ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

TECNOLOGIE WEB T

STUDIO E ANALISI DELLE PRESTAZIONI DI APPLICAZIONI BASATE SU PROTOCOLLI HTTP/2 E WEBSOCKET

CANDIDATO:
Giacomo Cavicchioli

RELATORE:
Chiar.mo Prof. Ing. Paolo Bellavista

CORRELATORE:
Dott. Ing. Carlo Giannelli

Anno accademico 2015/16

Sessione II

Parole Chiave:

- Sistemi Distribuiti
- Applicazioni Web
- HTTP/2
- HTTP/1.1
- WebSocket

Indice

INTRODUZIONE

Internet ha rivoluzionato le comunicazioni a livello globale come mai prima. La quotidianità delle popolazioni più civilizzate ha subito cambiamenti radicali e lo sviluppo di quelle meno avanzate ha trovato in Internet un prezioso strumento di crescita. Oggi, termini come “http://www.unibo.com” o “giacomo@unibo.it” sono diventati di uso comune, trovando un riscontro che non necessita di ulteriori spiegazioni per l’interlocutore medio. Internet rappresenta oggi uno degli esempi meglio riusciti dei benefici degli investimenti sostenuti, dell'impegno per la ricerca e dello sviluppo delle infrastrutture di informazione, nonché una partnership vincente tra l'industria e il mondo accademico per la continua evoluzione e distribuzione di questa nuova tecnologia.

Nel corso degli ultimi anni numerose sono le tecnologie per il Web che hanno contribuito allo sviluppo di Internet, alcune di queste hanno avuto un tempo di vita ridotto, altre sono ancora presenti fin dalla nascita del World Wide Web. Un esempio è sicuramente HTTP, il *protocollo di Internet*, il punto cardine che funge da fondamenta permettendo la comunicazione tra client e server sul web.

Questo lavoro di tesi focalizzerà l’attenzione principalmente sulla nuova versione di HTTP e su un protocollo più recente, WebSocket, ormai inserito radicalmente in molti contesti di sviluppo grazie alle caratteristiche che vanno a risolvere numerose problematiche legate alla natura di HTTP. Nei primi due capitoli verranno analizzati rispettivamente HTTP/2 e WebSocket, ponendo l’attenzione in un primo momento agli standard RFC e successivamente allo stato dell’arte, industriale e accademico, dei due protocolli. L’elaborato proseguirà poi con un terzo e ultimo capitolo di analisi concreta, basata su un’implementazione personale, sulle varie performance che HTTP/2 e WebSocket presentano, ponendo particolarmente l’attenzione su due tecniche differente ma complementari, quali tecnica a Polling e tecnica a Push.

1 PROTOCOLLO HTTP/2

Il World Wide Web venne proposto per la prima volta al CERN di Ginevra da un ricercatore britannico, Tim Berners-Lee. Inizialmente la tecnologia sarebbe dovuta servire semplicemente per condividere informazioni all'interno del centro di ricerca in modo da risolvere un problema già notato da Berners-Lee e colleghi: "*In those days, there was different information on different computers, but you had to log on to different computers to get at it. Also, sometimes you had to learn a different program on each computer. Often it was just easier to go and ask people when they were having coffee…*".

La soluzione prevedeva l'utilizzo di una tecnologia emergente chiamata Hypertext per la condivisione di documenti statici correlati da collegamenti ipertestuali serviti su una rete interna (lo sviluppo di Internet stava facendo enormi passi in quegli anni) grazie a un protocollo semplice, leggero ed affidabile. La proposta di Berners-Lee contrariamente a quanto si possa credere non venne accettata immediatamente: è oggi famosa la citazione "*Vague, but exciting*" del responsabile del dipartimento dello studioso britannico in risposta alla proposta [1].

Un anno più tardi, il 12 Novembre 1990, Tim Berners-Lee assieme a Robert Calliau presentò una proposta più formale per un sistema ipertestuale basato su tecnologia Cliente-Servitore, dove i tre punti cardine sono tutt'ora alla base del web utilizzato ai giorni nostri:

1. **HTML**: HyperText Markup Language, il **linguaggio** con cui venivano scritti i documenti online;
2. **URI**: Uniform Resource Identifier, un **identificatore** che rappresenta una risorsa in modo univoca sulla rete;
3. **HTTP**: HyperText Transfer Protocol, il **protocollo** che serviva per scambiare le risorse desiderate tra il cliente e il servitore.

Il primo sito Web vero e proprio fece la sua comparsa il 6 agosto 1991 esclusivamente per scopi accademici, all'interno del CERN stesso. La tecnologia diventò di dominio pubblico solamente qualche anno più tardi, il 30 Aprile 1993, grazie alla decisione di Berners-Lee e del CERN stesso di renderlo disponibili pubblicamente royalty-free [2].

Un concetto chiave alla base dello sviluppo del World Wide Web, nonché argomento di questo documento di tesi, è il protocollo su cui esso si basa: HTTP. La semplicità e l'affidabilità con cui questo protocollo riesce a servire risorse Web tra Clienti e Servitori è stato indubbiamente l'ingrediente chiave per lo sviluppo esponenziale del Web.

Dalla prima proposta avanzata da Tim Berners-Lee all'interno del CERN vi sono state diverse versioni di HTTP: HTTP/0.9, HTTP/1.0, HTTP/1.1 (con e senza pipeline) ed infine HTTP/2. Per semplicità, dal momento che l'argomento di tesi è l'ultima versione, HTTP/2, ci concentreremo solamente su due delle versioni precedenti, le principali, HTTP/1.0 e HTTP/1.1 per creare il giusto background in modo da rendere l'introduzione a HTTP/2 più chiara.

1.1 PROTOCOLLI HTTP/1.0 E HTTP/1.1

La prima versione standardizzata di HTTP è HTTP/1.0, un protocollo applicativo request/response stateless e *one-shot*. Sia la comunicazione in circolo dal client al server che quella dal server al client viene effettuata utilizzando uno stream TCP sottostante. Lo schema con il quale vengono richieste le risorse fa capo genericamente al seguente:

1. **server**: rimane in ascolto (*server passivo*) tipicamente sulla porta 80 in casi di protocollo HTTP o 443 in caso di protocollo HTTPS;
2. **client**: apre una connessione (*client attivo*) sulla porta 80 (o 443) del server. La porta locale sul client da cui parte la connessione non è importante in quanto una connessione TCP viene identificata in maniera univoca dalla quaterna: `<client-ip:client-port, server-ip: server-port>`;
3. **server**: accetta la connessione;
4. **client**: manda una richiesta;
5. **server**: invia la risposta e chiude la connessione.

HTTP/1.0 viene definito come un protocollo request/response in quanto il client dopo aver aperto la connessione invia una **request** per una determinata risorsa e il server risponde con una **response** contenente la risorsa desiderata. Ogni messaggio scambiato con protocollo HTTP, qualunque sia la natura, viene composto da due parti:

- **Header**: contente metadata sul messaggio;
- **Payload**: a volte assente, contente la risorsa vera e propria.

1.1.1 DIFFERENZE TRA HTTP/1.0 E HTTP/1.1

Il principale svantaggio presente in HTTP/1.0 è sicuramente il fatto di dover chiudere la connessione TCP subito dopo avere ricevuto una risorsa, per poi doverne successivamente riaprire una nuova anche se il destinatario, il server, è il medesimo. Con HTTP/1.1 uno dei primi accorgimenti che venne preso fu proprio il fatto di poter sfruttare una stessa connessione TCP anche per una serie di request e la corrispondente serie di response. Questo è reso possibile grazie al server, che nel casso offra supporto per HTTP/1.1, dopo aver inviato la risorsa lascia aperta la connessione TCP rimanendo in ascolto per altre richieste.

Se ad esempio si facesse una richiesta per una pagina HTML contenente 10 immagini con HTTP/1.0 ogni risorsa avrebbe avuto bisogno di una connessione TCP dedicata (1 per la pagina HTML, 10 per le immagini), mentre con HTTP/1.1 è invece necessaria una sola connessione. Il server HTTP dopo aver inviato le risorse al client, si preoccupa di chiudere la connessione solamente se all'interno dell'header del client è presente questa volontà. Se nell'header invece non compare lo specifico record il server attende un tempo di time-out prestabilito per poi chiudere comunque la connessione.

Oltre alle capacità sopracitate in una versione di HTTP/1.1 è presente anche la possibilità di effettuare **pipelining**: in questo caso la comunicazione consiste nell'invio di molteplici request da parte del client prima di terminare la ricezione delle response. Sarà poi compito del server restituire le rispettive response nell'ordine di ricezione.

1.2 DA SPDY AD HTTP/2

SPDY, pronunciato “SPeeDY”, è un protocollo network sviluppato principalmente da Google, Inc. per la trasmissione di contenuti Web in *realtime* [3]. Il protocollo agisce manipolando il traffico HTTP, inserendosi come intermediario con la funzione di ridurre il tempo di latenza, di ridurre lo spreco di banda e di migliorare la sicurezza.

SPDY utilizza diverse tecniche per adempiere al suo scopo, in particolare:

- permette al client e al server di comprimere gli header di request e di response. Questa tecnica permette di ridurre drasticamente la banda utilizzata quando header simili (come per esempio i cookie) vengono scambiati ripetutamente;
- permette contemporaneamente richieste multiple multiplexed[1] su una singola connessione. Queste tecnica rende efficace la gerarchia tra le risorse in quanto una risorsa prioritaria in questo modo non può bloccare una meno indispensabile;
- permette al server di fare il push di risorse da parte del server verso i client senza dover attendere che il client faccia la relativa richiesta. In questo modo il server riesce a sfruttare banda inutilizzata e ad eliminare eventuali tempi di latenza dovuti al polling.

