solo due utenti che rimangono isolati dagli altri. Quindi per "room" si intende, in generale, quello spazio dedicato solo a due utenti, dove possono comunicare in modo privato. In particolare, e più precisamente nel server PHP, per "room" si intende un oggetto che segue la notazione di un array json memorizzato su un file contenuto nel server. Tale oggetto contiene le informazioni necessarie per creare la connessione peer-to-peer (in particolare le rispettive sdp offer ed answer) e si traduce graficamente sull'index dell'applicativo come un rettangolo in cui un utente può leggere l'id e l'eventuale commento della room disponibile. A questo punto, se un utente vuole effettuare una videochiamata con l'utente che ha creato la room gli basterà cliccare sul bottone "JOIN" inserendo, eventualmente, la password associata alla room. Se non vi è stata associata nessuna password, bisogna lasciare il corrispettivo campo vuoto. Ogni rettangolo rappresenta una room diversa.

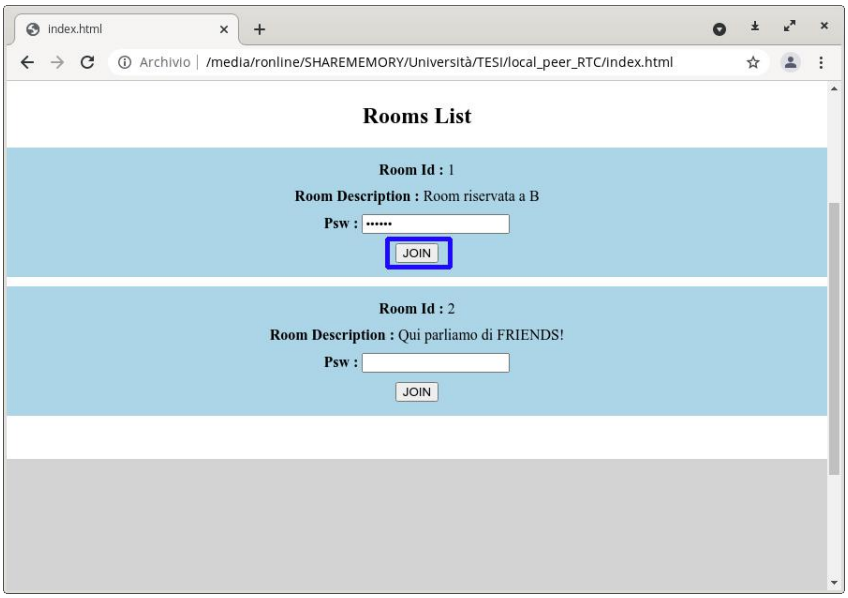

Figura 23 - Connessione con la Room che ha id 1

Inoltre, si indica con "room" anche la pagina che si presenta una volta instaurata la connessione, dove, cioè, è possibile inviare messaggi e comunicare in videochiamata con l'altro utente. Si distinguono in particolare le

[10] In modo analogo alla scelta di sviluppare un applicativo per fare videochiamate, la tecnologia webRTC riesce ad esprimere meglio il suo potenziale in una connessione tra due soli host.

pagine *offer_room* ed *answer_room*, che sono rispettivamente la pagine che vede il peer che ha generato l'offer sdp ed il peer che ha generato l'answer sdp. Dal punto di vista dell'utente, fra le due non c'è nessuna differenza, ma svolgono operazioni diverse nel processo di instaurazione della connessione.

