Alma Mater Studiorum · Università di Bologna

Campus di Cesena
Scuola di Scienze
CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

Progettazione, implementazione e valutazione di un simulatore basato su Web Real Time Communication (WebRTC).

Tesi di Laurea in Reti di Calcolatori

Relatore:
Gabriele D'Angelo

Presentata da:
Matteo Ciuffoli

Sessione I
Anno Accademico 2013/2014

A computer simulation is a computation that models the behavior of some real or imagined system over time

R.M. Fujimoto

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il *Web Computing* può essere definito come un particolare tipo di calcolo distribuito che richiede la collaborazione di più applicazioni remote.
Contrariamente ai sistemi distribuiti classici, con una configurazione hardware fissa all'interno di una rete locale, questo nuovo approccio fornisce delle funzionalità dinamiche che possono essere facilmente migrate ed eseguite su qualsiasi macchina connessa al Web. Questo nuovo stile di calcolo richiede connessioni a banda larga, un ambiente uniforme, e software flessibili.
Mai come negli ultimi tempi gli applicativi Web sono di così larga diffusione, portando sempre più il computing all'interno del browser. Già da tempo aziende produttrici di browser mettono a disposizione un vero mercato di applicazioni realizzate in HTML5, CSS e Javascript con prestazioni simili a quelle scritte in linguaggi nativi. Questo ha spinto a realizzare API e Framework con lo scopo di facilitare sempre più la sviluppo di applicazioni per browser, cercando, in alcuni casi, anche di sostituire applicazioni native.
Nel campo della comunicazione multimediale, i video sono disponibili nel World Wide Web (WWW) da prima del 1990, evolvendosi e adattandosi alle nuove tecnologie di collegamento (ad esempio DSL, 3G o EDGE). Per mezzo di prezzi sempre più accessibili, le macchine fotografiche digitali sono diventate parti integranti dei PC dei nostri giorni. Questi fattori, in concomitanza con la richiesta di applicazioni sempre più integrate al Web, sono alcune delle

motivazioni a monte di *Web Real Time Comunication (WebRTC).*
WebRTC è una tecnologia open source, nata principalmente per lo *stream* di dati multimediali, che permette videochat all'interno del browser. Ad oggi questa tecnologia si è espansa e migliorata, facilitando anche lo scambio di qualsiasi tipo di dato, permettendo così di realizzare applicativi distribuiti più complessi.
Lo scopo di questa tesi è di realizzare e valutare un simulatore *Web-Based* i cui nodi sono connessi tramite WebRTC e testare la sua efficienza mediante la simulazione di un semplice modello di mobilità.
La scelta rappresenta il caso pessimo di questo tipo di algoritmi, in quanto ogni singola entità simulata si sposta in maniera casuale ed indipendente dalle altre.
Nei capitoli a seguire si guiderà il lettore attraverso i principali concetti di simulazione e di WebRTC, fornendo le basi, laddove necessario, per una maggior comprensione del testo e delle scelte progettuali ed implementative.

Capitolo 2

La Simulazione

Le motivazioni per simulare un sistema, *modello di simulazione*, sono delle più svariate: esigenza di testare un sistema pericoloso, impossibilità di costruire il sistema stesso, lo studio di nuove soluzioni .
Modelli complessi possono richiedere anche settimane di calcolo per il numero elevato di elementi che interagiscono [1]. Un metodo per ridurre i tempi di esecuzione *(Wall-Clock Time, WTC)*[1] è utilizzare simulatori su architetture parallele e distribuite, suddividendo il carico di lavoro fra le varie unità di elaborazione.
L'utilizzo di tali architetture introduce una serie di problemi noti, tra cui: la sincronizzazione, il costo di comunicazione fra le unità di calcolo, la concorrenza e il rispetto del rapporto causa-effetto secondo un ordine cronologico, non definito a priori, di istruzione *(vincolo di casualità)*.
Nel corso degli anni sono stati introdotti diversi paradigmi, ognuno di questi presenta vantaggi e svantaggi. Non essendoci, in assoluto, un metodo migliore di altri, si valuta caso per caso quale strada percorrere.

[1] È la quantità di tempo necessaria per completare la simulazione, è usata come metrica per misurare l'efficienza di una simulatore. Non è adattabile in tutti gli scenari, esempio nel *Public Cluod.*

2.1 Simulazione monolitica

È l'approccio classico, il più semplice, alla simulazione: una singola unità di elaborazione si occupa dell'evoluzione dello stato del modello simulato. La simulazione monolitica permette di avere uno stato, tramite un insieme di variabili, e un orologio globale. Questi facilitano sensibilmente l'implementazione del modello rispetto ai simulatori su architettura distribuita, a discapito di un carico di lavoro concentrato su di un'unica unità di elaborazione *(Physical Execution Unit, PEU)* dove risulterà un limite, insieme alla complessità del modello, al tempo di simulazione.
In questo approccio non si presenta nessuna delle problematiche relative ai sistemi distribuiti, i cambiamenti di stato del sistema, chiamati evento, verranno elaborati secondo l'orologio globale. Questo paradigma viene chiamato *DES (Discrete Event Simulation).*

2.2 Simulazione parallela e distribuita

Contrariamente al DES, nella simulazione parallela e distribuita *(Parallel And Distributed Simulation, PADS)* l'esecuzione è affidata ad una rete interconnessa di PEU dove vi saranno in esecuzione più processi, chiamati *(Logical process)*, concorrenti in esecuzione. Questi PEU possono trovarsi in esecuzione sullo stesso calcolatore (nel caso di un'architettura multi-core oppure multi·CPU) , condividendo risorse , oppure su calcolatori distinti, connessi da una rete a bassa latenza. In ambienti reali la simulazione parallela e distribuita è un'unica realtà, composta da svariati calcolatori single core oppure multi-core e/o multi-CPU, connessi in rete, *(Figura 1.1)* .
In PADS il modello di simulazione viene suddiviso in componenti [2], LP, dove, in alcuni casi il partizionamento del modello è guidato dalla stessa semantica del sistema. In altri casi, dove il partizionamento risulta essere più difficoltoso, sarà fondamentale valutare alcuni aspetti, data la sensibilità della performance dipendente da essi: il bilanciamento del carico e la sincronizzazione delle comunicazioni sulla rete. In questo approccio è ovvio

che lo stato non è più globale, centralizzato, ma composto da un insieme di stati locali di ogni LP. In PADS ogni LP è in esecuzione su di PEU, e si occuperà solo di una data parte del sistema, interagendo, sincronizzandosi e scambiando informazioni con altre parti di esso in esecuzione su altri LP. I meccanismi d'interazione fra LP sono uno degli aspetti fondamentali della simulazione.

2.3 La sincronizzazione

La sincronizzazione nel PADS è basilare per ottenere un risultato corretto, e lo è solo se è identico a quello ottenuto dalla corrispondente simulazione sequenziale.
Esistono svariati metodi per sincronizzare gli LP ma, anche se ogni approccio presenta vantaggi e svantaggi, in qualsiasi caso si ha un notevole costo aggiuntivo alla simulazione.
Nel PADS è richiesto che ogni evento generato sia contrassegnato, *time-stamped*, e consegnato secondo un approccio a messaggi *(message·passing)*, rispettando quello che è chiamato vincolo di causalità tra eventi. Risulta determinante stabilire fra due eventi, della stessa catena causale, chi precede ed ha causato il secondo, per poterli processare in tale ordine. Questo significa che ogni LP deve processare esclusivamente gli eventi a lui destinati, nell'ordine temporale dato dal timestamp a loro associato *(Vincolo di casualità)*. Risulta ovvio che in una simulazione sequenziale sarà facile rispettare tale vincolo, processando gli eventi in ordine crescente di timestamp. Contrariamente, in un'architettura parallela o distribuita, questo risulta più difficoltoso per svariati motivi: differenti velocità di elaborazione e ritardi nella consegna dei messaggi a causa delle tecnologie di rete. Questi rendono indispensabile l'introduzione di algoritmi di sincronizzazione. A seguire si espongono i tre approcci più utilizzati tra quelli proposti negli ultimi anni.

2.3.1 Time-stepped

Consiste nel suddividere il tempo di simulazione in slot *(Fixed·size time-steps)*, dove ogni LP può procedere al timestep successivo solo se anche tutti gli altri LP hanno processato il timestep corrente.
Questa tecnica garantisce il rispetto del vincolo di causalità ed è facilmente implementabile, a discapito di un paradigma innaturale per alcuni modelli. La scelta della dimensione del timestep inficia sulla performance della simulazione: una timestep troppo piccolo equivale a più sincronizzazioni.

2.3.2 Sincronizzazione Conservativa

L'obiettivo di questo approccio, come nel Time·stepped, è prevenire gli errori di casualità, decidendo per ogni evento, prima di processarlo, se è ritenuto sicuro oppure no.
Nel caso dell'approvazione di un ipotetico evento con timestep t, in futuro non ci saranno più eventi con timestep inferiori a t. Tra gli algoritmi conservativi più conosciuti si ricorda il *Chandy·Misra·Bryant (CMB)*: gli LP, spesso, rimangono in attesa di ottenere le informazioni per decidere se un dato evento è sicuro o non sicuro. Per evitare che il sistema entri in stallo, *deadlock*[2], CMB , introduce dei messaggi senza significato semantico, detti *messaggi NULL*.

2.3.3 Sincronizzazione Ottimistica

Differentemente dai primi due approcci descritti, nell'ottimistico, ogni LP è libero di violare il vincolo di casualità (es. processando un evento con un timestep inferiore a quelli già processati) e non è presente alcun metodo decisionale che attesti se un dato evento è sicuro o non sicuro.
Il metodo ottimistico si basa sull'intercettare la violazione del vincolo di casualità per poi avviare dei meccanismi di *rollback*. Questi meccanismi, interni

[2]In informatica è una particolare condizione in cui i processi si bloccano a vicenda, nell'attesa che uno di questi processi esegua un'azione o renda disponibile una data risorsa.

all'LP, riportano il sistema in uno stato coerente per poi propagare le informazioni. Gli LP interessati avvieranno a loro volta i medesimi meccanismi.
Tra gli algoritmi di sincronizzazioni ottimistici più importanti, si ricorda *Jefferson's Time Warp* [4], [2].

2.3.4 Sistemi adattivi e bilanciamento

La scelta di quale approccio utilizzare, sequenziale o PADS, ha difficoltà intrinseche e per valutarne la performance è fondamentale utilizzare una metrica (WCT) oltre l'analisi di aspetti come il modello, l'ambiente e lo scenario. Ciò nonostante, la scelta risulta essere comunque molto complicata, in quanto non tutti i fattori sono noti a priori.
Altri fattori che concorrono all'efficienza del simulatore sono il costo della comunicazione ed il bilanciamento del carico. Ognuno degli approcci descritti, usati senza ulteriori altri meccanismi, presenta aspetti penalizzanti, es:

- l'algoritmo CMB, utilizzato per implementare la sincronizzazione conservativa, per evitare lo stallo tra i componenti introduce un messaggio senza significato semantico, NULL [3]. Questo approccio, a causa dell'alto numero di questi messaggi, in un sistema in cui il costo della comunicazione è elevato o la larghezza di banda, bandwidth[3] è limitata, risulterebbe inapplicabile per ovvi motivi;

- l'approccio ottimistico, come descritto, permette di violare il vincolo di casualità, ricorrendo a meccanismi di rollback per il ripristino del sistema ad uno stato coerente. Questo approccio assume che il costo della computazione sia inferiore a quello della comunicazione, di conseguenza i componenti del sistema, spesso, resteranno in attesa di ricevere informazioni dalla rete oppure saranno impiegati per eseguire i meccanismi di rollback. Il metodo ottimistico non è applicabile in sistemi in cui

[3]Misura l'intervallo di frequenze disponibili per la trasmissione di un segnale. È strettamente legato alla velocità di trasmissione [7].

la computazione è costosa, in quanto gran parte della computazione è impiegata per i meccanismi di rollback, es: *Public Cloud*;

Sulle basi di quanto fino ad ora descritto, si potrebbe dedurre che questi approcci abbiano come comuni denominatori la mancanza di adattività, la capacità di riuscire in un bilanciamento giusto, proporzionato alla possibilità delle unità di elaborazione. Un primo passo verso un sistema adattivo è un partizionamento dinamico del modello, suddividendolo in piccole parti chiamate *Simuleted Entities , SEs* [4]. Queste SEs saranno contenute, 20 Byte gestite e suddivise fra gli LP, durante la simulazione, libere di migrare da un LP ad un altro ripartendo, bilanciando equamente il carico di lavoro fra le unità di elaborazione. Le SEs sono piccole parti del modello simulato ed interagendo, comunicando fra loro, daranno vita al comportamento del modello. Infine, per ridurre il costo della comunicazione, si clusterizzano le SEs, con un ratio di comunicazione più alto in un unico LP: cioè, tenendo in considerazione il bilanciamento del sistema, si raggruppano in un LP quelle SEs che comunicano più frequentemente fra loro.