Sotto una particolare ottica si può dire che SPDY è il vero predecessore di HTTP/2, in quanto molte delle peculiarità di questo protocollo appartengono anche ad HTTP/2. Dopo aver valutato l’idea di procedere con una nuova versione del protocollo, SPDY/2, si è ritenuto opportuno implementare il nuovo protocollo basandosi sullo standard HTTP, formando quindi HTTP/2. Con il passaggio da SPDY ad HTTP/2 molti sono i cambiamenti, specialmente dovuti a discussioni all’interno di Working Group dedicato o a feedback dagli enti implementativi. Durante il processo di passaggio sono molti i tecnici e gli sviluppatori che sono passati da un gruppo di lavoro all’altro, inclusi Mike Belshe e Roberto Peon, i due creatori di SPDY. Nel Febbraio 2015 Google ha annunciato di voler rimuovere il supporto a SPDY in favore di HTTP/2 [4].

1.3 CARATTERISTICHE PRINCIPALI DI HTTP/2

Le principali caratteristiche di HTTP/2 rispecchiano parzialmente diverse soluzioni optate da SPDY per risolvere alcuni problemi con le versioni precedenti di HTTP. Le principali *features* del protocollo sono [5]:

- **HTTP/2 è binario invece che testuale**: i protocolli binari sono per definizione più facili da trattare in termini di parsing e ovviamente più compatti durante la trasmissione. Fondamentale il fatto di essere completamente meno error-prone,

[1] **Multiplexed**: termine derivante dalle telecomunicazioni, è il meccanismo per cui più canali trasmissivi in ingresso condividono contemporaneamente la stessa connessione, combinando flussi di dati in un solo segnale trasmesso in uscita

per esempio in HTTP/1.x[1] vi sono quattro metodi per effettuare il parsing [6], in HTTP/2 solamente uno. Ovviamente questa caratteristica può influenzare ogni operazione collegata al debugging: se con i protocolli HTTP/1.x bastava aprire una sessione telnet sulla porta 80 e richiedere una risorsa (`GET / HTTP /1.1` ad esempio) per ricevere la risorsa web in chiaro sul proprio terminale, con HTTP/2 non è più possibile dal momento che si riceveranno i byte codificati della risorsa. Sono già nati però alcuni strumenti per aggirare questo problema, due esempi possono essere Wireshark 2.0 e un tool fornito da Rebecca Murphey, chrome-http2-log-parser [7]

- **HTTP/2 è multiplexed**: HTTP/1.x ha sempre avuto fin dalla nascita alcuni problemi in fatto di multiplexing: solamente una richiesta poteva risiedere su una connessione. Vi sono stati alcuni tentativi di migliorare il protocollo sotto questo punto di vista con la versione con pipeline con HTTP/1.1, ma con scarsi risultati: una response troppo lenta o pesante poteva rallentare l'intera connessione, oltre a svariati problemi con gli intermediari della connessione (proxy). Con HTTP/2 questi problemi vengono risolti dalla capacità del protocollo di essere completamente multiplexed, quindi in grado di gestire più request/response sulla stessa connessione contemporaneamente.
- **HTTP/2 utilizza una sola connessione TCP**: Con HTTP/1.x ogni richiesta per una risorsa Web apre all'incirca dalle 4 alle 8 connessioni per ogni origin[2] e dal momento che ogni risorsa utilizza diverse origin questo potrebbe significare una trentina di connessioni per ogni request. Questo chiaramente va contro le assunzioni su cui si basa TCP. Per questo motivo con il protocollo HTTP/2 si è voluto fare in modo che una request venisse servita esclusivamente da una singola connessione TCP, come mostrato nella Figura 1.

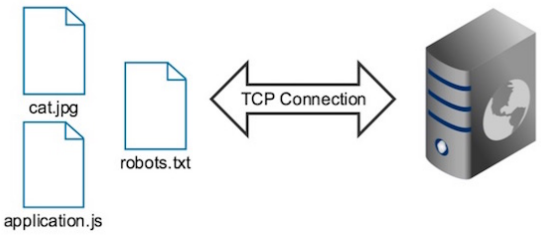


FIGURA 1 UNA SINGOLA CONNESSIONE TCP PER ORIGIN

- **HTTP/2 permette il push delle risorse**: Quando un browser richiede una risorsa Web a un determinato server quest'ultimo può spedirla nello stato attuale, senza curarsi se ci saranno o meno cambiamenti e se di conseguenza il client ne debba venire informato. Il push delle risorse, una delle features più acclamate di HTTP/2, risolve questo problema. Essa permette al server, chiaramente previa

[1] Per HTTP/1.x si fa riferimento a qualsiasi versione di HTTP precedente HTTP/2 come ad esempio HTTP/1.0 o HTTP/1.1

[2] **origin**: usato in questo contesto come sinonimo di client

una certa procedura di negoziazione, di inviare una certa risorsa web a un client senza che quest'ultimo debba effettuare alcuna request. Il vantaggio di questa tecnica è fondamentale dal momento che il client può ricevere "aggiornamenti" delle risorse precedentemente richieste andando così ad eliminare tecniche poco efficienti usate in passato, come ad esempio il polling a intervalli da parte del client.

- **HTTP/2 comprime l'header**: Come ha dimostrato Patrick McManus, network developer a Mozilla, se si assume che ogni pagina contenga 80 assets [1] (assunzione conservativa oramai ai giorni nostri) e ognuna di queste abbia 1400 bytes di header (di nuovo non particolarmente strano grazie per esempio ai Cookie), si potrebbe impiegare come minimo 7-8 round trips per ottenere gli header "on the wire", senza contare il tempo della response, solamente per inviare i dati dal client. Questo effetto è causato dal meccanismo di Slow Start di TCP, il quale limita I pacchetti spediti per i primi round-trip. Il compenso grazie alla compressione dell'header si potrebbe ottenere la spedizione e la relativa ricezione degli header solamente in un round-trip. Questo overhead è da tenere in considerazione, soprattutto considerando l'avvento di browser sui client mobile dove il tempo di latenza impiega tempi considerevoli che potrebbero andare a deteriorare l'intera UX[2]. In HTTP si è scelto di utilizzare come meccanismo di compressione HPACK [8], si veda la successiva sezione per approfondimenti.

1.4 HPACK – HEADER COMPRESSION PER HTTP/2

In HTTP/1.1 l'header HTTP non veniva compresso, ma con il passare degli anni sempre più record venivano aggiunti causando un sostanziale consumo di banda nello scambio di request/response tra client/server. Il protocollo SPDY per primo cercò una soluzione iniziando a utilizzare i metodi di compressione DEFLATE [9] e GZIP rispettivamente per la versione 1 e 2 del protocollo di Google, approcci sicuramente migliori da un punto di vista prestazionale e di semplicità di implementazione, ma peggiori per quanto riguardava la sicurezza, come è stato analizzato dopo l'attacco CRIME [10]. Con CRIME un attaccante, oltre che intercettare una connessione leggendo in chiaro tutti i record HTTP, poteva iniettare dati personali compromettendo potenzialmente l'intera connessione.

Con l'avvento di HTTP/2 risultò quindi evidente la necessità di un nuovo metodo di compressione per gli header HTTP. Venne creato così HPACK. Un sistema nuovo, sicuro, in grado di comprimere ogni record dell'HTTP header intendendolo come un

[1] **assets**: risorsa generica contenuta nella pagina, come per esempio immagini, file .css o .js

[2] **User Experience**: si intende ciò che un utente prova quando utilizza un prodotto, un sistema o un servizio. L'esperienza d'uso concerne gli aspetti esperienziali, affettivi, l'attribuzione di senso e di valore collegati al possesso di un prodotto e soprattutto all'interazione con esso, include anche le percezioni personali su aspetti quali l'utilità, la semplicità d'utilizzo e l'efficienza del sistema.

semplice ottetto rappresentato da una coppia nome-valore. Un header quindi viene visto come una serie di record dove possono presentarsi duplicati e dove l'ordine di questi ultimi viene preservato con la compressione e la decompressione. Ogni record viene poi quindi convertito con il **metodo di Huffman**, algoritmo per compressione lossless[1] già utilizzato per altri scopi simili, sviluppato da David A. Huffman durante il suo dottorato presso il MIT e pubblicato successivamente nel 1952 nel paper "A Method for the Construction of Minimum-Redundancy Codes" [11].

1.5 CREAZIONE CONNESSIONE HTTP/2

Una connessione HTTP/2 giace interamente al di sopra di una singola connessione TCP, rispettando il particolare protocollo. Come HTTP/1.x anche HTTP/2 utilizza gli schemi classici di URI (http:// o https://) e le stesse porte (rispettivamente 80 o 443). Sarà quindi compito dell'implementazione, al momento della richiesta di una certa risorsa, come ad esempio `http:www.example.com/foo` o `https://www.example.com/bar`, verificare se il server supporta o meno HTTP/2.

HTTP/2 prevede due identificativi di versione:

- La stringa `h2` identifica il protocollo quando la connessione sottostante utilizza il Transport Layer Security (TLS);
- La stringa `h2c` identifica il protocollo quando la connessione sottostante non utilizza TLS;
- La `c` che differenzia questo identificativo dal precedente sta per cleartext.