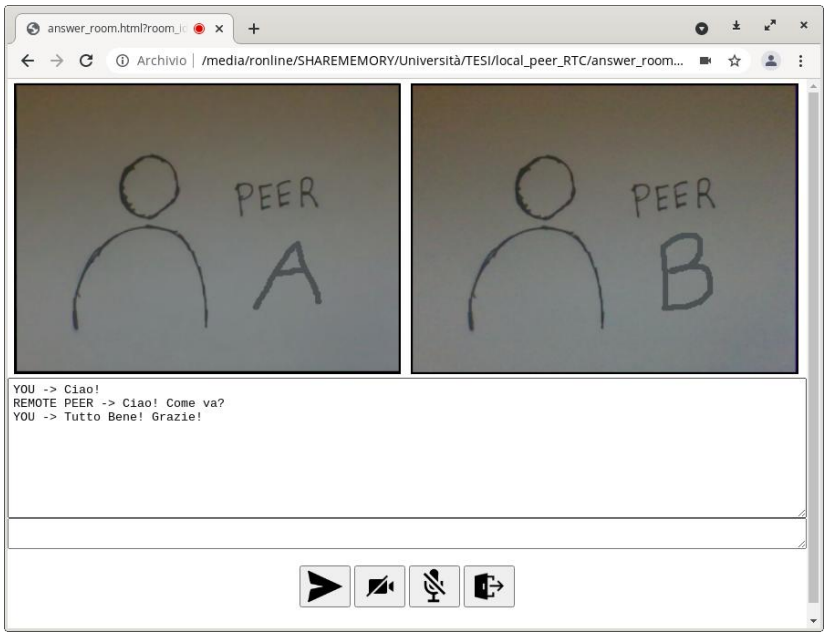

Figura 24 - Videochiamata fra due peer.

Una volta stabilita la connessione entrambi gli utenti possono abilitare/disabilitare video ed audio, scambiarsi messaggi oppure uscire dalla room (e quindi chiudere il collegamento).

5.2 Server PHP

Una volta che si sono create le sdp offer ed answer, i due peer devono in qualche modo scambiarseli. Ciò è necessario per l'instaurazione di una connessione peer-to-peer e non è supportato dalle API webRTC. Si deve, cioè implementare la *fase di Signaling*, decidendone anche il "come". Nella fattispecie si è deciso di inviare tali stringhe ad un server php che le memorizza in un oggetto json (salvato su file). Tale oggetto è, ovviamente, accessibile da qualsiasi host connesso ad Internet. Quindi, siano A e B due peer e posto che A voglia instaurare una connessione con B, allora, A clicca su "create room" per generare una nuova room come descritto sopra (si veda 5.1) . Si apre, quindi la

pagina offer_room e viene chiesto il permesso di accesso a videocamera e microfono (si veda 4.1). Se tale permesso viene negato la connessione non può avere luogo e la room non viene creata. L'utente viene re-indirizzato sulla schermata principale (index). Se, invece l'utente acconsente, il peer genera una sdp offer e la invia al server con la seguente istruzione:

```
send_offer_to_sig_server( JSON.stringify( lc.localDescription )    );
```

Figura 25 - Comando per l'invio dell'SDP Offer al server.

dove, lc è la variabile contenente la "local connection" e la sdp offer è data dalla "local description" della "local connection" lc[11]. Come sarà illustrato in seguito, *Signaling Server* prende sempre in input il parametro GET *cmd* (abbreviativo di command), il cui valore identifica le operazioni che deve compiere il server. Quindi, nella funzione *send_offer_to_sig_server* viene inviata una richiesta al server php (con l'istruzione *fetch*), per creare una nuova room con l'answer generata[12]. A questo punto la *offer_room* avvia un ciclo infinito, in cui ogni cinque secondi invia un richiesta al server per controllare se sia stata inserita una answer sdp nella room generata (attesa attiva). In caso di risposta affermativa, esce dal ciclo ed imposta la answer come "descrizione remota" di lc, instaurando così la connessione. Richiede infine al server di cancellare la room[13] dal file contenente la Rooms List in quanto ormai non è più utile.

Per quando riguarda il peer B, cliccando sul bottone "join" della room creata dall'*offer_room*, entra nella *answer_room*. Da qui richiede la offer sdp di A, da

[11] Una volta creata la offer sdp dal peer A, questo la imposta come "descrizione locale" della connessione lc (istanza di RTCConnection).

[12] In generale il server signaling lavora con GET, tuttavia la answer e la offer sdp sono troppo grandi per essere inviati tramite metodo GET così il signaling server è in progettato per recuperare questi dati sia in GET che in POST (senza doverlo specificare). Se entrambi i metodi contengono dei dati viene considerato il contenuto di POST. Nell'applicativo le richieste che vogliono inviare o la offer o la answer al server lo fanno tramite POST.

[13] In questo caso ci riferiamo all'oggetto json salvato sul Signaling Server e che si traduce graficamente in un rettangolo nell'index del sito.

cui genera la answer sdp che invia al *Signaling Server*. L'operazione, come appena descritto, è completata da A.

