Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Scuola Di Scienze

Corso di Laurea in Ingegneria e Scienze Informatiche

APPROCCIO ARCHITETTURALE AD ALTO LIVELLO PER REALIZZARE DTLS FAST RESUMPTION IN WEBRTC

Relazione finale in
System Integration

Relatore:
Prof. Vittorio Ghini

Presentata da:
Filippo Paganelli

II Sessione
Anno Accademico 2018/2019

Parole Chiave

DTLS
WebRTC
Fast Resumption
BoringSSL

Sommario

Al giorno d'oggi è molto facile trovarsi in situazioni nelle quali un device durante la comunicazione cambi frequentemente indirizzo IP. Il presente documento di tesi illustra le fasi di approccio alla libreria WebRTC nel tentativo di implementare le tecniche per il ripristino rapido della connessione, quali Fast Resumption e Session Resumption, attraverso il protocollo DTLS.
Il percorso di ricerca e sviluppo è partito dallo studio della struttura della libreria, poi si è sviluppato in tre strade differenti. Questi percorsi variano in base al livello di approccio verso la libreria.
Il primo approccio, al quale mi sono dedicato maggiormente, è stato quello ad alto livello. In questo approccio sono state analizzate le WebAPIs JavaScript standard e il loro possibile utilizzo per il nostro obiettivo. Dopo un attento studio e lo sviluppo di un applicativo di test, si è mostrata l'impossibilità di realizzare la Resumption in questo livello architetturale.
In seguito, mediante un approccio a medio livello al quale si è occupato principalmente il collega Milo Marchetti, è stato realizzato un applicativo interno alla libreria, che sfruttando le API della stessa, cercasse di istanziare un canale di trasmissione dati a cui applicare la Fast Resumption.
In fine, attraverso un approccio a basso livello al quale si è occupato maggiormente il collega Enrico Gnagnarella, si è verificata la presenza delle funzionalità di DTLS necessarie per riprodurre le tecniche di Resumption, si è proseguito quindi con il riadattamento della Fast Resumption sviluppata per OpenSSL all'interno della nostra libreria.
Questi ultimi due approcci sono stati ugualmente privi di risultati funzionanti. Nonostante ciò hanno permesso di trarre conclusioni riguardo al lavoro svolto, potendo così riflettere su un possibile approccio futuro alla libreria per realizzare la DTLS Fast Resumption.

Indice

Capitolo 1

Introduzione

In questo studio si analizza il caso in cui due sistemi stiano comunicando mediante l'utilizzo del protocollo DTLS. Si prenda in esame un caso in cui un client sia soggetto a frequenti cambiamenti di indirizzo IP; ad esempio perché il programma è in esecuzione su un sistema mobile che, spostandosi o entrando in lunghi periodi di sleep, è soggetto a reindirizzamento.
In condizioni normali il cambiamento di IP da parte del client porta ad un'interruzione della comunicazione e alla necessità di stabilire una nuova connessione facendo un handshake completo. Questo si verifica perché il server, non riconoscendo l'interlocutore, si comporta come in presenza di un client nuovo.
Questa dispendiosa operazione potrebbe necessitare di essere svolta molto frequentemente, aggiungendo dei delay significativi, che minano l'interattività della comunicazione.
Lo scopo del presente studio è quello di realizzare la Fast Resumption e la Session Resumption all'interno della libreria WebRTC. Tali tecniche permettono di effettuare un handshake abbreviato nel caso in cui un dispositivo cambi indirizzo IP.
La scelta di WebRTC, libreria che mette a disposizione strumenti semplici per la realizzazione di comunicazioni audio/video Real-Time in ambito Web, è stata dettata dalla sua larga diffusione negli ultimi anni. In aggiunta alla precedente motivazione, la suddetta libreria risulta particolarmente adatta al nostro scopo in quanto sfrutta il protocollo di sicurezza DTLS. Quest'ultimo è stato utilizzato per realizzare la versione OpenSSL delle tecniche di Resumption sopra citate.
L'obiettivo principale è dunque analizzare la libreria per capire al meglio come approc-

ciarsi ad essa e in che modo si dovrà implementare la soluzione, così da rendere la Resumption una funzionalità base della libreria.
Se ciò non sarà possibile, verranno individuate le origini delle limitazioni che ostacolano il raggiungimento dell'obiettivo preposto, si cercherà inoltre di capire se queste derivano dalla struttura di WebRTC o dalle specifiche della versione di DTLS utilizzata dalla libreria.
L'architettura a livelli, discussa nel Capitolo 7, ha permesso di organizzare il lavoro svolto in collaborazione con gli studenti Enrico Gnagnarella e Milo Marchetti. Personalmente mi sono occupato dello studio delle WebAPIs standard e della realizzazione di un semplice applicativo al fine di stabilire la fattibilità della Fast Resumption ad alto livello.
Per questo motivo, l'approccio da me adottato è definito ad alto livello e verrà descritto approfonditamente nel Capitolo 8.

Capitolo 2

WebRTC

2.1 Introduzione a WebRTC

WebRTC, acronimo di Web Real Time Comunication (comunicazione in tempo reale), è fondamentalmente un progetto open source promosso da Google che permette la comunicazione multimediale in tempo reale. Uno dei punti di forza di questa tecnologia è la possibilità di instaurare comunicazioni RTC su pagine web, senza necessitare l'installazione di plugins esterni, ma direttamente attraverso API JavaScript.
Ecco alcuni dei principali casi d'uso di WebRTC:

- Comunicazioni audio e/o video real-time
- Conferenze web
- Trasferimento diretto di dati tra browser

2.2 Architettura di WebRTC

WebRTC è studiato per consentire la comunicazione diretta di dati utilizzando un modello architetturale peer-to-peer (P2P). Come accennato nel paragrafo precedente uno dei punti di forza di questa tecnologia è quello di non necessitare l'installazione di plugins esterni, nonostante ciò, la comunicazione diretta fra i peer può avvenire solo successivamente ad una fase preliminare di scambio di metadati definita Signaling.

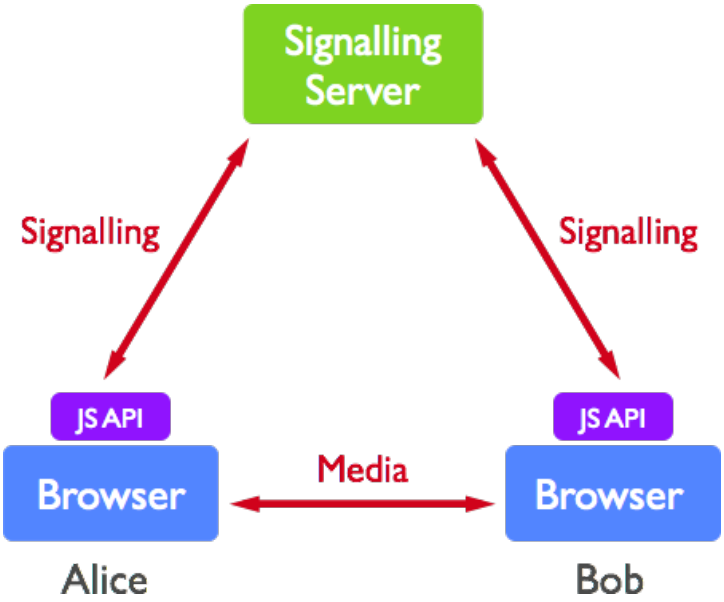

Figura 2.1: Architettura di WebRTC

2.3 Elementi Portanti della Comunicazione in WebRTC

WebRTC si basa su tre API, ognuna delle quali svolge una funzione specifica per consentire la comunicazione in tempo reale all'interno di un'applicazione Web. Senza soffermarci sulle API, in quanto verranno trattate più dettagliatamente in seguito, andiamo ad analizzare quali sono i presupposti principali di una comunicazione real-time.

- **Cattura di sorgenti audio e/o video**
  Per molti anni è stato necessario fare affidamento su plug-in di browser di terze parti come Flash o Silverlight per acquisire audio o video da un computer. Tuttavia, l'era di HTML 5 ha inaugurato l'accesso diretto all'hardware, a numerosi dispositivi e fornisce API JavaScript che si interfacciano con le funzionalità hardware sottostanti di un sistema.

- **Connessione tra peer (PeerConnection)**
  La connessione base tra peer, definita in WebRTC come PeerConnection, sta alla base della comunicazione e dello scambio dei dati. Una PeerConnection correttamente stabilita tra due peer deve essere in grado di consentire l'invio di dati audio e video in tempo reale come flusso di bit tra i browser. In sostanza la PeerConnection è responsabile della gestione dell'intero ciclo di vita di ogni connessione peer-to-peer, essa infatti incapsula in un'unica interfaccia tutte le informazioni e le

impostazioni necessarie alla corretta gestione di una comunicazione.

- **Canale dati (DataChannel)**
  Il canale dati, a differenza della PeerConnection, rappresenta il principale canale di comunicazione attraverso il quale avviene lo scambio di dati generici. Sebbene esistano molte di opzioni diverse per creare un canale di comunicazione (ad es. WebSocket, Server Sent Events, etc.), queste alternative non sono adatte ad una comunicazione peer-to-peer bensì ad una comunicazione Client-Server.

2.4 Protocolli e Tecnologie Sottostanti

Alla base di ciò che è stato precedentemente illustrato (DataChannel, PeerConnection) esistono diversi protocolli e tecnologie utilizzate, queste sono definite nel Protocol Stack di WebRTC illustrato qui di seguito.

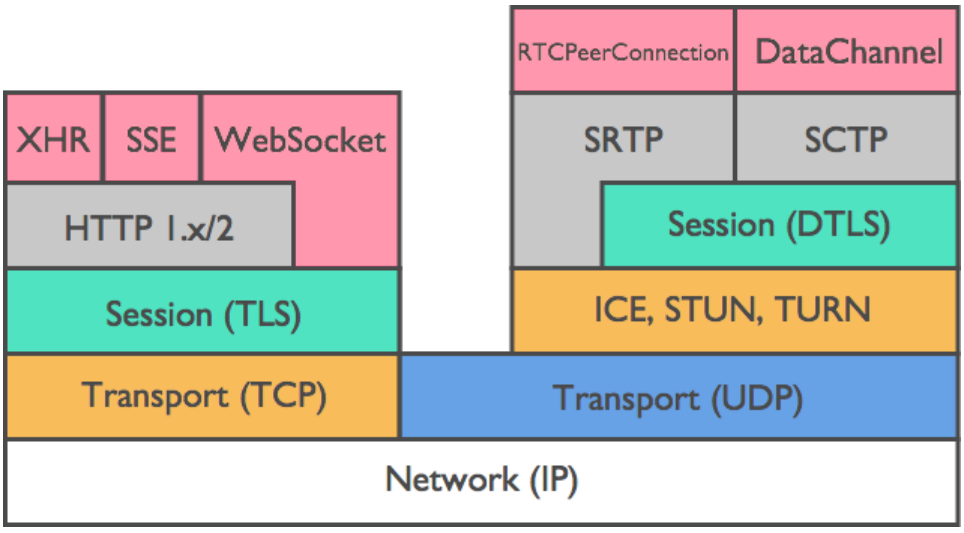

Figura 2.2: WebRTC Protocol Stack

ICE, STUN e TURN sono necessari per stabilire e mantenere una connessione peer-to-peer su UDP. DTLS viene utilizzato per proteggere tutti i trasferimenti di dati tra peer, poiché la crittografia è una funzione obbligatoria di WebRTC. Infine, SCTP e SRTP sono i protocolli applicativi utilizzati per multiplexare i diversi flussi, fornire congestione e controllo del flusso, fornire capacità di consegna parzialmente affidabile e altri servizi aggiuntivi su UDP.

2.4.1 SDP: Session Description Protocol

SDP è un formato realizzato per descrivere i parametri di inizializzazione di uno streaming di dati multimediali. Il formato SDP è utilizzato in WebRTC per definire le preferenze e le capacità di un peer che sta cercando di stabilire uno comunicazione real-time. Un oggetto SDP può contenere le seguenti informazioni:

- Capacità multimediali (video, audio) e codec utilizzati
- Indirizzo IP e numero di porta
- Protocollo di trasmissione dati peer-to-peer
- Larghezza di banda utilizzabile per la comunicazione
- Attributi della sessione come nome, identificativo, tempo attivo, ecc. (Informazioni non utilizzate in WebRTC)

Quando serializzato, un oggetto di tipo SDP, presenta la seguente struttura:

```
v = 0
o = - 3883943731 1 IN IP4 127.0.0.1
s =
t = 0 0
a = gruppo: BUNDLE audio video
m = audio 1 RTP / SAVPF 103 104 0 8 106 105 13 126

// ...

a = ssrc: 2223794119 etichetta:
H4fjnMzxy3dPIgQ7HxuCTLb4wLLLeRHnFxh810
```

Figura 2.3: Serialized SDP Object

Ad oggi SDP è ampiamente utilizzato nei contesti di Session Initiation Protocol (SIP), Real-time Transport Protocol (RTP) e Real-time Streaming Protocol (RSP).

2.4.2 Framework ICE

Finita la fase di segnalazione, in cui ci si scambiano le informazioni iniziali attraverso il formato SDP, WebRTC tenta di stabilire una connessione peer-to-peer. Questo processo viene svolto grazie all'ausilio del framework ICE.
ICE è un framework che consente a WebRTC di superare le complessità dovute al networking. Il compito di ICE è quello di trovare il modo migliore di connettere direttamente due peer cercando di risolvere le problematiche dovute alla presenza di NAT (Network

Address Translation).
A causa della continua prevalenza di indirizzi IPv4 avente una rappresentazione a 32 bit limitata, la maggior parte dei dispositivi abilitati alla rete non ha un indirizzo pubblico esclusivo con il quale risultare direttamente visibile su Internet. Il NAT funziona traducendo dinamicamente gli indirizzi privati in indirizzi pubblici quando una richiesta è in uscita e viceversa quando una risposta è in entrata. Di conseguenza, la condivisione di un IP privato spesso non è sufficiente per stabilire una connessione tra peer. ICE tenta di superare le difficoltà imposte dalla comunicazione attraverso i NAT così da trovare il percorso migliore per connettere i peer.
ICE tenta innanzitutto di stabilire una connessione utilizzando l'indirizzo dell'host ottenuto dal sistema operativo e dalla scheda di rete di un dispositivo; se fallisce (cosa che inevitabile quando si ha a che fare con dispositivi dietro NAT) ICE sfrutta due particolari tipi di server definiti STUN e TURN.

2.4.3 STUN

In caso di NAT Asimmetrico ICE userà un server STUN (Session Traversal Utilities for NAT). Un server STUN consente ai peer di scoprire i propri indirizzi IP pubblici e il tipo di NAT che li "protegge". Queste informazioni sono utili ad instaurare la connessione sulla quale viaggeranno i dati. Nella maggior parte dei casi STUN è utilizzato solo durate la fase di stabilimento della connessione, una volta che quest'ultima è stata stabilita i dati viaggiano direttamente tra i client.

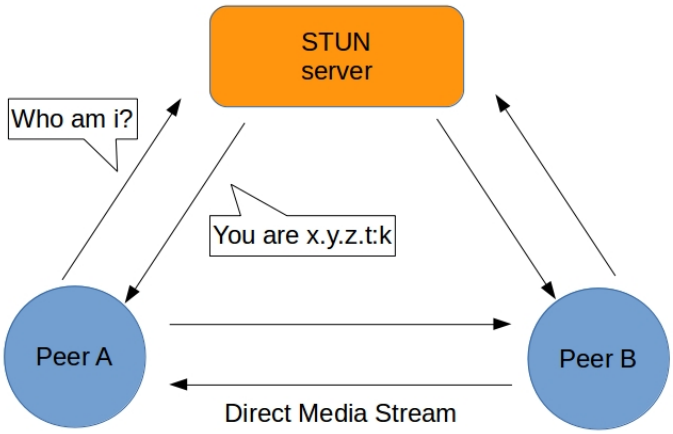

Figura 2.4: STUN Server

2.4.4 TURN

In caso il server STUN non riesca a stabilire la connessione, ICE può decidere di utilizzare TURN. TURN (Traversal Using Relay NAT) è un'estensione di STUN che consente ai dati di viaggiare attraverso NAT Simmetrici. Differentemente da STUN, il server TURN rimane anche dopo che la connessione è stata stabilita per trasmettere continuamente i dati che viaggiano tra due peer. Per questo motivo il server TURN è notoriamente definito "relay" server. Ovviamente e desiderabile non dover utilizzare un server TURN, ma non sempre ciò è possibile.

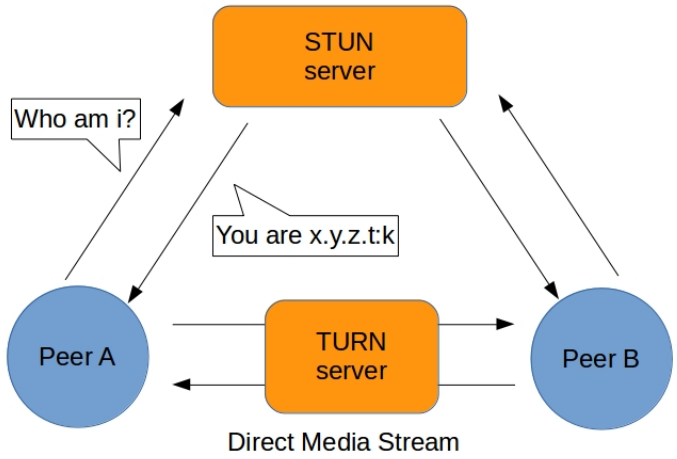

Figura 2.5: TURN Server

2.5 Ciclo di Vita di una Comunicazione Real-Time

Immaginiamoci la situazione in cui il Peer A voglia parlare con il Peer B. Il processo di instaurazione della comunicazione (mediante WebRTC) si divide in tre fasi:

1. Meccanismo di domanda e offerta (setup delle impostazioni multimediali) che sfrutta il formato SDP
2. Ricerca dei candidati ICE
3. Connessione diretta con ICE

Nella descrizione dettagliata di questi procedimenti non prenderemo in considerazione un livello architetturale specifico in quanto, come vedremo nei capitoli successivi, è possibile approcciarsi a WebRTC in diversi modi.

2.5.1 Meccanismo di Domanda e Offerta

Ecco in dettaglio il meccanismo di offerta/risposta:

1. A crea un oggetto PeerConncection
2. A crea un'offerta SDP (session description) sfruttando i metodi della PeerConnection
3. A utilizza l'offerta creata per impostare la session description locale
4. A serializza l'offerta e sfrutta il servizio di signaling per inviarlo a B
5. B imposta la session description remota sulla offerta ricevuta da A così che il suo PeerConncection sia a conoscenza del setup di A
6. B crea la risposta
7. B utilizza la risposta creata per impostare la session description locale
8. B sfrutta il servizio di signaling per inviare la risposta a A
9. A imposta la session description remota sulla risposta ricevuta da B così che il suo PeerConncection sia a conoscenza del setup di B.

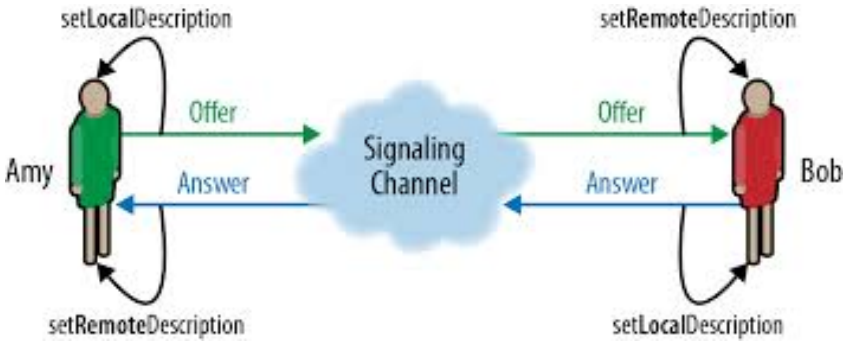

Figura 2.6: Answer/Offer Mechanism

2.5.2 Ricerca dei Candidati ICE

Attraverso l'espressione finding candidates ci si riferisce al processo di ricerca di interfacce di rete e porte che l'ICE Framework può utilizzare.

1. All'oggetto PeerConnection creato da A dovrà essere associato un gestore (handler) di candidati ICE
2. Questo gestore viene richiamato quando i candidati diventano disponibili
3. Il gestore notifica la disponibilità dei candidati i quali saranno, sotto forma stringata, inviati a B tramite il servizio di signaling
4. Una volta che i candidati hanno raggiunto il peer B esso li aggiunge alla descrizione remota (remoteDescription) tramite un metodo di PeerConnection

2.5.3 Connessione Diretta con ICE

Il Framework ICE compie dei tentativi al fine di stabilire la connessione. Se il tentativo corrente fallisce passa al successivo fino a raggiungere l'ultimo che consentirà sicuramente di stabilire la connessione.

1. Stabilire una connessione diretta mediante IP e Porta del Peer destinatario
2. Utilizzo di un server STUN per definire gli IP pubblici dei Peer che vogliono comunicare
3. Utilizzo di un server TURN come intermediario ("relay" server)

Capitolo 3

BoringSSL

Questo capitolo presenta una panoramica sulla libreria utilizza in WebRTC per realizzare comunicazioni end-to-end sicure su reti TCP/IP. Per molti anni e per diverse situazioni, Google ha utilizzato OpenSSL creando un gran numero di patch che sono state mantenute durante il suo tracciamento.
Con l'aumentare dell'offerta di prodotti Google sono aumentate anche le copie di OpenSSL. Per ovviare al mantenimento di tutte le patch si è deciso di effettuare un fork del repository OpenSSL realizzando una propria libreria, denominata BoringSSL. Tale libreria è attualmente utilizzata in Chrome, Chromium, Android ed altri programmi.

3.1 OpenSSL

OpenSSL è una libreria open source, scritta in linguaggio C, che fornisce delle implementazioni di SSL (Secure Socket Layer), TLS (Transport Layer Security) e DTLS (Datagram Transport Layer Security).
La struttura principale, chiamata SSL, può essere creata per ogni connessione SSL dal client o dal server solamente dopo aver creato e configurato il contesto (struttura SSL_CTX), poiché la struttura SSL eredita da essa determinati parametri. Le strutture utili ad immagazzinare dati riguardati la connessione a cui fa riferimento SSL sono le seguenti:

- **Context (SSL_CTX)**
  Struttura che viene istanziata una volta per tutta la durata del programma. Contiene informazioni statiche dell'ambiente, come ad esempio l'insieme dei cipher ammessi, la chiave ed il certificato. Contiene inoltre le configurazioni di default che verranno ereditate dagli oggetti SSL associati a quel context.

- **Method (SSL_METHOD)**
  Si stratta di una struttura dati contenente metodi e funzioni che implementano le varie versioni dei protocolli supportati (SSL, TLS, DTLS).