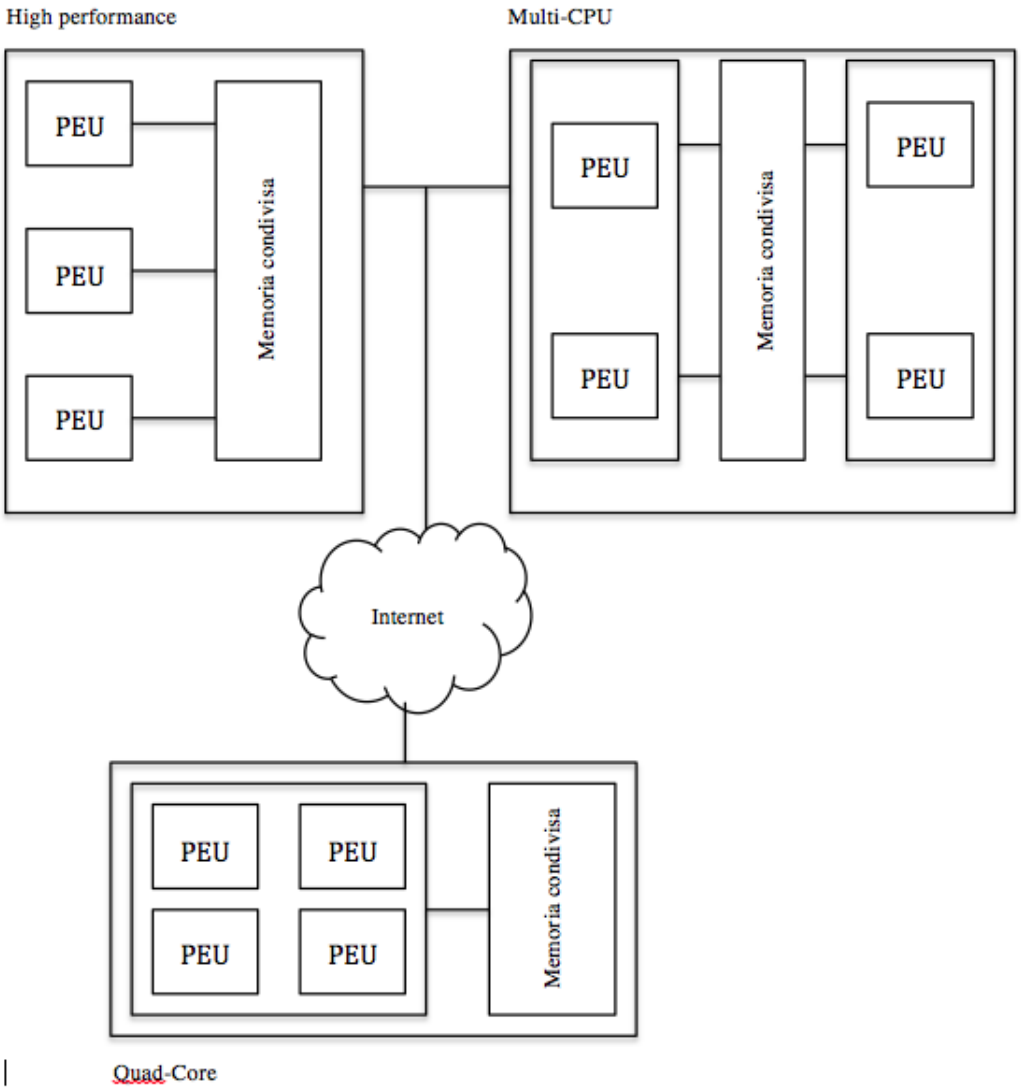

Figura 2.1: Rappresentazione di un sistema reale dove alcuni PEU, su diverse architetture, sono in comunicazione tramite Internet.

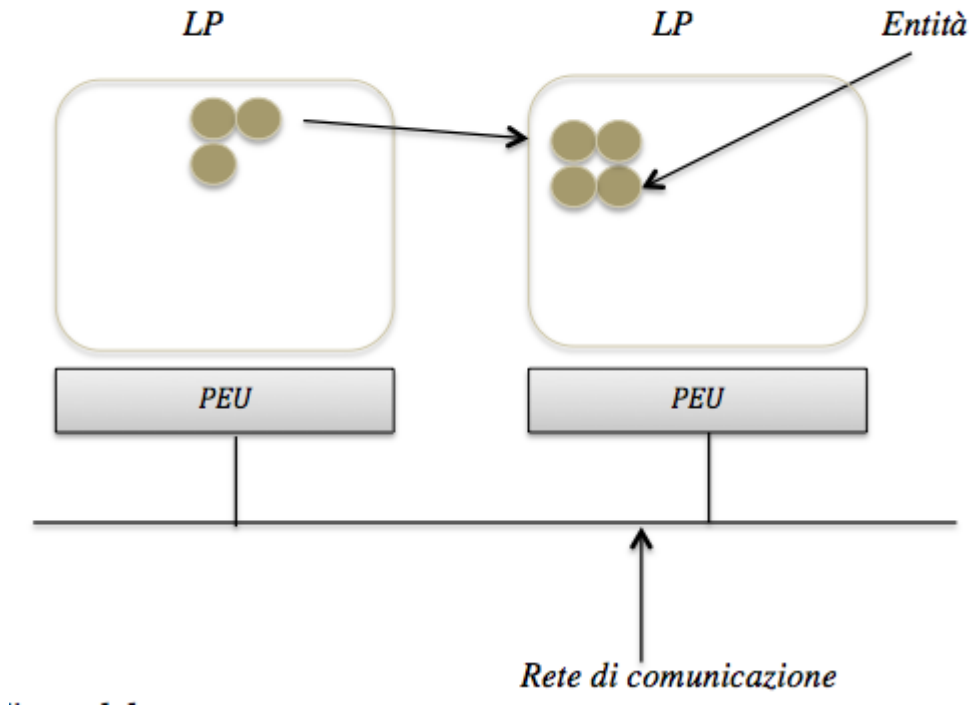

Figura 2.2: Rappresentazione di alcune entità distribuite su due LP.

Capitolo 3

WebRTC

Web Real Time Comunication, WebRTC è una tecnologia open source nata a metà del 2011 per lo scambio di dati multimediali. Oggi WebRTC è considerato uno standard *de facto*, incluso nel World *Wide Web Consortium (W3C)*[1] e supportato da alcuni browser tra cui *Chrome, Firefox e Opera*.
Il Web è un servizio Internet che ha origine durante i primi anni '90 su architettura *client-server*, come rete per la diffusione di documenti in formato elettronico. È un mezzo importantissimo per la comunicazione fra le persone ed esistono innumerevoli applicativi che sfruttano questo servizio. Negli ultimi anni si è cercato d'integrare questi applicativi nel browser, ma solo grazie a WebRTC, basato su architettura *peer-to-peer*, si può parlare di una soluzione standard che permette di realizzare applicativi senza l'ausilio di plug-in, sfruttando la tecnologia che accomuna tutti i browser: *Javascript*.

In questa breve introduzione sono citati architettura client·server e peer·to·peer, servizi ed altri termini correlati ad Internet. Nei concetti espressi da questi, sono contenuti i protocolli e le tecnologie di rete comunemente impiegati per il funzionamento del Web così come da noi percepito.
Come per il capitolo sulla simulazione, si cercherà, con la prossima sezione,

[1] È una associazione non governativa con lo scopo di stabilire degli standard tecnici inerenti ai linguaggi di markup e ai protocolli di comunicazione

d'introdurre le nozioni preliminari delle tecnologie di rete utilizzate, laddove necessarie, per una maggior comprensione del funzionamento di WebRTC.

3.1 I protocolli di rete

Una possibile definizione di Internet è infrastruttura di rete che fornisce servizi ad applicazioni distribuite, concepita su di un'architettura a livelli.
Tale architettura regola e classifica, associando a ogni livello logico i relativi protocolli di rete, in modo tale che ognuno di questi possa appartenere ad un solo livello.
In questa organizzazione ogni livello garantirà al suo diretto superiore un servizio, effettuando azioni al suo interno e sfruttando i servizi offerti dal livello sottostante *(modello a servizi)*. La stratificazione di protocolli *(pila protocollare figura 2.1)* vede alcuni di questi implementati via software, es. *SMTP*, altri solo hardware, alcuni in modo combinato, es. *Ethernet*. La pila dei protocolli di Internet prevede cinque livelli[2] : in un approccio top-down, applicazione, trasporto, rete, collegamento e fisico:

- applicazione *(Application Layer)*: vi si trovano le applicazioni di rete e i protocolli a livello applicazione, ad esempio vi risiedono gli applicativi sviluppati mediante WebRTC.
  I più comuni protocolli sono distribuiti su più sistemi e le informazioni ivi scambiate prendono il nome di messaggi;

- trasporto *(Transport Layer)*: si offre un servizio orientato alla connessione al livello superiore, applicazione, fornendo un canale logico di

[2]Esiste un'altro modello di riferimento: OSI a sette livelli.
Questo differisce per l'aggiunta dei livelli *Prestazione* e *Sessione* fra il livello Applicazione e quello di Trasporto. Nel primo sono presenti i protocolli che consentono alle applicazioni in esecuzione d'interpretare il significato dei dati scambiati , es. compressione e cifratura dei dati (DTLS).
Nel secondo la delimitazione e la sincronizzazione dello scambio di dati, compresi i mezzi per costruire uno schema di controllo e di recupero degli stessi [8].

collegamento fra due istanze dell'applicativo in esecuzione. I due protocolli più comunemente usati sono: *Trasmissin Control Protocol, TCP* [3] ed *User Datagram Protocol, UDP* [4].
I messaggi prendono il nome di segmenti e sono composti di un'intestazione, specifica del protocollo utilizzato, e come carico utile hanno il messaggio stesso proveniente dal livello superiore;

- rete *(Internet Layer)*: i relativi protocolli offrono come servizio un canale di comunicazione logico fra *host*, non direttamente connessi, col compito di instradare e indirizzare i dati verso la giusta destinazione, attraverso il percorso di rete più appropriato.
  Uno dei protocolli più usati è *Internet Protocol , IP*.
  Come per il livello superiore i segmenti sono incapsulati in datagrammi aventi un'intestazione specifica del protocollo di rete e come carico utile hanno il segmento stesso proveniente dal livello superiore;

- collegamento *(Link Layer)*: si occupa di inoltrare i datagrammi IP in una serie di host adiacenti fra l'origine e la destinazione. Fra i protocolli più utilizzati si ricorda Ethernet.
  Il datagramma è incapsulato in frame, come per il livello precedente, è composto da una sua intestazione specifica e come carico utile ha il datagramma;

- fisico *(Physical Layer)*: ha il compito trasferire i singoli bit del frame da un nodo a quello successivo;

[3] Alcune delle caratteristiche peculiari: controllo di flusso, di congestione ed affidabilità di trasporto. Quest'ultimo, come suggerisce lo stesso nome, garantisce, grazie ad una serie di meccanismi, la consegna del segmento al destinatario. TCP è considerato Strem-Oriented. Overhead intestazione 20 Byte.

[4] Diversamente da TCP non c'è alcun tipo di controllo sulla consegna e sull'andamento del canale di comunicazione. UDP è considerato *Message-Oriented*. Overhead intestazione 8 Byte.