5.2.1 Struttura Server Signaling PHP

Il Server Signaling PHP si occupa, come già detto, di gestire la Rooms List memorizzata sotto forma di array json in un file. Cioè, a seconda del valore assegnato al parametro GET *cmd* esegue un'operazione per manipolare la Rooms List, o restituire dati che la riguardano: ad esempio, inserire una nuova room, restituire la Rooms List[14], eliminare una room, ecc ...
È facile intuire che il server è formato sostanzialmente da uno *switch* sulla variabile *cmd* e che in base al valore di quest'ultima esegue una certa operazione.

```
// recupero comando
if( isset($_GET["cmd"]) ) $cmd = $_GET["cmd"];
else $cmd = "";

switch( $cmd ){

    // il peerA invia l' SDP Offer
    case "create_room":
        [ ... ]
        break;

    // cancella la room se levata dal peerA o se è avvenuta un collegamento con il peerB
    case "delete_room":
        [ ... ]
        break;

    // un peer richiede la lista delle SDP Offers disponibili
    case "get_rooms_list":
        [ ... ]
        break;
```

[14] Dalla lista restituita dalla Rooms List, il server toglie le password in quanto è un informazione riservata e non deve essere utilizzata nella costruzione di una rappresentazione grafica (con oggetti HTML) della Rooms List.

```
    // il peerB invia l' SDP Answer ricavata dalla SDP Offer del peerA
    case "join_room":
        [ ... ]
        break;

    // il peerA controlla se vi è stata associata una SDP Answer alla SDP Offer inviata
    // se è così, il server gliela invia e il peerA può effettuare la connessione Peer To Peer
    case "check_answer":
        [ ... ]
        break;

    // in caso il comando non viene trovato restituisce errore
    default:
        echo json_encode ( '{ "error" : "command not found" }' );

}
```

Figura 26 - Switch del server che per ogni comando esegue le corrispettive operazioni.

5.2.2 Comandi Server Signaling PHP

Si presenta di seguito la lista dei comandi accettati dal Server Signaling PHP spiegando, per ognuno di essi come viene utilizzato dall'applicativo.

- Il **create_room** serve ad aggiungere una nuova room alla Rooms List, dove salvare la SDP Offer del peer, in attesa che vi venga associata un SDP Answer. Oltre al parametro GET cmd, si aspetta l'inserimento della SDP Offer ed in modo opzionale della password e di un commento da associare alla room. La SDP Offer è passata come parametro POST per la sua (in genere) grande dimensione.

- Il **delete_room** serve a ripulire la Rooms List da una room attraverso la quale è stata già instaurata una connessione peer-to-peer. Infatti, a questo punto è sostanzialmente una "room morta", che, se non venisse cancellata, verrebbe mostrata nella pagina principale, fornendo una falsa informazione. In altre parole, verrebbe mostrata una room che in realtà non è più

disponibile. Tale comando necessita in input anche i valori dell'id e della password della room che si vuole eliminare. L'id serve per identificarla in modo univoco, mentre la password serve per essere certi che la richiesta proviene da un host autorizzato. Si può osservare che, nel caso in cui non sia stata definita una password, tale parametro corrisponde alla stringa vuota "".

- Il **get_rooms_list** serve ad ottenere la Rooms List (in notazione json), di modo che un peer possa vedere le SDP Offer disponibili e quindi individuare il peer a cui vuole connettersi. Chiaramente le password delle room non vengono restituite. Non necessita di ulteriori parametri.

- Il **join_room** serve ad un peer che, una volta individuata una room con cui vuole connettervi dalla Rooms List, vi associa, con questo comando, la SDP Answer generata dalla SDP Offer memorizzata in tale room. Oltre alla SDP Answer servono come parametri l'id e la password associati alla room per essere certi, rispettivamente, di modificare la room giusta e consentire l'accesso ad un utente autorizzato. Anche in questo caso, naturalmente, la mancanza di una password corrisponde ad averla impostata come stringa vuota "".

- Il **check_answer** serve come supporto per l'*attesa attiva*[15] dell'offer_room. Un peer, una volta creata una room, ricontrolla ogni cinque secondi se è presente una SDP Answer. Tale controllo avviene grazie a questo comando (check_answer) e restituisce la SDP Answer in caso positivo con

```
echo json_encode ( '{ "room_answer":'.json_encode($room["sdp_answer"]).' }' );
```

Figura 27 - Il server restituisce la SDP Answer della room.