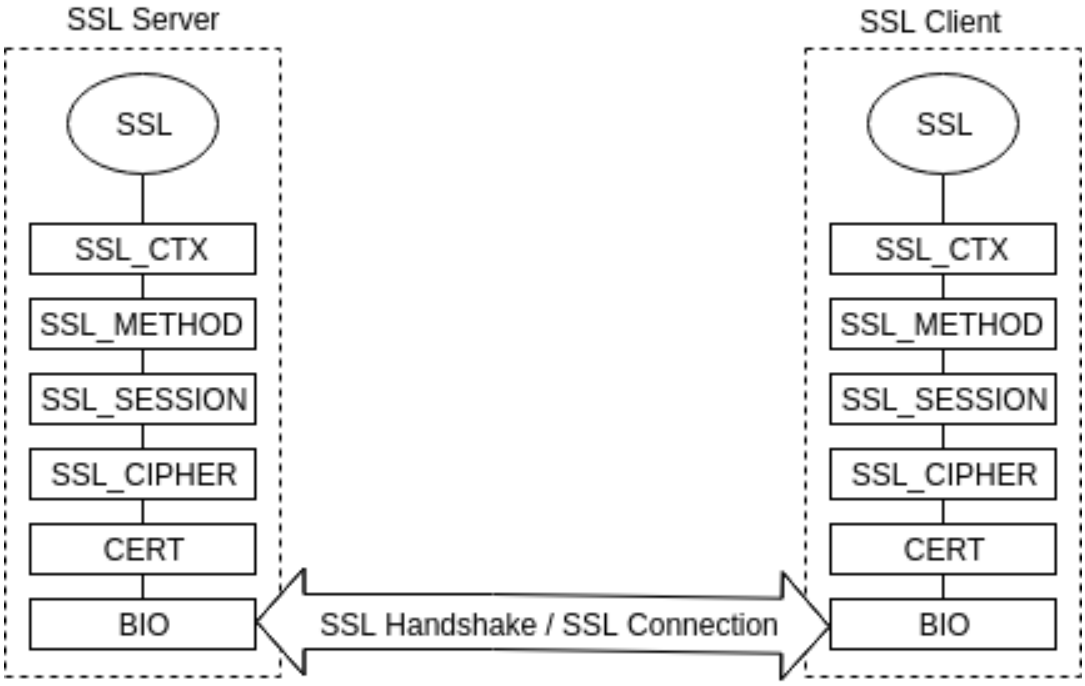

Figura 3.1: Schema Strutture SSL Connection

- **Session (SSL_SESSION)**
  Mantiene dati relativi alle connessioni (sia aperte che chiuse), contiene un set di parametri di sicurezza, può essere condivisa tra più connessioni e permette di limitare il numero di negoziazioni dei parametri di sicurezza poiché esse vengono fatte a livello di Session e non di Connection. Tuttavia, ogni connessione ha una chiave diversa.

- **Cipher (SSL_CIPHER)**
  Contiene le informazioni relative agli algoritmi di cifratura. I cifrari utilizzabili sono

configurati nel Context mentre quelli effettivamente utilizzati vengono memorizzati nella Session.

- **Certificate (CERT)**
  Struttura che mantiene le informazioni estratte dalla struttura X509 relative al certificato.

- **Basic Input Output (BIO)**
  Oggetto che rappresenta input stream ed output stream ed è utilizzato dalle connessioni per leggere e scrivere byte. Più BIO possono essere concatenati e si distinguono in questo caso tre tipi di BIO:

  – **Source BIO**, l'input non viene ricevuto da un altro BIO ma da una sorgente diversa (ad esempio un file).
  – **Filter BIO**, riceve dati in input da un altro BIO, li processa e li passa al BIO successivo.
  – **Sink BIO**, l'output non viene passato ad un altro BIO ma ad una destinazione diversa (ad esempio un file).

Figura 3.2: Schema di uno Stream BIO generico

3.2 OpenSSL vs BoringSSL

Tipicamente con il termine fork si intende la copia di un progetto a cui apportare le proprie modifiche. Invece, il processo che ha portato alla nascita di BoringSSL è iniziato con una directory vuota dove le funzionalità di OpenSSL (v1.0.2) sono state riformattate, pulite (o scartate) e documentate una ad una seguendo un determinato stile prima di esservi inserite. Non c'è garanzia della stabilità di API poiché, come già detto, BoringSSL è nato per un'esigenza dei prodotti Google e non per consumatori esterni.

3.2.1 Funzionalità Rimosse

L'approccio scelto per la rimozione di funzionalità da OpenSSL consiste nella loro inclusione in una sezione separata chiamata "decrepit", non presente in Chrome/Chromium o Android. Le principali funzionalità rimosse sono le seguenti: *Blowfish, Camellia, CMS, compression, Whirpool, IDEA, JPAKE, ENGINE, Kerberos, MD2, MDC2,*

OCSP, PKCS7, RC5, RIPE-MD, SEED, SRP e *timestamping*. OpenSSL contiene un numero elevato di differenti funzioni di inizializzazione utili alla configurazione degli algoritmi e alla gestione degli errori (come *SSL_library_init*, *ERR_load_crypto_strings*, ...).
A differenza di OpenSSL l'inizializzazione di BoringSSL non necessita delle funzioni rimosse, ma esse sono comunque dichiarate e definite, anche se senza corpo, così da mantenere la compatibilità. Anche il meccanismo delle asserzioni è stato rimosso poiché il wrapper *OPENSSL_assert* non aggiunge alcuna funzionalità alla funzione standard assert presente in *assert.h*.

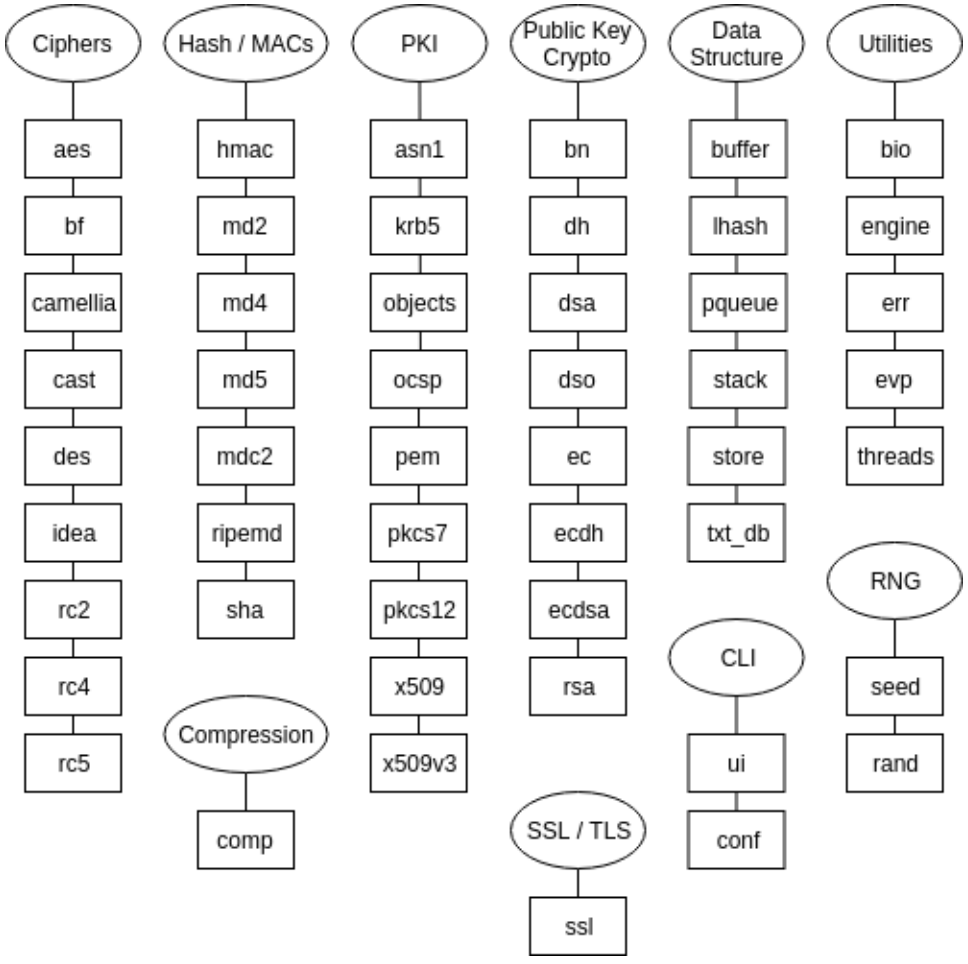

Figura 3.3: Componenti Libreria OpenSSL

3.2.2 Gestione della Memoria

Le funzioni fornite da OpenSSL per la gestione della memoria sono semplici wrappers delle funzioni standard, per questo motivo il loro utilizzo può essere alternato a piacere. In BoringSSL tali funzioni non sono semplici wrappers ma ad esse sono stati aggiunti ulteriori controlli sui parametri passati. Le funzioni utilizzabili sono dunque: *OPENSSL_malloc, OPENSSL_free, OPENSSL_memchr, OPENSSL_memcmp, OPENSSL_memcpy, OPENSSL_memmove, OPENSSL_memset.*

3.2.3 Reference Counter

Le principali strutture contengono un reference counter inizializzato ad uno al momento della creazione, esso viene poi incrementato o decrementato quando si chiamano determinati metodi. Alcuni utilizzatori esterni accedono direttamente ai reference counters tramite le chiamate a *CRYPTO_add* con il relativo *CRYPTO_LOCK_**.
Tale meccanismo, utile alla gestione di alcuni gravi errori, in BoringSSL è stato modificato e permette solamente le specifiche chiamate a funzione del tipo *FOO_up_ref*.

```
// EVP_PKEY_up_ref increments the reference count of |pkey| and returns one. It
// does not mutate |pkey| for thread-safety purposes and may be used
// concurrently.
OPENSSL_EXPORT int EVP_PKEY_up_ref(EVP_PKEY *pkey);
```

Figura 3.4: Esempio di Funzione Reference Counter della struttura EVP

3.2.4 Rinegoziazione TLS

Con rinegoziazione TLS si intende un nuovo handshake che stabilisce nuovi parametri crittografici e può essere iniziata sia dal client che dal server. La presenza di bug e vulnerabilità in questa funzionalità hanno portato alle seguenti modifiche durante l'inserimento di quest'ultima in BoringSSL:

- La rinegoziazione è disabilitata per default, è necessario abilitarla tramite le chiamate a *SSL_set_renegotiate_mode*
- Non c'è supporto per la rinegoziazione da parte del server
- Non c'è supporto per la rinegoziazione in DTLS
- Le chiamate alle funzioni *SSL_renegotiate* e *SSL_set_state* falliscono sempre (funzioni rimosse)

- Il server non può cambiare il proprio certificato durante la rinegoziazione in modo da evitare l'attacco conosciuto come Triple Handshake
- La rinegoziazione non fa parte del processo di Session Resumption: il client non può offrire una sessione durante la rinegoziazione e il server non può ripristinare nessuna sessione stabilità durante una rinegoziazione

3.2.5 Macro CTRL

L'enorme numero di funzioni implementato in OpenSSL attraverso la definizione di macro CTRL è stato convertito in funzioni vere e proprie.

CTRL value
DTLS_CTRL_GET_TIMEOUT
SSL_CTRL_CLEAR_MODE
SSL_CTRL_GET_READ_AHEAD

Funzione convertita
DTLSv1_get_timeout
SSL_CTX_clear_mode
SSL_CTX_get_read_ahead

Figura 3.5: Esempio di conversione da Macro CTRL a funzione

3.2.6 API Aggiuntive

L'uso delle API aggiuntive di BoringSSL pregiudica la compatibilità con OpenSSL e per questo non sono utilizzabili in caso di porting di codice. Ad esempio, per la creazione e la gestione dei messaggi TLS e ASN.1 sono stati aggiunti i due seguenti tipi di bytestrings:

- **CBB (CRYPTO ByteBuilder)**
  Consiste in un buffer che cresce con l'esigenza dell'applicazione e fornisce funzioni di creazione
- **CBS (CRYPTO Bytestring)**
  Rappresenta una stringa di byte in memoria e fornisce funzioni di utility

Un'altra importante funzionalità aggiunta da BoringSSL consiste nella modalità di memorizzazione dei certificati X509. Al posto delle costose strutture classiche utilizzate da OpenSSL si utilizzano delle bytestrings opache chiamate CRYPTO_BUFFER. È possibile inserire un pool di queste bytestrings in un *SSL_CTX* in modo che i certificati non siano duplicati tra varie connessioni e in particolare *SSL_CTX*.

Capitolo 4

DTLS

Datagram Transport Layer Security (DTLS) è un protocollo di sicurezza che, nella pila protocollare, si colloca tra il livello applicazione e il livello di trasporto. Esso è basato sul protocollo stream-oriented TLS, ma a differenza di quest'ultimo permette di avere al di sotto un protocollo di trasporto non affidabile, come UDP. Il suo ruolo è quello di cifrare i messaggi che vengono scambiati durante la comunicazione al fine di proteggerne il contenuto.
La versione attuale più recente di questo protocollo è la 1.2, rilasciata a Gennaio 2012. Questa release viene implementata in molteplici librerie come OpenSSL, BoringSSL, @nodertc/dtls etc...
Sono presenti dei draft della versione 1.3 che porteranno all'aggiunta di nuove funzionalità.

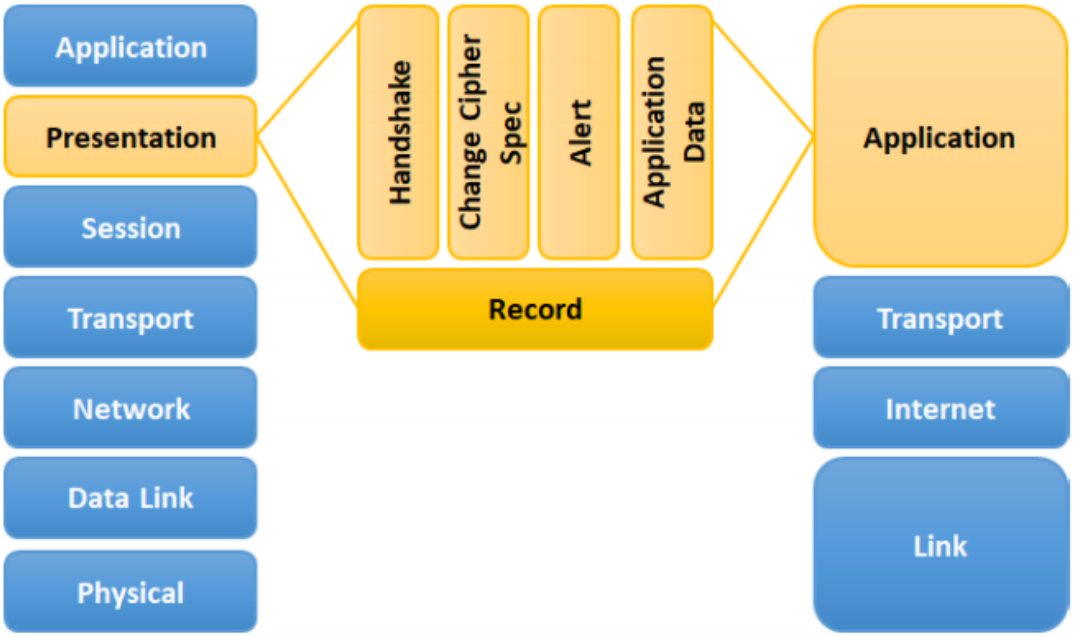

Figura 4.1: Collocamento del protocollo nella pila protocollare: confronto tra ISO/OSI e TCP/IP

4.1 Struttura di DTLS

Il protocollo DTLS è strutturato in quattro sotto protocolli, posti sullo stesso livello, che si occupano di generare i messaggi che poi verranno passati al livello sottostante, denominato Record Layer. Il compito del Record Layer è quello di frammentare, comprimere e criptare i messaggi, per poi inoltrarli al livello sottostante della pila protocollare. Tre dei quattro sotto protocolli vengono utilizzati durante la fase di handshake per consentire al client e al server di:

- Concordare i parametri di sicurezza per il Record Layer
- Autenticarsi
- Istanziare i parametri di sicurezza negoziati
- Segnalare eventuali errori

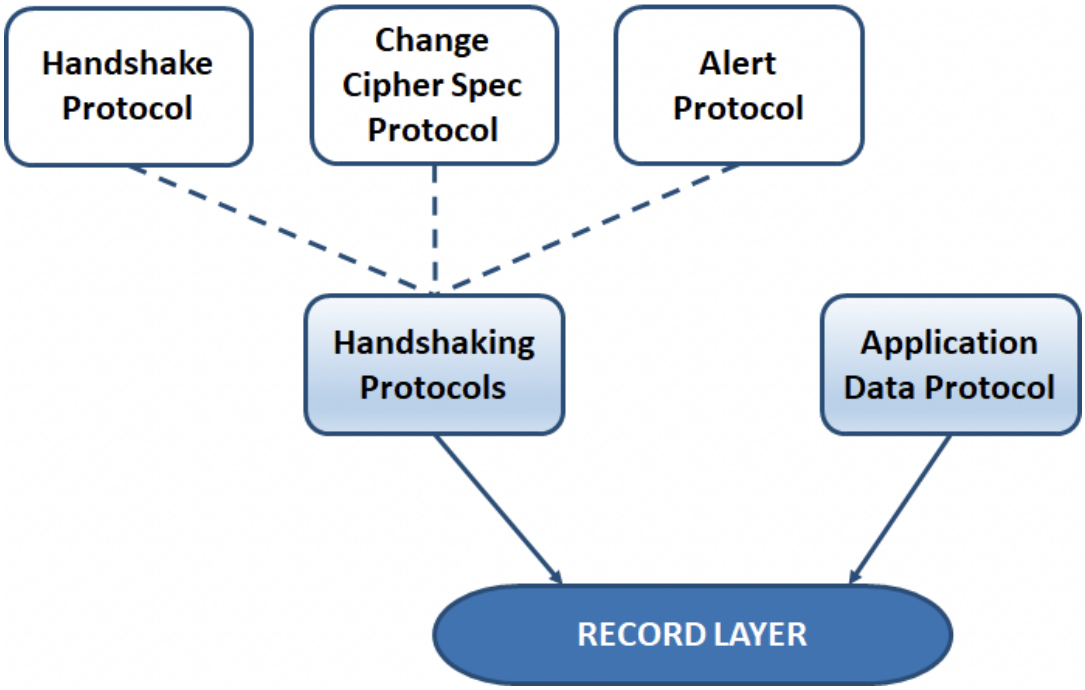

Figura 4.2: Rappresentazione dei protocolli di DTLS e delle relazioni tra essi

4.1.1 Handshake Protocol

Questo protocollo si occupa di negoziare la sessione ed entra in gioco all'inizio della comunicazione tra un client e un server. Mediante questo protocollo, i due sistemi si accordano sulla versione del protocollo DTLS da utilizzare e sugli algoritmi di crittografia; inoltre hanno la possibilità di autenticarsi scambiandosi i certificati.

4.1.2 Change Cipher Spec Protocol

CCSP è un protocollo semplice basato sulla gestione di un singolo messaggio costituito di un solo byte con valore uno, esso viene poi cifrato e compresso seguendo le impostazioni stabilite. Questo messaggio viene inviato sia dal client che dal server e la sua ricezione comporta l'aggiornamento delle informazioni relative ai parametri di cifratura.

4.1.3 Alert Protocol

L'Alert Protocol è utilizzato per trasportare messaggi di allarme, che si distinguono tra fatal e warning. In caso di un messaggio di tipo fatal, la connessione viene chiusa immediatamente e la sessione non è più riutilizzabile. L'alert più importante è il close-

notify, che viene inviato sia dal client che dal server nel momento in cui si vuole chiudere una connessione.

4.1.4 Application Protocol

L'Application Protocol gestisce la comunicazione tra i due sistemi e tutto ciò che ne concerne, dal termine della fase di handshake fino alla fine della comunicazione.

4.2 Handshake e Struttura Pacchetti

DTLS ha un handshake simile a quello di TLS, in quanto i messaggi e il loro flusso risultano essere gli stessi a parte queste tre differenze:

- Aggiunta di cookie stateless nello scambio dei messaggi, in modo da prevenire degli attacchi DoS
- Modifica dell'header dei messaggi di handshake per gestire meglio i messaggi persi, il riordinamento e la frammentazione
- Aggiunta di un timer di ritrasmissione per gestire i messaggi persi

L'handshake in DTLS ha inizio con l'invio di un messaggio da parte del client al server. Questo messaggio, chiamato ClientHello, è così definito:

```
1 struct {
2         ProtocolVersion client_version;
3         Random random;
4         SessionID session_id;
5         opaque cookie<0..32>;
6         CipherSuite cipher_suites<2..2^16-1>;
7         CompressionMethod compression_methods<1..2^8-1>;
8 } ClientHello;
```

Tutti i campi sono completi, tranne quello riguardante i cookie, che risulta vuoto. Il server risponde al client con l'HelloVerifyRequest, che presenta la seguente struttura:

```
1 struct {
2         ProtocolVersion server_version;
3         opaque cookie<0..32>;
4 } HelloVerifyRequest;
```

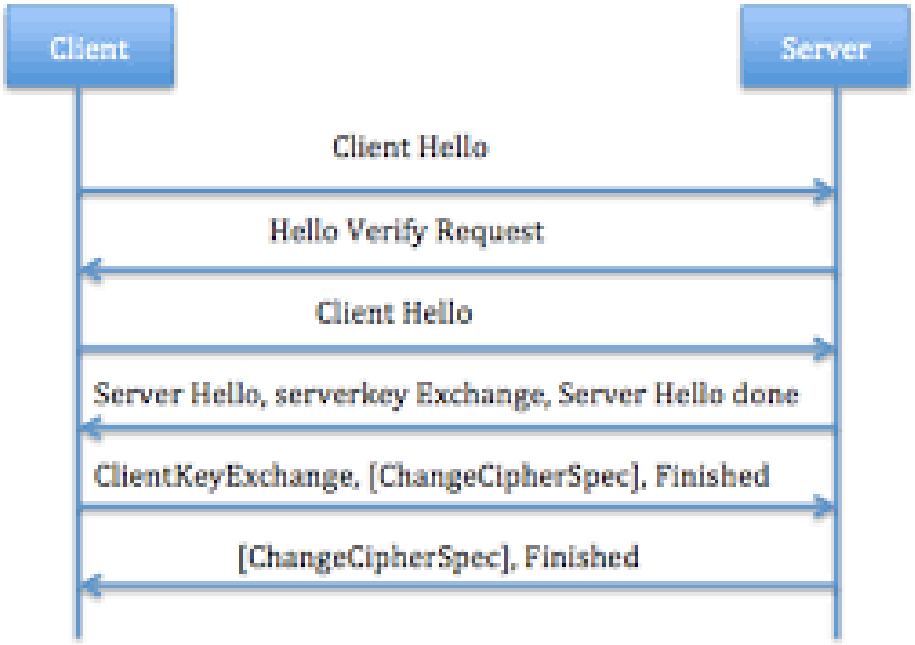

Figura 4.3: Rappresentazione dell'handshake DTLS

Questo messaggio contiene i cookie generati dal server. A questo punto, il client invia al server un ClientHello identico al precedente, al quale sono stati aggiunti i cookie ricevuti dal server.
Il server ora confronta i cookie ricevuti dal client, con quelli da lui precedentemente generati tramite la seguente funzione:

```
1 Cookie = HMAC(Secret, Client-IP, Client-Parameters);
```

In caso positivo server e client, mediante i pacchetti mostrati in figura 4.3, effettuano lo scambio delle rispettive chiavi e certificati. Questo permetterà di stabilire una comunicazione sicura e criptata.