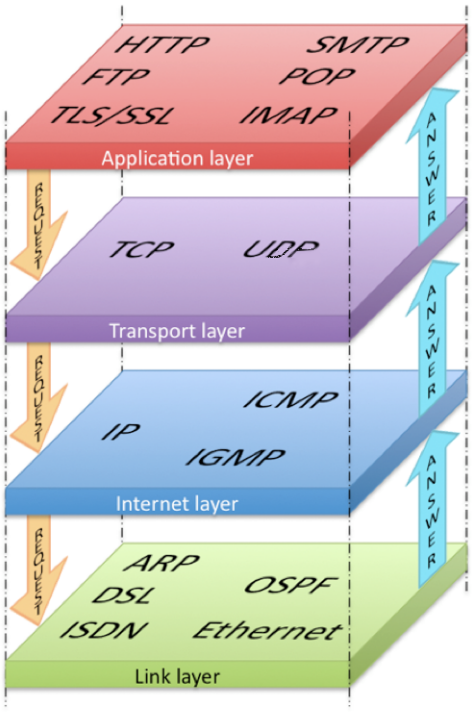

Figura 3.1: Riepilogo della stratificazione dei protocolli di rete con relativi esempi.

Alcuni protocolli utilizzati da WebRTC si posizionano al secondo livello della pila: SCTP, SRTP ed UDP; altri al primo livello: DTLS, STUN, TURN ed ICE.

3.2 Il Funzionamento

La combinazione dei sopra citati protocolli e le Javascript API di WebRTC, permette uno scambio dati peer·to·peer fra sessioni del browser (peer). Le tre principali API a disposizione sono:

- MediaStream: acquisizione e scambio del flusso audio e video con il peer remoto;

- RTCPeerConnection: è alla base della negoziazione della connessione fra la sessione locale e quella remota. Questo scambio di dati avviene tramite il protocollo *Session Descriptio Protocol, SDP* .

- RTCDataChannel: acquisizione e scambio dati generici dell'applicazione con il peer remoto;

Tutto quello di cui necessita WebRTC per la realizzazione di un'applicazione di videoconferenza è all'interno di queste tre API.
In seguito, trattandosi di una tesi incentrata sulla simulazione, non sarà argomentato MediaStream poichè non utile all'implementazione.
Uno degli obiettivi di WedRTC è la performance, riducendo al minimo la latenza della comunicazione fra origine e destinazione.
Questa esigenza risiede nella caratteristica principale che definisce il concetto di videoconferenza, ovvero l'istantaneità. È una delle motivazioni per cui il flusso dati fra i peer è trasportato da segmenti UDP, sostenendo la possibile perdita di dati nel durante, a favore di Real Time Communication.
Il solo protocollo UDP non è sufficiente ad avviare una comunicazione tra i peer, è indispensabile superare alcune avversità tra cui *il Network Address Translation, NAT* [5] [8], per poi negoziare alcuni parametri, es. la crittografia. È fondamentale che UDP sia supportato da una serie di protocolli, *figura 2.1.*

3.2.1 Signaling e Session Description Protocol (SDP)

Alcuni dei protocolli della figura 2.2 sono parte attiva di un processo che prende il nome di *Signaling.*

[5]Modifica gli indirizzi dei datagrammi IP in transito su router getaway, provenienti dalla rete interna verso quella esterna non rendendo indispensabile un IP pubblico per ogni dispositivo connesso alla rete. Per ogni terminale che attua una richiesta verso una rete esterna il getaway la effettuerà la richiesta al suo posto. Risulta ovvio che per un osservatore esterno tutte le richieste appaiono con l'IP del default getaway, impedendo una comunicazione diretta col dispositivo mascherato dal NAT.

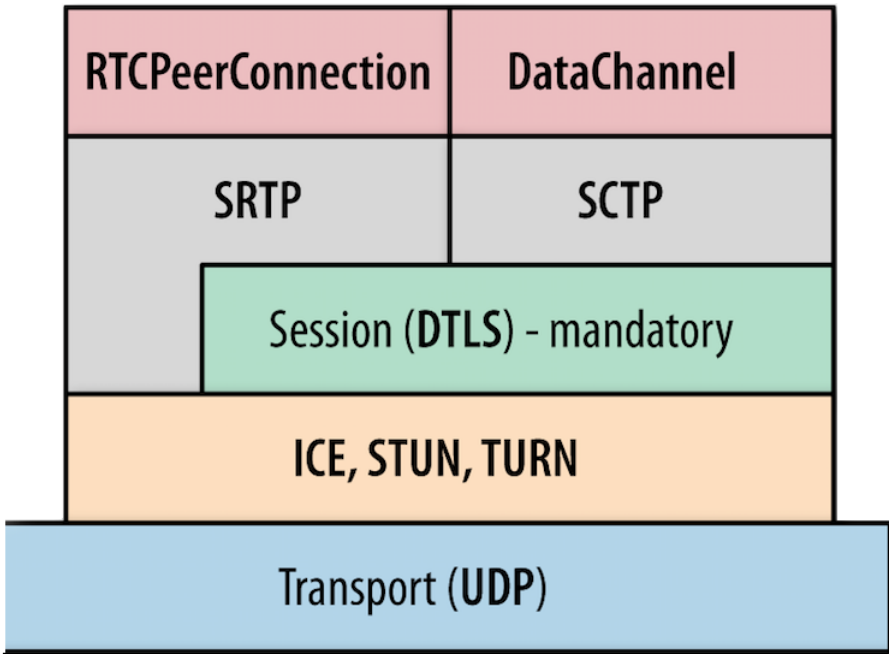

Figura 3.2: Riepilogo dei princiapli dei protocolli utilizzati da WebRTC.

La Signaling è la fase in cui i due peer attuano le operazioni necessarie per stabilire una canale di comunicazione condiviso.
Questi meccanismi non sono regolamentati dalle API di WebRTC, lasciando allo sviluppatore la possibilità di adottare le tecniche più opportune, si suppone per scelta.
È stato utilizzato *WebSocket* in esecuzione su *Node JS* per l'implementazione del canale condiviso. Questa parte è trattata nella sezione successiva, per il momento si invita il lettore ad assumere l'esistenza di tale canale.
WebSocket e Node JS hanno in comune lo stesso linguaggio di programmazione utilizzato per il resto del progetto, Javascript.
Una volta instaurato un canale di comunicazione condiviso, si può procedere con i primi meccanismi per avviare una connessione WebRTC.
Il primo passo consiste nello scambiarsi, attraverso il suddetto canale, i parametri della connessione peer·to·peer che si andrà ad instaurare. Per fare ciò WebRTC utilizza il *Session Description Protocol, SDP*. Questo protocollo offre un formato per descrivere ed elencare i parametri di inizializzazione di

uno streaming media, es: rete, codec utilizzato e sue impostazioni, informazioni sulla larghezza di banda, tipi di dati da scambiare (video, applicazioni e dati), metadati, candidato ICE (trattato nella sezione 2.2.3 STUN,TURN e ICE), etc..
SDP fornisce una descrizione dei dati che sono scambiati ma non il loro trasporto.
Si riporta un esempio generato da un'istanza del simulatore sviluppato. Figura 2.3:

```
answer ——— v=0
o=- 6708113802255427709 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE data
a=msid-semantic: WMS
m=application 1 RTP/SAVPF 101
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:T3Ft58tC24wW+PC2
a=ice-pwd:1bv505M1d8vKF0+KH5DjiIsb
a=fingerprint:sha-256 2A:14:90:B8:6A:0F:BE:45:74:C6:F7:CE:37:6D:E0:9C:82:1F:67:47:B9:B7:B6:F0:28:AD:9D:7D:73:93:C0:75
a=setup:active
a=mid:data
b=AS:1638400
a=sendrecv
a=rtcp-mux
a=rtpmap:101 google-data/90000
a=ssrc:358612100 cname:w65SKAofpUoIXyVR
a=ssrc:358612100 msid:RTCDataChannel RTCDataChannel
a=ssrc:358612100 mslabel:RTCDataChannel
a=ssrc:358612100 label:RTCDataChannel
```

Figura 3.3: L'esempio riportato è specifico di una sessione di Chrome.

I meccanismi utilizzati di scambio dell'SDP sono implementati all'interno dei metodi dell'oggetto RTCPeerConnection.
Una volta definita la sessione locale, tramite il metodo *setLocalDescripton*, su questa si genera un messaggio che prende il nome di *offerta* ed inviato attraverso il canale condiviso al peer remoto, tramite il metodo *createOffer*. L'offerta è trasferibile al destinatario in qualsiasi formato si ritenga opportuno, es. stringa di testo oppure XML.
Nell'implementazione si è utilizzato un oggetto specifico RTCSessionDescription, fornito dalle API di WebRTC [6].
Il peer ricevente, tramite l'oggetto RTCPeerConnection, salva il messaggio ricevuto in una sessione locale specifica, col metodo *setRemoteDescripton*, e risponde al suo interlocutore con un'offerta tramite il metodo *createAnswer*. Il mittente, in modo analogo, salva la risposta del peer remoto nella sua sessione apposita. Figura 2.4.

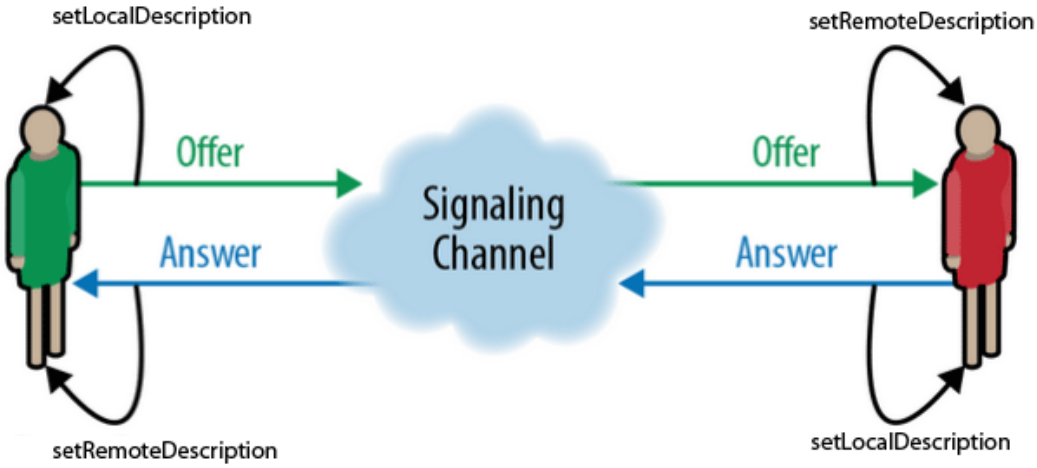

Figura 3.4: Scambio di sessione fra i peer.

Come anticipato lo scambio della sessione è implementato nei metodi dell'oggetto RTCPeerConnection; si riporta parte dell'implementazione del procedimento descritto:

Nell'implementazione esposta si trovano degli accorgimenti relativi al browser, poichè ognuno di questi chiama ed implementa in modo diverso le API di WebRTC.