Si analizzerà ora come inizializzare una connessione con HTTP/2, distinguendo due casi in base alla presenza o meno del TLS

1.5.1 INIZIALIZZAZIONE CONNESSIONE CON HTTP

Un client che esegue una richiesta HTTP generica, senza prima avere la certezza se il server che sta interrogando supporta o meno HTTP/2, procederà seguendo lo schema definito dall'*HTTP Upgrade Mechanism*: il client procederà eseguendo una request con protocollo HTTP/1.1 ma contente nell'header un campo contenente un `h2c` token, per esempio:

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

[1] Una compressione **lossless** è una generica compressione di dati che non porta a nessun perdita di informazioni durante la fase di compressione/decompressione

In questo modo il server potrà rispondere in maniera differente a seconda che esso supporti o meno HTTP/2. In particolar modo, se non supporta HTTP/2 la response sarà simile alla seguente:

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html
. . .
```

Ignorando così il token h2c. Se invece il server supporta HTTP/2 la risposta avrà un *response-code* pari a 101 (Switching-Protocols) e sarà simile alla seguente:

```
HTTP/1.1 101 Switching Protocols
Content: Upgrade
Upgrade: h2c

[HTTP/2 connection . . .
```

In questo modo il client riceverà sia la response della request con protocollo HTTP/1.1, sia la response alla risorsa richiesta con il nuovo protocollo. Grazie a questo meccanismo non sarà necessario nessun invio di acknowledgement da parte del server, dal momento che la response 101 possiede già il concetto in maniera intrinseca.

1.5.2 Inizializzazione connessione con HTTPS

Il procedimento per stabilire una connessione con TLS è completamente analogo al precedente, con l'unica differenza che invece che il token `h2c` sarà necessario inviare un token `h2`.

1.5.3 Connection Preface

Un client potrebbe essere già a conoscenza del fatto che un server possa supportare HTTP/2, per connessioni passate o per altri motivi di implementazione. In tal caso subito dopo la request con protocollo HTTP/1.x verrà eseguita quella con protocollo HTTP/2 senza attendere la response della prima, in modo da ridurre la latenza. Il server è comunque tenuto a mandare una **Connection Preface** all'inizio della comunicazione su protocollo HTTP/2 nonostante sia già in arrivo l'apertura della connessione sul nuovo protocollo da parte del client. La Connection Preface consiste in una stringa di 24 ottetti prestabilita, seguita da un frame Settings, eventualmente vuoto, che dovrà poi essere "confermato" dal client attraverso un acknowledgement. Si può considerare questa procedura come una specie di handshake dove si vanno a negoziare e stabilire i parametri che la connessione successivamente dovrà seguire. Il client sarà tenuto per tutta la durata di vita della connessione a onorare questi parametri e in caso contrario si potrebbero verificare errori che in alcuni casi potrebbero interrompere interamente la connessione.

1.6 Frame HTTP/2

I Frame sono l'entità base sulla quale ogni connessione HTTP/2 si basa, sono formati da un header di lunghezza costante e da un payload di lunghezza variabile.

1.6.1 HEADER DI UN FRAME HTTP/2

Ogni header è composto da 9 ottetti e precede il payload contenente i dati veri e propri. La Figura 2 schematizza i vari campi mettendo in risalto quanti byte occupa ognuno di questi.

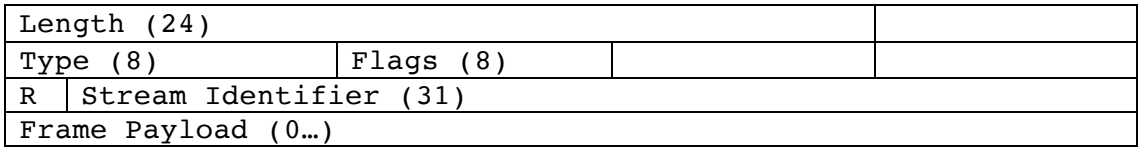


FIGURA 2 FRAME LAYOUT

I vari campi dell'header di un frame HTTP/2 sono i seguenti:

- **`Length`**: la lunghezza del payload contenuto nel frame espressa con un unsigned 24-bit integer. I valori più grandi di 2^14 (16,384) non devono essere mandati a meno che il server ricevente non abbia impostato un valore diverso da quello di default nelle impostazioni con il campo `SETTINGS_MAX_FRAME_SIZE`;
- **`Type`**: la tipologia del frame, espresso con 8 bit. Questa tipologia determina il formato e la semantica del frame intero. Le implementazioni devono rifiutare ogni frame contenente una tipologia non riconosciuta. Per maggiori dettagli si veda il capitolo **Error! Reference source not found.** per la definizione dei singoli frame;
- **`Flags`**: un campo a 8-bit riservato per booleani specifici per la tipologia di frame;
- **`R`**: campo a singolo bit riservato per future specifiche, deve rimanere non settatto (0x0) e quando ricevuto deve essere ignorato;
- **`Stream Identifier`**: identifier dello stream espresso da un intero unsigned a 31-bit. Il valore 0x0 è riservato per frame associati alla connessione stessa.

1.6.2 TIPOLOGIE DI FRAME

L'RFC ufficiale di HTTP/2 definisce una serie di tipologie di frame, ognuna identificata da un codice a 8 bit unico. Verranno in seguito elencati ed illustrati.

- **`Data Frame`**: ha come codice 0x0 e vengono utilizzati per trasportare il payload di request o response HTTP. Deve essere associato a uno stream e in caso contrario scatena un connection error;
- **`HEADER Frame`**: ha come codice 0x1 e viene utilizzato per aprire uno stream. È utilizzabile esclusivamente su uno stream nello stato: idle, reserved (local), open o half-closed (remote), come viene indicato nel capitolo 1.7.1;
- **`PRIORITY Frame`**: ha come codice 0x2 e viene utilizzato per richiedere, da parte del mittente, la priorità per un determinato stream. Può essere utilizzato in qualsiasi stato lo stream si trovi, nel caso questo si trovasse nello stato di closed o idle sarebbe da intendere come una re-prioritizzazione degli stream collegati;
- **`RST_STREAM Frame`**: ha come codice 0x3 e viene utilizzato per terminare immediatamente uno stream in seguito al verificarsi di un errore. Nel payload contiene un singolo unsigned int a 32 bit contenente il codice dell'errore che ha causato la chiusura dello stream;

- **`SETTINGS Frame`**: ha come codice 0x4 e viene utilizzato per configurare i parametri che condizionano la comunicazione tra gli endpoint. Alcuni dei parametri negoziabili più importanti sono:
    - **`SETTINGS_ENABLE_PUSH`**: serve per disattivare o attivare il push da parte del server. Nel caso fosse disattivata e il server mandasse comunque una promise si dovrebbe scatenare un `PROTOCOL_ERROR`. Di default l'impostazione è attivata;
    - **`SETTINGS_MAX_CONCURRENT_STREAMS`**: serve per indicare il numero massimo di stream concorrenti che il server può garantire. Non viene indicato alcun limite, ma viene consigliato di non superare 100;
    - **`SETTINGS_MAX_FRAME_SIZE`**: serve per indicare la dimensione massima di un frame, nel caso venisse superato questo valore verrebbe scatenato un `PROTOCOL_ERROR`. Il valore massimo iniziale è 2^14 e può essere aumentato fino a un massimo di 2^24-1;
- **`PUSH_PROMISE Frame`**: ha come codice 0x5 e viene utilizzato per notificare l'endpoint peer in anticipo riguardo a successive push di frame da parte del server;
- **`PING Frame`**: ha come codice 0x6 e viene utilizzato per verificare il round-trip minimo dal sender al receiver, oltre ovviamente al controllare che il ricevente sia attivo e funzionante;
- **`GOAWAY Frame`**: ha come codice 0x7 e viene utilizzato per avviare la chiusura della connessione o per segnalare gravi errori su di essa;
- **`WINDOW_UPDATE Frame`**: ha come codice 0x8 e viene utilizzato il flow control dello stream (si veda il capitolo 1.7.2). Quest'ultimo come indicato nel capitolo dedicato agisce su due livelli: su un individuo stream o sull'intera connessione;
- **`CONTINUATION Frame`**: ha come codice 0x9 e viene utilizzato per notificare la volontà continuare a spedire altri blocchi dell'header.

1.7 STREAMS E MULTIPLEXING

Uno *stream* è una sequenza indipendente di scambio di frame su un canale bidirezionale tra due endpoint. Nel nostro caso uno stream farà riferimento a un canale bidirezionale basato su una connessione HTTP/2. Ogni stream ha particolari caratteristiche:

- una connessione HTTP/2 può contenere diversi stream aperti in maniera concorrente;
- ogni stream, dopo essere stato aperto, può essere utilizzato unilateralmente o condiviso tra client e server;