Capitolo 5

Metodi di Resumption

Fast Resumption e Session Resumption hanno lo scopo di ripristinare una connessione precedentemente instaurata e successivamente interrotta fra un client e un server. Gli approcci di Session Resumption e Fast Resumption possono coesistere in un'unica implementazione.
I due metodi non vanno in conflitto perché utilizzano meccanismi completamente differenti: la Session Resumption utilizza come strumento le sessioni (che per essere ripristinate devono essere valide), mentre la Fast Resumption opera a un livello più basso, sfruttando il comportamento delle socket UDP non connesse.
L'indipendenza dei due approcci permette di ottenere un'implementazione che sfrutta i vantaggi forniti da entrambi:

1. La possibilità di gestire in maniera efficiente il cambiamento di indirizzo IP lato client, offerta dalla Fast Resumption.

2. La possibilità di ristabilire una sessione tramite handshake abbreviato, in caso di disconnessione prolungata, messa a disposizione dalla Session Resumption.

La Fast Resumption risulta essere utile solo quando un client cambia IP e continua a comunicare, ma è possibile che il client non riesca ad ottenere il nuovo indirizzo IP in tempi brevi. Se il server non riceve alcun pacchetto entro la scadenza del timeout chiude la comunicazione lanciando la *close-notify* al vecchio IP del client.
In questo scenario la Session Resumption sopperisce ai problemi della Fast Resumption.

Se il client non riesce più a contattare il server può comunque validare la sua sessione lanciando la *close-notify* e riconnettendosi al server tramite la Welcome Socket. In questo caso, grazie alla Session Resumption, il client potrà utilizzare un handshake abbreviato, risparmiando tempo e computazione.

5.1 Session Resumption

La Session Resumption basa la propria implementazione sui Session Ticket. Questi ultimi sono un'estensione nata per TLS, ma pur non essendo ufficializzati in nessun documento o RFC a riguardo, sono presenti anche in DTLS.

5.1.1 Session Ticket

I Session Ticket sono strutture cifrate create dal server per racchiudere le informazioni riguardanti la sessione. Queste strutture vengono memorizzate lato client così da poter essere rinviate al server in caso di ripristino della sessione.
Il server, partendo dal ticket, risale a tutte le informazioni necessarie a riprendere la sessione, ricostruendo quelle mancanti, come ad esempio le chiavi usate e gli algoritmi di compressione.

5.1.2 Utilizzo del Session Ticket

I Session Ticket compaiono durante l'operazione di handshake. Se il client vuole richiedere il ripristino di una nuova sessione, include il relativo ticket nel ClientHello, in caso contrario, invia al server un ticket vuoto. Di seguito saranno mostrate le due richieste possibili che il client può effettuare.

5.1.3 ClientHello con Session Ticket vuoto

Un server che implementa il meccanismo di Resumption mediante Session Ticket, nel momento in cui riceve un ClientHello con Session Ticket vuoto, comprende che l'interlocutore non desidera ripristinare la sessione precedente. È necessario dunque creare una nuova sessione, generare un ticket a partire da essa ed inviarlo al client in modo che possa memorizzarlo per ripristinare in futuro la sessione appena creata.
Di seguito vi è una raffigurazione di un handshake completo tratta dall'RFC 5077. È necessario tenere conto del fatto che, visto che si parla di TLS, l'immagine non rappresenta lo scambio dei cookie, che in DTLS potrebbero essere abilitati. In più non è sempre detto che il client e il server supportino i ticket.

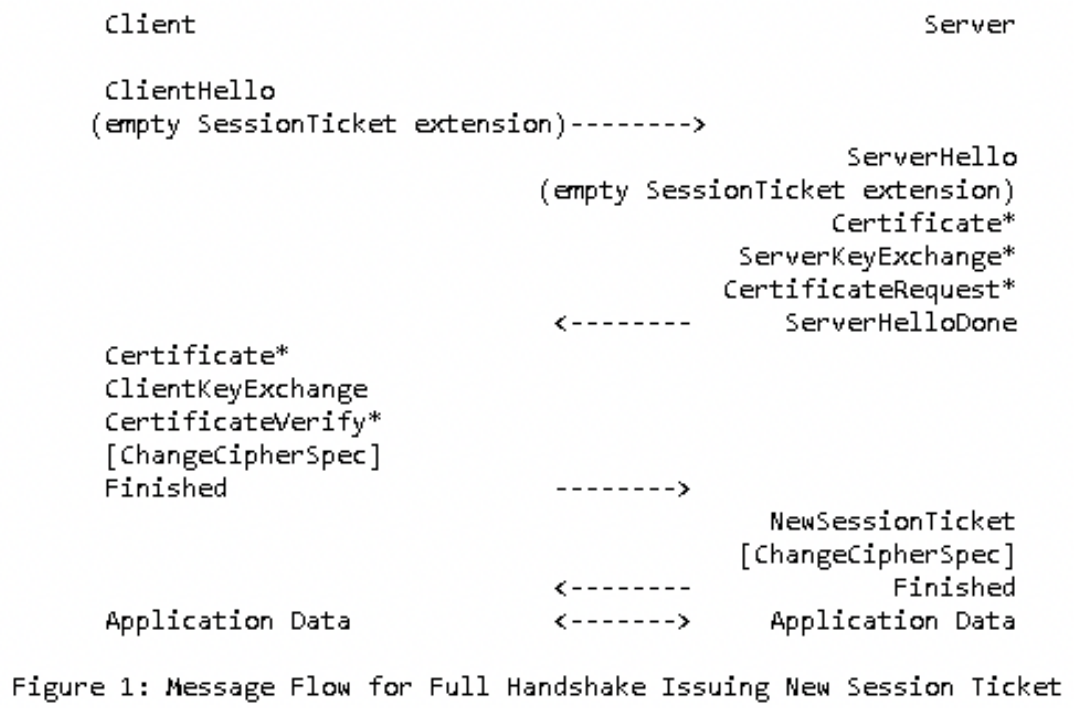

Figure 1: Message Flow for Full Handshake Issuing New Session Ticket

Figura 5.1: Ticket: Schema del caso in cui il client non mandi il ticket e viene effettuato un handshake completo ordinario

1. Il client manda un ClientHello con ticket vuoto
2. Il server risponde a sua volta con un ticket vuoto
3. L'handshake continua e, dopo aver creato una nuova sessione, il server manda al client il ticket relativo
4. Se il client vuole ristabilire la connessione in futuro, memorizza il ticket, pur non comprendendone il contenuto
5. Alla chiusura della connessione, il server dealloca le risorse relative e non salva nulla perché le informazioni necessarie vengono mantenute memorizzate dal client

5.1.4 ClientHello con Session Ticket non vuoto

Il server, dopo aver ricevuto il messaggio ClientHello con il ticket di sessione presente, deve verificare che la sessione che il client vuole ristabilire sia valida. I motivi che possono portare al rifiuto di ripristinare la sessione da parte del server sono i seguenti:

- Il server non implementa l'estensione Session Ticket
- Il server non riesce a interpretare il ticket perché errato
- La sessione non è stata chiusa correttamente, tramite il messaggio *close-notify* dell'Alert Protocol
- Una delle connessioni relative alla sessione che si sta cercando di ristabilire ha generato un errore di tipo fatal
- La sessione che si sta cercando di ripristinare è scaduta

Nel caso in cui il server, a causa di uno dei motivi sopra indicati, non riesca o non voglia ripristinare la sessione, esso potrebbe scegliere di proseguire con un handshake completo ordinario, senza utilizzare l'estensione del Session Ticket. Tale scenario è raffigurato nello schema seguente, tratto dall'RFC 5077.

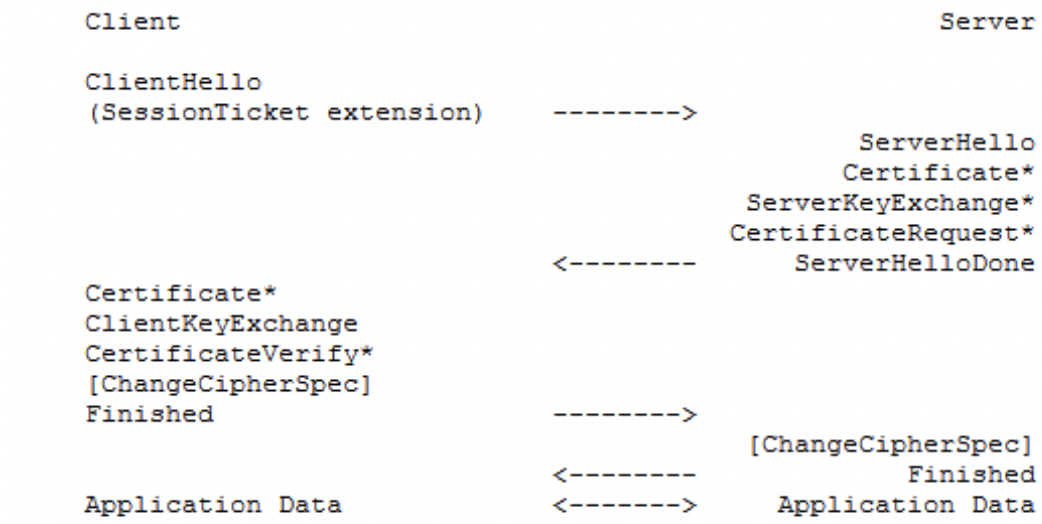

Figure 3: Message Flow for Server Completing Full Handshake Without Issuing New Session Ticket

Figura 5.2: Ticket: Schema del caso in cui il client mandi un ticket, ma il server non ripristina una sessione precedente

1. Il client manda un ClientHello con ticket non vuoto
2. Il server decide di non proseguire con i ticket e continua con un normale handshake che non utilizza tale estensione

3. L'handshake prosegue e il server non invia nessun ticket al client, che quindi non può memorizzarlo per un futuro ripristino della sessione

Se, invece, il server accetta la richiesta del Client di ripristinare la sessione precedente, esso risponde al Client con un Session Ticket vuoto.
Il meccanismo è dettagliato nella figura seguente:

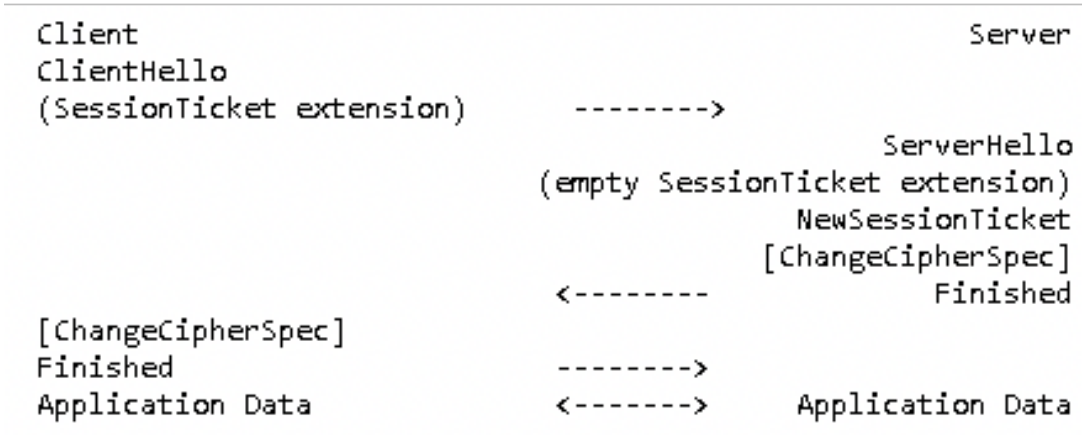

Figura 5.3: Ticket: Schema del caso in cui il client mandi un ticket non nullo e, quindi, voglia ristabilire la sessione precedente

1. Il Client manda il suo ClientHello con un ticket (che supponiamo valido)
2. Il server verifica il ticket e, se è valido, risponde con un ticket vuoto, per fargli capire che il ticket era valido e che si darà inizio a un handshake abbreviato

5.1.5 Risultato

Questa tecnica, illustrata nei paragrafi precedenti, permette di continuare una comunicazione interrotta ristabilendola attraverso un handshake abbreviato. Quest'ultimo serve solo per assicurarsi di ripristinare i parametri concordati in precedenza senza doverli nuovamente negoziare.
Nonostante non si è del tutto esenti da tempo di computazione e delay, in questo modo si ha un approccio molto più leggero rispetto al classico full handshake.

5.2 Fast Resumption

La Fast Resumption si basa principalmente sull'utilizzo di socket non connesse lato server.

5.2.1 Socket UDP

Prima di procedere con la spiegazione del metodo di Resumption, è necessario introdurre alcune nozioni riguardanti le socket UDP, al fine di facilitare la comprensione dei paragrafi successivi. Le informazioni da conoscere sono le seguenti:

- Più socket possono essere bindate sulla stessa porta della stessa interfaccia
- Le socket possono essere connesse o non connesse. Nel primo caso la socket riceve solo i messaggi provenienti dall'indirizzo al quale è connessa. Nel secondo caso, invece, la socket può ricevere messaggi da chiunque
- Una socket UDP può ricevere messaggi ICMP solo se è connessa
- Se su una porta è presente una sola socket e questa non è connessa, essa riceverà tutto il traffico indirizzato a quella porta, indipendentemente dal mittente
- Se sulla stessa porta sono bindate due socket, una connessa all'indirizzo X e l'altra no, il kernel indirizzerà i messaggi provenienti dall'indirizzo X sulla socket connessa e tutti gli altri sulla socket non connessa
- Se su una porta sono bindate N socket connesse e solo una non connessa, il comportamento è analogo: ogni socket connessa riceverà i messaggi provenienti dall'indirizzo a cui è connessa, mentre la socket non connessa riceverà i restanti
- Quando si hanno N socket non connesse bindate sulla stessa porta, solo una delle N socket riceve i messaggi che non sono di competenza di una socket connessa. La scelta di tale socket dipende dall'implementazione del sistema operativo; di norma si tratta o della prima ad essere stata bindata o dell'ultima

5.2.2 Utilizzo della Socket Connessa Temporanea (SCT)

L'approccio della Fast Resumption consiste nel creare una socket temporanea lato server, bindata sulla stessa porta della Welcome Socket e connessa al Client, con lo scopo di utilizzarla per portare a termine l'handshake. Una volta completato l'handshake, il server creerà un'altra socket non connessa, bindata su una porta diversa che gestirà il resto della comunicazione. Le dinamiche di questo meccanismo sono rappresentate nello schema in figura:

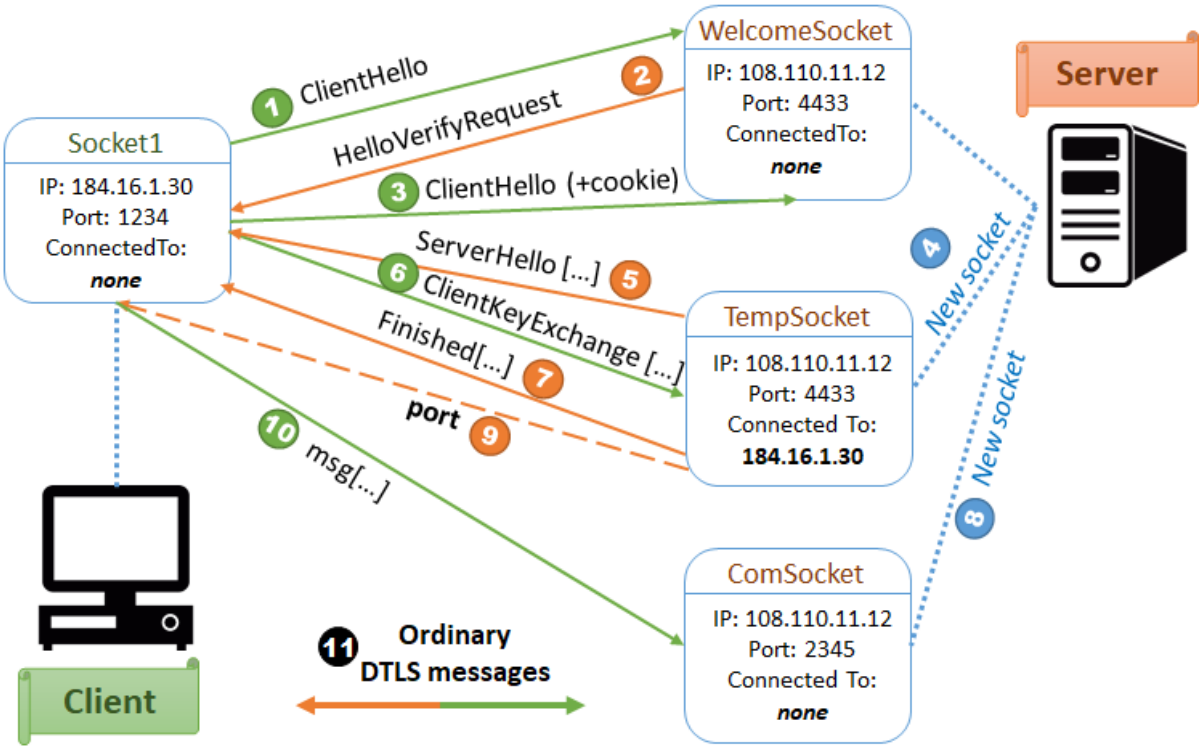

Figura 5.4: Si utilizza una socket temporanea, connessa al client, per terminare l'handshake

1. Il client contatta il server inviando un ClientHello all'indirizzo 108.110.11.12, sulla porta 4433 dove è in ascolto la Welcome Socket

2. Il Server risponde con un HelloVerifyRequest, contenente un cookie

3. Il Client manda un nuovo ClientHello, contenente il cookie ricevuto

4. Il Server crea una nuova socket, bindata sulla stessa porta della Welcome Socket (4433) e connessa al Client (184.16.1.30:1234)

5. Il Server invia il ServerHello al client dalla nuova socket. Così facendo lascia libera la Welcome Socket. In questo modo il problema del NAT non si presenta perché, essendo la nuova socket bindata sulla stessa porta della Welcome Socket, l'indirizzo e la porta del mittente viene riconosciuto

6. Il Client continua l'handshake, inviando i messaggi al solito indirizzo (184.16.1.30:1234). I messaggi vengono ricevuti dalla nuova socket, poiché essa è connessa all'indirizzo del Client.

7. Il Server conclude l'handshake dalla socket temporanea Il Server crea una nuova socket allo scopo di utilizzarla per gestire la comunicazione con quel Client. Quest'azione è necessaria perché la socket attuale è connessa all'indirizzo del Client, di conseguenza se quest'ultimo cambiasse indirizzo IP, i messaggi da esso provenienti verrebbero ricevuti dalla Welcome Socket, la quale li scarterebbe perché si aspetta solo dei ClientHello. La nuova socket, pertanto, non deve essere connessa ma deve essere bindata su una porta diversa

8. Il Server, come primo messaggio DTLS, invia la porta alla quale vuole essere contattato

9. La connessione è stabilita e i due sistemi possono iniziare a scambiarsi normalmente i dati criptati. In questo momento il canale DTLS stabilito tra i due interlocutori è resistente al cambiamento di indirizzo IP, sia lato client che lato server

5.2.3 Pregi e Difetti

Questa tecnica vanta molteplici pregi. In primo luogo, non si ha il problema del Port Restricted NAT in quanto si alloca una nuova socket solo dopo aver verificato l'identità del Client. In questo modo si è protetti dall'IP Spoofing e non vengono modificati messaggi propri del protocollo. Inoltre, questa soluzione non espone la porta in chiaro permettendo al server di essere meno vulnerabile. Un punto negativo però è il fatto che questa tecnica risulta piuttosto pesante, questo perché per ogni client dovranno essere create due socket, una delle quali verrà poi distrutta.

Capitolo 6

Strumenti e Configurazione

In questo capitolo andremo ad illustrare quelli che sono i due principali strumenti utilizzati durante lo svolgimento di questa tesi, in particolare nelle fasi relative al testing delle applicazioni.

6.1 Oracle VM VirtualBox

Oracle VM VirtualBox è un software gratuito e Open Source per l'esecuzione di macchine virtuali (con una versione ridotta distribuita secondo i termini della GNU General Public License) per architettura x86 e 64bit che supporta Windows, GNU/Linux e Mac OS come sistemi operativi host, ed è in grado di eseguire Windows, GNU/Linux, OS/2 Warp, BSD e infine Solaris e OpenSolaris come sistemi operativi guest.
VirtualBox è risultato essere uno strumento essenziale nello svolgimento della tesi questo perché, come descritto nel Capitolo 9, la libreria necessita di una particolare versione di Linux per essere correttamente compilata. Questo ha portato alla scelta di virtualizzare il sistema operativo in questione senza doverlo per forza installare su un disco fisico.
La scelta del sistema di virtualizzazione è ricaduta su VirtualBox in quanto è un sistema gratuito e Open Source adatto a fornire un'ottima virtualizzazione di svariati sistemi operativi. Oltretutto presenta un'interfaccia utente estremamente semplice ed intuitiva.

6.1.1 Virtualizzazione

Per quanto riguarda la virtualizzazione del sistema operativo non è stato fatto nulla di particolare. Qui di seguito andremo ad elencare una breve lista di caratteristiche che le macchine da noi create presentano.

- **Sistema Operativo**, Ubuntu
- **Memoria RAM**, Almeno 3 GB
- **Tipo di Archiviazione**, Disco virtuale a dimensione fissa
- **Dimensione del Disco** Almeno 20 GB

Ricordiamo inoltre che se si utilizzano degli applicativi che catturano dati audio e video da dispositivi hardware (built-in o meno) sarà necessario consentire alle macchine virtuali di averne accesso tramite le impostazioni di VirtualBox.

6.1.2 Modalità di Networking

Qui di seguito andremo a descrivere quali sono le diverse modalità di rete che VirtualBox mette a disposizione.