[15] Ovvero, in cui è il componente in attesa che periodicamente controlla l'arrivo dei dati di cui necessita.

altrimenti NULL (come stringa)[16] con

```
echo json_encode ( '{ "room_answer" : "NULL" }' );
```

Figura 28 - Nel caso la room non contiene ancora nessuna Answer restituisce la stringa NULL.

Anche in questo caso servono id e password per gli stessi motivi descritti in alcuni comandi sopra riportanti (per esempio in join_room).

5.2.3 Server php in locale

L'applicativo presentato come esempio è funzionante senza la necessità di ulteriori configurazioni. Basta aprire con Google Chrome[17] il file "index.html" della cartella "local_peer_RTC" ed è pronto all'uso. Tuttavia, si vuole descrivere brevemente un possibile modo per utilizzare il server in locale. Nella fattispecie si è deciso di utilizzare XAMPP (Cross-Platform Apache MySQL Php Perl) scaricabile all'indirizzo

https://www.apachefriends.org/it/index.html .

È scaricabile una versione per tutte le principali piattaforme (Linux, Windows, OS X). Una volta installato, bisogna inserire il server "signaling_server" nella cartella "htdocs" contenuta nella cartella di installazione di XAMPP. Per "attivare" il server si avvia il service Apache dallo *XAMPP Control Panel [1]*.

Pertanto, l'indirizzo locale del server è

http://127.0.0.1/signaling_server/signaling_server.php .

---

[16] Si è scelto di identificare la mancanza di una SDP Answer con *la stringa* "NULL", tuttavia si poteva decidere una qualsiasi altra stringa, come per esempio "none" o l'effettivo valore null. L'importante è che il client (cioè. l'offer_room) sia in grado di "capire" tale informazione.

[17] Il funzionamento è garantito da desktop su Google Chrome perché è stato pensato e progettato per essere avviato con tale browser; *tuttavia* teoricamente, come descritto in questo documento dovrebbe essere compatibile anche con Firefox, Edge, Opera, Safari e "Mobile".

Si può, ora, richiamare una funzione piuttosto che un'altra inserendo i parametri GET nell'URL come strutturato di seguito:

http://127.0.0.1/signaling_server/signaling_server.php
?cmd=<COMANDO>&<nome_parametro1>=<VALORE1>
&<nome_parametro2>=<VALORE2>& ...

Come già detto, l'applicativo non ha bisogno di avere installato in locale il signaling_server perché fa riferimento al server

https://serversignaling.altervista.org/signaling_server.php

caricato su Altervista. Pertanto, per usare il server locale, è necessario modificare il valore della costante "signaling_server" contenuta nel file "/local_peer_RTC/js/js_consts.js". L'applicativo è strutturato in modo tale che questa costante sia visibile a tutti gli script js e, quindi, tutte le richieste al server siano indirizzate all'indirizzo indicato da essa.

Dalla sperimentazione della tecnologia webRTC, l'avere installato in locale il server è risultato utile in fase di sviluppo e di modifica del server stesso, rendendo queste operazioni più semplici e veloci. In caso di modifica del server, è possibile riportare velocemente le modifiche su Altervista tramite il protocollo FTP.

5.3 Rooms List

Con Rooms List si intende l'insieme di room disponibili. L'utente può consultare tale lista nella pagina principale (index) dell'applicativo. Ogni tre secondi, l'applicativo invia una richiesta al server php richiedendo tale lista. Quindi verifica se è uguale a quella corrente e se è così non fa nulla, altrimenti la aggiorna.

```
// Ogni tot di sec, se la rooms list è cambiata, aggiorna le rooms html
( async function () {

    while( true ){

        // aspetta un tot di sec
        await new Promise(resolve => setTimeout(resolve, 3000));

        // parametri passati
        // is_create_room = false & is_check_rooms = true
        get_rooms_list_from_server( false, true );
    }

}) ();
```

Figura 29 - Controllo di eventuali cambiamenti nella Rooms List

In particolare si usa

```
await new Promise(resolve => setTimeout(resolve, 3000));
```

Figura 30 - Aspetta tre secondi prima di ricontrollare la Rooms List.

per aspettare 3 secondi da una richiesta ad un'altra, e

```
get_rooms_list_from_server( false, true );
```

Figura 31 - Richiesta Rooms List al Signaling Server.

per effettuare la richiesta al server_php.