- ogni stream può essere potenzialmente chiuso da entrambi gli endpoint;
- l'ordine in cui i frame vengono inviati su uno stream è significativo, in particolare l'ordine degli header e dei payload deve essere processato nello stesso ordine con cui vengono inviati;
- ogni stream è identificato da un intero assegnato dall'endpoint che prende l'iniziativa e apre la connessione.

1.7.1 STREAM STATES

Il lifecycle generico di uno stream è rappresentato nella Figura 3:

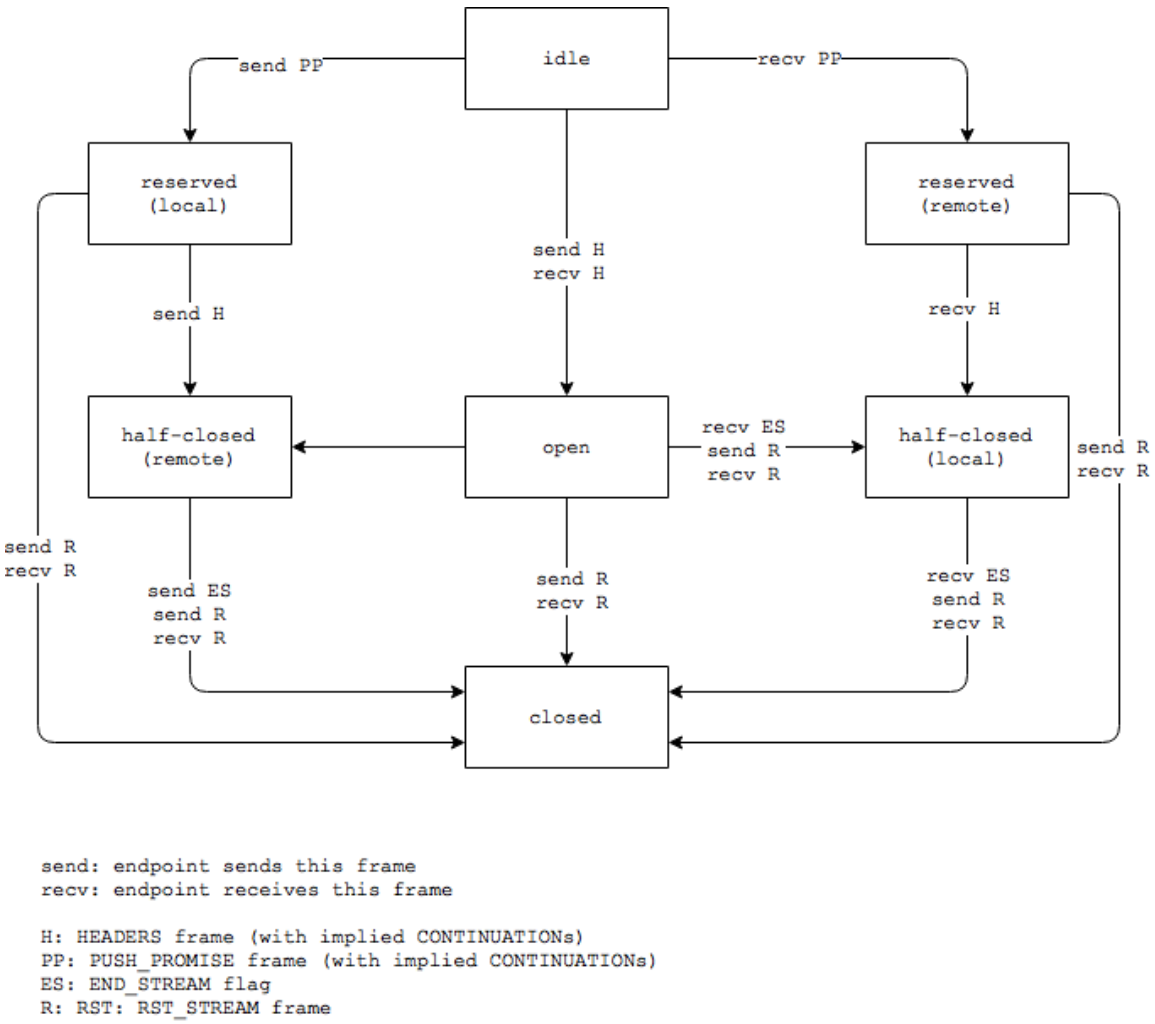


FIGURA 3 STREAM LIFECYCLE

Verranno analizzati ora tutti gli stati.

- **Idle**: ogni stream parte nello stato di idle. Da questo stato le transizioni valide possono essere:

    - L'invio o la ricezione di un frame `HEADER` causa l'apertura dello stream, portandolo nello stato **open**;
    - L'invio di un frame `PUSH_PROMISE` porta lo stream nello stato di **reserved (local)**;
    - La ricezione di un frame `PUSH_PROMISE` porta lo stream nello stato di **reserved (remote)**;
- **Reserved (local)**: uno stream arriva in questo stato perché gli è stata "promessa" una push. Le possibili transizioni possono essere:
    - L'endpoint manda un `HEADER` frame, rendendolo **half-closed (remote)**;
    - Entrambi gli endpoint possono inviare un frame `RST_STREAM` che rende l'intero stream **closed**;
- **Reserved (remote)**: uno stream in questo stato è stato riservato per un peer remoto. Le possibili transizioni sono:
    - La ricezione di un frame `HEADER` porta lo stream in una posizione **half-closed (local)**;
    - Entrambi gli endpoint possono inviare un frame `RST_STREAM` che rende l'intero stream **closed**;
- **Open**: uno stream in questo stato può essere utilizzato da entrambi gli endpoint per inviare frame di qualsiasi tipo. In questo stato entrambi gli endpoint possono mandare un frame con attivato il flag `END_STREAM`, in modo da portare lo stream in uno dei due stati half-closed:
    - La ricezione di un frame `END_STREAM` porta nello stato **half-closed (remote)**;
    - L'invio di un frame `END_STREAM` porta nello stato **half-closed (local)**;
- **Half-closed (local)**: in questo stato uno stream si porta nello stato **closed** alla ricezione di un frame `END_STREAM`;
- **Half-closed (remote)**: uno stream in questo stato non viene più usato dal peer per l'invio di frame;
- **Closed**: questo è lo stato terminale del lifecycle di uno stream. Ogni frame ricevuto deve scatenare un errore di tipo `STREAM_CLOSED`.

1.7.2 Flow Control

L'utilizzo di stream introduce chiaramente dei possibili conflitti nell'utilizzo della connessione TCP sottostante, con il rischio di bloccare lo stream se la connessione si satura. Un particolare metodo chiamato **Flow Control**, offerto da HTTP/2 grazie ai frame `WINDOWS_UPDATE`, si preoccupa che più stream sulla stessa connessione TCP non interferiscano tra di loro, in modo da non creare conflitti.

Alcune delle caratteristiche più importanti del Flow Control sono:

- il Flow Control è caratteristico di una singola connessione. Sia i Flow Control di uno stream, sia quelli della connessione hanno due singoli endpoint, vale a dire i due host di un singolo hop, non dell'intera connessione end-to-end;
- il Flow Control è basato sui frame HTTP/2 `WINDOWS_UPDATE`. I riceventi informano i mittenti su quanti ottetti sono disposti (capaci) di ricevere. Questo schema viene detto credit-based scheme;

- il Flow Control è unilaterale, con il controllo principale riservato all'endpoint ricevente. Quest'ultimo potrebbe impostare qualsiasi dimensione della finestra nella `WINDOWS_UPDATE`, i mittenti (che questi siano client, server o semplici intermediari) sono tenuti a rispettare la volontà indicata fino ai propri limiti di risorse;
- il valore iniziale della finestra `WINDOWS_UPDATE` è impostato a 65,535 ottetti;
- il Flow Control viene applicato a seconda del tipo di frame (si veda successivamente per le diverse tipologie), solamente i frame di tipo `DATA` sono soggetti a a essere processati dal Flow Control, in quanto tutti le altre tipologie di frame non occupano risorse di connessione (banda);
- il Flow Control non può essere disattivato;
- l'RFC 7540 di HTTP/2 non impone come e quando la `WINDOWS_UPDATE` debba venire impostata, è compito del client implementare con gli algoritmi che più soddisfano le necessità dell'applicazione.

Il Flow Control è stato creato per proteggere gli endpoint con risorse limitate. Un esempio potrebbe essere quello dei Proxy che si vedono costretti a condividere molta memoria con più connessioni, magari con un upstream limitato rispetto al downstream. In alcuni casi il Flow Control si può disattivare semplicemente impostando la finestra massima con il frame `WINDOWS_UPDATE` (2^31 – 1) e mantenendola con un nuovo invio del frame ogni qualvolta risulti necessario; in questo modo di fatto si disattiva il Flow Control.

1.7.3 Priorità negli Stream

Una delle principali caratteristiche che differenziano HTTP/2 dalle versioni precedenti è la possibilità di dare un *peso* a ogni frame in modo da impostare una priorità di alcuni di essi rispetto ad altri. Per fare questo viene impostato il valore del *peso* nel frame `HEADERS` che apre lo stream e nel caso fosse necessario cambiare il valore dopo che lo stream è stato inizializzato si utilizzerà un frame specifico di tipo `PRIORITY`. La prioritizzazione a differenza dell'indicazione del Flow Control è solamente un suggerimento che l'host può fare al proprio peer, non si può obbligare quest'ultimo a gestire i frame in un determinato ordine.

1.7.4 Gestione degli Errori

HTTP/2 permette due tipologie di errori:

- un errore che rende inutilizzabile l'intera connessione;
- un errore che rende inutilizzabile solamente un singolo stream.

1.7.4.1 Connection Error

Quando un endpoint incontra un errore che condiziona e rende inutilizzabile un'intera connessione per prima cosa deve mandare un frame di tipo `GOAWAY` con lo stream identifier dell'ultimo stream utilizzato e funzionante, contente il motivo dell'errore, successivamente chiudere la connessione TCP.

1.7.4.2 Stream Error

Quando un endpoint incontra un errore che condiziona e rende inutilizzabile uno stream per prima cosa deve mandare un frame di tipo `RST_STREAM` contenente l'identifier dello stream e il codice dell'errore.

1.8 HTTP Message Exchange

HTTP/2 nasce per essere completamente retro-compatibile con le versioni precedenti, per questo da un punto di vista dell'applicazione le features del protocollo sono rimaste quasi interamente immodificate. Per ottenere questo la semantica classica delle request e response e i vari scheme di uri (`http://` e `https://`) è rimasta la stessa che si aveva con i protocolli HTTP/1.x, ma la relativa sintassi è cambiata.

1.8.1 Scambio di Request e Response HTTP

Generalmente un client manda una nuova request utilizzando un stream con un identifier nuovo, non ancora utilizzato. Il server risponde con una response sullo stesso stream. Ogni messaggio HTTP, che sia una request o che sia una response, è composto da:

1. esclusivamente per le response, zero o più frames HEADER contenenti i messaggi dell'header delle risposte di info (1xx);
2. un frame HEADER contenente l'header del messaggio;
3. zero o più frame DATA contenenti il payload del messaggio;
4. opzionalmente uno o più frame HEADER seguito da più frame CONTINUATION con la parte restante del messaggio.

L'ultimo frame dell'intera serie dovrà avere attivo il flag `END_STREAM`.

Uno scambio completo di request e response vede il consumo totale dello stream utilizzato, quindi quando il server invia (o "quando il client riceve", a seconda del punto di vista) un frame contenente il flag `END_STREAM`, con la successiva request si utilizzerà uno stream nuovo.

1.8.2 Compressione dei Cookie

In HTTP/2 è presente ovviamente il supporto ai cookie nell'header HTTP. La differenza rispetto alle precedenti versioni è che per ottenere una migliore compressione viene fatto uno split tra i cookie in modo da ottenerne uno singolo per ogni record dell'header. Per esempio se in HTTP/1.x si poteva avere:

```
cookie: a=b; c=d; e=f
```

In HTTP/2 si avrà la versione analoga ma separata:

```
cookie: a=b
cookie: c=d
cookie: e=f
```

1.8.3 ESEMPI

Verranno in seguito presentati alcuni esempi comparativi tra i protocolli HTTP/1.x e HTTP/2. In particolare verranno analizzate le seguenti situazioni:

- Request con soli header;
- Response con soli header;
- Request con payload;
- Response con payload.

Si noti che per la parte di HTTP/2 è stata inserita la versione testuale dal momento che nello scambio effettivo i frame HTTP/2 sono spediti e ricevuti in formato binario.

1.8.3.1 REQUEST CON SOLI HEADER

HTTP/1.X	HTTP/2

`GET /resource HTTP/1.1`
`Host: example.org`
`Accept: image/jpeg`
	`HEADERS`
	`+ END_STREAM`
	`+ END_HEADERS`
	`:method = get`
	`:scheme = https`
	`:path = /resource`
	`host = example.org`
	`accept = image/jpeg`

Si noti che sono stati settati i flag `END_STREAM` e `END_HEADERS` nel frame, oltre alla convenzione (che se non rispettata potrebbe scatenare un errore di tipo malformed request) delle lettere minuscole per tutti i record dell'header.

1.8.3.2 RESPONSE CON SOLI HEADER

In maniera simile una response che include soli response haders è trasmessa come un frame `HEADER` e la traduzione sarà simile alla seguente:

HTTP/1.X	HTTP/2

`HTTP/1.1 304 Not Modified`
`ETag: "xyzzy"`
`Expires: Thu, 23 Jan`
	`HEADERS`
	`+ END_STREAM`
	`+ END_HEADERS`
	`:status = 304`
	`etag = "xyzzy"`
	`expires = Thu, 23 Jan …`

Come nel caso precedente viene trascritto lo status code (in questo caso d'esempio `304`) in un particolare "header record", senza la traduzione letteraria (`Not Modified`), insieme ai rimanenti (`etag` ed `expires`).

1.8.3.3 REQUEST CON PAYLOAD

Una request HTTP fatta con metodo `POST` che include anche un payload potrebbe essere trasmessa come un frame `HEADER` seguita da zero o più frame `CONTINUATION` con il flag `END_HEADERS` e successivi frame `DATA`, dei quali l'ultimo avrà il flag `END_STREAM`.

HTTP/1.X	HTTP/2
`POST /resource HTTP/1.1` `Host: example.org`	`HEADERS` `- END_STREAM`

```
Content-Type: image/jpeg                    - END_HEADERS
Content-Length: 123                               :method = POST
                                                  :path = /resource
{binary data}                                     :scheme = https

                                          CONTINUATION
                                                + END_HEADERS
                                                  :content-type =
                                          image/jpeg
                                                  :host = example.org
                                                  :content-length =
                                          123