- **Not Attached**
  In questa modalità una scheda di rete virtuale è abilitata per ogni macchina virtuale, ma viene emulato un cavo di rete non collegato. Di conseguenza le macchine virtuali non hanno alcuna connessione alla rete.
- **NAT**
  In questa modalità, ad ogni VM (Virtual Machine) viene assegnato lo stesso indirizzo IP (10.0.2.15) perché ogni VM è su una rete isolata. Quando invia traffico attraverso il gateway (10.0.2.2) VirtualBox riscrive i pacchetti per farli apparire come se fossero originati dall' Host, piuttosto che dal guest (in esecuzione all'interno dell'Host). Ciò significa che il client funzionerà anche se l'host si sposta da una rete all'altra (ad esempio, computer portatile in movimento da un luogo all'altro). Questa è la modalità predefinita per le nuove VM ed è da preferire nella maggior parte delle situazioni in cui il guest è di tipo "client" (nel senso che il guest fa generalmente richieste in uscita e non in entrata). Per questo motivo i server che eseguono su VM che necessitano di connessioni dall'esterno preferiscono una diversa modalità di rete.
- **Bridged Network**
  Bridged Networking viene utilizzato quando si desidera che la VM debba essere un

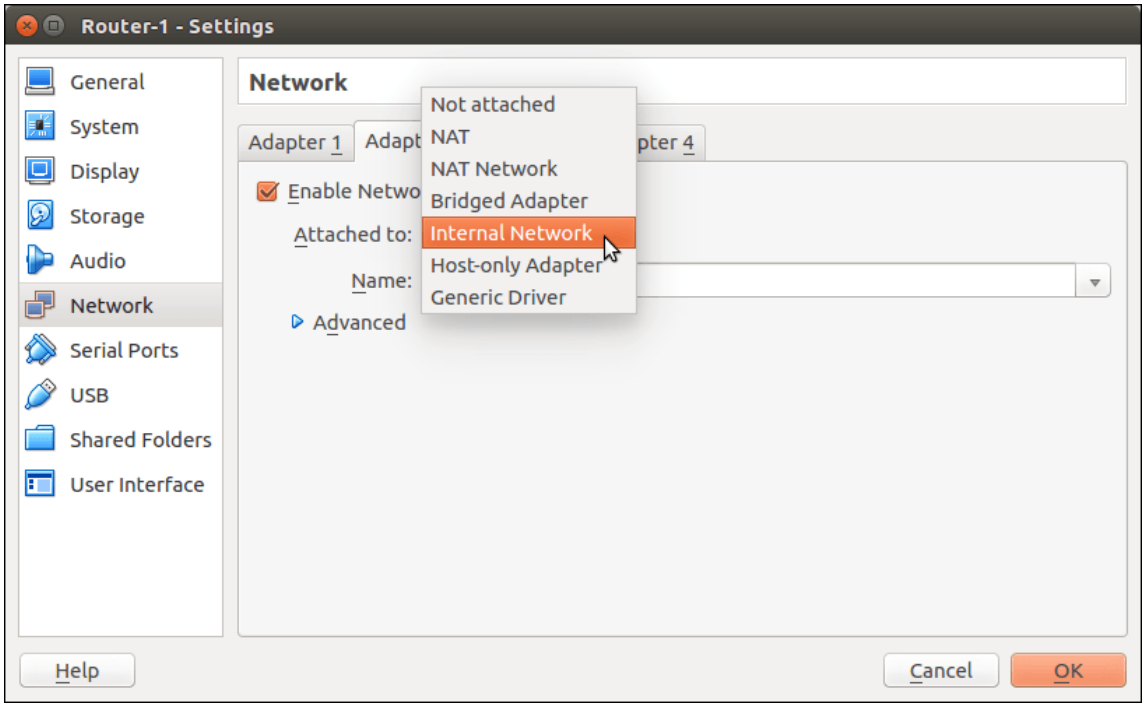

Figura 6.1: Set Network Adapter

componente della rete a tutti gli effetti, ovvero debba essere esposto sulla rete come l'host o come un qualsiasi altro PC sulla rete. In questa modalità, una scheda di rete virtuale è il "ponte" per la scheda di rete fisica sul vostro host. Grazie a questo, ogni VM ha accesso alla rete fisica nello stesso modo dell'host. Si può accedere a qualsiasi servizio sulla rete come, ad esempio, i servizi DHCP, servizi di ricerca, le informazioni di routing allo stesso modo in cui lo fa l'host fa.
L'aspetto negativo di questa modalità è che se si eseguono molte VM si può restare rapidamente a corto di indirizzi IP. Inoltre, se l'host dispone di più schede di rete fisiche (ad esempio, wireless e via cavo), oppure se ci si sposta da una rete all'altra (nel caso di portatili è una cosa frequente) sarà necessario riconfigurare le VM.
Quindi, la rete è simile a:

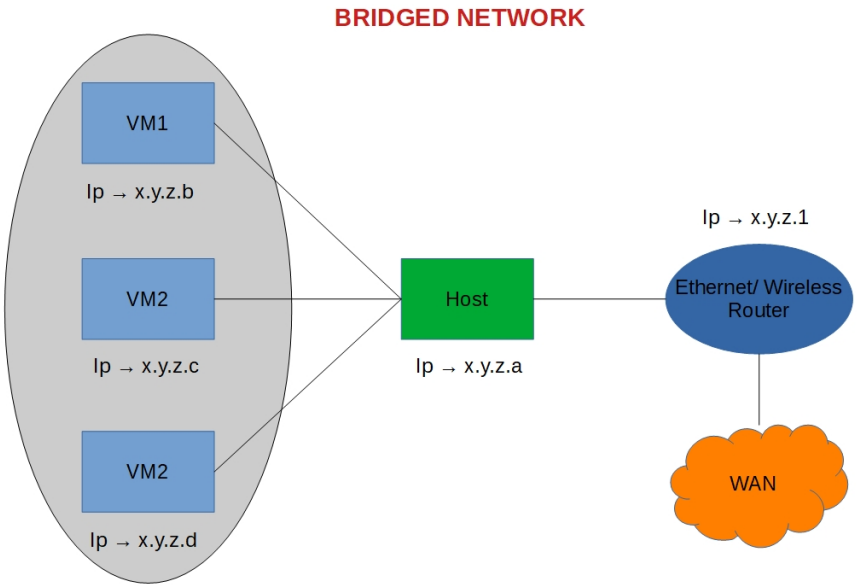

Figura 6.2: Bridged Network

- **Internal Network**
  Quando si configurano una o più macchine virtuali per stare su una rete interna, VirtualBox fa sì che tutto il traffico su tale rete sia visibile alle sole VM su quella rete virtuale.
  La rete interna è una rete totalmente isolata. Questa è una buona soluzione per i test quando si ha bisogno di creare sofisticate reti interne con VM che forniscono i loro servizi alla sola rete interna (ad esempio di Active Directory, DHCP, ecc.) Da notare che nemmeno il Guest è un membro della rete interna, ma questa modalità permette alle VM di funzionare anche quando l'host non è connesso a una rete.

  Si noti che in questa modalità, VirtualBox non fornisce alcun servizio (es DHCP), e quindi le VM devono essere configurato staticamente oppure una delle VM deve gestire un server DHCP. È possibile creare/gestire diverse reti interne ed è possibile configurare le VM per avere più schede di rete di cui una interna ed una di tipologia diversa.

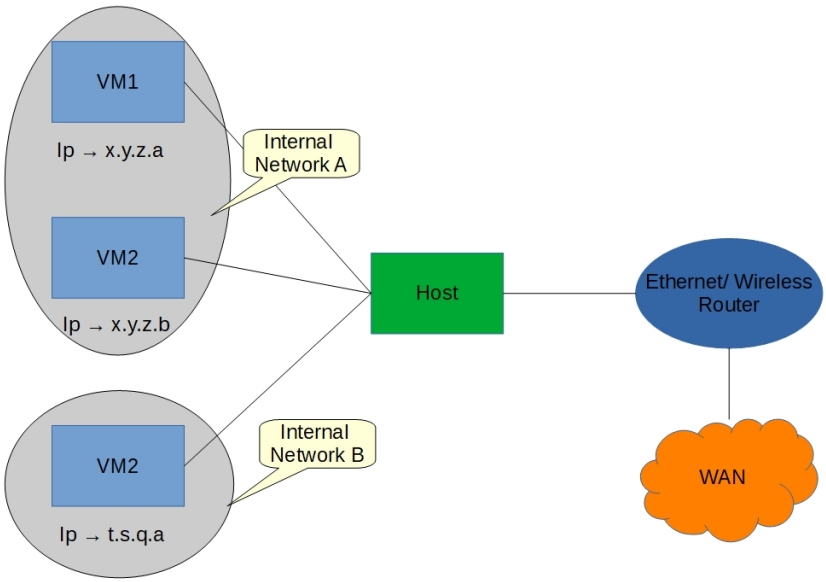

Figura 6.3: Internal Network

6.2 Wireshark

Wireshark (precedentemente chiamato Ethereal) è un software per analisi di protocollo o "packet sniffer" utilizzato per la soluzione di problemi di rete, per l'analisi e lo sviluppo di protocolli o di software di comunicazione e per la didattica. Inoltre, possiede tutte le caratteristiche di un analizzatore di protocollo standard.
Wireshark è distribuito sotto una licenza Open Source; gira sulla maggior parte dei sistemi Unix e compatibili (inclusi GNU/Linux, Sun Solaris, FreeBSD, NetBSD, OpenBSD e Mac OS) e sui sistemi Microsoft Windows appoggiandosi al toolkit di grafica multipiattaforma Qt. Wireshark riesce a "comprendere" la struttura di diversi protocolli di rete, è in grado di individuare eventuali incapsulamenti, riconosce i singoli campi e permette di interpretarne il significato. Per la cattura dei pacchetti Wireshark non dispone di proprio codice, ma utilizza libpcap/WinPcap, quindi può funzionare solo su reti supportate da libpcap o WinPcap. Wireshark è risultato essere uno strumento estremamente utile durante lo svolgimento di questa tesi. Grazie ad esso è stato possibile verificare il comportamento ed il giusto o meno funzionamento dei vari applicativi.

Capitolo 7

Livelli Architetturali

In questo breve capitolo andremo a definire i diversi livelli architetturali che caratterizzano la struttura di WebRTC. Per livelli architetturali non si intende qualcosa di definito dalle specifiche di WebRTC, ma bensì i diversi livelli che, durante il corso del nostro studio, abbiamo definito e approcciato. Anche se tutto ciò verrà poi approfondito successivamente si è ritenuto doveroso dare una panoramica iniziale per consentire al lettore di comprendere meglio le scelte fatte e il flusso di lavoro. Come risulterà poi dai capitoli successivi, durante il corso della nostra ricerca ci siamo addentrati sempre di più nei meandri di questa tecnologia a causa di problemi riscontrati o a causa limiti dei livelli architetturali affrontati. Qui di seguito sono riportati i vari livelli architetturali e le loro caratteristiche.

7.1 Architettura Alto Livello

Per architettura ad alto livello si intende tutto ciò che riguarda il modo più semplice per accedere alle funzionalità di WebRTC.
Essendo WebRTC studiato per facilitare le comunicazioni peer-to-peer tra browser senza l'ausilio di alcun plugin, il primo punto di accesso per lo studio di questa tecnologia è il JavaScript. WebRTC viene maggiormente sfruttato attraverso le API che mette a disposizione, infatti risultano essere le uniche API che presentano una documentazione ufficiale.

Di seguito sono riportate le API di alto livello che WebRTC espone per svolgere le funzionalità principali definite nel Capitolo 2.3:

- MediaStream
- RTCPeerConnection
- RTCDataChannel

7.2 Architettura Medio Livello

In questo livello ci ritroviamo all'interno della libreria e siamo in grado di utilizzare le API, analoghe a quelle JavaScript, che consento di svolgere essenzialmente le stesse cose ma astraendoci dal mondo del Web.
In sostanza, a questo livello architetturale, è possibile generare degli applicativi slegati dal browser che sfruttino a pieno le potenzialità della comunicazione real-time che WebRTC mette a disposizione.
Qui di seguito come fatto per il JavaScript andiamo a riportare le interfacce di medio livello che WebRTC espone:

- MediaStreamInterface
- PeerConnectionInterface
- DataChannelInterface

Il Livello Intermedio verrà trattato successivamente in quanto si discosta lievemente dal Medio Livello in questo momento non risulta essere rilevante. Verrà trattato poi nel Capitolo 11.

7.3 Architettura Basso Livello

Nel caso dell'Architettura a Basso Livello invece si intende l'utilizzo di quelle tecnologie che WebRTC stesso sfrutta per dare vita alle API viste qui sopra. Infatti, a basso livello non troviamo le solite API, ma funzionalità di WebRTC di basso livello e librerie di terze parti, come ad esempio BoringSSL, che vengono utilizzate per far funzionare la libreria.

7.4 Approccio ai Vari Livelli

L'idea iniziale era quella di riuscire ad insediarci all'interno di un livello architetturale, applicare le nostre modifiche per sviluppare le tecniche di Resumption e renderle disponibili così ai livelli superiori.

Nella seguente immagine verranno illustrati i vari livelli e dove i vari approccio si sarebbero collocati. Ovviamente per raggiungere l'obiettivo dello studio non è necessario modificare tutti e tre i livelli, bensì uno solo.

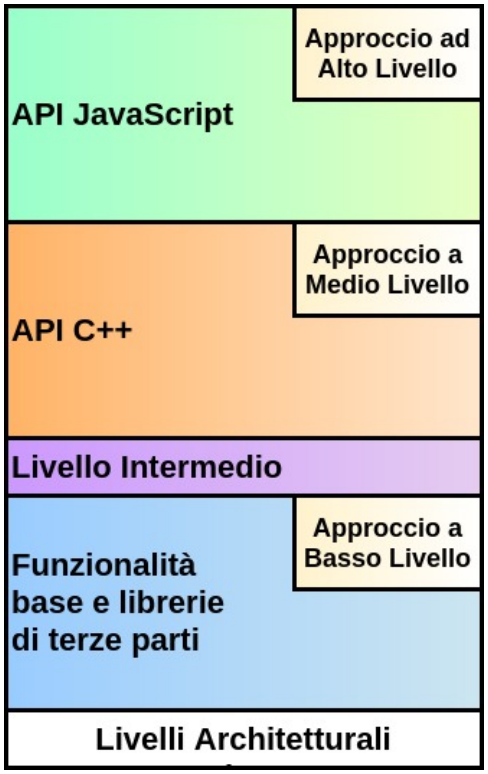

Figura 7.1: Approccio Ideale

Questo approccio ideale, come risulterà dai capitoli successivi, non verrà realizzato per diverse ragioni. La mancanza di documentazione adeguata e la grande complessità della libreria hanno fatto si che nostro lavoro si è spostato verso la costruzione di un'applicazione che, ad un determinato livello della libreria, implementi i metodi di Resumption senza intaccare quelli superiori.

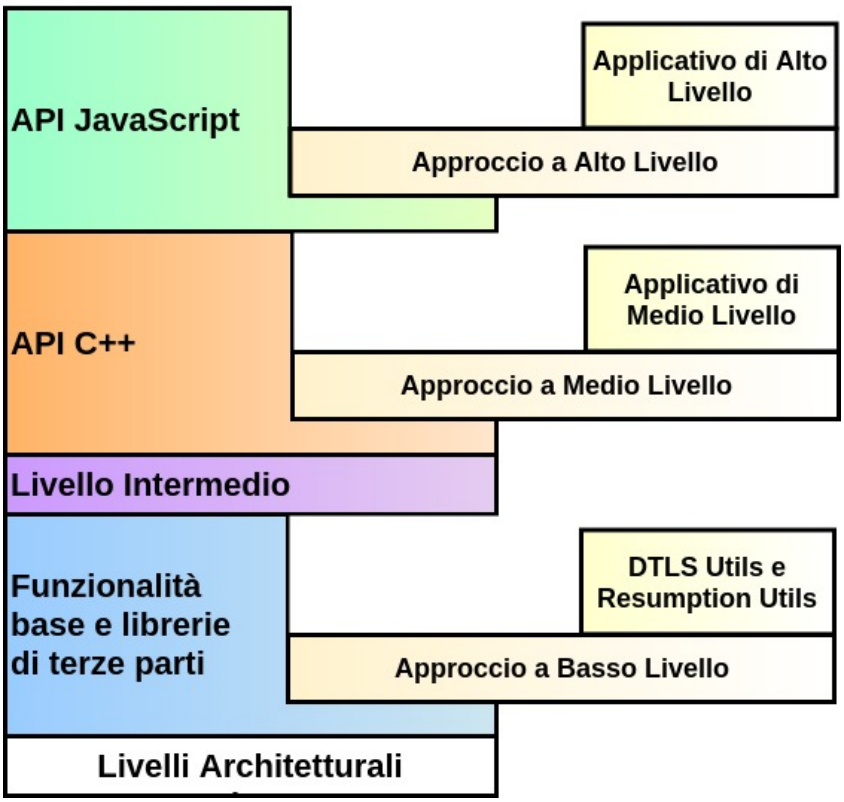

Figura 7.2: Approccio Reale

Capitolo 8

Approccio ad Alto Livello

Considerate le tecnologie descritte nei capitoli precedenti, si tratta ora di come si è evoluto il primo tentativo di realizzazione e studio di un'applicazione utile alla comunicazione real-time tra due peer e dei risultati che questo ha portato.
Il lavoro descritto nei paragrafi seguenti si è definito approccio ad alto livello perché, riferendosi allo stack delle API di WebRTC già descritto, ci si allontana dalla reale implementazione delle librerie native dei browser e si utilizza un insieme "limitato" di WebAPIs. Queste permettono agli sviluppatori di applicazioni web di utilizzare le potenzialità di WebRTC senza occuparsi di vari aspetti e problematiche che riguardano i livelli più bassi.

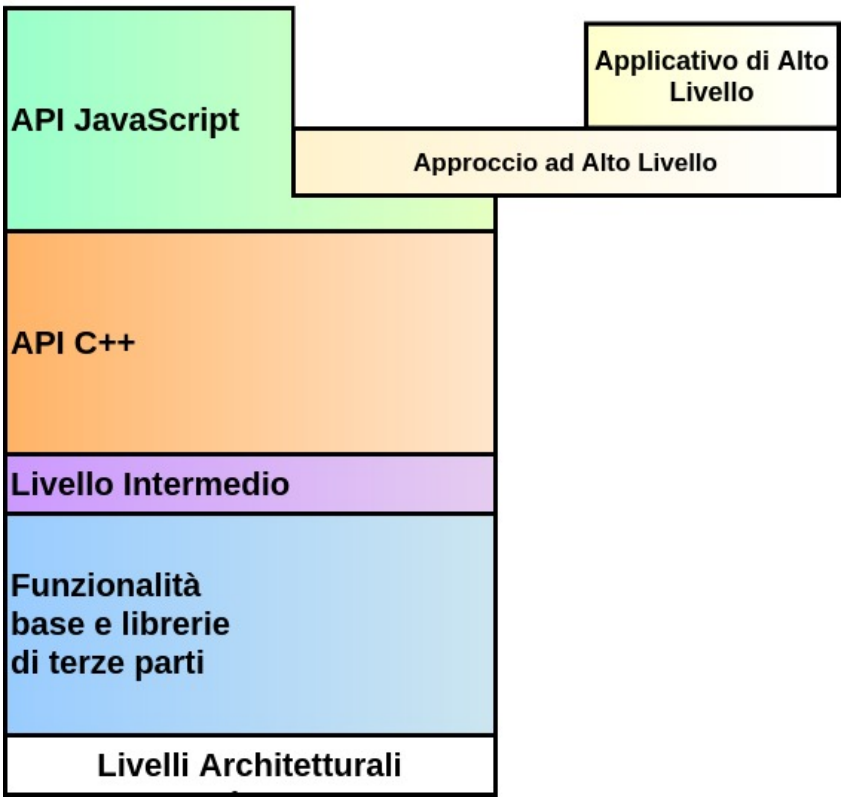

Figura 8.1: Livelli Architetturali - Alto Livello

8.1 WebAPIs

Creato nel 2011, il Web Real-Time Communications Working Group si è occupato di definire e standardizzare le WebAPIs lato client in linguaggio JavaScript (EICMAScript). Le funzioni JavaScript corrispondenti ai tre concetti principali su cui sono basate le comunicazioni in WebRTC sono:

- **MediaStream**
  Rappresenta uno o più stream audio/video gestendo l'accesso alle sorgenti di dati multimediali e la loro cattura tramite essi. Un MediaStream può essere:
  - **locale**, stream di output proveniente da una sorgente dell'user-agent "chiamante".
  - **remoto**, stream di input proveniente dall'user-agent "chiamato".
- **RTCPeerConnection**
  Rappresenta il canale diretto fra i peer che necessitano di comunicare tramite lo scambio di dati audio/video

- **RTCDataChannel**
  Rappresenta il canale diretto fra i peer che necessitano di comunicare tramite lo scambio di dati generici

In una raffigurazione strutturata delle WebAPIs troviamo dei blocchi che riferiscono ai concetti sopra citati e altri che si riferiscono ad ulteriori funzionalità di utility:

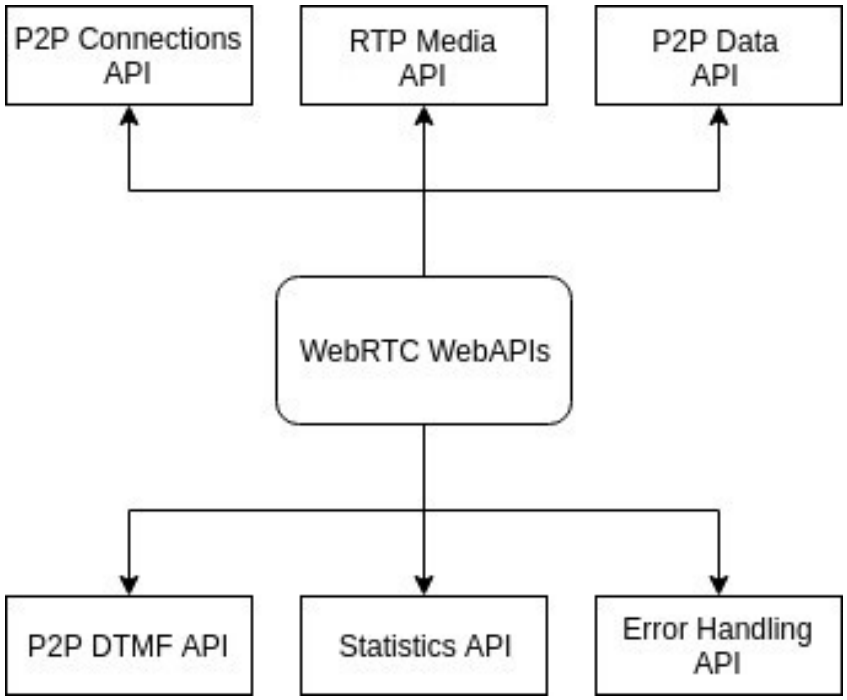

Figura 8.2: Struttura WebRTC WebAPIs

- **P2P Connections API**
  Fornisce funzioni utili alla configurazione del protocollo ICE, alla gestione delle identità dei peer, alla gestione del protocollo SDP, alla gestione di più flussi di dati multimediali tramite il framework di priorità Quality of Service e a tutti gli aspetti riguardanti la creazione, mantenimento e chiusura del canale di comunicazione

- **RTP Media API**
  Tali WebAPIs permettono ad una applicazione web di agganciare uno o più stream di dati multimediali ad un canale precedentemente creato tramite RTCPeerConnection e di gestire tutti gli aspetti specifici alle modalità di comunicazione quali frame rate, volume, sample size

- **P2P Data API**
  Permette di creare un canale di comunicazione partendo da un RTCPeerConnection dove i dati scambiati sono generici e non puramente audio/video

- **P2P DTMF API**
  Fornisce un insieme di funzioni per inviare segnali DTMF attraverso il canale di comunicazione. Con DTMF (Dual-Tone Multi-Frequency) si intende un sistema di codifica utilizzato nella telefonia per codificare codici numerici sotto forma di segnali sonori in banda audio

- **Statistics API**
  Danno la possibilità di monitorare determinati oggetti al fine di ottenere statistiche sul comportamento dell'applicazione. Ogni selettore, ovvero l'oggetto da monitorare, fornisce la funzione getStats la quale ritorna un insieme di statistiche rilevanti allo specifico oggetto

- **Error Handling API**
  Gli errori sono definiti attraverso un'interfaccia apposita nelle WebAPIs chiamata RTCError la quale estende da DOMException. Il tipo specifico di errore che si verifica viene descritto da una serie di campi e valori che identifica una determinata situazione in punto preciso della connessione, come ad esempio dtls-failure, sdp-syntax-error, hardware-encoder-error, ...