3.2.2 Node JS e WebSocket

Node JS e WebSocket sono stati impiegati per implementare la parte server del simulatore, permettendo l'avvio della Signaling sopra descritta.
Node JS è un framework open source che permette di sviluppare applicazioni server-side in Javascript, basato su motore V8 realizzato da Google. V8 è incluso in Chrome.
La sua particolarità risiede nella possibilità di accedere alle risorse del sistema operativo ospitante in modalità event·driven[6]. Questo approccio dovrebbe

[6]Programmazione ad eventi, paradigma di programmazione in cui il flusso del programma è determinato dal verificarsi di eventi esterni.

```
var RTCPeerConnection = window. mozRTCPeerConnection || window.webkitRTCPeerConnection;
var RTCSessionDescription = window.mozRTCSessionDescription || window.RTCSessionDescription;

var Offer = {
      createOffer: function(config) {
         var peer = new RTCPeerConnection(iceServers, optionalArgument);
         RTCDataChannel.createDataChannel(peer, config);
         if (isChrome) {
            peer.createOffer(function(sdp) {
               sdp = serializeSdp(sdp, config);
               peer.setLocalDescription(sdp);
            }, null , offerAnswerConstraints);
         } else if (isFirefox) {
            navigator.mozGetUserMedia({
                  audio: true,
      }, function(stream) {
         peer.addStream(stream);
         peer.createOffer(function(sdp) {
                     peer.setLocalDescription(sdp);
                     config.onsdp({
                        sdp: sdp,
                        userid: config.to
                     });
                  }, null, offerAnswerConstraints);
               }, null);
         }
         this.peer = peer;
         return this;
      }, setRemoteDescription: function(sdp) { this.peer.setRemoteDescription(new
RTCSessionDescription(sdp), onSdpSuccess, null) }, null } };
```

Figura 3.5: Implementazione offerta.

massimizzare le prestazioni dell'applicazione.
Si riporta uno stralcio dell'implementazione del WebServer:

A Node JS si sono aggiunti i seguenti moduli per lo sviluppo:

- EXPRESS: Framework aggiuntivo per lo sviluppo di applicazioni Web;
- SOCKET.IO: modulo per lo sviluppo di WebSoket;

Come si legge dal codice, si è implementato un WebServer in ascolto su porta 8888 TCP. Lo stesso server avvia una WebSocket.
La WebSocket fornisce una comunicazione full·duplex attraverso una singola connessione TCP, offrendo ai peer un canale per scambiarsi informazioni per l'avvio di una comunicazione peer·to·peer.
Le sue API sono standardizzate dal W3C.

```
var app = require('express')(),
   server = require('http').createServer(app),
   io = require('socket.io').listen(server);

server.listen(8888);
io.sockets.on('connection', function (socket) {
...
socket.on('message', function (data) {
            socket.broadcast.emit('message', data);
        });
});
app.get('/', function (req, res) {
   res.sendfile(__dirname + '/index.html');
});

app.get('/js/socket.io.js', function (req, res) {
   res.setHeader('Content-Type', 'application/javascript');
   res.sendfile(__dirname + '/js/socket.io.js');
});
app.get('/js/DataChannel.js', function (req, res) {
   res.setHeader('Content-Type', 'application/javascript');
   res.sendfile(__dirname + '/js/DataChannel.js');
});
....
```

Figura 3.6: Implementazione server Node JS.

3.2.3 STUN, TURN e ICE

Analizzata questa prima fase di scambio di sessione, attraverso il canale condiviso, si giunge ad affrontare il problema del NAT.
Alcuni dei protocolli in figura 2.1 sono impiegati per avviare la comunicazione peer·to·peer, oltrepassando le problematiche introdotte dal NAT.

- STUN *(Session Traversal Utilities for NAT)*: è un protocollo di livello applicazione su architettura client·server.
  Questo può fornire all'applicazione in esecuzione informazioni sulla presenza di eventuali NAT che si interpongono fra il dispositivo ospitante dell'applicazione e le reti esterne.
  STUN mette a disposizione le informazioni inerenti a porte e IP che utilizza il NAT per rendere il dispositivo visibile alle reti esterne. Tali informazioni sono richieste attraverso un segmento UDP a uno dei server STUN.

- TURN *(Traversal Using Relays around NAT)* : permette di ricevere dati provenienti da una rete esterna tramite relay di un server TURN. Con questo protocollo due peer possono scambiarsi dati oltrepassando il NAT.

- ICE *(Intereactive Connectivity Establishment)* : insieme di metodi che permettono di stabilire una comunicazione fra i dispositivi utilizzando, laddove possibile, STUN.
  Nel caso in cui questo fallisse, esiste la possibilità di ripiegare su TURN, fig 2.7.

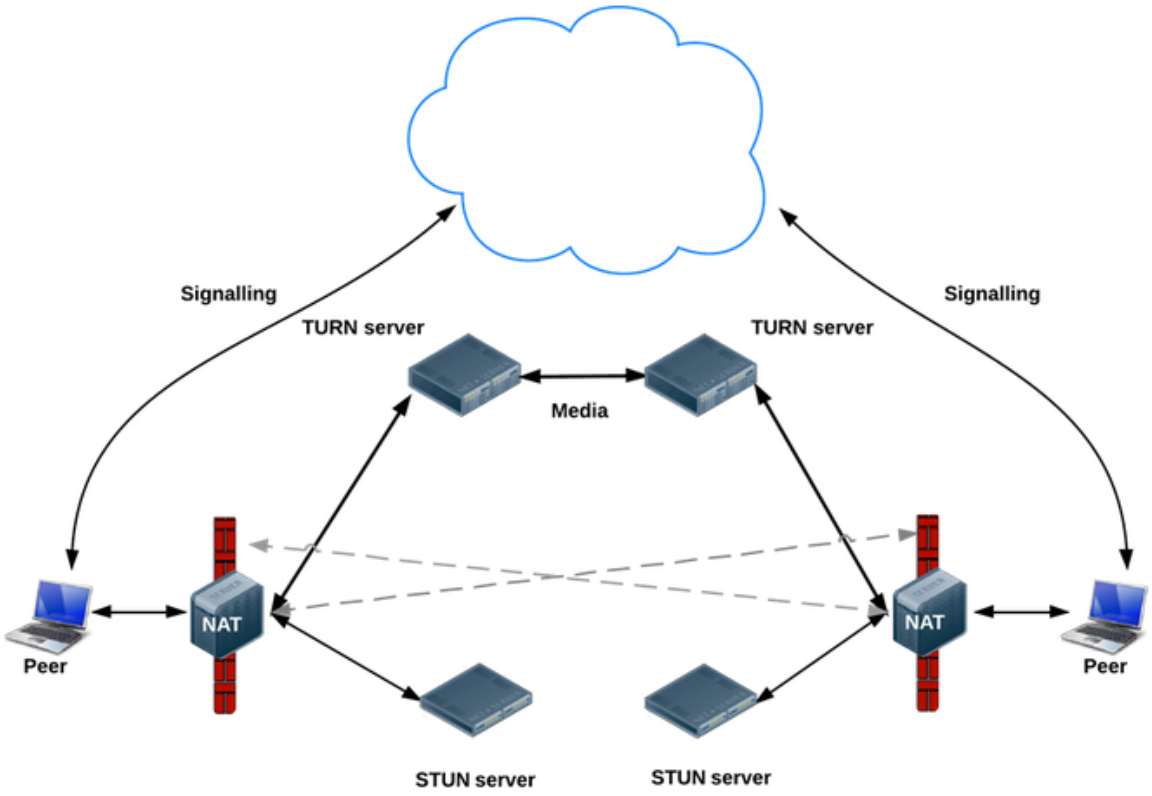

Figura 3.7: Rappresenazione Signaling, TURN e STUN.

I protocolli descritti sono al primo livello della pila protocollare e rappresentano una parte fondamentale per WebRTC in quanto, a differenza dall'iter

previsto da un'architettura client·server dove il secondo è sempre disponibile ad instaurare una nuova connessione con un client, esso basa le sue fondamenta su un'architettura peer·to·peer, dove i peer spesso sono nascosti dal NAT che impedisce di accettare una comunicazione dall'esterno.
I suddetti protocolli non necessitano di un'implementazione da parte dello sviluppatore, ma sono offerti da WebRTC, mettendo a disposizione quanto serve per riuscire ad instaurare un canale peer-to-peer.
Nei metodi dell'oggetto RTCPeerConnection sono implementate le funzioni del framework ICE responsabile per l'avvio e la gestione della connessione, in particolar modo:

- recupero dell'indirizzo IP locale;
- interrogazione del server STUN per il recupero dell'IP esterno e porta su cui essere contattato;
- interrogazione del server TURN nel caso la risposta al server STUN non dia esito positivo, se richiesto;
- notifica della scoperta di un nuovo interlocutore (candidato) tramite una funzione di callback chiamata *onicecandidate*;
- controllo dello stato di connettività dei candidati;

Alcune di queste operazioni sono eseguite in automatico tramite delle funzioni di callback, dopo aver definito la sessione locale: in altre parole, una volta definita la sessione locale, si eseguono le operazioni di ricerca delle informazioni dai server STUN o TURN a seconda delle specifiche. Ottenute tali informazioni (la coppia IP e porta prende il nome di ICE Candidate) si procede col generare l'offerta SDP per poi spedirla tramite il canale condiviso al peer remoto [5].
Una volta che l'ICE Candidate è ricevuto e la sessione remota definita, i peer tenteranno una prima comunicazione. Se questa darà esito positivo, caso in cui si è richiesto l'ausilio di STUN, si è instaurata una comunicazione peer·to·peer. Nella figura 2.8 codice Javascript dei processi sopra descritti.

```
var STUN = { url: isChrome ? 'stun:stun.l.google.com:19302' : 'stun:23.21.150.121' };
var iceServers = { iceServers: [STUN] };
if (isChrome) {  iceServers.iceServers = [STUN]; }
var Offer = {
      createOffer: function(config) {
         var peer = new RTCPeerConnection(iceServers, null);
         RTCDataChannel.createDataChannel(peer, config);
         function sdpCallback() {
            config.onsdp({
               sdp: peer.localDescription,
               userid: config.to
            });
         }
         peer.onicecandidate = function(event) {
            if (!event.candidate) sdpCallback();
         };
         peer.ongatheringchange = function(event) {
            if (event.currentTarget&& event.currentTarget.iceGatheringState === 'complete')
               sdpCallback();
         };
         peer.oniceconnectionstatechange = function() {
             if (!!peer && peer.iceConnectionState == 'disconnected') { peer.close();}
         };
         if (isChrome) {
            peer.createOffer(function(sdp) {
               sdp = serializeSdp(sdp, config);
               peer.setLocalDescription(sdp);
            }, onSdpError, offerAnswerConstraints);
         } else if (isFirefox) {
            navigator.mozGetUserMedia({
                  audio: true
               }, function(stream) {
                  peer.addStream(stream);
                  peer.createOffer(function(sdp) {
                     peer.setLocalDescription(sdp);
                     config.onsdp({
                        sdp: sdp,
                        userid: config.to
                     });
                  }, onSdpError, offerAnswerConstraints);
               }, mediaError);
         }
         this.peer = peer;
         return this;
      },  setRemoteDescription: function(sdp) {
         this.peer.setRemoteDescription(new RTCSessionDescription(sdp), null, null);
      }, addIceCandidate: function(candidate) {
         this.peer.addIceCandidate(new RTCIceCandidate({
            sdpMLineIndex: candidate.sdpMLineIndex,
            candidate: candidate.candidate
         }));
      }
   };
```

Figura 3.8: Il codice in figura riporta l'implementazione blablabal

Grazie all'utilizzo di questi protocolli i due peer ora hanno a disposizione un canale di comunicazione diretto basato su UDP.
Per sua natura UDP non serve nessun controllo di congestione e di flusso del

canale, non è un protocollo sicuro nè tantomeno fair sul canale di comunicazione.
Questi gap di UDP sono colmati tramite altri protocolli, es. DTLS e SCPT, anch'essi a disposizione mediante le API di WebRTC.

3.2.4 Sicurezza e affidabilità

WebRTC, per garantire la sicurezza dei dati trasmessi, si avvale di un altro protocollo a livello applicazione, DTLS.
Questo presenta le principali caratteristiche di TLS [7], aggiungendo i meccanismi necessari per garantire un trasporto affidabile durante la stretta di mano iniziale, *handshake*:

- numero di sequenza: garantisce l'odine di arrivo dei segmenti;
- frammentazione: anche i segmenti più grandi possono essere spediti, associando a ogni frammento un identificativo e dimensione. Garantisce il ri·assemblaggio del segmento;
- timer: Implementa un timer per ogni messaggio dell'handshake. Allo scadere vi è una ritrasmissione;

Terminato l'handshake di DTLS, si procede con lo scambio di dati fra le applicazioni.