                                          DATA
                                                + END_STREAM
                                                  {binary data}
```

1.8.3.4 RESPONSE CON PAYLOAD

Una response che include sia header che payload è trasmessa con un frame `HEADER` seguito da zero o più frame `CONTINUATION` seguito da uno o più frame `DATA` dei quali l'ultimo nella sequenza avrà il flag `END_STREAM`.

HTTP/1.X	HTTP/2

`HTTP/1.1 200 OK`
`Content-Type: image/jpeg`
`Content-Length: 123`
 	
`{binary data}`
	`HEADERS`
	`- END_STREAM`
	`- END_HEADERS`
	`:status 200`
	`:content-type = image/jpeg`
	`:content-length = 123`
	
	`DATA`
	`+ END_STREAM`
	`{binary data}`

1.8.4 HTTP SERVER PUSH

Il protocollo HTTP/2 permette al server di mandare una "response" (non sarà in realtà una vera response: come si può parlare di una response se non è stato "requested" nulla?) a un client in riferimento a un request precedentemente effettuata da quest'ultimo. Questo metodo può tornare utile quando il server sa che il client ha bisogno di ulteriori informazioni da processare per ultimare la request. Un client può modificare questa funzione sul server attivando o disattivando il flag `SETTINGS_ENABLE_PUSH`. Le *promise*[1] del server devono poter essere salvate nella cache e non devono contenere un request body, nel caso queste due condizioni non fossero rispettate il client che riceve la promise deve mandare uno stream error di tipo `PROTOCOL_ERROR`.

[1] **promise**: letteralmente "promesse", le offerte di push fatte dal server al client

1.8.4.1 *Push Requests*

Quando il server esegue una push verso un client la semantica è identica a una response standard, quindi a una response di una request. La differenza risiede nel fatto che la request in questo caso è eseguita sempre dal server sotto forma di un frame `PUSH_PROMISE`.

La sequenza in questo caso sarà simile alla seguente:

1. il client esegue una request standard per una risorsa contenente link esterni (immagini, file .css o .js…);
2. il server risponde con un frame `PUSH_PROMISE`;
3. il server risponde con il frame `DATA` contenente la risorsa richiesta contenente a sua volta i link;
4. il server esegue la push verso il client con le risorse richieste.

Da notare l'importanza dell'ordine con cui vengono elencati i passaggi: se il server non avesse fatto una `PUSH_PROMISE` prima di aver mandato la risorsa contenente i link esterni il client avrebbe potuto richiederli in maniera standard, rendendo di fatto inutile il meccanismo intero.

Ovvio è che i frame di tipo `PUSH_PROMISE` sono riservati al server: un client non si troverà mai in un contesto in cui dovrà mandare un frame `PUSH_PROMISE`.

Appena prima che il server mandi una `PUSH_PROMISE` lo stream deve essere o in uno stato "open" o in un stato "half-closed (remote)", una volta spedita la `PUSH_PROMISE` invece passerà a "reserved (local)" e "reserved (remote)" rispettivamente per il server e il client.

1.8.4.2 *Push Response*

Dopo aver mandato la `PUSH_PROMISE` il server può quindi iniziare a mandare le `PUSH_RESPONSE` come se fossero standard response, con la differenza che lo stream sarà identificato come inizializzato dal server e sarà "half-closed" per il client dal primo frame `HEADER` spedito dal server.

Quando un client riceve una `PUSH_PROMISE` e decide di accettare le seguenti push provenienti dal server non deve eseguire nessuna request finché il *promised stream* non viene chiuso dal server.

1.8.5 Il metodo CONNECT

In HTTP/1.x lo pseudo metodo `CONNECT` è usato per convertire una connessione HTTP in un *tunnel* tra client e server, principalmente utilizzato con proxy HTTP per stabilire una connessione TLS.

In HTTP/2 il metodo `CONNECT` viene utilizzato in modo analogo per stabilire un tunnel su uno stream HTTP/2, qualora un proxy ricevesse una richiesta per un metodo `CONNECT` da un client. Il suo compito sarà di creare una connessione TCP verso il server ed inoltrare il payload di ogni frame `DATA` ricevuto dal client verso il server sulla connessione appena creata, nel caso il server ricevesse frame di altro tipo deve segnalare uno stream

error. Il tunnel rimane consistente finché il proxy non riceve un frame con il flag `END_STREAM`, il quale oltre a chiudere lo stream (comportamento standard) chiude anche il tunnel sul proxy, che quindi avrà il compito di chiudere la connessione TCP verso il server.

1.9 STATO DELL'ARTE DI HTTP/2

Ogni volta che nasce la necessità di rinnovare un protocollo (ma il concetto potrebbe essere esteso a diversi altri ambiti) è per risolvere problematiche presenti con le versioni attuali. Nella fattispecie il protocollo HTTP nasce nella mente di Tim Berners Lee per servire pagine *semplici*, dove è presente quasi esclusivamente testo. Pagine quindi che non hanno praticamente nulla a che fare con i moderni siti web a cui siamo abituati al giorno d'oggi, dove spesso le risorse testuali rappresentano una ristretta minoranza in confronto a immagini o video, fogli di stile CSS o script JS. Facendo riferimento alla lista di siti web più visualizzati al mondo fornita da Alexa si è raccolto [12] nella Tabella 1 il numero di risorse delle varie tipologie.

	WEBSITE	HTML	CSS	JS	IMAGES
1	Google	2	1	5	4
2	Youtube	7	6	18	45
3	Wikipedia	3	2	5	5
4	Yahoo	4	2	8	38
5	Baidu	9	0	13	16
6	Amazon	4	28	40	45
7	Taobao	7	11	55	93

TABELLA 1 STATISTICHE DEI PRINCIPALI SITI WEB

Ad esempio possiamo notare come il sito web Amazon abbia nella propria home page solamente 4 risorse HTML, mentre un ordine di grandezza in più di risorse CSS/JS fino ad arrivare a 45 immagini.

Risulta quindi evidente come i moderni siti web abbiano poco in comune con una semplice pagina HTML. Di conseguenza un protocollo nato per condividere documenti di quest'ultimo tipo necessita di essere aggiornato, in modo da mantenersi in pari con il materiale da servire. In [13] vengono analizzati, sempre facendo riferimento ai primi 10000 siti secondo Alexa, quali supportino o meno HTTP/2. Nell'ottobre 2012 solamente 208 (2.1%) mentre a distanza di pochi mesi, nell'aprile 2013, già 271 (2.7%). Questi numeri stanno a significare la sensibilità del mondo commerciale all'innovazione del protocollo.

Viene stilata in seguito una lista, Tabella 2, che mette in risalto gli 8 principali[1] siti web che implementano HTTP/2, tralasciando eventuali siti web collegati (come ad esempio si considera facebook.com (#1) ma non fbcdc.net (#202)).

[1] sempre secondo la lista fornita da Alexa

SITE	ALEXA RANK	HTTP/2 RANK	WEB SERVER
FACEBOOK	1	1	OpenCompute
GOOGLE	2	2	Google Web Server
YOUTUBE	3	3	Apache
BLOGSPOT	11	4	Google Servlet Engine
TWITTER	13	6	Twitter Frontend
WORDPRESS	23	8	Nginx
IGMUR	96	24	Nginx
YOUM7	485	65	Nginx

TABELLA 2 HTTP/2 ENABLED WEBSITES

Anche in questo caso si può notare come anche alcuni dei siti web più visitati al mondo non siano HTTP/2 enabled. Questo si può capire dal fatto che man mano che si procede verso il basso si crea una discrepanza tra l'indice generale e l'indice caratteristico HTTP/2.

1.9.1 HTTP/2 È PIÙ VELOCE DI HTTP/1.1?

Una volta consolidata la necessità di un nuovo protocollo risulta spontaneo chiedersi se la versione aggiornata di HTTP risulti effettivamente più performante (che per la natura del contesto risulta sinonimo in parte di *veloce*) del predecessore. Nell'analisi fornita in [12] vengono fatti alcuni test simulando situazioni reali: una rete locale (LAN) e una connessione a Internet broadband. In entrambi i casi servendosi di un client e un server, entrambi compatibili sia con HTTP/1.1 che con HTTP/2, e di un contenuto da servire, clonato grazie a un'estensione di Firefox seguendo la lista offerta da Alexa. Le caratteristiche peculiari delle due tipologie di rete sono state simulate invece utilizzando il Linux Traffic Control.

L'intero esperimento ha delle limitazioni, come sottolineato dagli autori, in quanto non si tiene conto dei seguenti aspetti:

- clonando i siti web e servendoli da un singolo host non si simula il *domain sharding*;
- utilizzando indirizzi IP in chiaro non si ha il passaggio attraverso un server DNS;
- si utilizza un modem 3G collegato direttamente al computer, non avendo quindi limitazioni CPU come si potrebbero avere invece su uno smartphone.

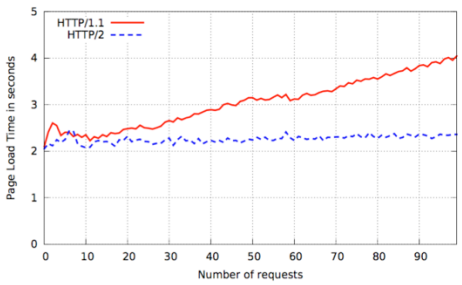


FIGURA 4 TEMPO DI CARICAMENTO VS NUMERO DI REQUEST

L'esperimento conduce a risultati interessanti in quanto è possibile notare in Figura 4 come HTTP/2 sia effettivamente più performante rispetto a HTTP/1.1 soprattutto con siti

web complessi che richiedono l'interazione con un alto numero di risorse, come ad esempio siti web di news o e-commerce; siti web minimalistici invece e più semplici invece non beneficiano dell'upgrade di protocollo. Per quanto riguarda invece la connessione a Internet con il modem 3G risulta evidente come il miglioramento sia nettamente minore: diversi studi su SPDY avevano già messo in risalto come il protocollo non rispondesse in maniera ottimale alla perdita di pacchetti su una rete leggermente instabile come potrebbe risultare una connessione 3G.

Infine si sono testate alcune features innovative presenti in HTTP/2 che non avevano un corrispettivo in HTTP/1.1: la push da parte del server e la prioritizzazione dei frame in un singolo stream.

Il fatto di poter stabilire un comportamento proattivo per l'implementazione del server evidenzia un miglioramento del 10% nel tempo di caricamento delle pagine web. Ad oggi però, come sottolineato dagli autori, non sono presenti algoritmi per la push di pagine web generiche, risulta quindi necessario definire una propria implementazione, caratteristica del proprio caso.