Nonostante siano standard, fra le Web APIs fornite dai browser delle varie aziende facenti parte del W3C esistono alcune differenze che rendono un porting diretto impossibile. Per garantire la portabilità di codice fra browser differenti è stata realizzata una libreria chiamata *Adapter.js* la quale fa uso di shim e polyfill, ovvero codice scaricabile che fornisce dei servizi che non fanno parte di un browser Web.

W3C Standard	Chrome	Firefox
getUserMedia	webkitGetUserMedia	mozGetUserMedia
RTCPeerConnection	webkitRTCPeerconnection	RTCPeerConnection
RTCSessionDescription	RTCSessionDescription	RTCSessionDescription
RTCIceCandidate	RTCIceCandidate	RTCIceCandidate

Figura 8.3: Differenza fra le principali funzioni WebRTC

Dato l'obbiettivo di questo studio si è deciso di non approfondire le WebAPIs riguardanti la manipolazione degli stream dati in modo da focalizzarsi sugli aspetti di connessione e comunicazione che rientrano in P2P Connections API e P2P Data API.

8.1.1 RTCPeerConnection

La responsabilità del funzionamento dell'intero ciclo di vita di una comunicazione peer-to-peer è affidata all'interfaccia RTCPeerConnection la quale incapsula al suo interno tutte le impostazioni, stati e metodi di gestione dell'intera connessione:

- **createOffer**
  Genera un blob SDP contenente la configurazione supportata per la sessione, la descrizione dei MediaStream locali che riferiscono ad un preciso *RTCPeerConnection*, le codifiche supportate ed i parametri DTLS e ICE

- **createAnswer**
  Genera un blob SDP allo stesso modo della *createOffer*

- **setLocalDescription/getLocalDescription**
  Imposta/ritorna i parametri locali della connessione precedentemente negoziati

- **setRemoteDescription/getRemoteDescription**
  Imposta/ritorna i parametri remoti della connessione precedentemente negoziati

- **addStream/removeStream**
  Aggiunge/rimuove un *MediaStream* dalla connessione

- **setConfiguration/getConfiguration**
  Aggiorna/ritorna la configurazione dell'oggetto *RTCPeerConnection* (setConfiguration in passato si chiamava updateIce)

- **addIceCandidate**
  Fornisce un candidato remoto all'ICE Agent

8.1.2 RTCDataChannel

Un *RTCDataChannel*, come già detto, permette lo scambio bidirezionale di dati generici tra due peer emulando il funzionamento delle WebSocket, infatti consente anche di configurare le proprietà della connessione come affidabilità e ordinamento dei pacchetti. Il protocollo applicativo su cui si basa la comunicazione creata tramite RTCDataChannel è SCTP (Stream Control Transmission Protocol) che fa utilizzo di un canale sicuro DTLS.

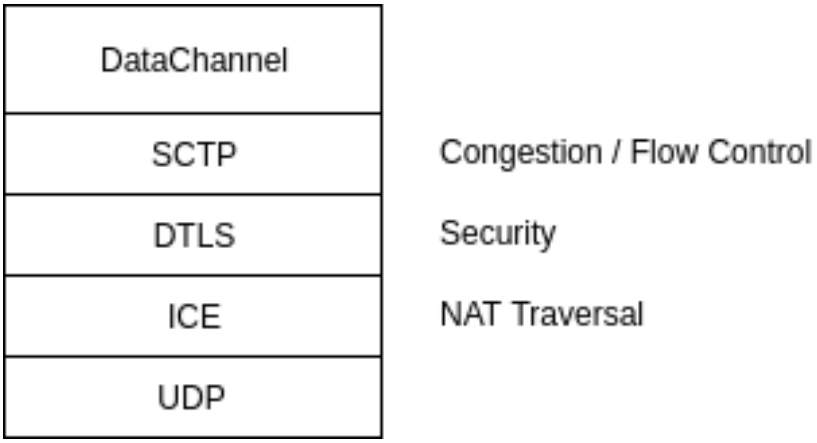

Figura 8.4: Stack Protocolli DataChannel

Il metodo send, fornito dall'oggetto *RTCDataChannel*, permette di inviare i seguenti tipi di dati:

- **String**
  I dati inviati consistono in un buffer di byte codificati in UTF-8
- **Blob**
  I dati inviati consistono in un oggetto di byte codificati come un file. Il Blob risulta preferibile quando non si devono modificare o spezzare in parti più piccole i dati da inviare
- **ArrayBuffer**
  I dati inviati consistono in un oggetto simile al Blob ma preferibile quando si devono effettuare elaborazioni su di essi

8.1.3 RTCCertificate

Il meccanismo utilizzato da una RTCPeerConnection (e quindi anche da una *RTCDataChannel*) per autenticare i peer fa uso degli *RTCCertificate*.

```
[Exposed=Window, Serializable]
interface RTCCertificate {
  readonly attribute DOMTimeStamp expires;
  static sequence<AlgorithmIdentifier>
getSupportedAlgorithms();
  sequence<RTCDtlsFingerprint> getFingerprints();
};
```

Figura 8.5: Definizione Standard JavaScript RTCCertificate

Un peer può creare un certificato *X.509* e la corrispondente chiave privata chiamando la funzione statica apposita fornita da *RTCPeerConnection*: in questo modo è possibile controllare il certificato offerto nella sessione DTLS durante la creazione di un nuovo *RTCPeerConnection*. Una volta creato il certificato è possibile impostare la sua data di scadenza tramite la funzione *RTCCertificateExpiration*.

```
RTCPeerConnection.generateCertificate({
  name: 'RSASSA-PKCS1-v1_5',
  modulusLength: 2048,
  publicExponent: new Uint8Array([1, 0, 1]),
  hash: 'SHA-256'
}).then(function(cert){
  var fingerprints = cert.getFingerprints();
  console.log("CERTIFICATO CREATO: "+fingerprints[0].value);
});
```

Figura 8.6: Esempio Creazione Fingerprint

La struttura RTCDtlsFingerprint consiste in un dizionario utile a memorizzare coppie chiave-valore del tipo:

- **algorithm**, specifica la funzione hash
- **value**, specifica il valore del fingerprint come stringa, secondo la sintassi standard RFC4572

8.1.4 RTCDtlsTransport

RTCDtlsTransport è il risultato della chiamata alle funzioni *setLocalDescription* e *setRemoteDescription*: permette all'applicazione di accedere alle informazioni riguardo il canale DTLS utilizzato per lo scambio dei pacchetti SCTP in un RTCDataChannel o dei

pacchetti RTP/RTCP in un RTCPeerConnection. Lo stato del canale DTLS, rappresentato dall'oggetto RTCDtlsTransport, è dato da uno dei possibili valori definiti nella enum RTCDtlsTransportState:

- **new**, la negoziazione DTLS non è ancora iniziata
- **connecting**, è in atto la negoziazione di una connessione sicura e la verifica dei fingerprint remoti
- **connected**, la negoziazione e la verifica dei fingerprint remoti è terminata ed è andata a buon fine
- **closed**, il canale DTLS è stato chiuso intenzionalmente in seguito alla ricezione del messaggio *close_notify* (alert)
- **failed**, si è verificato un errore, come la ricezione di un alert o il fallimento della verifica del fingerprint remoto

```
[Exposed=Window]
interface RTCDtlsTransport : EventTarget {
  [SameObject] readonly attribute RTCIceTransport
iceTransport;
  readonly attribute RTCDtlsTransportState state;
  sequence<ArrayBuffer> getRemoteCertificates();
  attribute EventHandler onstatechange;
  attribute EventHandler onerror;
};
```

Figura 8.7: Definizione Standard JavaScript RTCDtlsTransport

8.2 Tecnologie Utilizzate

Per la realizzazione dell'applicativo e dei test su di essi compiuti, sono state adottate le seguenti tecnologie:

- **VirtualBox**, software di virtualizzazione utilizzato per testare l'applicativo
- **Ubuntu 16.04 LTS**, sistema operativo installato su ogni macchina virtuale (peer e server)

- **Wireshark**, software utilizzato per analizzare il traffico di rete
- **Google Chrome**, browser adottato per l'esecuzione dei peer
- **webrtc-internals**, tool di Chrome utile all'analisi delle comunicazioni tramite WebRTC
- **Node.js**, software utilizzato per l'esecuzione del server

8.2.1 WebRTC-Internals

Nel browser Google Chrome è disponibile di default un tool accedendo all'URL interno chrome://webrtc-internals in una nuova scheda o pagina. Questo tool permette di effettuare il debug delle sessioni WebRTC attive e di determinare i problemi che si riscontrano durante lo sviluppo o l'esecuzione di una applicazione.
Oltre alle operazioni di debug, webrtc-internals può essere utilizzato per monitorare le prestazioni dell'applicazione in quanto fornisce dati dettagliati, statistiche e grafici in tempo reale sull'andamento della comunicazione. Il funzionamento di tale tool è basato sulle WebAPIs classificate come Statistics API, in particolare getStats e RTCStatsReport. È importante citare i seguenti report fra i vari tipi disponibili:

- **googCertificate**, contiene informazioni riguardo i certificati DTLS, i fingerprint e gli algoritmi di hashing utilizzati
- **googComponent**, contiene informazioni riguardo certificati e connessione come la ciphersuite utilizzata per DTLS

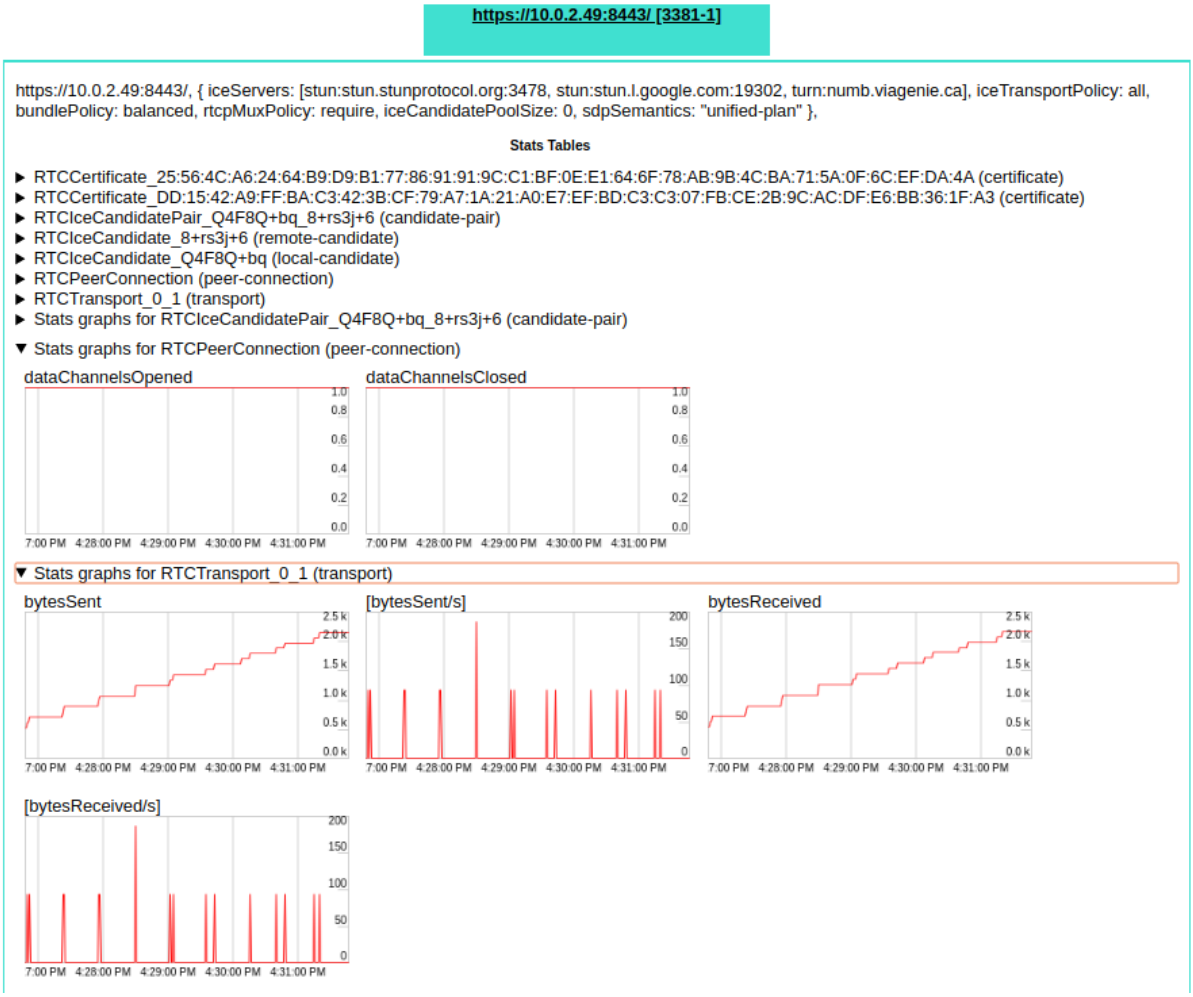

Figura 8.8: Istantanea Tool webrtc-internals durante un test

8.3 Scenario

Similmente a quanto descritto nei capitoli precedenti sullo scenario tipico di comunicazione real-time fra peer, si tratta ora dello scenario realmente implementato per l'esecuzione e l'analisi dell'applicativo realizzato. L'obbiettivo è quello di simulare la situazione in cui due peer comunicano via browser utilizzando le WebAPIs WebRTC ed uno dei due cambia IP durante la comunicazione.
I peer A, B e il server sono stati connessi a due reti di tipo NAT, individuate internamente con le interfacce di nome enp0s3 e enp0s8, alle quali è stato disattivato il servizio DHCP in modo da avere una maggior flessibilità di configurazione di rete per le varie entità comunicanti e poter quindi testare facilmente il cambio IP/rete.

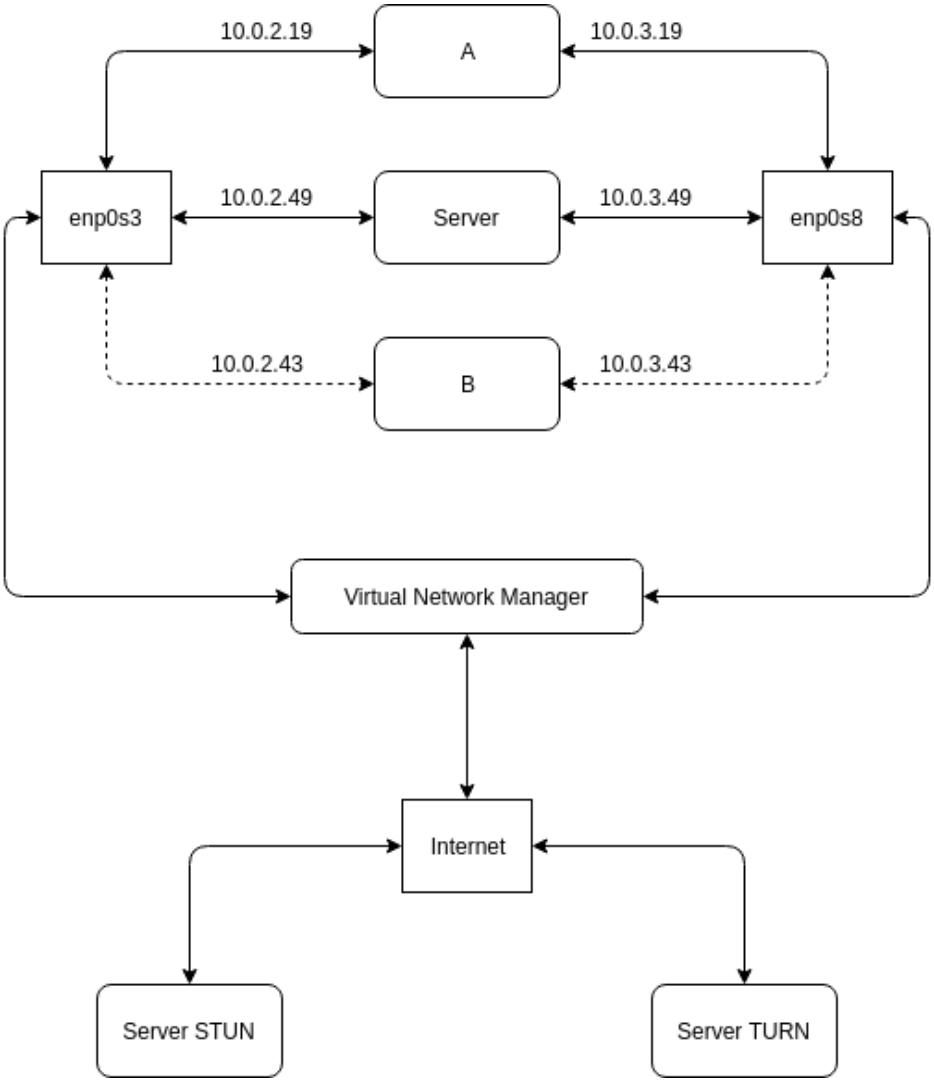

Figura 8.9: Scenario Complessivo realizzato

8.4 Realizzazione

Si descrive ora l'applicativo realizzato partendo dal funzionamento completo dei peer e del server discutendo sulle parti fondamentali di una comunicazione real-time che, come già visto nel capitolo su WebRTC, sono:

- RTCPeerConnection
- Signaling e negoziazione
- SDP (Session Description Protocol)
- ICE (Interactive Connectivity Establishment)

8.4.1 JSEP

WebRTC permette all'applicazione di avere il pieno controllo sul meccanismo di signaling al fine di aumentare l'interoperabilità fra i vari protocolli di signaling esistenti come *Jingle*, *ISUP* e *SIP*.
L'architettura standard di offerta/risposta viene chiamata *JavaScript Session Establishment Protocol* e rappresenta come è possibile escludere il browser dal meccanismo di signaling affidando l'intero compito all'applicazione.

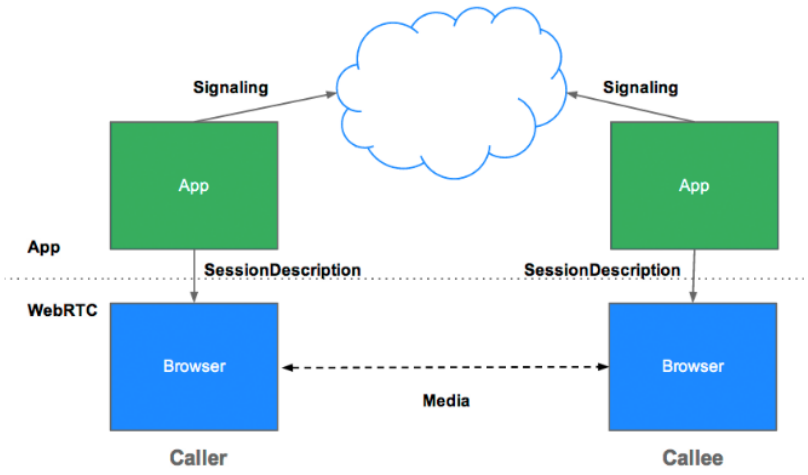

Figura 8.10: Architettura JSEP

8.4.2 Peer

Nelle comunicazioni fra peer, ciascuno dei partecipanti può sia dare inizio che dare fine alla comunicazione e per questo è necessario che essi vengano identificati per svolgere determinate funzioni, fra cui:

- creare il DataChannel
- creare ed inviare l'offerta SDP
- inviare la risposta SDP ed effettuare la join al DataChannel

Il peer definito chiamante è colui che crea il DataChannel ed invia l'offerta SDP attraverso il canale di signaling per il quale sono state impiegate le WebSocket. L'oggetto RTCPeerConnection è il punto principale su cui sono basate tutte le altre WebAPIs utili alla comunicazione: si ha infatti la necessità di istanziare correttamente un oggetto di questo per poter creare un DataChannel.
Dopo aver definito alcune differenze fra peer chiamante e peer chiamato, è possibile elencare alcune funzioni che rimangono comuni ad entrambi:

- identificazione tramite la creazione di UUID (Universally Unique Identifier)
- utilizzo delle WebSocket per la fase di signaling, tramite esso vengono scambiati sia le offerte e le risposte SDP che i candidati ICE

Per evidenziare le differenti funzionalità svolte dai peer, nel caso siano chiamanti o chiamati, e per ridurre la complessità dei diagrammi, sono riportate di seguito due macchine a stati finiti specifiche per ogni caso. In realtà l'applicativo lato client realizzato è unico e stabilisce il ruolo del peer verificando la presenza del DataChannel alla pressione del pulsante start.