Come accennato in precedenza WebRTC è in grado di maneggiare sia dati provenienti da uno streaming audio/video che gestire dati di forma più

[7] Protocollo a livello applicazione derivato da *Secure Socket Layer, SSL*, permette a due applicazioni di comunicare in sicurezza. TLS è composto da:

- *TLS Record Protocol*: usato per incapsulare i dati provenienti dal livello superiore, applicazione, tra cui anche Handshake Protocol.
- *TLS Handshake Protocol*: si occupa dalla fase di negoziazione delle chiavi segrete e condivise con l'interlocutore.

astratta.
Per applicativi di videochat il flusso dati scambiati fra i peer, presenta una gestione e l'impiego di protocolli differenti rispetto a quelli che scambiano dati generici; in particolare, nei primi, s'impiegano principalmente due protocolli a livello di trasporto: *Secure Real-time Transport Protocol, SRTP e Secure Real·Time Control Transport Protocol, SRTCP* .
WebRTC fornisce la possibilità di impostare dei vincoli nell'acquisizione di dati provenienti dalle periferiche audio/video mediante l'oggetto RTCPeer-Connection. Una volta impostati sono un limite superiore della qualità dei dati inviati, es. se chiesta una qualità video di 480p WebRTC, anche se avesse disponile banda a sufficienza per trasferire dati per una qualità maggiore, si limita a rispettare il vincolo.
Diversamente a quanto detto, nel caso in cui la banda a disposizione non sia sufficiente per la qualità richiesta, i due protocolli suddetti attueranno i meccanismi di riduzione della qualità in base alla banda disponibile, salvaguardando l'istantaneità della comunicazione.
Come per altre parti di WebRTC non si approfondirà oltre a quanto citato non essendo utile al progetto.
Altri sono i protocolli e i meccanismi impiegati da WebRTC per lo scambio di dati generici. Come per il simulatore implementato in questa tesi, alcune applicazioni richiedono una garanzia del trasporto dei dati, contrariamente a quanto offre UDP.
WebRTC ricorre all'ausilio di un altro protocollo a livello di trasporto, *Strem Controll Trasmission Protocol, SCTP*, che unisce alcune delle caratteristiche di TCP e altre di UDP.
In dettaglio SCTP offre la possibilità di controllo di flusso e congestione sul canale, una comunicazione affidabile e una trasmissione orientata ai messaggi. I dati in arrivo dal livello superiore possono essere frammentati, *chunk*, e incapsulati in segmenti SCTP per poi essere ri·assemblati alla destinazione. SCTP, figura 2.4, applica 28 Byte di overhead, 12 Byte in comune più 16 Byte specifici per ogni tipo di chunk, che unito ad una stretta di mano a

quattro vie portano ad un sensibile rallentamento delle prestazione dell'applicazione.
Un'altra particolarità di SCTP è di fornire la scelta di una configurazione ad hoc dei servizi da lui offerti, es. è possibile richiedere a SCTP un trasporto affidabile ma un trasferimento fuori ordine.
Questo protocollo non è sempre correttamente supportato dai router, costringendo a un incapsulamento di SCTP all'interno di un altro protocollo dello stesso livello della pila protocollare: *tunneling*.
Nel caso specifico qui trattato, SCTP, si trova incapsulato all'interno di DTLS che a sua volta è contenuto in un segmento UDP.

Bits	Bits 0 - 7	8 - 15	16 - 23	24 - 31
+0	Source port		Destination port	
32	Verification tag			
64	Checksum			
96	Chunk 1 type	Chunk 1 flags	Chunk 1 length	
128	Chunk 1 data			
...	...			
...	Chunk N type	Chunk N flags	Chunk N length	
...	Chunk N data			

Figura 3.9: La parte in blu rappresenta l'header in comune a ogni chunck. la verde solo il primo chunk e la rossa i restanti.

3.3 DataChannel

Stabilita la connessione tra i peer, mediante RTCPeerConnection, DataChannel permette lo scambio di dati generici tra due sessioni di browser differenti tramite un canale bidirezionale.
DataChannel è impiegato su diversi fronti: applicativi di messaggistica, on·line gaming, simulazione, etc..
Come anticipato, si appoggia, dove possibile, su SCTP a differenza del flusso dati audio/video che utilizza SRTP, specificando all'interno della sessione

SDP la scelta di un protocollo al posto dell'altro.
Successivamente allo scambio delle sessioni, si procede con la spedizione di un messaggio ad hoc per alcune specifiche di DataChannel, *DATA_CHANNEL_OPEN*. Questo contiene informazioni aggiuntive sulla gestione del canale di comunicazione già in parte definite tramite SCTP. Nel dettaglio, nel caso in cui si sia richiesto a SCTP un canale parzialmente affidabile, si può specificare il tempo per la ritrasmissione dei presunti pacchetti persi.
A seguire nella figura 2.10 un parte dell'implementazione.

```
var RTCDataChannel = {
    createDataChannel: function(peer, config) {
        var channel = peer.createDataChannel('RTCDataChannel', {
            reliable: true
        });
        this.setChannelEvents(channel, config);
    },
    setChannelEvents: function(channel, config) {
        channel.onopen = function() {
            config.onopen({
                channel: channel,
                userid: config.to
            });
        };

        channel.onmessage = function(e) {
            config.onmessage({
                data: e.data,
                userid: config.to
            });
        };

        channel.onclose = function(event) {
            config.onclose({
                event: event,
                userid: config.to
            });
        };

        channel.onerror = function(event) {
            config.onerror({
                event: event,
                userid: config.to
            });
        };
    }
};
```

Figura 3.10: Parte dell'implementazione di DataChannel [9].

Come anticipato WebRTC nasce per videoconferenza, solo in seguito è stato aggiunto DataChannel che permette la manipolazione di dati più astrat-

ti.

Proprio su questo è stato possibile realizzare un simulatore distribuito in cui si è mappato logicamente il concetto di LP, della simulazione ortodossa, in una sessione del browser. I PEU realizzati da qualsiasi dispositivo, es. smartphone, che abbia la possibilità di eseguire una versione di browser che supporti le API di WebRTC. Ultimo, ma non per importanza, il calcolo parallelo, a oggi, è supportato da tutti i browser compatibili con le API di HTML5 tramite *WebWorker*[8].

I WebWorker combinati alle API di WebRTC hanno permesso di realizzare un PADS.

[8]JavaScritp API che permettono di sfruttare il multi·core e CPU, permettendo di eseguire simultaneamente più script.

Capitolo 4

Implementazione e valutazione sperimentale

Attraverso i capitoli precedenti si sono introdotti i concetti principali della simulazione (capitolo 1) e di WebRTC (capitolo 2) arrivando fino a questa sezione dove verranno trattate le parti salienti dello sviluppo di un simulatore WebBased con sincronizzazione Time Step.
Di seguito si desscriveranno ed illustreranno le strutture dati, le strutture dei messaggi ivi scambiati, la sincronizzazione fra gli LP e le euristiche di bilanciamento del carico di lavoro fra i nodi. A conclusione, una serie di test comparativi per testare l'efficienza dell'applicativo tramite un algoritmo di mobilità, *Random Waypoint Model*, con l'utilizzo o meno di tecniche di bilanciamento.
Trattandosi di una implementazione in JavaScript si accenneranno anche alcune delle problematiche ad esso legate e riscontrate in fase si sviluppo.

4.1 Implementazione

Avviato il server socket, tramite il comado da shell *node signaling.js*, l'applicativo presenta una interfaccia grafica avviando lo scambio delle sessioni (capitolo 2 paragrafo 2.2). Questa si presenta con due bottoni (figura 3.1):

il primo, *New Data Connection*, utilizzato solo dal primo nodo per aprire il canale di comunicazione condiviso sul server; il secondo, *Join Connection*, utilizzato da tutti gli altri nodi per comunicare sullo stesso.

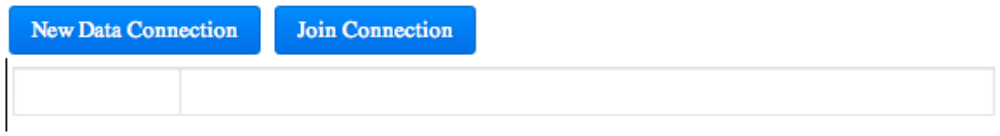

Figura 4.1: Interfaccia iniziale del simulatore per l'avvio della connessione fra gli LP.

Terminata la fase di signaling l'interfaccia stampa a video l'identificativo degli LP con cui si è instaurata una comunicazione peer·to·peer.

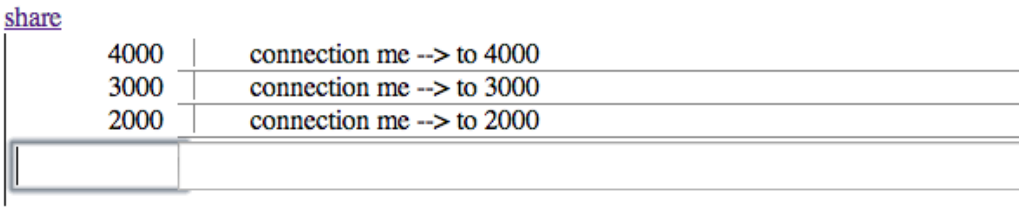

Figura 4.2: Interfaccia grafica del simulatore al termine della connessione fra gli LP.

Questi identificativi, scelti progressivamente a multipli di mille (partendo da 1000), sono utilizzati come base di calcolo per definire le caratteristiche delle entità simulate, tra queste caratteristiche il loro identificativo e le coordinate di partenza. In questo modo ad ogni istanza dell'applicativo si avranno sempre gli stessi identificativi in relazione al numero di LP partecipanti. Ad esempio per 2 LP identificativi 1000 e 2000, per 3 LP identificatvi 1000, 2000

e 3000.

Le funzioni JavaSciprt dell'interazione tra l'interfaccia grafica e l'avvio della simulazione sono contenute all'interno del file *Adapter.js* :

- *('setup-new-connection').onclick*: come anticipato, con questa funzione, si apre un canale di comunicazione condiviso sul server socket. Al suo interno sono contenuti i metodi *check e setup* dell'oggetto *DataConnection* per l'inizializzazione dell'SDP.
  In questo oggetto DataConnection, contenuto in DataChannel.js, è implementato il necessario per il funzionamento di WebRTC già descritto nel capitolo 2;

- *('join-connetcion').onclick*: per mezzo di quest funzione è possibile collegarsi al server socket, sul canale condiviso esistente, ed iniziare lo scambio delle sessioni con gli altri peer collegati sullo stesso canale;

All'interno del medesimo file *Adapter.js* sono presenti altre due importanti metodi dell'oggetto *DataConnection* non collegati direttamente all'interfaccia grafica:

- *onmessage*: questo metodo intercetta tutti i messaggi in arrivo sul canale di comunicazioni con gli altri nodi per poi inoltrarli, a sua volta, all'oggetto *hubObj*;

- *onopen*: una volta instaurata una comunicazione diretta fra i nodi, in base al numero di connessioni peer to peer instaurate ed al numero degli LP definiti a priori, viene manda un messaggio *start* che identifica l'avvio della simulazione;

In fine in questo file è istanziato l'oggetto *hubObj*, definito nel file *Hub.js*, che rappresenta il cuore del simulatore.

All'interno di *hubObj* sono presenti gli *handler* per l'interpretazione dei messaggi proveniente dagli altri LP e dai *WebWorker*. Esso, inoltre, definisce le strutture dati per la gestione e la sincronizzazione delle entità simulate, stabilisce se e quando migrare una certa enità per bilanciare il carico di lavoro di ogni LP.

4.1.1 Strutture Dati

In hubObj sono presenti diverse strutture dati, le quali si dividono fra quelle impiegate per la gestione delle entità simulate e quelle per la gestione dei messaggi in ricezione ed in invio.
Fra le strutture dati per la gestione delle entità si riportano le seguenti:

- *DataEntity*: in questo particolare oggetto, contenuto nel file DataEntity.js, sono implementati i metodi per la gestione dei dati delle entità sia in fase di inizializzazione sia in fase di migrazione.
  DataEntity contiene al suo interno anche due vettori: *interacIntra* ed *interacExtra* dedicati al conteggio del numero di interazioni con le altre entità simulate sia interne che esterne all'LP. Questa operazione viene svolta tramite i metodi *getInteractionExtra* e *getInteractionIntra*;
  Ogni istanza di DataEntity ha i seguenti parametri in ingresso:
  - *channel*: LP di appartenenza dell'entità;
  - *id*: identificativo dell'entità, è sempre dato dalla somma dell'LP (multiplo di 1000) ed un progressivo;
  - *SW_K*: utilizzato per determinare la dimensione dei vettori *interacIntra* e *interacExtra* per il conteggio del numero d'interazioni con le altre entità interne ed esterne;
  - *LP*: numero degli LP partecipanti alla simulazione;
- *DataEntityExtra*: questo oggetto, contenuto nel file *DataEntityExtra.js*, contiene i dati dell'entità in gestione dagli altri LP;

Rispetto a DataEntity tiene il minimo dei dati utili dell'entità; precisamente id, coordinate, velocità e LP di appartenza.