Per quanto riguarda la prioritizzazione dei frame invece nel documento in analisi si è attuata una strategia semplicistica, ma che comunque ha influito positivamente riducendo i tempi di caricamento della pagina web del 6%. Si è dato priorità alle risorse secondo la tipologia, seguendo il seguente schema: HTML > CSS > JS > immagini, dove il simbolo ">" indica "prioritario rispetto a".

1.9.2 UTILIZZO DI HTTP/2 IN CASI SPECIFICI

Con la continua evoluzione delle Tecnologie Web sempre più servizi vengono implementati online, dimostrando così l'importanza dei protocolli che ne stanno alla base. HTTP è sicuramente il primo che viene in mente, nonché argomento di tesi. Si è raccolto una serie di documenti scientifici che analizzano l'utilizzo del protocollo in contesti professionali specifici, come ad esempio l'implementazione di un servizio cloud-based per la sanità [14] o i test delle performace di uno streaming DASH [15]. Si procederà ora con l'analisi di ogni articolo.

1.9.2.1 IMPLEMENTAZIONE DI UN SERVIZIO MHEALTH CLOUD-BASED CON HTTP/2

I servizi cloud stanno rappresentando sempre più un segmento centrale nel settore commerciale informatico. Gli utenti preferiscono spesso un'opzione pay-per-use decentralizzata, piuttosto che soluzioni costose, pesanti dal punto di vista computazionale e con i vari problemi che ne derivano, primo tra tutti la necessità di effettuare costanti aggiornamenti. Come molti altri settori anche il l'Health System fa un uso intenso dell'evoluzione tecnologica, mantenendosi costantemente aggiornato, in special modo utilizzando soluzioni mobile. Studi dimostrano come implementazioni mobile, come ad esempio il "remote monitoring", possano ridurre le spese per l'healthcare di $197 miliardi nei prossimi 25 anni nei soli USA.

Viene presentato quindi, in [14] un'implementazione cloud-based cross-platform per l'analisi dell'elettrocardiogramma (ECG) remoto. Il progetto è composto da 4

componenti: il dispositivo mobile dell'end-user, un server HTTP/2 (la scelta è ricaduta su un'instanza EC2 su Amazon AWS), un database centralizzato per lo storage delle informazioni sanitarie e un'applicazione MATLAB per il data-processing e la rappresentazione delle informazioni attraverso grafici. L'intera applicazione è stata implementata utilizzando HTML5 per garantire il cross-platform più esteso.

L'intera esperienza porta alla luce come le caratteristiche cloud-based e cross-platform siano la chiave dell'intero progetto, garantendo così una decentralizzazione per l'utente finale e una copertura di mercato sicuramente più ampia. Come spunti di lavoro futuro gli autori propongono infine l'implementazione di push-notification grazie alla feature HTTP/2 delle server- push.

1.9.2.2 UTILIZZO DI HTTP/2 CON DASH PER VIDEO STREAMING

MPEG Dynamic Adaptive Streaming over HTTP (DASH) è uno standard per lo streaming video e audio recentemente promosso a International Standard (IS). In [15] viene sottolineato come il segmento dell'entertainment real-time sia uno dei più importanti nel settore, coprendo il 50% dell'intero traffico internet negli Stati Uniti, Netflix da solo ricopre il 30%. Questi numeri fanno sicuramente riflettere sull'importanza di uno streaming di qualità su HTTP. Vengono quindi proposte due esperienze di test differenti, presentate in [15] e in [16], che si propongono rispettivamente di indagare le performance di DASH su HTTP/2 in maniera generica e di testare i costi di utilizzo (soprattutto riferiti al consumo energetico) utilizzando Server Push con DASH.

In [15] vengono effettuati test di performance con l'obbiettivo di testare uno streaming che utilizzi DASH sopra HTTP/2 con l'intento in particolare di misurare le variazioni di overhead facendo il tuning su specifiche impostazioni. L'intero esperimento è eseguito grazie all'utilizzo di una rete simulata come in Figura 5.

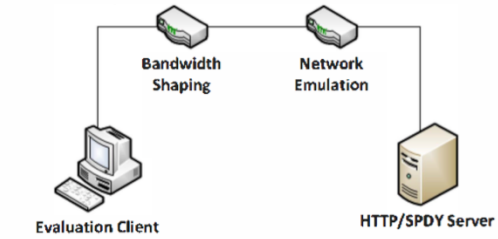
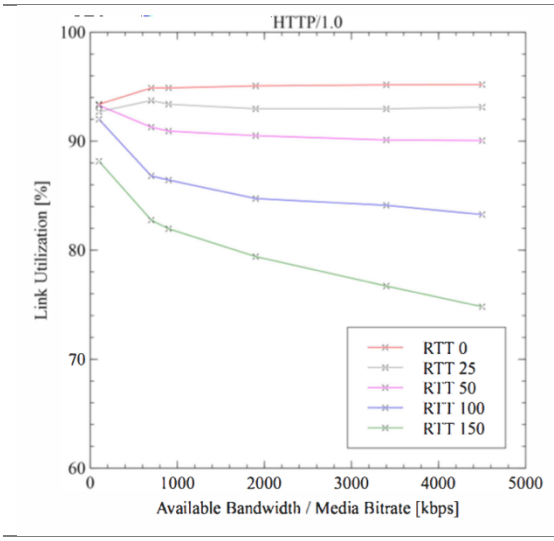


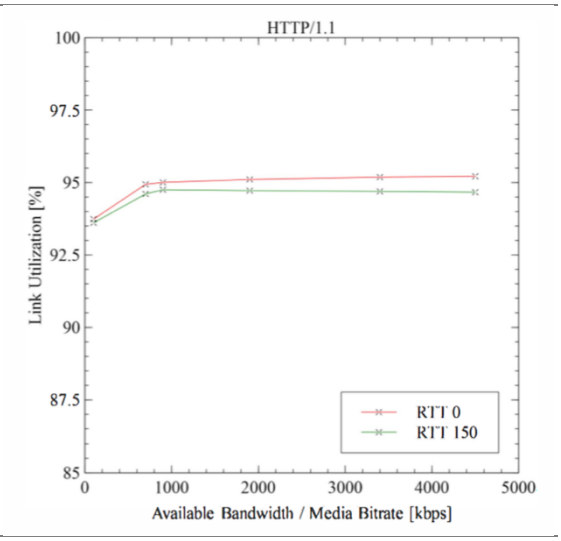
FIGURA 5 NETWORK SETUP

I quattro nodi presenti sono impostati in modo da simulare le condizioni di una rete reale, eccezione fatta per l'Evaluation Client e il Server, che non necessitano di spiegazioni, il nodo Bandwith Shaping è utilizzato per inserire restrizioni all'interno della rete, utilizzando il Linux Traffic Control (tc) e l'Heriarchal Token ucket (htb), il nodo Network Emulation è utilizzato invece per impostare il RTT dell'esperimento.

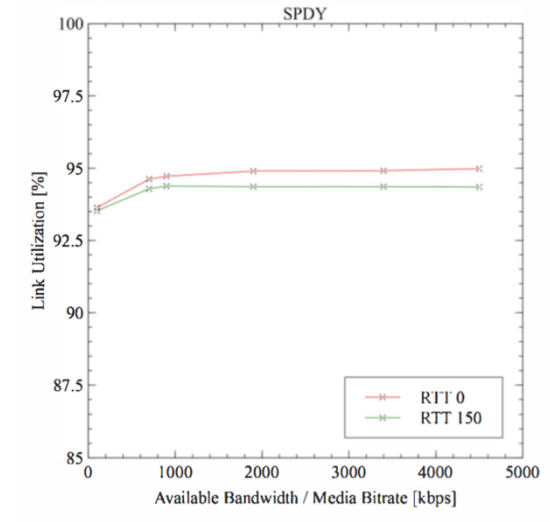
L'overhead testato nell'esperimento si concentra sul layer data-link e dimostra come le differenze rispetto a HTTP/1.1 e HTTP/2 siano minime, mentre siano nette considerando l'utilizzo di HTTP/1.0

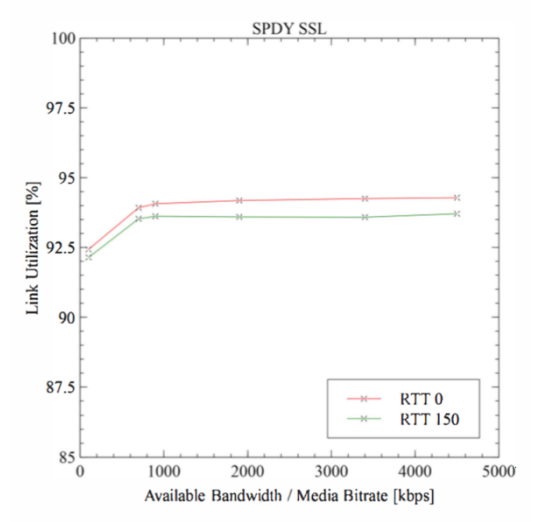
Link Utilization con HTTP/1.0	Link Utilization con HTTP/1.1





Link Utilization con HTTP/2 | **Link Utilization con HTTP/2 con SSL**





L'intero esperimento si conclude mettendo in luce due caratteristiche riguardanti l'utilizzo di HTTP/2:

- HTTP/2 non è performante come HTTP/1.1 per quanto riguarda l'overhead su Link a causa dei vari frame che vengono scambiati sulla rete;
- con l'aumentare del RTT HTTP/2 è più performante e stabile di HTTP/1.1 grazie al fatto che viene utilizzata una sola connessione TCP durante tutta la comunicazione. Inoltre viene risolta in maniera implicita la problematica riguardante HOL[1].

1.9.3 Performance di uno streaming su HTTP/2 con utilizzo di Server Push

Una delle caratteristiche principali di HTTP/2 è la possibilità di effettuare Push da parte del Server verso la cache del client. Questa, nonostante venga presentata in maniera

[1] Head of Line blocking

“vaga” dallo standard RFC viene accolta molto positivamente dal pubblico di sviluppatori dal momento che permette uno sviluppo con numerosi gradi di libertà. In [16] viene indagato come l’utilizzo di questa caratteristica possa essere impiegato nel campo dell’entertainment, in particolare dello streaming video: si parla quindi ancora di DASH.

Vengono quindi implementate diverse strategie per permettere al Server di effettuare il Push dei vari segmenti audio/video verso la cache del client, il quale durante la riproduzione del contenuto multimediale si vedrà ovviamente facilitato il compito. Il flow dell’intero processo viene rappresentato in maniera schematica in Figura 6: una volta che il client effettua la request HTTP verso il server questo inizia il calcolo, seguendo una determinata strategia, dei segmenti che il client necessiterà in seguito, dopodiché inizierà ad effettuare la push verso la cache.

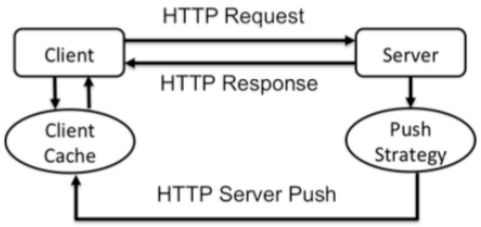


FIGURA 6 SERVER PUSH FLOW

Le strategie implementate e testate sono principalmente due. Elenchiamo per comodità anche l’opzione senza Server Push (tecnica quindi utilizzata con HTTP/1.x):

1. **No Push**: il client effettua una request per il primo segmento video, alla ricezione di quest’ultimo il client procede con una request per l’analogo segmento audio. Alla ricezione di quest’ultimo il client può processare il materiale e procedere con la request per il segmento successivo. Figura 7;
2. **Audio Push**: il client effettua la request per il primo segmento video, il server alla ricezione della request invia il segmento video ed effettua la push verso la cache del client del rispettivo segmento audio. Figura 7;
3. **Audio/Video K-Push**: il client effettua la request per il primo segmento video, il server alla ricezione della request risponde inizialmente come con la strategia Audio Push dopodiché procede con la push di ulteriori K (dove K è un numero intero maggiore di zero) coppie di segmenti audio/video verso la cache del client. Figura 7;

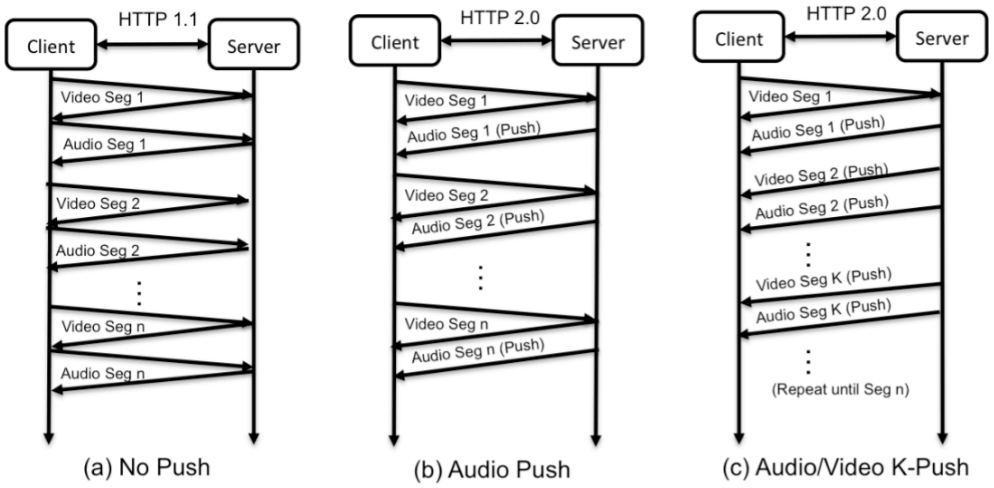


FIGURA 7 STRATEGIE PER SERVER PUSH

Il problema principale sorge quando avviene uno switch di bitrate, quando quindi l'utente (o il contesto d'utilizzo a seconda della natura dell'applicazione) richiede al server l'invio del contenuto multimediale con un bitrate differente. In questo caso ovviamente la strategia più performante è la prima, quella dove la push da parte del server è assente, dal momento che appena nel client viene modificata questa impostazione la request successiva rispecchierà nel segmento ricevuto le nuove caratteristiche. La strategia meno performante è invece la K-Push dal momento che il Server ha probabilmente già avviato l'invio delle coppie di segmenti audio/video dopo lo switch del bitrate, a questo punto quindi le opzioni che può intraprendere l'applicazione sono principalmente due:

- il server alla ricezione dello switch finisce le K Push verso il client dopodiché inizia a lavorare con la nuova impostazione. Questa opzione non ha overhead nella cache del client e nella rete dal momento che tutti i segmenti vengono utilizzati, ma presenta un ritardo (eventualmente anche consistente) nella UX, dal momento che dopo lo switch l'utente dovrà attendere il consumo dei segmenti con il bitrate iniziale presenti nella cache (nel caso peggiore pari a K-1 segmenti) prima di poter ricevere i nuovi. Questa opzione è sicuramente da preferire quando il valore K è basso;