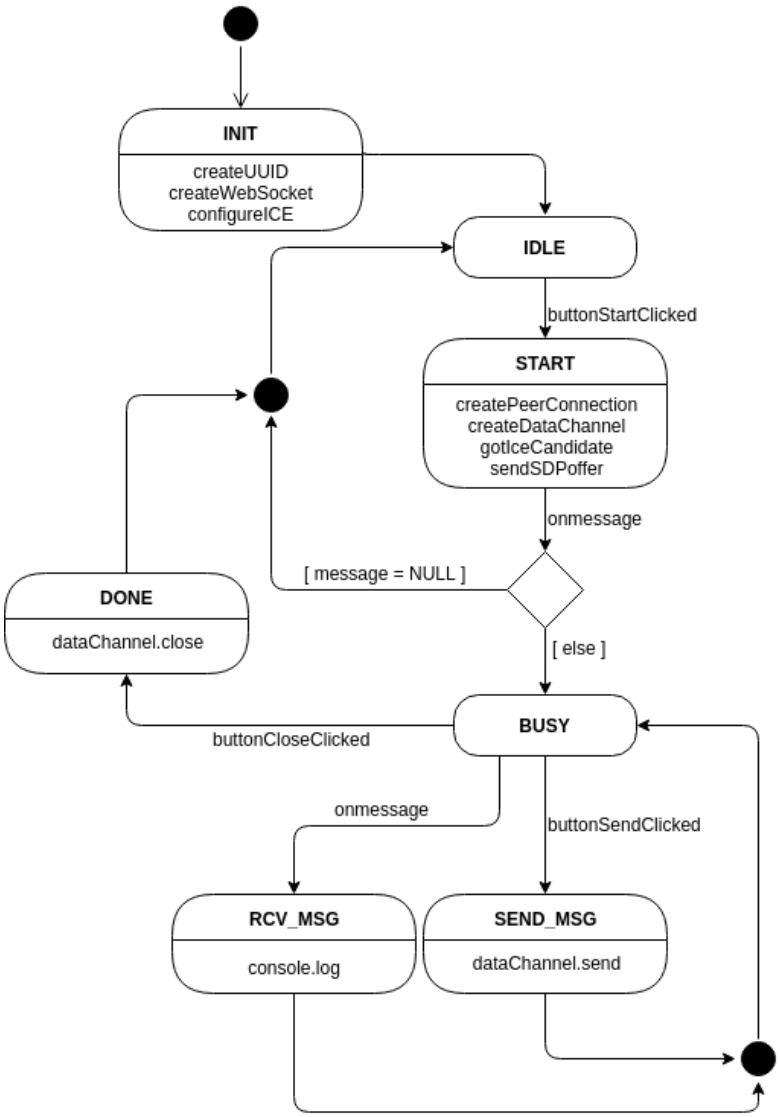

Figura 8.11: FSM Peer Chiamante

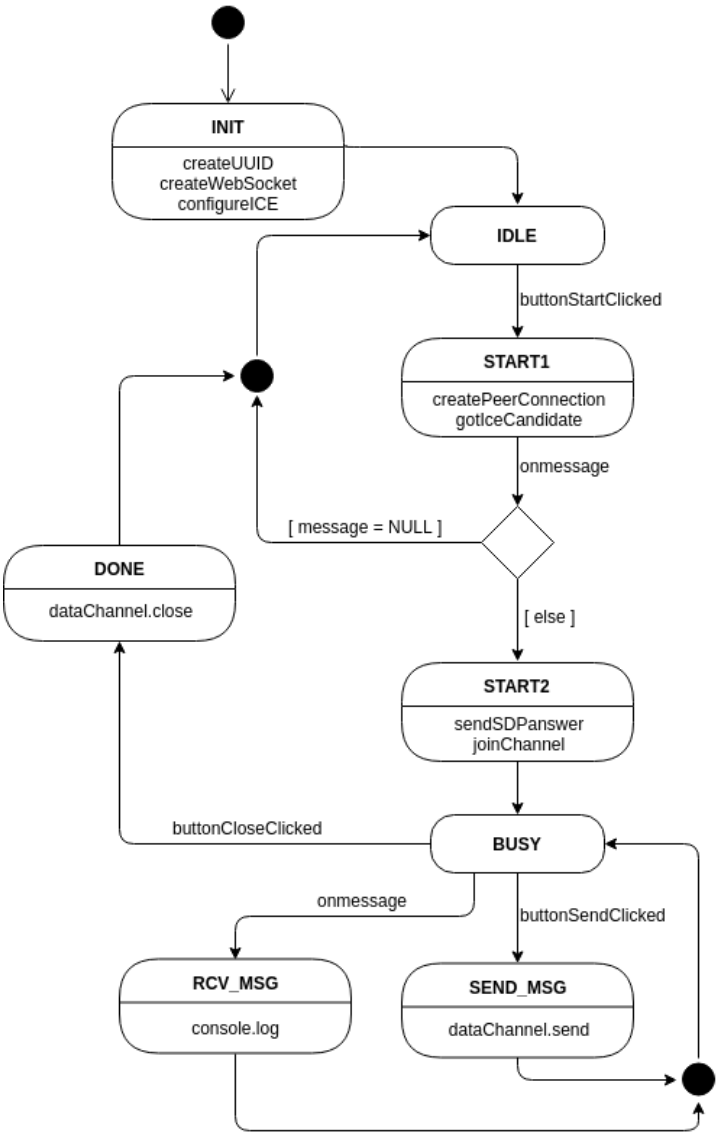

Figura 8.12: FSM Peer Chiamato

8.4.3 Server

Il server, implementato anch'esso in JavaScript, esegue in una terza macchina virtuale, come indicato nello schema generale precedente, grazie agli applicativi Node.js e npm. I moduli utilizzati dal server sono:

- **fs (File System)**
  Tale modulo fornisce dei wrapper delle funzioni standard POSIX ed è utilizzato in questo caso per accedere al certificato (cert.pem) e alla chiave (key.pem) del server

- **https**
  Il server utilizza questo modulo per mettersi in ascolto, sulla porta 8443, delle richieste HTTP in modo sicuro

- **ws (WebSocket)**
  Anche in questo caso il modulo è utilizzato per mettere il server in ascolto delle richieste WebSocket

Posizionandosi nella directory contenente il sorgente del server è possibile mettere in esecuzione il server lanciando il comando npm start.
Come definito nella macchina a stati finiti, il semplice server realizzato svolge due funzioni:

- rispondere alle richieste HTTP dei peer che vogliono utilizzare l'applicazione realizzata per comunicare tra di loro

- tramite le WebSocket si occupa di inviare a tutti i client, connessi al server, i dati ricevuti da un'altra WebSocket connessa. Naturalmente, come indicato nella sezione precedente, i dati ricevuti dal server ed inoltrati in broadcast sono offerte/risposte SDP o candidati ICE

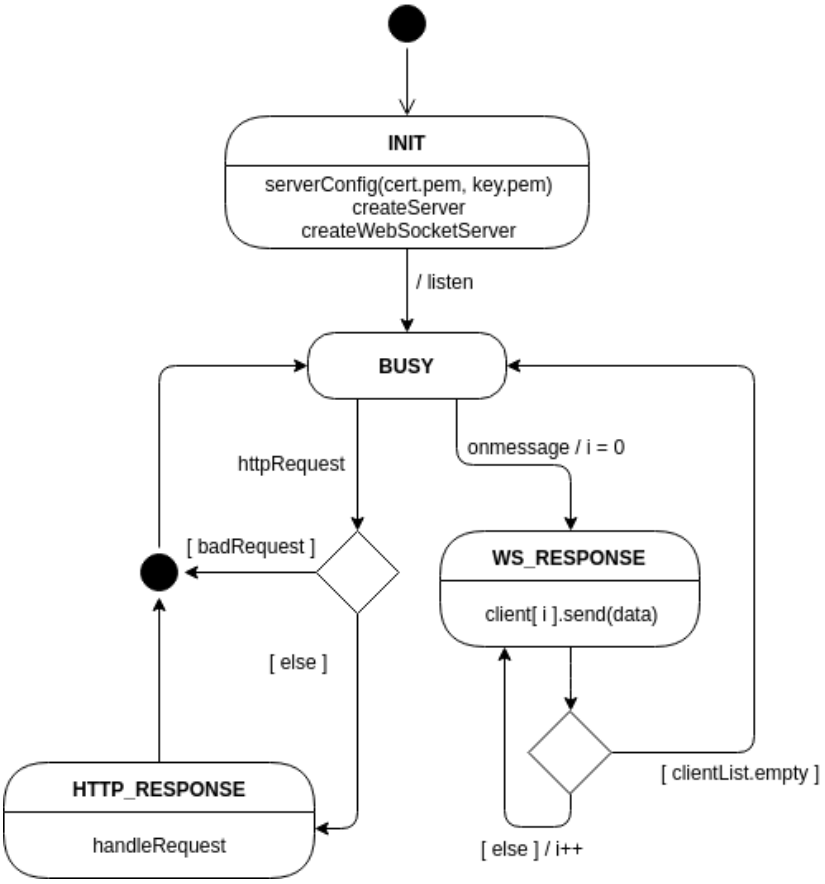

Figura 8.13: FSM Server

8.5 Test Applicazione

Definiti gli attori, lo scenario e la loro implementazione, si tratta in questa sezione dell'effettiva esecuzione dell'applicativo. Il test che simula al meglio lo scenario ideale è dato dal cambio IP tramite il cambio di interfaccia. Si ottiene infatti lo stesso effetto che si ha, in un caso reale, quando un dispositivo mobile passa dalla rete dati (3G, ...) alla rete fissa domestica (WiFi) o viceversa. Per ogni test effettuato si assume che:

- il peer B è connesso sempre solo ad una delle due interfacce
- il peer A è il primo a connettersi al server ed avvia la comunicazione creando il DataChannel
- il peer B è il secondo a connettersi al server ed effettua la join al DataChannel
- ogni manipolazione di rete viene effettuata sul peer B

In entrambi i peer è stato avviato il browser ed instaurata correttamente la comunicazione tramite la "pressione" del pulsante Start nel peer A. Dopo aver scambiato qualche messaggio di prova, per verificare il normale comportamento dell'applicativo, è stato possibile eseguire i test.
Il cambio dell'indirizzo IP tramite connessione ad altra interfaccia è stato effettuato con due scenari differenti. Nel primo, il server ed il peer A sono connessi ad entrambe le interfacce (enp0s3 e enp0s8) ed il peer B, inizialmente connesso all'interfaccia enp0s3, si disconnette connettendosi all'interfaccia enp0s8 una volta stabilita la connessione con l'altro peer. Nel secondo, il server è connesso ad entrambe le interfacce mentre i peer A e B sono connessi alla interfaccia enp0s3.

8.6 Conclusioni

L'approccio ad alto livello adottato in questa prima fase ha permesso di stabilire l'impossibilità di realizzare la Fast Resumption utilizzando le WebAPIs standard fornite per lo sviluppo di applicazioni WebRTC.
In entrambi i test effettuati non è stato possibile continuare la comunicazione in seguito al cambio di IP di uno dei due peer. Nonostante tali test siano stati fondamentali per la comprensione del comportamento di una comunicazione peer-to-peer non è stato possibile agire sulle parti interessate. Si ha infatti un insieme di API che incapsula tutti gli aspetti di più basso livello della comunicazione e fornisce solo alcuni metodi per accedere in sola lettura ad alcune di queste informazioni.
Per questo motivo si è deciso di scendere di livello e quindi di abbandonare le WebAPIs per utilizzare direttamente quello che fornisce la libreria nativa di WebRTC.

Capitolo 9

Ricerca e Compilazione

9.1 Ricerca

WebRTC è un progetto aperto e gratuito che fornisce, ai browser e alle applicazioni mobili, funzionalità di comunicazione in tempo reale (RTC) tramite semplici API. I componenti WebRTC sono stati ottimizzati per servire al meglio questo scopo.
Tuttavia, tra i vari componenti della libreria è disponibile l'implementazione del protocollo DTLS. Tale protocollo, pur non essendo stato pensato per implementare delle tecniche di Resumption, potrebbe comunque permetterci di sviluppare i seguenti metodi: Fast Resumption e Session Resumption, i quali si appoggiano appunto su DTLS, nelle modalità spiegate nel Capitolo 4.
Il primo approccio verso la libreria WebRTC non ha portato ai risultati sperati, infatti non potendo sfruttare le API JavaScript fornite ad alto livello per implementare i meccanismi di Resumption siamo dovuti entrare nel vivo della libreria. La limitatezza della documentazione e la complessità della struttura di tale libreria andranno ad influenzare notevolmente i successivi passi del nostro percorso di ricerca e sviluppo.
La versione di WebRTC con cui andremo a lavorare e che useremo per i nostri test è quella sviluppata e mantenuta da Google Inc. e supportata anche da altri sistemi come Mozilla e Opera.

9.2 Requisiti

La libreria per essere scaricata e compilata necessita di uno specifico ambiente ospitante, con una serie di strumenti di sviluppo installati.
Il sistema operativo richiesto è Ubuntu 14.04 o 16.04 nella versione x64, cioè con architettura a 64 bit. Il sistema deve essere aggiornato e su di esso deve essere presente anche il pacchetto di Python.
Successivamente è necessario installare i Depot Tools, una serie di strumenti sviluppati da Google, per lavorare con la libreria di WebRTC. Questi vengono forniti come un repository da clonare mediante Git e da salvare all'interno della stessa cartella di lavoro in cui sarà presente la libreria vera e propria.

9.3 Strumenti di Compilazione

Ninja e Gn sono due sistemi inclusi nei Depot Tools ma anche disponibili in modo separato nei corrispettivi repository, necessari per compilare la libreria e creare poi una "nostra" versione custom di WebRTC. Di seguito verranno illustrati più dettagliatamente.

9.3.1 Gn

GN è un applicativo a linea di comando che ha il compito di generare la cartella di build all'interno della libreria e tutti i file necessari a Ninja per ultimare la compilazione di WebRTC.
Mediante il seguente comando: gn gen out/"my_folder" sarà possibile realizzare tale cartella che prenderà il nome "my_folder" memorizzata dentro la cartella "out". All'interno delle cartelle saranno presenti i file di compilazione con estensione .ninja. I suddetti file, utilizzati poi dall'applicativo Ninja per compilare l'intera libreria e le applicazioni al suo interno, vengono realizzati a partire dai CMakeList, cioè una lista di tutti i file sorgenti presenti nella libreria.

9.3.2 Ninja

Ninja è un build system di piccole dimensioni focalizzato sulla velocità. Si differenzia dagli altri sistemi per due aspetti principali: è progettato per avere i suoi file di input di alto livello e per eseguire la compilazione il più velocemente possibile.
I file di build di Ninja, riconoscibili mediante l'estensione .ninja, sono infatti leggibili dall'uomo ma non sono particolarmente comodi da scrivere a mano, per questo vengono generati mediante GN. Al loro interno contengono le regole con le quali vengono compilare le applicazioni da generare e i file oggetto necessari ognuna di queste.
Questi file sono molto utili e consentono a Ninja di effettuare una compilazione rapida

ed incrementale, questo perché vengono compilati solo i sorgenti non compilati già in precedenza. La versione di Ninja utilizzata è la v1.9.0, rilasciata il 30 gennaio 2019.

9.4 Compilazione

La fase di compilazione prevede l'utilizzo dei due strumenti illustrati in precedenza, quali GN e Ninja. Dopo aver proceduto al download del repository ufficiale di WebRTC attraverso i Depot Tools, si procede nel seguente ordine:

```
1 #Creazione dei file per compilare la libreria:
2 gn gen out/Default
3
4 #Compilazione della libreria:
5 ninja -C out/Default
```

Terminato questo procedimento, la cui durata varia dal quantitativo dei sorgenti da compilare, si troveranno i nostri file eseguibili appena compilati nel percorso: *workspace/webrtc-checkout/src/out/Default*.
Durante la compilazione è possibile aggiungere al comando ninja l'argomento -v, in questo modo si rende l'operazione "verbose" e durante la sua esecuzione verranno mostrati i vari log. Questi risultano utili nel caso di realizzazione di nuove applicazioni per riscontrare errori nel codice già durante la fase di compilazione, senza dover attendere l'output della compilazione stessa.

9.5 PeerConnection

WebRTC come risultato della compilazione, mette a disposizione una serie di applicazioni d'esempio, che mostrano le principali funzionalità della libreria e possono essere trovati al seguente percorso: *workspace/webrtc-checkout/src/examples*
La libreria compilata presenta due applicativi che fruttano la PeerConnection:

- **Applicazione Server**, con nome target *peerconnection_server*
- **Applicazione Client**, con nome target *peerconnection_client* (non attualmente supportato su Mac / Android)

L'applicazione client permette di effettuare una comunicazione audio e video con altri client, sfruttando microfono e webcam integrata nel pc. Il server invece ha lo scopo di inizializzare una comunicazione fra i client gestendo lo scambio di messaggio di segnalazione fra questi ultimi.

9.5.1 Funzionamento

Dopo l'avvio di peerconnection_server tramite il comando *./peerconnection_ server* verrà visualizzato il messaggio: "Server listening on port 8888". A questo punto si possono avviare un numero qualsiasi di peerconnection_client e collegarli al server specificando l'IP e la porta del server.
Una volta connesso il client interfaccia utente permette di svolgere:

1. **Connessione ad un client**, ci si può connettere a un client selezionandolo dalla lista dei client connessi allo stesso server. Una volta connessi due client comincia la comunicazione audio/video tra di essi

2. **Chiusura della sessione di chat**, premendo Esc, sarà possibile tornare al menù di selezione di un peer

3. **Chiusura della connessione**, premendo Esc, sarà possibile tornare al menù di selezione del server

9.5.2 Limiti

Le applicazioni PeerConnection_client e PeerConnection_server presentano una grande complessità dovuta a quello che è il loro scopo, cioè l'instaurazione di una comunicazione tra peer che consenta di compiere videochiamate.
Le funzionalità di questi applicativi che riguardano la cattura e la trasmissione di dati di tipo audio e video, la gestione dell'accoppiamento dei client e l'utilizzo del protocollo UDP per la trasmissione dei dati non appartengono alla nostra sfera di interesse. Tuttalpiù aumentano notevolmente la complessità delle applicazioni e di conseguenza rendono maggiormente difficili modifiche per adattarlo al nostro scopo.
Nonostante questo, sono state utili per capite come funzionano le applicazioni che sfruttano direttamente la libreria tramite le chiamate alle API C++.

Capitolo 10

Creazione di un'applicazione in WebRTC

In questo capitolo andremo ad illustrare le problematiche incontrate nel tentativo di generare una propria applicazione in WebRTC. Il risultato che si tenta di ottenere è quello di un'applicazione che, analogamente a *peerconnection_client* e *peerconnection_server*, venga compilata assieme alla libreria.

10.1 Obiettivi

Gli obiettivi della creazione di un'applicazione personalizzata sono principalmente due. Il primo è chiaramente quello di acquistare manualità e consapevolezza nell'uso della libreria. La creazione di un'applicazione infatti non riguarda soltanto ciò che è legato alle funzionalità che la libreria mette a disposizione, ma anche ai tool utilizzati dalla libreria stessa per compilare il codice e generare eseguibili. Il secondo invece riguarda la possibilità di sviluppare successivamente un'applicazione più complessa che risulti più adatta di quelle fornite nativamente dalla libreria (*peerconnection_client* e *peerconnection_server*).

10.2 Complessità degli Applicativi già Esistenti

Come spiegato nel capitolo precedente *peerconnection_client* e *peerconnection_server* presentano una grande complessità dovuta a quello che è il loro scopo, cioè l'instaurazione di una comunicazione tra peer che consenta di compiere videochiamate.
Tra le diverse caratteristiche dell'applicativo che esulano dalla nostra sfera di interesse troviamo:

- La necessità di catturare e trasmettere dati di tipo audio e video
- L'accoppiamento di peer connessi al server
- L'utilizzo del protocollo UDP per il passaggio dei dati

Tutte queste caratteristiche non solo aumentano sensibilmente la complessità dell'eseguibile, e con essa riducono la possibilità di modificarlo per compiere dei test, ma soprattutto risultano poco utili al nostro studio.
In particolare, l'utilizzo dei pacchetti di tipo UDP per i dati risulta poco pratico in quanto il nostro caso di studio ideale deve presentare una comunicazione interamente basata sul protocollo DTLS.

10.3 Realizzazione

Al fine evitare ulteriori complessità si è optato per la creazione di un'applicazione dummy, in altre parole un'applicazione quasi del tutto priva di un'effettiva funzionalità.
L'applicazione, che è stata successivamente denominata *test_peerconnectionfactory*, ha come unico scopo quello di generare un eseguibile che crei un oggetto di tipo PeerConnectionFactory. Inizialmente si è pensato di creare un'eseguibile del tutto privo di contenuto (cioè un main vuoto essenzialmente), ma l'opzione è stata scartata in quanto non presentava alcun collegamento con le altre classi della libreria. L'oggetto *PeerConnectionFactory* è stato scelto in quanto necessita l'inclusione di diverse classi della libreria per essere istanziato ed inoltre è uno degli elementi fondamentale per l'instaurazione di una comunicazione tra peer in WebRTC.
In questa prima fase di realizzazione è stata aggiunto al path degli */examples* una cartella con all'interno il file sorgente. Ora la cosa da fare è modificare i file incaricati a definire le regole di compilazione del sorgente.

10.3.1 Reverse Engineering

La maggior parte delle problematiche riscontrate, durante qualsivoglia tipo di attività svolta sul codice nativo C++ della libreria, è dovuta alla mancanza di documentazione. L'assenza di documentazione ha reso necessario un processo di reverse engineering

per capire dove e come venissero scritte le regole di compilazione dei file necessari alla generazione di un eseguibile.

10.3.2 Ninja

Durante il processo di reverse engineering, come era possibile immaginare, ci si è imbattuti in una serie di file con estensione .ninja. Come visto nel capitolo precedente Ninja è un build system utilizzato da Google per compiere la build della libreria. I file Ninja generati dalla build verranno successivamente utilizzati per la compilazione.
Qui di seguito sono illustrati i file Ninja essenziali per la compilazione di un eseguibile:

- **build.ninja**
  Il file build.ninja fornisce un elenco di regole insieme a un elenco di istruzioni di compilazione che definiscono come dovrà essere creato il file. Concettualmente, le istruzioni di compilazione vanno a descrivere il grafico delle dipendenze del progetto, mentre le regole descrivono come generare i file di una determinata zona del grafico

- **toolchain.ninja**
  Il file toolchain.ninja è richiamato all'interno del file build.ninja ed è lui che definisce quali eseguibili verranno compilati. Ciò è possibile grazie alla keyword subininja che consente al subninja file di utilizzare le regole e le variabili del file ninja padre. All'interno di toolchain.ninja sarà necessario aggiungere nella sezione apposita il nome dell'eseguibile che vorremo compilare e le relative regole di compilazione, definite in un ulteriore file, utilizzando sempre la keyword subninja

- **File Ninja da creare**
  Ogni applicativo che la libreria presenta è caratterizzato da un proprio file Ninja specifico. Come per gli applicativi già presenti nella libreria è necessario che il nostro applicativo presenti un file Ninja che, come visto nel punto precedente, dovrà essere richiamato nel file toolchain.ninja con la keyword subninja. Questo file, chiamato per semplicità *test_peerconnectionfactory.ninja*, presenta diverse sezioni:

  - **defines**, definizioni di macro
  - **include_dirs**, cartelle da includere
  - **cflags**, flags di compilazione di codice C
  - **cflags_cc**, flags di compilazione di codice C++
  - **label_name**
  - **target_out_dir**, cartella dove verranno inseriti i file oggetto risultanti dalla compilazione

- **target_output_name**, nome dell'eseguibile
- **build**, regole per compilare i file da noi creati
- **build link**, lista dei file oggetto necessari alla creazione dell'eseguibile
- **Ldflags**, flags del linker
- **Libs**, librerie incluse
- **output_extension**, estensione dell'eseguibile
- **output_dir**, cartella dove verrà creato l'eseguibile

Una volta che il file è stato scritto adeguatamente e che verrà lanciato sulla console il comando di compilazione, i file Ninja da noi creati e modificati genereranno l'applicativo secondo le regole stabilite.