I parametri false e true passati si riferiscono rispettivamente ai flag *is_create_room* e *is_check_room*. Impostando il flag *is_create_room* a *true*, dopo aver ottenuto la Rooms List dal server sotto forma di array json, si crea un insieme di oggetti html, ognuno dei quali rappresenta una room contenuta

nella lista; questo insieme di oggetti viene inserito in un *container*[18] contenuto nella pagina. Tale flag viene impostato a *true* quando si accede alla pagina per la prima volta oppure si aggiorna la lista[19].
Il flag *is_check_rooms*, invece viene impostato a *true* quando si vuole controllare che la Rooms List corrente sia uguale a quella ottenuta dal server. In questo caso, gli elementi html che rappresentano graficamente le room vengono aggiornati solo se la nuova lista è diversa da quella corrente, salvata nella variabile *current_json_rooms_list*.

5.4 Offer e Answer Room

L'offer_room.js e l'answer_room.js rappresentano il cuore dell'applicativo e della sperimentazione della tecnologia webRTC. Infatti contengono lo script responsabile della creazione e della gestione della connessione RTC tra due peer. Di seguito descriviamo il loro funzionamento.

1. Entrambi gli script, come prima azione, si preparano a creare la connessione, recuperando le informazioni necessarie dai parametri GET passati tramite l'url. Infatti, quando vengono cliccati i bottoni "create room" e "join" vengono richiamate,rispettivamente le pagine offer_room e answer_room. La prima richiede i parametri *room_description* (il commento) e *room_psw;* la seconda *room_id, room_psw e room_index* (indice della room nella Rooms List).

2. Entrambi gli script recuperano lo stream dei dispositivi attraverso il metogo *getUserMedia()* dell'oggetto *navigator.mediDevices*. Questo implica che viene richiesto il permesso per accedere ai dispositivi e se viene negato (dall'utente) allora il browser viene reindirizzato sull'index.

[18] Si tratta di un div che viene recuperato nello script javascript e vi vengono inseriti gli oggetti html che rappresentano le room della Rooms List.

[19] Per aggiornare graficamente la Rooms Lista, prima si eliminano tutti gli oggetti html dal container delle room, quindi si può creare la nuova lista di room. In questo modo nella pagina non saranno più presenti le room della lista precedente.

3. La offer_room genera la SDP Offer da un oggetto RTCConnection; quindi, la invia al signaling server e si mette in attesa attiva che vi venga associata una SDP Answer.

4. La answer_room invia la SDP Answer al server. Riesce ad individuare l'esatta room con cui vuole mettersi in comunicazione grazie ai parametri GET passati (room_id,room_psw, room_index), che la identificano in modo univoco.

5. La offer_room si accorge della SDP Answer e con essa instaura la connessione.

Una volta instaurata la connessione sarà visibile sull'oggetto video di sinistra lo stream di video e audio del peer che ha generato l'SDP Offer e sull'oggetto video di destra quello del peer che ha generato l'SDP Answer. Sono anche disponibili i pulsanti per inviare un messaggio all'altro peer, mutare video ed audio e uscire dalla videochiamata.

5.5 Scambio di Messaggi in una Videochiamata

Una volta avviata una videochiamata è possibile inviare dei messaggi all'altro peer. Lo scambio di messaggi avviene tramite DataChannel. Quindi, nella fattispecie la offer_room crea una connessione lc (istanza di RTCConnection), da cui crea un oggetto dc DataChannel con il comando

```
const dc = lc.createDataChannel("channel");
```

Figura 32 - Creazione DataChannel dalla connessione RTCConnection lc.

D'altra parte, la answer_room non deve creare un nuovo DataChannel, ma deve recuperare quello associato alla connessione. Quindi, dopo aver creato un

oggetto rc RTCConnection, si mette in ascolto per l'evento "datachannel" che cattura un evento *e* di tipo RTCDataChannelEvent. Si può recuperare il datachannel con il comando

```
var dc = e.channel;
```

Figura 33 - Recupero DataChannel dalla connessione.

Una volta che è stato creato il DataChannel dc e che entrambi i peer ne hanno accesso, per inviare un messaggio basta usare il comando

```
dc.send("Messaggio Da Inviare");
```

Figura 34 - Invio messaggio tramite DataChannel.

mentre l'arrivo di un messaggio è gestito mettendosi in ascolto dell'evento "message" che cattura un evento *e* di tipo MessaggeEvent. Da *e* possiamo ricavare il testo del messaggio con *e.data* .