I parametri di DataEntity sopra citati sono utilizzati come seme per la libreria *Random.js* [10][1] per ottenere un sequenza di numeri pseudocasuali per una qualsiasi scelta randomica del simulatore tra cui, coordinate di partenza e di destinazione delle entità, velocità di spostamento delle entità, tempo di sospensione e cosi via.
Gli oggetti DataEntityExtra e DataEntity sono organizzati in *HashTable* con chiave l'id dell'entità e valore l'oggetto stesso. La scelta di HashTable è dovuta alle sue performance in termini di confronti sia in fase di inserimento che di recupero[2].
L'utilizzo di HashTable non è limitato solo a DataEntity ed a DataEntityExtra ma vede il suo impiego anche in:

- *hashTableEos*: tiene traccia degli *end of step* provenienti dagli altri N-1 LP per ogni step.
  Come anticipato, trattandosi, di un simulatore con sincronizzazione Time Step, hashTableEos è un punto fondamentale per la sincronia fra gli LP. Come chiave ha l'LP mittente e come valore il Time Step nel quale è stato spedito. Al termine di ogni passo di simulazione, hashTableEos, è svuotato mediate un suo metodo *clear* di HashTable;

- *hashTableMig*: ha come chiave l'id dell'entità prossima alla migrazione e come valore il Time Step nel quale avverrà la migrazione;

[1] La scelta di una libreria che garantisse la generazione di numeri pseudocasuali è obbligata in quanto il linguaggio impiegato non mette a disposizione tali funzionalità. Al fine di ottenere istanze del simulatore identiche e confrontabili, la stessa libreria è impiegata per tutte le scelte casuali nel progetto.

[2] JavaScript non mette a disposizione HashTable, ne è stata implementata una ad hoc tramite *Array* associativi, file HashTable.js, col supplemento di alcune ottimizzazioni per il conteggio degli elementi presenti nel vettore.

4.1.2 Strutture Messaggi

A differenza delle strutture dati impiegate per la gestione delle entità, i messaggi scambiati fra LP sono contenuti nell'oggetto *DataMessage*, implementato nel file *DataMessage.js*. Prima di essere spediti DataConnection li incapsula all'interno di oggetti *JavaSctip Object Notation, JSON*[3].

L'oggetto DataMessage definisce tre variabile al suo interno:

- *DataMessage*: LP destinatario del messaggio;
- *data*: contenuto del messaggio;
- *timeSteps*: specifica a quale step di simulazione deve essere spedito il messaggio;

I messaggi sono organizzati in due *Heap*: *minHeap* per i messaggi in spedizione e *minHeapR* per i messaggi in ricezione[4].
L'heap utilizzato ha come ordinamento il valore Time Step dell'oggetto DataMessagge, e posiziona in cima all'albero i messaggi da processare, in invio o in ricezione, con Time Step più basso.
Questo tipo di organizzazione dei messaggi è stata oppositamente scelta per poter processare i dati con un ordine temporale, cioè di Time Step.
Durante la simulazione alcuni messaggi chiedono di essere processati con un Time Step preciso rispetto a quello in cui vengono generati. Si portano come esempio i messaggi di migrazione effettuata con un Time Step successivo a quello in cui viene deciso.
I messaggi scambiati fra gli LP sono:

- EOS *(End Of Step)*: ogni LP, al termine di tutte le attività computazionali e di input/output, manda questo tipo di messaggio in broadcast per comunicare a tutti gli LP la fine delle attività.

[3]È un semplice formato per lo scambio di dati basato su due tipi di strutture: coppia chiave/valore o un elenco ordinato di valori.

[4]Come per le HashTable è stato implementato un MinHeap ad hoc, nel file *MinHeap.js*

Successivamente si sospende in attesa di ricevere gli N - 1 messaggi di EOS, relativi al passo corrente, per procedere allo step successivo.
I messaggi di EOS, in concomitanza delle strutture dati sopra citate, risultano essere i cardini della sincronizzazione Time Step fra gli LP;

- INTERAC: questo messaggio rappresenta il ping fra entità simulate;

- TIME: utilizzato per notificare il tempo impiegato per eseguire tutte le operazione per il passo di simulazione appena terminato.
  Tramite questo messaggio sarà possibile successivamente implementare l'euristica di bilanciamento del carico di lavoro;

- EXC: al suo interno sono racchiuse le nuove posizioni raggiunte dalle entità simulate durante il passo di simulazione corrente.
  I dati contenuti in questo messaggio sono il risultato dell'algoritmo che rappresenta il sistema simulato;

- MIG_EXT: notifica a tutti gli LP che ad un determinato passo di simulazione avverrà la migrazione di una certa entità dall'LP mittente del messaggio all'LP specificato come destinatario;
  Questo messaggio richiede la modifica della HashTable che mantiene le entità esterne all'LP.

- MIG: il messaggio ha come destinatario il ricevente dell'entità prossima alla migrazione, richiedendo la modifica di hashTableEntityExtra e hashTableEntity;

- start: viene generato da Adapter.js, una volta raggiunti gli LP - 1 di connessioni WebRTC, per avviare il passo 0 della simulazione.
  In questo specifico momento si istanziano le strutture dati ed i Web-Worker per poi sospendersi in attesa degli EOS;

4.1.3 WebWorker

Come già anticipato, i WebWorker permettono l'esecuzione di più script contemporaneamente, riuscendo a sfruttare il *multithreading*[5] del calcolatore.

In questo progetto si è cercato di separare, laddove possibile, la parte riguardante l'implementazione del sistema simulato dall'implementazione del simulatore. Per fare ciò si è concentrata l'implementazione dell'algoritmo, usato per i test finali, nel file eseguito in contemporanea dai WebWorker, ottenendo cosi l'esecuzione in parallelo del sistema simulato.
I WebWorker per scambiarsi i dati col resto dell'applicativo, a loro volta, utilizzano degli specifici handler:

- *onmessage*: impiegato per la ricezione dei messaggi.
  Questi possono essere *start* (utilizzato solo all'avvio del simulatore) e *move* (chiamato ad ogni passo di simulazione per l'esecuzione dell'algoritmo implementato);

- *postMessage*: utilizzato per l'invio dei messaggi.
  L'unico messaggio ad essere inviato dai WebWorker è *UPD* il quale trasporta il risultato dell'esecuzione dell'algoritmo implementato, nella fattispecie trasporterà le nuove posizioni raggiunte dalle varie entità simulate;

Lo scambio dei messaggi avviene esclusivamente con l'oggetto hubObj che implementa handler specifici per l'elaborazione dei dati contenuti nei messaggi inviati e ricevuti dai WebWorker.
Tra questi:

- *onmessage*: utilizzato per la ricezione dei messaggi dai WebWorker.
  Gli unici messaggi sono *EOS_S*, utilizzato solo all'avvio del simulatore, ed *UPD*, che aggiorna per ogni entità in hashTableEntity i relativi dati trasportati dal messaggio;

[5]Il termine multithreading definisce un processore in grado di eseguire più theard contemporanemente migliorando cosi le prestazione del programma eseguito.

- *postMessage*: impiegato per l'invio dei messaggi.
  Come in precedenza i messaggi sono due: *start* e *move*;

Nel passo di simulazione corrente, per ogni messaggio UPD ricevuto, viene incrementato un contatore *EOS_E* dove, quando questo raggiunge il numero delle entità gestite direttamente dall'LP quindi dopo aver eleborato tutte le entità simulate, l'applicativo procede col resto delle operazione previste.

4.1.4 Sistema adattivo e bilanciato

Come introdotto nel capitolo 1 è importante che ogni simulatore sia adattivo e vanti di un sistema di bilanciamento delle risorse per evitare colli di bottiglia, al fine di ottimizzare le performance durante la simulazione.
In questo progetto sono stati considerati ed implementati al meglio entrambi gli aspetti con l'introduzione di alcune euristiche di bilanciamento.

4.1.4.1 Adattività

Il primo passo per realizzare un sistema adattivo consiste nel partizionare il problema. Come si spiegherà nelle prossime sezioni, questo partizionamento avviene in maniera naturale, perchè suggerito dell'approccio stesso.
Un secondo punto fondamentale consiste nel dotare il simulatore della libertà di migrare da un LP ad altro le entità simulate, con la finalità di clusterizzare nello stesso LP le entità che comunicano più di frequente fra di loro [12].
Questa tecnica di clusterizzazione, prima di procedere con la scelta di quali entità migrare e la loro destinazione, necessita che per ogni l'LP e per ogni sua entità, vengano individuate quali altre entità si trovano entro una certa distanza scelta[6] per poi inviare a queste un messaggio di INTERAC.
Per ottimizzare questo passaggio e cioè il calcolo della distanza fra le entità,

[6]Corrisponde alla distanza Euclidea fra due punti sul piano cartesiano a 2 dimensioni. Questa distanza è data da

$$\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$$

una volta definita una costante RADIUS come distanza limite, si considerano due quadrati con al centro l'entità in esame; il primo con lato il doppio di RADIUS [11] ed il secondo con lato

$$\sqrt{RADIUS}$$

.

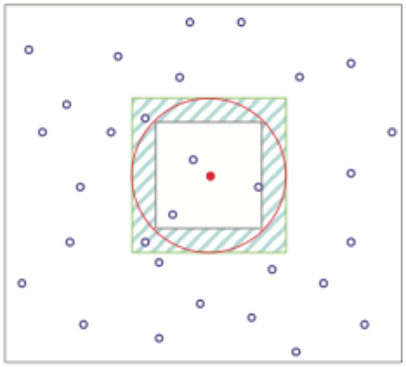

Figura 4.3: Rappresentazione dell'euristica dei due quadrati.

Come da figura 3.3, solo per le entità simulate fra i due quadrati verrà eseguito il calcolo della distanza Euclidea così riducendo sensibilmente la computazione per individuare le entità all'interno di RADIUS.
Ovviamente tutte le entità al di fuori del quadrato più esterno saranno scartate contratriamente a quelle contenute nel quadrato più interno.
Queste operazioni vengono effettuate per ogni entità dell'LP ad ogni passo di simulazione. Per giungere alla scelta di quale entità migrare e su quale LP, vengono presi in esame solo gli ultimi k passi di simulazione.
Le informazioni relative ai messaggi di INTERAC ricevuti e inviati alle entità esterne ed interne all'LP, sono organizzate nel vettore sopra citato interacExtra, mediante i metodi setInteractionIntra, setInteractionExtra, getInteractionIntra e getInteractionExtra di DataEntity.
Tramite l'ausilio di questi metodi si è implementata una finestra di scorrimento con dimensione k, che restituisce due valori:

- ε: totale delle interazioni avvenute dall'entità in esame verso l'esterno;
- ι: totale delle interazioni avvenute all'interno dell'LP;

Il rapporto $\frac{\varepsilon}{\iota}$ produce un valore α dove se risulta essere maggiore di MK[7] e di MT[8] l'entità è candidata alla migrazione con destinazione l'LP con cui ha scambiato il maggior numero di messaggi.
La tecnica descritta porta ad una clusterizzazione delle entità che più di frequente comunicano, avviando una riduzione del numero di messaggio di INTERAC immessi sul canale di comunicazione e di conseguenza un risparmiando tempo.

4.1.4.2 Bilanciamento

Una ulteriore caratteristica fondamentale di un simulatore è il riuscire a bilanciare nel miglior modo possibile il carico di lavoro fra i vari LP. Questo risulta indispensabile per più motivi, ad esempio non tutti i calcolatori partecipanti alla simulazione hanno la stessa capacità di calcolo.
Fondamentale, in termini puramente prestazionali, significa bilanciare equamente il carico di lavoro al fine di limitare il più possibile l'effetto collo di bottiglia fra LP partecipanti.
Sulla base di ciò è stata implementata una euristica di bilanciamento che ha come finalità l'individuare l'LP meno prestante e successivamente evitare di migrare verso quest'ultimo nuove entità.
Per individuare questo l'LP, tramite i messaggi TIME, si calcola ogni k passi l'LP che ha avuto la media dei tempi di elaborazione peggiore. Per i k passi successivi, gli LP partecipanti non migreranno nuove entità verso quest'ultimo.