Capitolo 11

Approccio a Medio Livello

In questo capitolo andremo ad affrontare il Medio Livello della tecnologia WebRTC. Per medio livello intendiamo, come già anticipato nel Capitolo 7, l'approccio interno alla libreria nativa attraverso la generazione di applicazioni e attraverso lo studio delle API che essa fornisce.
Nei successivi paragrafi andremo a definire i passi svolti durante questo approccio a WebRTC. L'approccio a Medio Livello, analogamente a quello di Alto livello, è sperimentale e si basa sulla creazione di un applicativo funzionante e che sfrutti le funzionalità di WebRTC. Questo al fine di comprendere se sia sensato o meno lavorare a questo livello per implementare lai metodi di Resumption.

11.1 Caratteristiche dell'Applicazione

Una volta definito e accertato il modo per generare degli applicativi personalizzati all'interno della libreria, il passo successivo risulta essere quello di creare un applicativo più significativo di quelli già presenti. Come visto nel 10.2, le ragioni per cui *peerconnection_client* e *peerconnection_server* si sono rivelati poco adeguati sono molteplici.

11.2 Ricerca e Adattamento

Come ripetuto in precedenza la problematica principale, per quanto riguarda lo sviluppo di codice C++, è la mancanza di documentazione. Senza di essa e senza un diagramma UML per capire come utilizzare correttamente le API della libreria si è optato per la ricerca sul web di un qualsivoglia tipo di esempio da poter studiare e adattare. Essendo WebRTC una tecnologia relativamente recente non è facile trovare degli esempi, in particolare per quanto riguarda quelli scritti in C++. Dopo molte ricerche però è stato trovato un progetto che è risultato essere abbastanza in linea con ciò che ci serviva. Infatti, l'obiettivo di questo applicativo è quello di stabilire una connessione fra due peer e consentire l'invio, attraverso un DataChannel, di stringhe di testo.
Ovviamente è stato necessario un discreto lavoro di adattamento e modifica per far sì che questo applicativo, che da ora in poi per semplicità chiameremo *test_ simple_ datachannel*, risultasse compilabile e funzionante. Sono stati corretti alcuni errori e si sono svolte le dovute modifiche per poterlo integrare correttamente nella libreria come fosse un eseguibile analogo a *test_ peerconnectionfactory*.

11.3 Struttura dell'Applicazione

In questa sezione andremo ad analizzare la struttura e le parti più importanti di *test_ simple_ datachannel*. Tale applicativo è suddiviso principalmente in due sezioni.
La prima è costituita dal main e da delle di funzioni private che consentono, attraverso una serie di comandi passati al terminale, di compiere tutte le azioni necessarie a far sì che i due peer (cioè due istanze dell'applicazione) possano connettersi e comunicare.
La seconda invece è una classe di nome Connection, costituita a sua volta da un insieme di classi innestate, le quali racchiudono tutti gli elementi necessari al main e alle funzioni private per svolgere il loro compito.

11.3.1 Dettagli Implementativi

Come detto all'inizio di questo capitolo è presente una classe Connection che contiene molti degli elementi necessari al funzionamento di *test_ simple_ datachannel*.
Le funzioni che costituiscono il fulcro del funzionamento dell'applicazione riguardano lo scambio di dati in formato SDP e le relative operazioni da esse svolte sono:

- **cmd_sdp1()**
    1. La PeerConnectionFactory crea un oggetto PeerConnection associandogli l'observer PCO
    2. Tramite l'oggetto PeerConnection viene creato il DataChannel a cui viene associato l'observer DCO

3. L'oggetto PeerConnection crea l'offerta SDP che verrà stampata a video grazie all'observer CSDO

- **cmd_sdp2(string parameters)**

  1. La PeerConnectionFactory crea un oggetto PeerConnection associandogli l'observer PCO
  2. Viene creata una SessionDescription grazie ai parametri passati alla funzione
  3. L'oggetto PeerConnection imposta la remoteDescription utilizzando la sessionDesciption precedentemente creata
  4. L'oggetto PeerConnection crea la risposta SDP che verrà stampata a video grazie all'observer SSDO

- **cmd_sdp3(string parameters)**

  1. Viene creata una SessionDescription grazie ai parametri passati alla funzione
  2. L'oggetto PeerConnection imposta la remoteDescription utilizzando la sessionDesciption precedentemente creata

Le funzioni che gestiscono lo scambio delle informazioni relative al framework ICE e le relative operazioni sono:

- **cmd_ice1()**

  1. Stampa i candidati ICE da passare al secondo peer

- **cmd_ice2(string parameters)**

  1. Vengono creati degli IceCandidate grazie ai parametri passati alla funzione
  2. I candidati creati precedentemente vengono aggiunti alla PeerConnection che tramite essi svolge la procedura di instaurazione effettiva del canale dati fra i peer

In fine troviamo delle funzioni, che non andremo a descrivere in dettaglio in quanto ovvie, che consentono l'invio di stringhe e di chiudere correttamente il programma.

11.3.2 Flusso di Controllo

Le funzioni poc'anzi descritte, in particolare quelle relative a SDP e ICE, non necessitano di essere richiamate tutte da ogni peer per stabilire la connessione. Qui sotto è riportata un'immagine che descrive le chiamate a funzione svolte da ogni peer (che denomineremo per semplicità A e B) per instaurare la connessione.

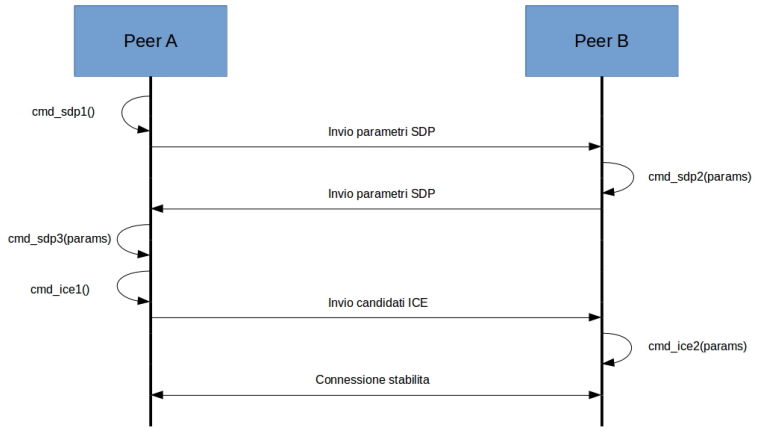

Figura 11.1: Flusso di controllo di test_simple_datachannel

11.4 Test dell'Applicazione

Al termine della creazione dell'applicazione si dà inizio, come fatto per la parte JavaScript, alla fase di test.
Lo scopo è capire se attraverso la modifica delle API richiamate da *test_simple_datachannel* è possibile andare a lavorare sul protocollo DTLS e quindi sviluppare la Fast Resumption.

11.4.1 Test_simple_datachannel in ambiente di test

Per testare l'applicazione è stata inserita una versione della libreria contenente i sorgenti di *test_simple_datachannel* in ognuna delle macchine virtuali Linux 16.04 (descritte approfonditamente nel Capitolo 12).
Dopo aver compilato i sorgenti sono state svolte le dovute operazioni per trovarci nel seguente stato.
Sia la prima che la seconda macchina sono state collegati alla rete virtuale 192.168.1.X rispettivamente attraverso l'interfaccia enp0s8 e enp0s9.
A questo punto è stato avviato l'applicativo, sono state svolte le operazioni necessarie per stabilire la connessione tra i peer ed è stata verificata la corretta funzionalità del canale.
Dopo di che nella seconda macchina è stato lanciato lo script netSwap.sh. Lo script, come spiegato dettagliatamente nel Capitolo 12.4, consente alla macchina di sganciarsi dall'interfaccia di rete corrente ed agganciarsi all'interfaccia enp0s9 che la collega alla

sottorete 192.168.2.X (sempre raggiungibile per mezzo di un Router).
Ovviamente in fase di test è stato provato anche il processo contrario, cioè che le due macchine inizino la comunicazione stando in sottoreti diverse e successivamente vengano agganciate alla stessa sottorete.

11.4.2 Risultato del Test

Dopo aver svolto diverse prove analizzando il traffico dati, attraverso Wireshark, è risultato che i peer durante tutta la comunicazione si scambiano dei pacchetti tramite il protocollo STUN. Dopo ulteriori ricerche è stato appurato che questi pacchetti che viaggiano da un peer all'altro sono definiti "ping STUN".
In WebRTC il termine "ping STUN" viene utilizzato per fare riferimento ai controlli di connettività ICE. Vengono utilizzati inizialmente all'avvio di ICE per trovare una coppia candidata iniziale (ai fini di stabilire la connessione) e, successivamente, vengono comunque inviati ad una velocità inferiore per mantenere aggiornato lo stato della connessione e per verificare che l'altro peer acconsenta ancora alla ricezione dei dati.
Durante i test è stato riscontrato però che dal momento in cui il primo ping non arriva a destinazione, a causa del cambio di IP dell'altro peer, la frequenza di invio dei "ping STUN" incrementa sensibilmente.

11.5 Conclusioni

L'invio costante dei "ping STUN" durante l'intero periodo di connessione dei peer risulta essere problematica nell'ottica dello sviluppo della Fast Resumption. Questo perché modificare le main API della libreria (es. PeerConnection, DataChannel, etc..) negli aspetti relativi al protocollo DTLS non comporterebbe una soluzione sicura. L'invio continuo di questi ping infatti risulta comunque essere un problema anche se il sottostante protocollo DTLS gestisce correttamente il cambio IP.
In uno scenario ipotetico in cui la Fast Resumption venga correttamente implementata c'è il rischio che risulti necessario svolgere ulteriore lavoro altrettanto grande. Questo per evitare che tutti gli elementi della PeerConnectionche che gestiscono lo stato di failure della connessione si attivino, evitando inoltre di non rimuovere la gestione di una normale caduta della connessione.
Per queste ragioni si è deciso che questa strada da intraprendere risulta, non necessariamente impossibile, ma estremamente complessa e con scarse probabilità di risultato.

11.6 Livello Intermedio

Finora non si è mai trattato singolarmente del Livello Intermedio in quanto è parte integrante del Medio Livello. Il Livello Intermedio è caratterizzato da un insieme di classi e funzionalità che non fanno parte delle API e delle interfacce qui sopra descritte. Possiamo considerare questo livello come un punto di accesso della libreria in C++ alle librerie di terze parti e ad altre funzionalità base per la comunicazione real-time. Uno dei tipi di classe più interessanti, fra quello del Livello Intermedio, sono gli Adapters. Le classi Adapters (come ssladapters, openssladapters, socketadapters etc...) sono molto importanti in quanto hanno lo scopo di fornire un'interfaccia, accessibile e adeguatamente personalizzata, di ciò che rappresentano.
Le classi di questo livello vengono utilizzate per implementare le API principali che abbiamo visto nei paragrafi precedenti durante l'approccio a Medio Livello.

Capitolo 12

Ambiente di Test

In questo capitolo andremo a definire il lavoro svolto al fine di costruire un ambiente su cui testare la libreria nativa in C++ e gli applicativi da noi sviluppati.

12.1 Obiettivo

Ciò che si cerca di ottenere è un ambiente che presenti due macchine in grado di contenere e compilare la libreria, due reti a cui le macchine possono liberamente collegarsi e scollegarsi ed infine un meccanismo che consenta ad una delle due macchine di cambiare velocemente interfaccia di rete e di conseguenza IP.

12.2 Oracle VM VirtualBox

L'ambiente di test è stato realizzato non in maniera fisica ma virtuale, non solo per mancanza di mezzi ma anche per avere una maggiore affidabilità del sistema. Il software scelto su cui lavorare, come spiegato nel Capitolo 6, è VirtualBox. Senza dilungarci troppo diciamo che Oracle VM VirtualBox è un software gratuito e open source per l'esecuzione di macchine virtuali (con una versione ridotta distribuita secondo i termini della GNU General Public License) per architettura x86 e 64bit che supporta Windows, GNU/Linux e Mac OS come sistemi operativi host, ed è in grado di eseguire Windows,

GNU/Linux, OS/2 Warp, BSD e infine Solaris e OpenSolaris come sistemi operativi guest.

12.3 Realizzazione

In questo paragrafo andremo a vedere nel dettaglio come è stato creato l'ambiente.

12.3.1 Macchine Virtuali

Per raggiungere il nostro scopo, come detto in precedenza, saranno necessarie due macchine virtuali dove poter caricare e compilare la libreria. Sono state quindi create due macchine, VM1 e VM2, installando su di esse (come definito nel Capitolo 9) il sistema operativo Ubuntu Desktop 16.04 LTS.
Successivamente si è considerata la necessità di una terza macchina che, in un'ottica di due reti virtuali, svolga il ruolo di forwarder di pacchetti per consentire alle altre due macchine di comunicare anche se non si trovano nella stessa sottorete. Per questa ragione è stata creata una terza macchina denominata Router su cui, visto il lavoro che deve compiere, è stato installato un Ubuntu Server 16.04 LTS.

12.3.2 Reti Virtuali

Per quanto riguarda le reti invece si è pensato di crearne due, neta e netb, a cui le macchine possono connettersi. Visto che per svolgere i test non è necessario che entrambe le macchine possano cambiare IP, si è deciso di consentire solo alla seconda macchina di potersi connettere ad entrambe le reti attraverso due diverse interfacce.
Una volta definito ciò, sono state impostate le adeguate network adapter per ogni macchina in VirtualBox e per far sì che tutto funzioni correttamente è stato modificato, all'interno di ogni macchina, il file */etc/network/interfaces*.
Abbiamo rispettivamente:

- **VM1**
  La VM1 presenta due network adapter, il primo che utilizza il NAT e il secondo la rete virtuale denominata neta

- **VM2**
  La VM2 presenta invece tre network adapter. Il primo rimane sempre quello relativo al NAT, il secondo alla rete virtuale neta e il terzo alla rete netb

- **Router**
  Il Router funziona analogamente a VM2

Infine, per consentire al Router di funzionare correttamente è stato abilitato il forwarding de commentando la seguente riga *net.ipv4.ip_forward=1* nel file */etc/sysctl.conf*. Una volta svolte queste operazioni la situazione che ci si presenta è riassumibile nella seguente immagine:

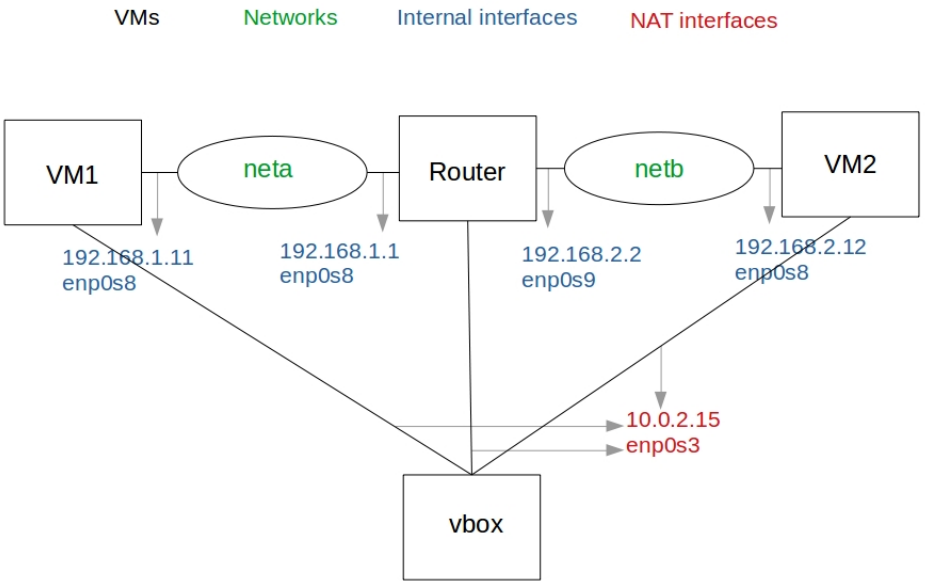

Figura 12.1: Struttura ambiente di test.

12.4 Test Cambio IP

La motivazione per cui la macchina VM2 è collegata a entrambe le retti è che, come accennato in precedenza, essa deve essere in grado di cambiare IP rimanendo comunque raggiungibile alla macchina VM1. Per svolgere queste operazioni si è scelto di scrivere il seguente script *netSwap.sh*, che attraverso due semplici comandi sgancia VM2 dall'interfaccia di rete enp0s9 e aggancia alla macchina l'interfaccia enp0s8:

```
1 # Network interfaces swapper
2 sudo ifup enp0s8
3 sudo ifdown enp0s9
```

Ovviamente per funzionare correttamente al momento del lancio dello script VM2 dovrà risultare agganciata solo all'interfaccia enp0s9.

Capitolo 13

Approccio a Basso Livello

Lo studio della libreria a basso livello è risultato necessario dal momento in cui si è preso coscienza delle problematiche relative agli altri approcci. La ricerca è stata mirata a capire se fossero presenti all'interno della libreria gli strumenti e le funzioni per poter implementare realmente la Resumption.
L'analisi infine ha portato a risultati non molto positivi per quanto riguarda la disponibilità di tali funzionalità. Di seguito verrà illustrato in modo più approfondito il lavoro svolto.

13.1 Sviluppo

Partendo dalla documentazione delle tecniche di Resumption, sono state raccolte funzioni necessarie all'implementazione di quest'ultima all'interno di OpenSSL.
La maggior parte di queste funzioni risulta implementata e dichiarata all'interno della libreria BoringSSL, la quale come già illustrato nel Capitolo 3 è una libreria di terze parti, inclusa all'interno di WebRTC. L'uso di tali funzioni è risultato di conseguenza molto semplice, in quanto è stato sufficiente richiamarle all'interno del nostro codice per sfruttarle. Le restanti funzioni hanno comportato un discreto lavoro, che verrà descritto nei paragrafi successivi.
I risultati di questa ricerca ci hanno permesso, in un primo momento, di generare un header file e un file sorgente, denominati rispettivamente *DtlsUtils.h* e *DtlsUtils.c*.
Le DtlsUtils sono una raccolta di funzionalità utilizzate per lo sviluppo della Resumption

in OpenSSL presenti anche in BoringSSL.
Lo sviluppo di questi file sorgenti ci ha permesso di ottenere un primo risultato nello studio a basso livello, permettendoci così di proseguire nella creazione di un applicativo che svolga Fast e Session Resumption a questo livello di astrazione.

13.2 Funzioni non esposte da BoringSSL

Delle restanti funzioni necessarie ad implementare la Resumption, alcune di queste risultavano presenti ed implementate all'interno di BoringSSL, ma non richiamabili nei nostri file sorgenti. In un secondo momento si è infatti compreso che WebRTC mette in atto delle politiche di sicurezza per esporre ai livelli superiori solo le funzioni delle librerie sottostanti di cui essi necessitano.
La libreria BoringSSL espone perciò solo parte dei metodi, delle macro e delle strutture dati implementate e lo fa attraverso libboringssl. Libboringssl è un file generato durante la compilazione che racchiude al suo interno tutte le funzionalità che BoringSSL mette a disposizione di WebRTC. Così facendo rende richiamabili solo determinate funzioni all'interno dei sorgenti presenti nella libreria. Si è perciò dovuto cercare una soluzione per andare ad esporre le funzioni seguenti così da poterle utilizzare: *OPENSSL_free(uint8_t *p), OPENSSL_assert(int n), OPENSSL_malloc(unsigned int length), SSL_CTX_set_read_ahead(SSL_CTX *ctx, int yes)* .
Tale compito non è stato dei più semplici a causa della scarsità di documentazione della struttura di WebRTC e delle sotto librerie di cui è composto, ma vedrà soluzione nella realizzazione delle ResumptionUtils.

13.3 Funzioni non Implementate in WebRTC

Un'altra tipologia di funzione necessarie allo sviluppo delle tecniche di Resumption dentro la libreria di WebRTC sono quelle non implementate.
Alle funzioni totalmente assenti però si affiancano anche quelle dichiarate all'interno dei file header corrispondenti ma con l'implementazione rimossa in specifici commit, in quanto considerata non necessaria dagli sviluppatori di WebRTC.
Per questo insieme di funzioni non si è riuscito a trovare totalmente una soluzione e questo ha portato al fallimento dell'approccio a basso livello, ma questo aspetto verrà analizzato più approfonditamente di seguito.

13.4 Resumption Utils

Le ResumptionUtils nascono come uno strumento per esporre le funzionalità di BoringSSL mancanti, potendo così sviluppare la Resumption a basso livello. Per produrre questo

strumento è stato necessario applicare delle modifiche nei file di compilazione e strutturare un nuovo file sorgente all'interno di WebRTC col seguente nome *resumption_utils.c*. La realizzazione di queste utilità non è stata immediata, in quanto non sono state trovate informazioni riguardanti libboringssl, ma tutto il lavoro svolto è stato frutto di molteplici tentativi e reverse engineering.

13.4.1 File Sorgente

Il file sorgente delle ResumptionUtils è situato nel seguente percorso: *workspace/webrtc-checkout/src/third_party/boringssl/src/ssl* Tale locazione del file ha permesso di richiamare le funzioni non esposte da BoringSSL ma comunque presenti, in quanto vengono richiamate all'interno della libreria stessa.
Le funzioni non implementate in WebRTC per prima cosa sono state dichiarate nel file header corrispondente. Essendo funzioni di tipo SSL_CTX la dichiarazione è stata inserita all'interno del file *ssl.h*. In questo file sono state poi eseguite modifiche alle strutture dati su cui tali funzioni operano per permettere il salvataggio e la lettura dei dati.
Le funzioni sono state poi implementate all'interno del file *resumption_utils.c*. Il comportamento della funzione da implementare è stato ricercato all'interno della libreria OpenSSL, per poterlo poi riprodurre nel nostro lavoro.

13.4.2 Regole di Compilazione

Modificare le regole di compilazione di libboringssl, per poter esporre le ResumptionUtils create, era obbligatorio altrimenti il lavoro precedentemente svolto sarebbe stato inutilizzabile. Di seguito andremo ad elencare i passaggi svolti.
Per aggiungere un modulo a libboringssl.a è necessario aggiungere la classe *resumption_utils.cc* nella cartella:
/workspace/webrtc-checkout/src/third_party/boringssl/src/ssl/
cioè nello stesso percorso in cui si trova anche *ssl_lib.cc*, e aggiungere le regole di compilazione del modulo nel file *boringssl.ninja* presente in:
/workspace/webrtccheckout/src/out/Modified/obj/third_party/boringssl/boringssl.ninja.
Per quanto riguarda i metodi non basta normalmente implementarli in *resumption_utils.cc*, ma per essere correttamente richiamabili è necessario aggiungere le definizioni in altri due file.