A questo punto rimane da mostrare come permettere questo scambio tramite elementi grafici, cioè tramite gli oggetti HTML della pagina. Consideriamo, quindi la offer_room (vale lo stesso per la answer room) e inseriamo nella pagina html due *textbox* ed un *button: sa*ranno rispettivamente il contenitore dei messaggi, il campo dove scrivere il messaggio da inviare ed il bottone per inviarlo. Associamo tali elementi a delle variabili javascript in modo da poterli manipolare all'interno dello script, assegnando loro un id e recuperandoli tramite esso come fatto nel seguente esempio:

```
var messages_container = document.getElementById("messages_container");
var write_msg_area = document.getElementById("write_msg_area");
var bt_send_msg = document.getElementById("bt_send_msg");
```

Figura 35 - Associazione di elementi HTML a delle variabili javascript.

Per inviare un messaggio si inserisce un listener sull'evento "click" sul bottone *bt_send_msg*. Quando viene catturato tale evento, si invia tramite *dc.send()* il

testo contenuto in *write_msg_area*, cioè *write_msg_area.value*. *Si a*ggiunge tale stringa anche nel *messages_container* con la seguente istruzione:

```
messages_container.value += "YOU -> " + write_msg_area.value + "\n";
```

Figura 36 - Concatenazione del messaggio inviato al contenuto della textarea messages_container.

ed infine si ripulisce l'area di testo per prepararlo alla scrittura del prossimo messaggio con l'istruzione

```
write_msg_area.value = "";
```

Figura 37 - Prepara alla scrittura del prossimo messaggio svuotando la textarea dedicata.

Per gestire la ricezione di un messaggio, nel listener dell'evento "message" sul DataChannel dc, si recupera il testo ricevuto tramite *e.data (*dove *e* è il MessageEvent catturato) e lo si concatena al contenuto di *write_msg_area*.

```
dc.onmessage = e => {
    messages_container.value += "REMOTE PEER -> " + e.data + "\n" ;

    // Eventualmente scolla in basso la textarea
    messages_container.scrollTop = messages_container.scrollHeight;
}
```

Figura 38 - Concatenazione del messaggio arrivato alla textarea contenente tutti i messaggi.

6. CONCLUSIONI

In generale, lo sviluppo di questa tecnologia ha portato semplificazione ed efficienza nell'ambito della programmazione web. Infatti, prima standard *de facto* e recentemente (26 Gennaio 2021) divenuto ufficialmente standard [29], ha messo in mano ai programmatori una serie di API che permettono di

implementare con poche istruzioni e senza componenti aggiuntivi (*plugin-free*) tutta una serie di complesse funzionalità, come, per esempio, instaurare una connessione peer-to-peer efficiente o inviare/ricevere stream video ed audio.
Dal fatto che necessita di una fase di Signaling la cui implementazione è lasciata completamente al programmatore, si può dedurre che questa tecnologia non è nata per fornire un metodo in grado di mettere in connessione due host, ma per farlo nel modo più efficiente possibile. Infatti nella fase di Signaling i due host sono, di fatto, già in comunicazione tra di loro. Ma, quando i due peer ottengono le informazioni necessarie, si abbandona tale connessione (in genere con architettura client/server) per aprirne una nuova con la tecnologia webRTC. Quest'ultima è sicuramente molto più efficiente di quella usata nella fase di Signaling, perché è stata generata partendo da tutte le possibili configurazioni e scegliendo la migliore tra di esse; inoltre, essendo una connessione peer-to-peer, i messaggi non devono passare per server che fanno da intermediari, rallentando lo scambio dei dati.
Gli svantaggi di questa tecnologia si sono individuati nell'uso di server TURN e nella connessione di un numero di peer maggiore di due. Riguardo al primo punto, ci sono situazioni in cui non si può creare una connessione peer-to-peer e, pertanto, essa deve essere instaurata tramite il server TURN, che inoltra i messaggi da un peer all'altro. Nella maggior parte dei casi si può usare un server STUN con una connessione peer-to-peer, ma quando non è possibile, utilizzando un server TURN si perde l'efficienza della connessione webRTC; inoltre un server TURN è costoso da mantenere.