[7] *Fattore di Migrazione* è una costante per tener controllata la migrazione. Per l'implementazione è stata usato con valore 2

[8] *Soglia di Migrazione* come MF è una costante, scelta con valore 5, che indica quanti passi di simulazione devono passare prima che una data entità possa migrare nuovamente dalla sua ultima migrazione.

Il metodo di bilanciamento appena descritto può risultare, durante la simulazione, in contrapposizione alla euristica di clusterizzazione sopra citata, ma si è scelto di privilegiare per i test finali la tecnica di bilanciamento.
Oltre all'euristica di bilanciamento si è aggiunto un vincolo riguardante il numero minimo di entità in ogni LP; questo per evitare che un nodo inizialmente partecipe alla simulazione possa rimanere senza alcuna entità simulata da processare.
Per i test comparativi k è stato scelto a 3 e vincolando ogni LP a non poter cedere più del 30% delle entità inizialmente istanziate.

4.2 Algoritmo di Mobilità (Random Waypoint Model)

Per testare la funzionalità del simulatore implementato e in particolar modo l'efficienza delle tecniche di bilanciamento e di adattività, si è scelto l'algoritmo di mobilità *Random Waypoint Model*.
Questo modello vede i propri nodi, nella fattispecie rappresentati da entità simulate, muoversi in maniera casuale e scegliere destinazione e velocità randomicamente ma soprattutto in maniera completamente indipendetemente l'uno dagli altri.
Il comportamento di ogni nodo segue le seguenti semplici regole:

- ogni nodo inizializzato si sospende per un numero fisso di secondi;
- sceglie una destinazione casuale nell'area di simulazione;
- sceglie una velocità compresa fra 0 e la velocità massima consentita;
- il nodo si sposta alla velocità scelta con moto uniforme verso la destinazione;
- raggiunta la destinazione si sospende per un tempo casuale di secondi, per poi ricominciare;

Ogni nodo dell'algoritmo è espresso da un'istanza dell'oggetto *Entity*, implementato in *Entity.js*, dove al suo interno vi è il necessario per l'algoritmo descritto.
Gli spostamenti dei nodi avvengono in un piano cartesiano 2D circolare, cosi facendo se un certa entità raggiunge un bordo continua il suo percorso senza deviazioni.
Random Waypoint Model è spesso impiegato per rappresentare dispositivi mobili all'interno di reti wireless [13].

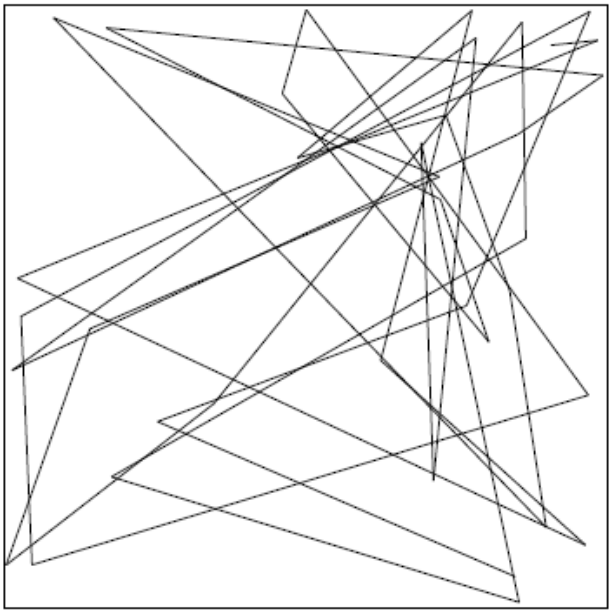

Figura 4.4: Esempio di un movimento di un nodo secondo Random Waypoint Model.

4.3 Valutazioni sperimentali

Per l'applicativo descritto sono stati effettuati una serie di test comparativi con 2,3 e 4 calcolatori per valutarne l'efficienza delle tecniche di bilanciamento e di migrazione.
I calcolatori impiegati sono cosi equipaggiati:

- PC1: Notebook con Processore Intel® Core i5-2430M da 2.40 Ghz, 8GB (2 core, 4 thread), sistema operativo OSX 10.9.
- PC2: Desktop Processore Intel® Core i3-3220 da 3.3 GHz, 4GB (2 core, 4 thread), sistema operativo Window 7
- PC3: Desktop Processore Intel® Core i3-3220 da 3.3 GHz, 4GB (2 core, 4 thread), sistema operativo Window 7
- PC3: Notebook Processore Intel® Celeron B830 Dual-core 1.80 Ghz, 8GB (2 core, 2 thread), sistema operativo Window 8.1

I vari calcolatori, per ogni test, hanno ospitato sempre e solo un LP ciascuno, con la particolarità del PC1. Questo ha sempre eseguito il server socket per il canale di comunicazione condiviso, che partecipasse o meno alla simulazione.
I calcolatori sopra elencati sono collegati mediante connessione cablata 100 Mbit ad un router con switch 4 porte, figura 3.5.

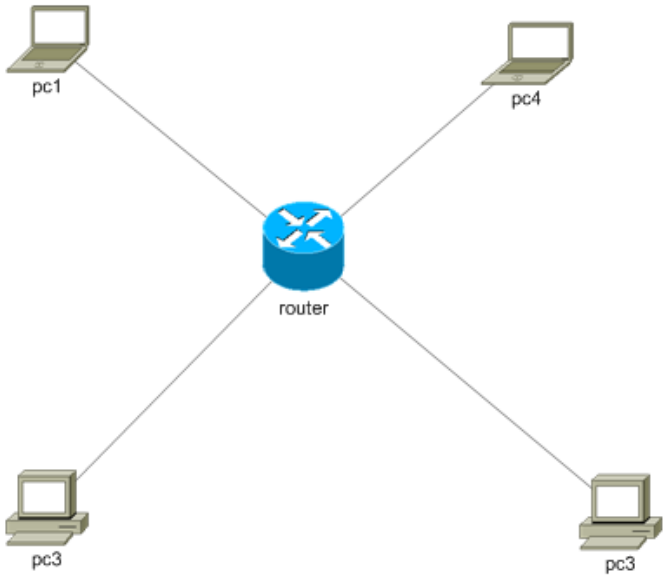

Figura 4.5: Topologia di rete utilizzata per i test.

Per poter effettuare istanze identiche dell'applicativo e garantire la riproducibilità dei test effettuati , a parità di condizioni di calcolatori, numero LP partecipanti , numero di entità e passi di simulazione, come già introdotto si utilizza per tutte le scelte casuali la libreria Random.js con seme l'identificativo dell'LP. Questo per poter garantire che ad ogni istanza dell'applicativo ogni LP riproduca sempre gli stessi identificativi per le entità simulate, dove a loro volta queste seguano sempre lo stesso comportamento portando come risultato istanze compali.
Per comprovare che il modello simulato abbia sempre la stesso evoluzione nella simulazione, si è generato un *trace*[9] per ogni LP, tracciando tutti gli spostamenti delle entità nel piano. Questo trace al termine della simulazione è spedito da ogni LP al PC1[10] che ospita il server socket e produce una stampa cosi eleborata:

[9]Si registra dell'evoluzione nel tempo di una o più parti del programma, nel caso specifico delle entità simulate.

[10]In Javascript lato client non è possibile, nella maggioranza dei browser, poter accedere al filesystem del calcolatore per questioni di sicurezza.

```
------------

- LP : 2000

------------

-- Step : 0

-- Step : 1
--- ID : 2001
--- Speed : 3
--- X : 223
--- Y : 221

--- ID : 2002
--- Speed : 19
--- X : 435
--- Y : 35

--- ID : 2003
--- Speed : 5
--- X : 169
--- Y : 276
```

Figura 4.6: Trace da un LP con identificativo 2000 per le entità simulate con identificato 2001,2002 e 2003.

Ogni test comparativo è composto da 10 misurazioni differenti, ognuna di queste composta da 100 passi di simulazione, ripartendo equamente fra gli LP 300 entità simulate. Ai tempi di computazione rilevati, misurati in secondi, si riportano media e varianza con relativa differenza, Δ, percentuale [11].

[11] Fornisce una misura della variabilità dei valori assunti, nello specifico quanto discosta quadraticamente dalla media.

Media E_t e varianza σ^2 dei tempi di elaborazione in secondi.

	Con migrazione		Senza migrazione		
Calcolatori	E_t	σ_t^2	E_t	σ_t^2	Δ
PC2,PC4	203,5	248,125	248,125	0,5469	17,98%
PC2,PC3,PC4	173,9	399,105	205,5	174.25	15,38%
PC1,PC2,PC3,PC4	144,675	11,12	154,75	3,6875	6,51%

Tabella 4.1: Riepilogo dei vari scenari con e senza migrazione

A seguire grafici dell'andamento dei tempi di computazione per ogni 10 passi di simulazione nei vari scenari di calcolatori interessati, dove H rappresenta l'andamento della simulazione senza migrazione attivata, η con migrazione attivata.