13.4.3 Pregi

Per quanto l'uso delle ResumptionUtils sia momentaneamente limitato all'interno del nostro progetto, e riguardi solamente la libreria BoringSSL, la possibilità di aggiungere nuove funzionalità ad una libreria di terze parti presente in WebRTC e di poterle sfruttare nelle funzionalità ad alto livello è una scoperta molto interessante.

Questo aspetto non è illustrato in nessuna documentazione da noi consultata ma può tornare utile per realizzare altre molteplici estensioni di WebRTC, permettendo un ampliamento notevole delle funzionalità che la libreria può fornire.

13.5 Creazione di Client e Server

Terminato lo sviluppo delle ResumptionUtils e delle funzioni comuni aggiunte alle DtlsUtils, si è proceduto con lo sviluppo di un ambiente per poter testare tutto il lavoro svolto finora e verificarne il corretto funzionamento.
Il lavoro in questa fase di studio mirava a realizzare due applicativi, uno lato client e uno lato server, che sfruttassero le funzioni realizzate e che riproducessero l'architettura client-server sfruttata nella realizzazione originale della Resumption in OpenSSL, spiegata nel Capitolo 5. Sia per quanto riguarda il client che per il server, attraverso lo sviluppo delle ResumptionUtils sono stati risolti la maggior parte problemi dovuti alla mancanza di funzioni all'interno di WebRTC.
Sfortunatamente questo percorso non è stato terminato a causa dell'assenza di due specifiche funzioni, che verranno illustrate meglio nel Capitolo 13.7. Queste funzionalità non è stato possibile implementarle all'interno delle ResumptionUtils, come quelle illustrate in precedenza, a causa della loro complessità interna e delle innumerevoli dipendenze da altre classi all'interno di OpenSSL le quali non risultavano presenti in BoringSSL.

13.6 Arresto nello Sviluppo

Durante lo sviluppo dell'applicativo a lato client ci siamo imbattuti nella mancanza della seguente funzione: *BIO *BIO_new_dgram(int fd, int close_flag);*
Lo scopo di tale funzione, come illustrato nella documentazione della libreria OpenSSL, è quello di creare un datagramma di tipo BIO che possa essere usato dentro una sessione DTLS. Tale funzione non è il motivo principale dell'arresto nello sviluppo, infatti, si è pensato che con possibili adattamenti la funzione *BIO *BIO_new_fd (int fd, int close_flag);* possa in qualche modo sostituirla. La differenza principale fra le due funzioni risulta essere che la seconda non lavora su un datagramma ma direttamente sui File Descriptor della sessione. Tale soluzione è stata messa in pratica, ma non è stata testat in quanto non si è riuscito a completare l'applicativo lato server. Nell'applicativo a lato server la funzione che ha bloccato il nostro percorso di studio è stata: *int DTLSv1_listen(SSL *ssl, BIO_ADDR *peer);* Questa funzione non è presente in nessuna classe della libreria WebRTC e non sono state trovate alternative valide per poterla sostituire. Il suo scopo è quello di mettere il server in attesa di nuove connessioni DTLS in entrata. Nel caso in cui venga ricevuto un *ClientHello* non contenente un cookie, *DTLSv1_listen()* risponde con *HelloVerifyRequest.*

Se viene ricevuto un *ClientHello* con un cookie verificato, il controllo viene restituito al codice utente per consentire il completamento dell'handshake. Dopo innumerevoli tentativi questa soluzione è stata abbandonata in quanto anche se ad un passo dalla realizzazione finale, era impossibile procedere nel lavoro.

13.7 Limiti Architetturali

Nella documentazione di BoringSSL non è spiegato il motivo della rimozione della funzione *DTLSv1_listen()* dalla libreria. L'unica informazione risale ad un commit di qualche anno fa (*60e799276419e843b6af13de69f26582a97ed67e*) in cui viene specificato solamente che la rimozione di questa funzione è dovuta al non utilizzo all'interno della libreria WebRTC.
Analizzando logicamente il problema abbiamo concluso che l'assenza di questa funzione è dovuta alle caratteristiche architetturali di WebRTC. La libreria BoringSSL presente in WebRTC infatti, è studiata per poter lavorare su un'architettura peer-to-peer e non su un'architettura clientserver. Questa differenza architetturale spiega perché alcune funzionalità presenti in OpenssL sono state rimosse da BoringSSL, come la *DTLSv1_listen()*.

Capitolo 14

Sviluppi Futuri

In questo capitolo di sviluppi futuri illustriamo come, visto il fallimento dei vari approcci affrontati, si potrebbe continuare il lavoro da noi svolto finora. L'utilizzo e/o estensione funzionalità relative alle classi del Livello Intermedio (trattato nel Capitolo 11.6), potrebbe essere la chiave per un nuovo approccio alla libreria e per lo sviluppo dei metodi di Resumption.
L'idea è quella di evitare di utilizzare le API preconfezionate della libreria, come fatto durante l'approccio a Medio Livello, ed evitare allo stesso tempo di scendere ad un livello troppo basso. Se si lavora a Livello Intermedio, potrebbe essere possibile sfruttare le classi necessarie a stabilire una comunicazione real-time peer-to-peer per sviluppare i metodi di Resumption. Questo però comporta un indiretto allontanamento ai metodi di Resumption da noi analizzati e presi in considerazione, in quanto dovranno essere massicciamente rivisitati. Tuttavia, non si hanno certezze della reale fattibilità di questo approccio in quanto non è stato svolto uno studio abbastanza approfondito.

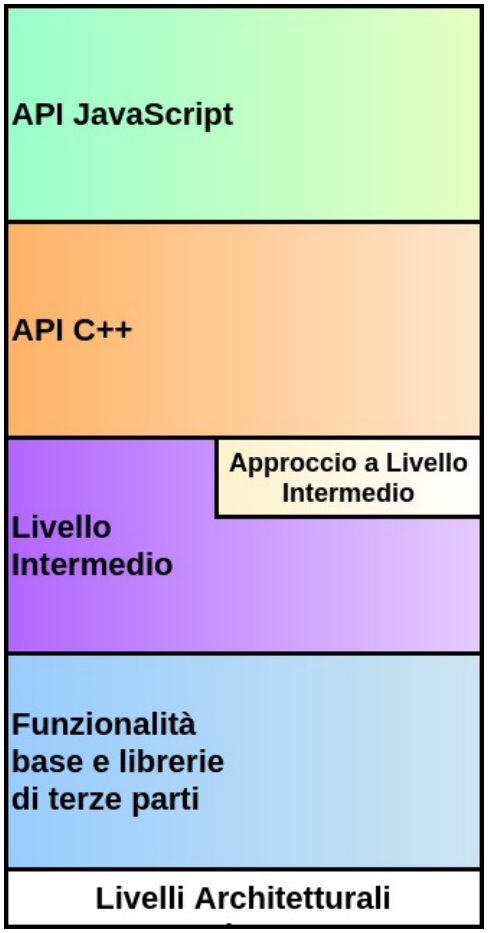

Figura 14.1: Livelli Architetturali – Livello Intermedio

Capitolo 15

Conclusioni

Il lavoro svolto ha portato alla luce informazioni molto interessanti, alcune intuitive, altre del tutto inaspettate.
Le dimensioni notevoli della libreria hanno reso l'approccio a più livelli una scelta ottimale per studiarla in modo approfondito. Il lavoro svolto non ha comunque portato alla realizzazione di una implementazione funzionante.
Nel corso dello studio l'approccio ad alto livello, al quale mi sono dedicato principalmente, ha permesso di stabilire l'impossibilità di realizzare la Fast Resumption utilizzando le WebAPIs standard per lo sviluppo di applicazioni WebRTC. Le interfacce fornite, come descritto nel Capitolo 8, incapsulano tutti gli aspetti di basso livello e forniscono metodi per accedere in sola lettura ad un insieme limitato di informazioni su cui si dovrebbe lavorare. Per questo motivo, nella prima fase di lavoro, è stato possibile realizzare solamente un semplice applicativo. Tale applicativo ha però permesso di prendere dimestichezza con quelli che sono i meccanismi di comunicazione Real-Time fra browser ed in modo particolare come essi vengono gestiti in WebRTC.
L'approccio a medio livello, al quale si è occupato maggiormente il collega Milo Marchetti, ha mostrato come il problema principale sia l'invio costante dei "ping STUN" durante l'intero periodo di connessione dei peer. Modificare le main API della libreria negli aspetti relativi al protocollo DTLS non comporterebbe una soluzione sicura, in quanto l'invio continuo di questi ping risulta comunque essere un problema anche se il sottostante protocollo DTLS gestisce correttamente il cambio IP. In uno scenario ipotetico in cui la Fast Resumption venga correttamente implementata c'è il rischio che risulti

necessario svolgere ulteriore lavoro altrettanto grande. Questo per evitare che tutti gli elementi della PeerConnectionche che gestiscono lo stato di failure della connessione si attivino, evitando inoltre di non rimuovere la gestione di una normale caduta della connessione.
Per queste motivazioni si è deciso che questa strada da intraprendere risulta, non necessariamente impossibile, ma estremamente complessa e con scarse probabilità di risultato.
L'approccio a basso livello, al quale si è dedicato principalmente il collega Enrico Gnagnarella, è stato l'ultimo approccio adottato. Intrapreso per scoprire se tutte le funzionalità necessarie ad implementare la Fast Resumption fossero presenti, tale approccio ha mostrato come questo non fosse vero. La versione di BoringSSL presente fra le librerie di terze parti è stata adattata a WebRTC, e per questo motivo sono state rimosse le funzionalità di ascolto dei pacchetti in entrata necessarie all'implementazione del server. La ragione di questo adattamento è dovuta dalle caratteristiche architetturali della libreria, la quale infatti si basa su un'architettura peer-to-peer e non clientserver. Questa differenza di architettura porta all'impossibilità di realizzare la Resumption in questo livello.
I risultati ottenuti ci hanno permesso di capire che l'approccio a WebRTC sia da rivalutare fin dal principio, riprogettando una tecnica di ripristino della connessione che consideri l'architettura peer-to-peer ed il framework ICE.

Capitolo 16

Bibliografia e Sitografia

Durante lo sviluppo della tesi sono stati utilizzate varie fonti fra cui siti, documenti, documentazione ufficiale ed RFC. Qui sono riportati i documenti e i testi principalmente utilizzati, inoltre è allegata anche qualche piccola nota esplicativa.

16.1 RFC

1. **RFC 5077**
   Da questo RFC sono state presi i diagrammi per mostrare la Session Resumption

2. **RFC 5746**
   Questo RFC mostra il processo di rinegoziazione TLS, importante per descrivere le differenze tra BoringSSL e OpenSSL.

3. **RFC 4572**
   Mostra la negoziazione dei parametri SDP in una comunicazione TLS.

4. **Draft JSEP**
   Descrizione del meccanismo di session establishment in comunicazioni real-time web.

16.2 Documenti

5. **W3C WebRTC API Documentation** Questo documento definisce le API utilizzate per le seguenti funzionalità:

    - Connessione a peer remoti utilizzando tecnologie NATTraversal come ICE, STUN e TURN
    - Invio di dati generici direttamente a peer remoti
    - Inviare e ricevere "tracks" verso e da peer remoti

    Questa specifica è stata sviluppata dal gruppo **IETF RTCWEB**.

6. **OpenSSL Documentation** La documentazione di OpenSSL è servita nello studio a basso livello, per l'implementazione delle Resumption Utils.

7. **BoringSSL Documentation**
    Documentazione di BoringSSL.

8. **Stabilizzazione di un canale DTLS in sistemi IOT mobili**
    Standard, limiti e proposte risolutive Documento di tesi triennale di Sara Kiade e Gyordan Caminati da cui abbiamo appreso e studiato i metodi di Resumption

16.3 Siti

9. **Sito ufficiale di WebRTC**
    Il sito ufficiale della libreria da noi presa in considerazione per lo studio, al suo interno sono contenuti svariate informazioni e link utili.
    Principalmente è stato utilizzato per capire come:

    - **Compilare la libreria**
    - **Approcciarsi inizialmente alla libreria**

10. **Getting Started with WebRTC**
    Questo sito, come anche il successivo, è stato molto utile durante il primo approccio a WebRTC in quanto descrive alcuni dei meccanismi principali della tecnologia.

11. **A Study of WebRTC Security**
    Analogo al precedente ma con un occhio di riguardo per la sicurezza.

12. **Discuss WebRTC – Google Groups**
    Forum di discussione su WebRTC. Durante l'approccio a medio livello questo forum è stato estremamente d'aiuto in quanto il medio livello non è in nessun modo

documentato. Attraverso la ricerca per parole chiave è stato possibile analizzare discussioni su ciò che si stava trattando o scoprire qualcosa di nuovo fino a quel momento ignorato.

13. ICE Framework Over WebRTC
Descrizione dettagliata del funzionamento del Framework ICE in WebRTC.

14. Modalità di Networking in VirtualBox Da questo sito sono state prese le informazioni necessarie a comprendere le diverse modalità di rete in VirtualBox in fase di costruzione degli ambienti di test per le applicazioni.

15. Esempio VideoChat
Esempio che apre una VideoChat sfruttando le API JavaScript.

16. Esempio DataChannel
Esempio modificato, adattato e studiato nell'approccio a medio livello.

17. WebRTC Adapter

18. BoringSSL repository
Repository ufficiale della libreria di BoringSSL.

16.4 Libri

19. Pravir Chandra, Matt Messier, John Viega, *Network Security with OpenSSL*, O'Reilly, 2002

20. Simon Pietro Romano, Salvatore Loreto, *Real-Time Communication with WebRTC*, O'Reilly, 2014

21. Ilya Grigorik, *High Performance Browser Networking*, O'Reilly, 2013

22. Dan Ristic, *Learning WebRTC*, Packt Publishing, 2015

Capitolo 17

Appendice

Nell'appendice verranno descritti nello specifico alcuni processi realizzativi riassunti nei vari capitoli della tesi. Questo al fine di consentire a eventuali successori del nostro studio di mettersi nelle condizioni di partire dal punto in cui noi ci siamo fermati senza dover perdere tempo su cose già affrontate.

17.1 BoringSSL

17.1.1 Prerequisiti

```
1 sudo apt install cmake
2
3 sudo apt install ninja-build
4
5 sudo add-apt-repository ppa:longsleep/golang-backports
6 sudo apt-get update
7 sudo apt-get install golang-go
```

17.1.2 Installazione

```
1 #Download repository ufficiale della libreria:
2 git clone https://boringssl.googlesource.com/boringssl
```

```
3
4 #Creazione cartella per la compilazione:
5 mkdir build && cd build
6
7 #Compilazione:
8 cmake .. -GNinja
9 ninja
```

In caso i comandi della sezione Compilazione diano problemi riguardo alla versione di ninja eseguire il comando *cat CMakeCache.txt | grep ninja* dalla cartella build per verificare che ci sia la seguente stringa *CMAKE_MAKE_PROGRAM:FILEPATH=/usr/bin/ninja*. In caso negativo correggere l'errore ed eseguire il comando *sudo ln -s /usr/bin/ninja /usr/bin/ninja-build*.
A questo punto, lanciando il comando *ninja* dalla cartella build dovrebbe partire la compilazione.

17.1.3 Realizzare Applicazione C++ con BoringSSL

Creare ed editare i file (.h e .cc) della propria applicazione nella cartella indicata dal percorso: *boringssl/ssl/test*.
Modificare il file *boringssl/ssl/test/CmakeLists.txt* inserendo le regole utili al generatore Ninja di CMake di ricreare il file *build.ninja* in modo da poter compilare la propria applicazione.

```
19 add_executable(
20   dtls_server          (1)
21
22   dtls_server.cc       (2)
23   dtls.cc
24 )
25
26 add_dependencies(dtls_server global_target)
27
28 target_link_libraries(dtls_server test_support_lib ssl crypto)
29
```

Figura 17.1: Definizione Nuovo Eseguibile con CMake

1. Nome dell'eseguibile da creare
2. Elenco dei file .cc che compongono l'applicazione

Tornare nella cartella build (*cd ../../build*) e lanciare i comandi indicati nella sezione compilazione.
Terminata la compilazione si deve trovare l'eseguibile creato nel percorso *boringssl/build/ssl/test*. Per eseguire, nel nostro caso lancio il comando *./dtls_server* dal percorso sopra indicato.

17.2 WebRTC

Nel decimo capitolo è stata affrontata la costruzione di un'applicazione all'interno della libreria WebRTC, qui di seguito saranno elencati i passi chiave necessari alla realizzazione dell'applicazione.

1. Accedere alla directory *workspace/webrtc-checkout/src/examples* della libreria

2. Creare una cartella con un nome significativo, in quanto dovrà essere utilizzato più avanti, e all'interno inserire il/i file sorgenti. Nel nostro caso il sorgente è dato da un file *main.cc*

3. A questo punto spostarsi nella directory relativa ai file generati dalla build della libreria *workspace/webrtc-checkout/src/out/****/*. L'ultima directory dipenderà dal nome dato in fase di build della libreria, nel nostro caso la cartella si chiama */Modified*. All'interno della suddetta directory sono presenti i file generati da gn e Ninja per la compilazione della libreria, e sarà lì che noi dovremmo apportare modifiche al fine di dare le dirette istruzioni a Ninja per compilare il/i nostro/i sorgente/i.

4. A questo punto sarà necessario accedere alla cartella *workspace/webrtc-checkout/src/out/Modified/obj/examples* dove andremo ad inserire un file da noi creato con estensione *.ninja* che definirà le regole di compilazione del nostro eseguibile. Qui di seguito è riportato come esempio il file *test_simple_datachannel.ninja*

5. Dopo di che sarà necessario tornare alla directory precedente, cioè *workspace/webrtc-checkout/src/out/Modified/obj*, e lì andremo a modificare il file *toolchain.ninja* per far si che il file creato al punto 4 venga correttamente letto e in fase di compilazione. Per fare questo lavoro sarà necessario specificare alla riga 247 il nome associato all'eseguibile, definito nel file *.ninja* del punto 4, di seguito al comando di build. Infine, sarà necessario specificare dove il compilatore dovrà andare a prendere il file .ninja tramite il comando *subninja*. Come per il punto precedente di seguito è riportato come esempio quello relativo a *test_simple_datachannel*

6. Infine sarà necessario tornare alla directory *workspace/webrtccheckout/src/* e lanciare il comando *../../depot_tools/ninja -C out/Modified/* e verificare che tutto sia stato compilato correttamente

17.3 Ambiente di Test

Nel dodicesimo capitolo è stata affrontata la creazione di un'ambiente di test per le applicazioni della libreria WebRTC, qui di seguito saranno elencati i passi chiave necessari alla realizzazione dell'ambiente.

17.3.1 Creazione Macchine Virtuali

Per prima cosa sarà necessario lanciare VirtualBox e creare tre macchine virtuali con le seguenti caratteristiche. Le prime due, VM1 e VM2, avranno le seguenti caratteristiche:

1. Sistema Operativo: Ubuntu
2. Memoria RAM: Almeno 3 GB
3. Tipo di Archiviazione: Disco virtuale a dimensione fissa
4. Dimensione del Disco: Almeno 20 GB

Mentre l'ultima, il Router, avrà:

1. Sistema Operativo: Ubuntu
2. Memoria RAM: Basta 1 GB
3. Tipo di Archiviazione: Disco virtuale a dimensione fissa
4. Dimensione del Disco: Il minimo possibile

Sulle prime due dovrà essere installato Ubuntu 16.04 LTS versione Desktop, mentre nell'ultima la versione Server. In fine su ogni macchina dovranno essere presenti, per comodità e necessità, i seguenti applicativi:

1. openssh-server
2. man
3. manpages-dev
4. nano

17.3.2 Configurazione Reti

La VM peer1 presenterà due Network Adapters, il prima di tipo NAT e l'altro di tipo Internal Network denominata neta.
Le VM peer2 e router invece presenteranno tre Network Adapter, la prima sempre di tipo NAT mentre le altre due di tipo Internal Network denominate rispettivamente neta e netb.

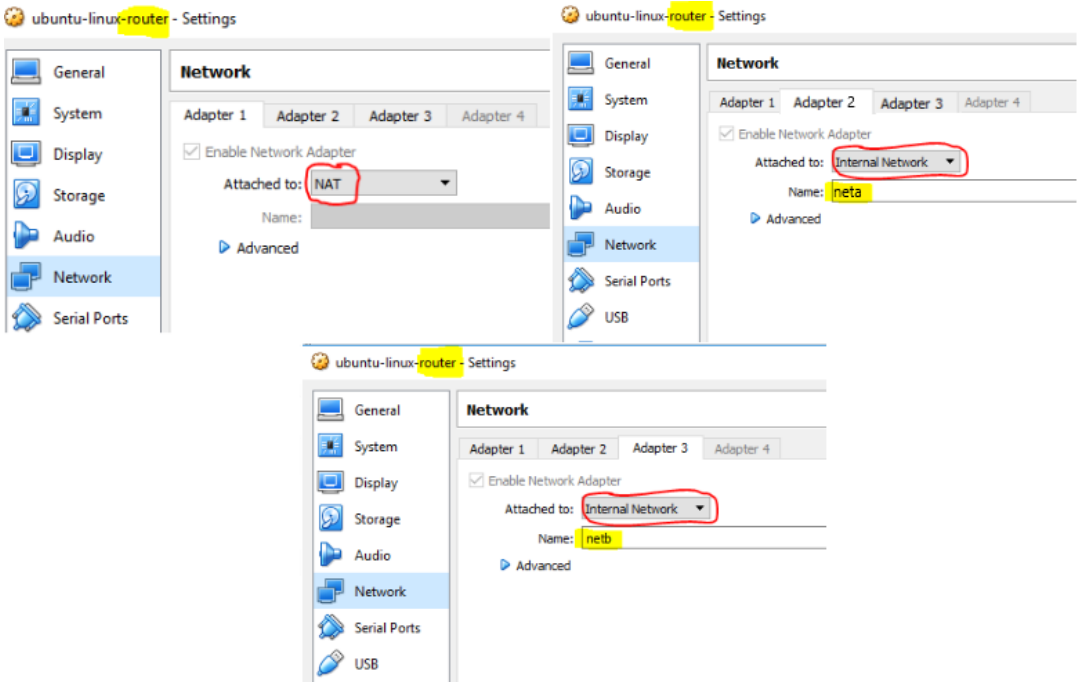

Figura 17.2: Router VirtualBox Network Settings

Per ogni VM dovrà essere modificato il file /etc/network/interfaces come mostrato nel Capitolo 12. Infine, per consentire al Router di funzionare correttamente è stato abilitato il forwarding de commentando la seguente riga *net.ipv4.ip_forward=1* nel file */etc/sysctl.conf*. Una volta svolte queste operazioni la situazione che ci si presenta è riassumibile in Figura 12.1.