Nel caso in cui si voglia collegare più di due peer, è necessario stabilire, per ciascuno di essi, una connessione con tutti gli altri singolarmente. Cioè ogni peer è collegato direttamente ad ogni altro peer. Quindi, si può notare come si arriva facilmente ad un numero spropositato di connessioni peer-to-peer, tanto che sarebbe preferibile una connessione client/server in cui tutti i client sono collegati ad un server centrale [12].

Concludendo, l’avvento di questa tecnologia ha portato semplificazione ed efficienza nello sviluppo di funzionalità sempre più richieste, ma allo stesso tempo ci sono situazioni in cui l’uso della tecnologia webRTC è semplicemente poco adatta.

Bibliografa

[1] apachefriends.org - Introduction to XAMPP - https://www.youtube-nocookie.com/embed/h6DEDm7C37A

[2] Dutton Sam - After signaling: Use ICE to cope with NATs and firewalls - https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/#after-signaling-using-ice-to-cope-with-nats-and-firewalls

[3] Dutton Sam - Build the backend services needed for a WebRTC app STUN, TURN, and signaling - https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/

[4] Dutton Sam - Get started with webRTC - https://www.html5rocks.com/en/tutorials/webrtc/basics/

[5] Frasca Pietro - Sistemi Operativi, seconda parte: Reti di Calcolatori - Lezione 13 (37) - http://www.informatica.uniroma2.it/upload/2020/SOR/RETI%20LEZ13.pdf

[6] Frasca Pietro - Sistemi Operativi, seconda parte: Reti di Calcolatori - Lezione 14 (38) - http://www.informatica.uniroma2.it/upload/2020/SOR/RETI%20LEZ14.pdf

[7] Frasca Pietro - Sistemi Operativi, seconda parte: Reti di Calcolatori - Lezione 17 (41) - http://www.informatica.uniroma2.it/upload/2020/SOR/RETI%20LEZ17.pdf

[8] Google - codice sorgente webRTC -licenza - https://webrtc.googlesource.com/src/+/refs/heads/main/LICENSE

[9] Google Developers - Real-time communication with WebRTC: Google I/O 2013 - https://www.youtube.com/watch?v=p2HzZkd2A40&t=670s

[10] Harald (speaking for Google) - Google release of WebRTC source code; https://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html

[11] IETF - Rtcweb Status Pages - https://tools.ietf.org/wg/rtcweb/

[12] Nasser Hussein - WebRTC Crash Course - https://www.youtube.com/watch?v=FExZvpVvYxA&t=404s

[13] Trilogy-LTE - How WebRTC Is Revolutionizing Telephony - Communication is Moving To The Web - https://blogs.trilogy-lte.com/post/77427158750/how-webrtc-is-revolutionizing-telephony

[14] WebRTC Ventures - webrtc signaling stun vs turn - https://webrtc.ventures/2020/12/webrtc-signaling-stun-vs-turn/

[15] Mozilla developers - Capabilities, constraints, and settings - https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API/Constraints

[16] Mozzilla developer - Codec - https://developer.mozilla.org/en-US/docs/Glossary/Codec

[17] Mozzilla developer - ICE - STUN - NAT - TURN - SDP - https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols

[18] Mozzilla developer - ICE - https://developer.mozilla.org/en-US/docs/Glossary/ICE

[19] Mozilla developer - MediaStream - https://developer.mozilla.org/en-US/docs/Web/API/MediaStream?retiredLocale=it

[20] Mozzilla developer - SDP - https://developer.mozilla.org/en-US/docs/Glossary/SDP

[21] Mozzilla developer - Signaling - https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity

[22] Open Source Initiative - BSD 3 Clause - https://opensource.org/licenses/BSD-3-Clause

[23] WebRTC - licenza webRTC - https://webrtc.org/support/license

[24] WebRTC - connessione tra pari - https://webrtc.org/getting-started/peer-connections

[25] WebRTC - Getting started with media devices - https://webrtc.org/getting-started/media-devices?hl=en

[26] WebRTC - licenza webRTC - https://webrtc.org/support/license

[27] Wikipedia - BSD Licenses - https://en.wikipedia.org/wiki/BSD_licenses

[28] Wikipedia - WebRTC - https://en.wikipedia.org/wiki/WebRTC#cite_note-5

[29] W3C- Web Real-Time Communications (WebRTC) transforms the communications landscape; becomes a World Wide Web Consortium (W3C) Recommendation and multiple Internet Engineering Task Force (IETF) standards - https://www.w3.org/2021/01/pressrelease-webrtc-rec.html.en