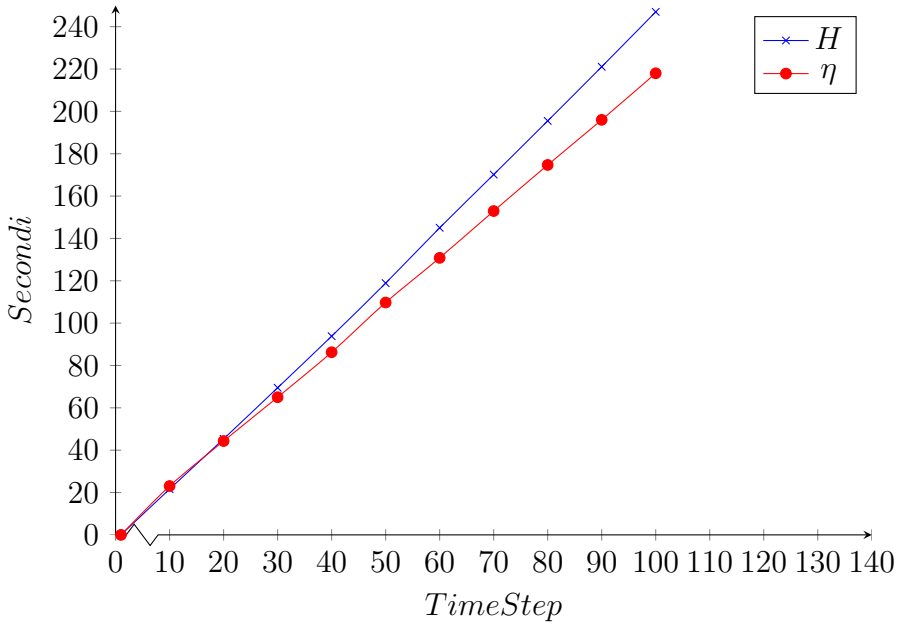

Figura 4.7: Scenario con 2 calcolatori PC2 e PC4

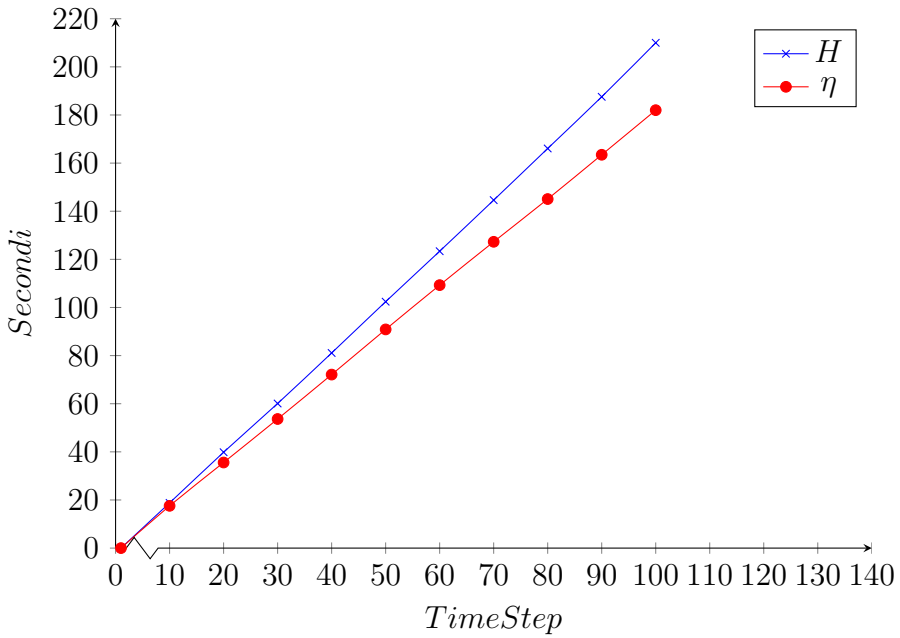

Figura 4.8: Scenario con 3 calcolatori PC2 , PC3 e PC4

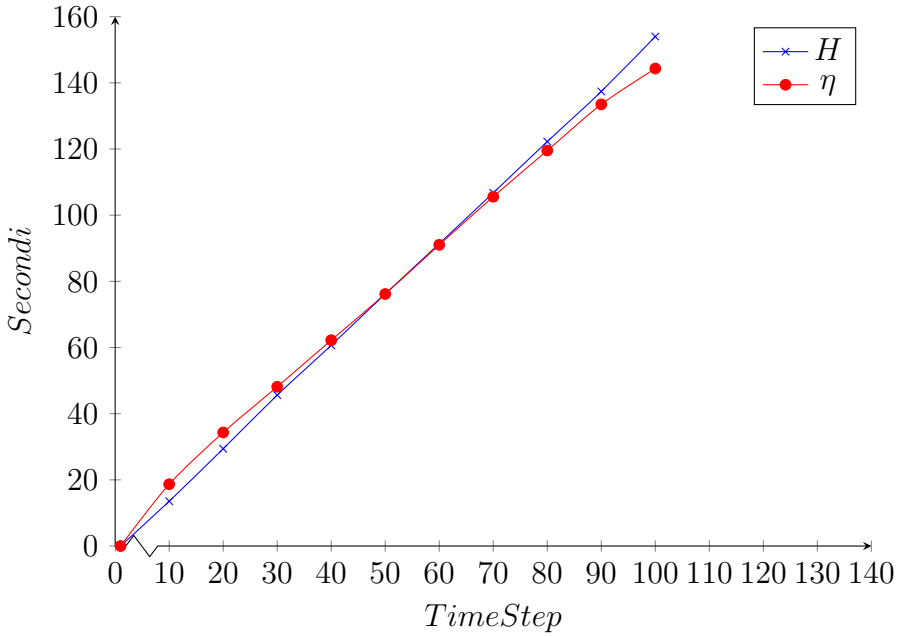

Figura 4.9: Scenario con 4 calcolatori PC1,PC2,PC3 e PC4

I dati raccolti tramite i test come descritti, confermano un miglioramento delle prestazioni con migrazione attivata.
Si nota una diminuzione della percentuale di miglioramento al crescere del numero degli LP partecipanti alla simulazione; si suppone che il calo del Δ miglioramento sia dovuto al numero delle entità simulate in relazione al numero degli LP partecipanti.
Su questa considerazione è stato effettuato un altro test comparativo col doppio delle entità simulate, 600, nello scenario di 4 PC coinvolti.

Media E_t e varianza σ^2 dei tempi di elaborazione in secondi.

	Con migrazione		Senza migrazione		
Calcolatori	E_t	σ_t^2	E_t	σ_t^2	Δ
PC1,PC2,PC3,PC4	563,77	357,59	792,695	109,41	28,87%

Tabella 4.2: Scenario con 4 LP, 600 entità e 100 passi di simulazione

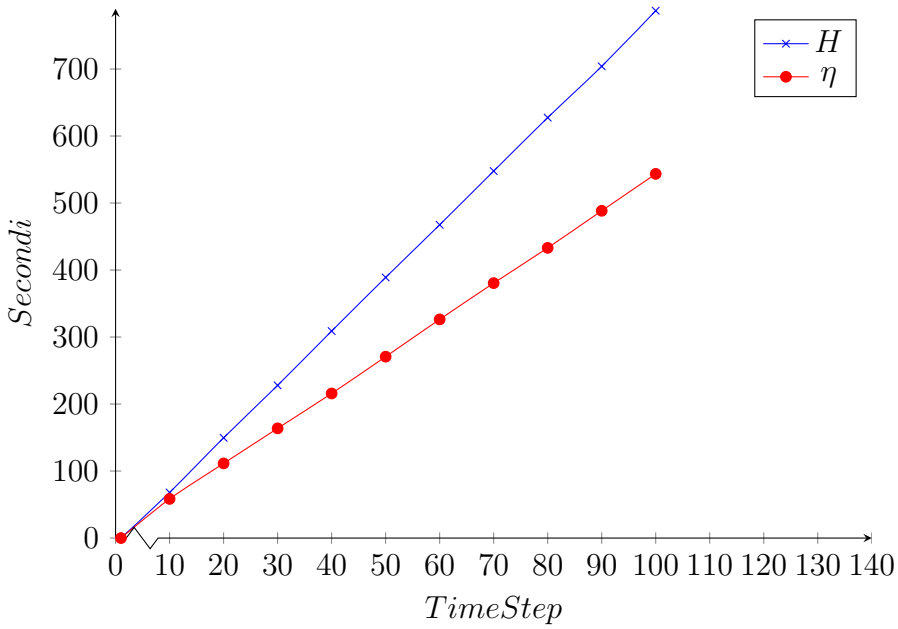

Figura 4.10: Scenario con 4 calcolatori PC1,PC2,PC3 e PC4

Un ultimo test è stato effettuato nello scenario appena descritto con la differenza di uno dei protocolli impiegato a livello di trasporto da WebRTC, SCTP.

Questo è stato sostituito da *Real-time Transport Protoco, RTP* avendo un overhead di 12 Byte rispetto ai 28 di SCTP.

Media E_t e varianza σ^2 dei tempi di elaborazione in secondi.

	Con migrazione		Senza migrazione		
Calcolatori	E_t	σ_t^2	E_t	σ_t^2	Δ
PC1,PC2,PC3,PC4	497,88	320,75	710,17	98,4	29,89%

Tabella 4.3: Scenario con 4 LP, 600 entità e 100 passi di simulazione con RTP

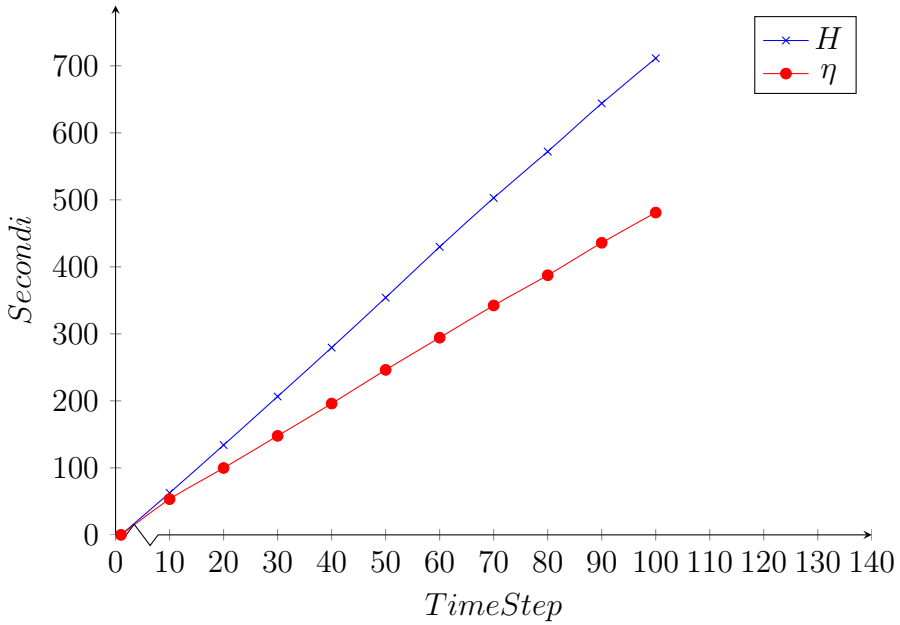

Figura 4.11: Scenario con 4 calcolatori

Dai dati raccolti, utilizzando RTP al posto di SCTP, si rileva un incremento dell'11% circa di prestazioni a parità di condizioni. Ciò non toglie che RTP è un protocollo differente da SCTP in quando non offre nessun meccanismo di affidabilità nella trasmissione dei dati, creando così potenzialmente problemi nella sincronizzazione fra gli LP. Non sono comunque state riscontrate anomalie di alcun genere nell'esecuzione dei test sopra citati.
Tutti i dati elencati sono stati raccolti eseguendo l'applicativo col browser Google Chrome.

Capitolo 5

Conclusioni

WebRTC è una tecnologia relativamente giovane dove a oggi già conta numerosi impieghi. Nasce per applicativi di videoconferenza, evolvendosi fino a proporsi come un nuovo standard per lo scambio di dati fra browser. Ciò nonostante non tutti i browser al momento supportano questa nuova tecnologia, Google Chrome, Mozilla Firefox e Opera sono stati i primi ad abbracciare questa innovativa proposta quando altri, Safari di Apple e Explorer, ancora non sono fra questi.
In questa tesi si è proposto un nuovo utilizzo di questa tecnologia così diffusa, approcciando ad una argomentazione diametralmente opposta agli impieghi dei suoi albori: la simulazione.
È stato possibile realizzare un applicativo distribuito, mettendo in pratica tecniche note ma anche la rielaborazione di alcune di queste, e testare la loro efficienza mediante uno dei più noti algoritmi di mobilità.
Le prove sopra citate portano alcune considerazioni in merito ad un suo utilizzo in un ambito professionale, riconsiderando un suo eventuale impiego solo in limitate circostanze, dettate dai limiti prestazionali indotti dal linguaggio di programmazione.
Questo linguaggio presenta anche altre carenze come ad esempio le estensioni messe a disposizione per la programmazione in multithreading risultano ancora non equiparabili alle librerie di altri linguaggi, sebbene è incontrover-

tibile la sua vasta diffusione.

Il largo utilizzo è stato sicuramente, in parte, agevolato dalla sua semplicità; in questo progetto aggiungere un nuovo calcolatore alla simulazione è sufficiente aprire la sessione di un browser che supporta WebRTC.

Gli odierni applicativi per la simulazione vertono la lora attenzione su due punti fondamentali: efficienza ed usabilità [2]. Proprio su quest'ultimi si ritiene che, con gli eventuali sviluppi prestazionali di questa nuova tecnologia, essa possa rivelarsi un prezioso ausilio per futuri progetti di simulazione.

Bibliografia

[1] Gabriele D'Angelo and M. Bracuto.
Distributed simulation of large-scale and detailed models. International Journal of Simulation and Process Modelling, Vol. 5, No. 2, 2009

[2] Gabriele D'Angelo.
Parallel and Distributed Simulation: from Many Cores to the Public Cloud. International Conference on High Performance Computing and Simulation (HPCS), 2011

[3] Kalyan Perumalla.
Parallel and Distributed Simulation (PADS):Traditional Techniques and Recent Advances. WSC '06 Proceedings of the 38th conference on Winter simulation.

[4] Richard M.Fujimoto.
Parallel and Distributed Simulation Systems: From Chandy·Misra to High Level Architetture and Beyond.
John Wiley and Sons, 2000.

[5] Sam Dutton
WebRTC in the real world: STUN, TURN and signaling, 2014
http://www.html5rocks.com/

[6] Ilya Grigorik
High-Performance Browser Networking, 2013
O'Reilly Media, Inc.

[7] Wikipedia
http://en.wikipedia.org/wiki/Bandwidth_(computing).

[8] Reti di Calcolatori e Internet. Un approccio top·down.
James F.Kurose, Keith W.Rose *Pearson Addison wesley, 2008*

[9] Muaz Khan
https://www.webrtc-experiment.com

[10] Maneesh Varshney *Number Generation*
http://simjs.com/random.html

[11] Luciano, Bononi Michele Bracuto, Gabriele D'Angelo, Lorenzo Donatiello
Proximity Detection in Distributed Simulation of Wireless Mobile Systems.
MSWiM '06 Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems.

[12] Gabriele D'Angelo.
The simulation model partitioning problem: an adaptive solution based on self-clustering, Work in progress.

[13] Johnson, D. B.; Maltz, D. A.
Dynamic Source Routing in Ad Hoc Wireless Networks (1996).
Mobile Computing