- il server alla ricezione dello switch interrompe l'invio dei K segmenti e riavvia immediatamente l'algoritmo, iniziando da subito a inviare i segmenti con il bitrate nuovo. Questa opzione ovviamente risolve il problema del ritardo nella UX del caso precedente, ma deve preoccuparsi di eliminare i segmenti presenti nella cache nel client con il bitrate iniziale. Questa opzione inoltre presenta un overhead variabile (ma, a differenza del caso precedente, sempre presente) dal momento che i segmenti ormai già inviati hanno consumato banda e spazio nella cache del client seppur non vengano poi effettivamente utilizzati.

2 PROTOCOLLO WEBSOCKET

Il protocollo WebSocket permette la creazione di connessioni a bassa latenza full duplex[1], tra client e server, in grado di scambiare dati in tempo reale. Prima di WebSocket venivano utilizzate altre tecnologie a polling come Ajax o Comet in grado di simulare solamente in parte ciò che invece con WebSocket è diventato successivamente possibile.

2.1 CENNI STORICI

Il protocollo HTTP è stato inteso fin dalla prima versione ideata da Tim Berners-Lee come metodo per recuperare risorse remote in maniera *semplice*: una richiesta per ogni pagina Web, ogni immagine o l'invio di dati da rendere persistente. Con il passare degli anni però, all'incirca intorno al 2004, lo sviluppo di applicazioni Web subì una forte accelerazione dovuta all'introduzione di una nuova tecnologia, **Ajax**, che grazie alla potenza di Javascript fu in grado di creare e gestire richieste HTTP asincrone tramite funzioni di callback dedicate. Seguendo l'evoluzione delle applicazioni Web molte applicazioni prevedevano una UX orientata al real-time. Esempi possono essere applicazioni di chat, videogames multiplayer o sistemi di notifiche, tutte applicazioni che la sola tecnologia Ajax (o simili, come connessioni HTTP persistenti COMET) poteva simulare solo in parte con sistemi di polling poco performanti e complessi da implementare.

La soluzione arrivò quando ci si rese conto che la risposta a questi problemi risiedeva effettivamente nel protocollo stesso: HTTP sfrutta a livello di rete la suite TCP/IP, connection-oriented, usata in altri contesti singolarmente per connessioni full-duplex. Nacque così il protocollo **WebSocket** con un ottimo tempismo considerando l'avvento contemporaneo di HTML5 e altre tecnologie dedicate all'open-source che contribuiranno successivamente alla diffusione del protocollo. [17]

2.2 DESIGN PHILOSOPHY

WebSocket è stato inteso fin dalle prime versione come un protocollo costruito sopra il layer application di supporto, HTTP, minimizzando il framing in modo da renderlo sempre più vicino a uno stream continuo invece che uno scambio di messaggi. I punti chiave di WebSocket potrebbero essere individuati tra i seguenti:

- WebSocket è un layer sopra TCP che aggiunge un modello di sicurezza origin-based ai browser;
- aggiunge un meccanismo di supporto per più servizi su una stessa porta e uno stesso indirizzo IP;

[1] **Full-duplex**: è possibile l'invio contemporaneo di dati tra i due estremi della connessione

- aggiunge un sistema di handshake per la chiusura della comunicazione che supporta la presenza di intermediari, come proxy.

Oltre a questo WebSocket non aggiunge nulla, lo scopo principale è di permettere l'utilizzo di una connessione TCP in circolazione sul browser.

2.3 CARATTERISTICHE DEL PROTOCOLLO

Concettualmente, dal punto di vista dello sviluppatore, WebSocket non è l'unico strumento per la comunicazione in real-time tra client e server, ma è un protocollo, relativamente semplice da utilizzare, che può coesistere senza difficoltà con connessioni HTTP e strutture HTTP intermediarie (proxy), oltre ad avere un modello di sicurezza alla base che lo rende un'ottima e flessibile soluzione.

Il protocollo può essere suddiviso in due differenti parti: un handshake iniziale e l'effettivo trasferimento dei dati.

2.3.1 HANDSHAKE DI WEBSOCKET

L'apertura dell'handshake fa affidamento sugli stessi principi di HTTP in modo che una singola connessione possa fare affidamento sul software server-side strutturato per ricevere request HTTP. Inoltre una stessa porta viene utilizzata sia per le connessioni HTTP che WebSocket. La prima request che effettua il client sarà dunque una richiesta di upgrade del protocollo correlata da alcuni header come `Sec-Websocket-Key`, `Origin`, `Sec-Websocket-Protocol` e `Sec-Websocket-Version`. Una volta ricevuta la request il server dovrà rispondere con una reponse con status code 101 (Switching Protocols) e un header del tipo `Sec-Websocket-Accept` contenente la chiave mandata dal client concatenata a un GUID[1] specifico, fatto l'hash con SHA-1 e infine l'encode con base64. Alcuni header aggiuntivi possono essere inseriti anche nella response del Server, come ad esempio `Sec-WebSocket-Protocol` dove il server può specificare quale dei protocolli richiesti dal client verrà utilizzato nella comunicazione.

L'handshake da parte del client sarà simile al seguente:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Mentre dal punto di vista del server:

[1] **GUID**: Globally Unique Identifier, nel caso di WebSocket: 258EAFA5-E914-47DA-95CA-C5AB0DC85B11

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Una volta ultimato l'handshake il client e il server hanno la possibilità di scambiarsi *messaggi* composti da *frame* (si veda in seguito per la definizione formale di frame e messaggio).

La chiusura dell'handshake è nettamente più semplice rispetto all'apertura. Sia il server che il client possono mandare un particolare frame di controllo dove richiedono l'effettiva chiusura della connessione. Chi riceve tale frame è tenuto a mandare un frame conclusivo del tipo Close. Questa procedura deve servire solamente da supporto alla chiusura effettiva della connessione TCP sottostante.

2.3.2 WebSocket URIs

La specifica RFC stabilisce due differenti tipologie di URI per rappresentare una risorsa remota di tipo WebSocket:

1. ws-URI = `ws://HOST[:PORT]/PATH[?QUERY]`;
2. wss-URI = `wss://HOST[:PORT]/PATH[?QUERY]`.

Dove in entrambi i casi i seguenti parametri significano:

- HOST: l'host dove risiede la risorsa;
- PORT: la porta sul quale l'host rende disponibile la risorsa. Questo apramentro è opzionale e se non specificato viene inteso:
    - Porta 80 per connessioni WS
    - Porta 443 per connessioni WSS
- PATH: il percorso all'interno dell'host dove trovare la risorsa;
- QUERY: la query da compiere sulla risorsa.

Proprio come con le URI HTTP anche in questo caso i caratteri ASCII speciali devono essere escaped attraverso la loro traduzione. [18]

2.3.3 Relazioni con TCP e HTTP

WebSocket è un protocollo indipendente da HTTP nonostante sia basato anch'esso su TCP. L'unica relazione con HTTP risiede nel fatto che l'handshake viene interpretato dal server come un HTTP Upgrade Request. Di default WebSocket utilizza la porta 80 per connessioni regolari e la porta 443 per connessioni sicure che utilizzano il tunnelling con TLS.

2.4 WebSocket e HTTP/2

Lo scopo di HTTP/2, se il lettore permette una semplificazione che rischia di essere estrema, è quello di standardizzare SPDY e fare un upgrade del protocollo, non quindi

rendere obsoleto WebSocket anche se come possiamo notare nella Tabella 3 HTTP/2 vs WebSocket

sono molte di più le similitudini che HTTP/2 ha con WebSocket piuttosto che con il suo predecessore HTTP/1.x.

	HTTP/2	WEBSOCKET
HEADERS	Compressed (HPACK)	None
BINARY	Yes	Binary or Textual
MULTIPLEXING	Yes	Yes
PRIORITIZIATION	Yes	No
COMPRESSION	Yes	Yes
DIRECTION	Client/Server + Server Push	Bidirectional
FULL DUPLEX	Yes	Yes

TABELLA 3 HTTP/2 VS WEBSOCKET

Risulta evidente come i due protocolli, nonostante abbiano un percorso storico completamente differente, siano strettamente legati come features e possibili utilizzi. Risulta quindi naturale la domanda: WebSocket sopravviverà quando HTTP/2 sarà lo standard? La risposta è si, dal momento che le loro features, simili, sono destinati a scenari d'uso differenti. Si prenda ad esempio uno scenario in cui si debba implementare una logica di gioco mulitplayer che debba mantenere tutti i giocatori sincronizzati, in questo caso la scelta più naturale sarà WebSocket dal momento che l'upstream è quasi equivalente al downstream. Se si considera invece come scenario un'applicazione con il compito di mostrare news in realtime, market data o chat forse la scelta migliore potrebbe risultare HTTP/2 con push da parte del server, grazie al fatto di essere multiplexed i tempi di latenza potrebbero essere trascurabili. Inoltre WebSocket spesso risulta pericolo in termini di compatibilità, dal momento che fa affidamento a un upgrade della connessione HTTP ma che con questa ha poco a che fare. Infine è importante considerare anche dettagli di scalabilità e sicurezza: molti componenti web (firewalls, load balancers, etc…) sono strutture definite tenendo in mente il protocollo HTTP e che spesso fatto affidamento sulle relative peculiarità, una connessione con WebSocket portata in contesto ampio potrebbe riscontrare problemi in termini di elasticità, scalabilità e sicurezza. [19]

2.5 STATO DELL'ARTE DI WEBSOCKET

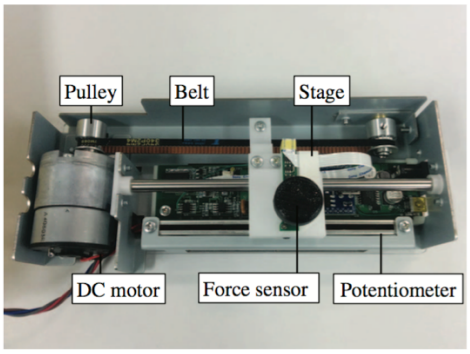
A fianco dello studio di HTTP/2 ci si vuole concentrare anche sul protocollo WebSocket, ormai largamente utilizzato per i più variegati scopi, dai video giochi online a stazioni metereologiche realtime. Ci si concentrerà ora su alcuni casi specifici di utilizzo, dove il protocollo WebSocket copre un ruolo fondamentale.

2.5.1 SVILUPPO DI UN SISTEMA LUDICO TELECONTROLLATO MULTIUTENTE

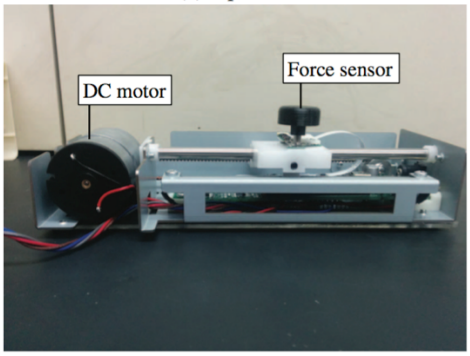
Al di là dei casi d'utilizzo più comuni WebSocket viene impiegato costantemente anche in applicazioni più particolari e dettagliate, è il caso ad esempio di [20] dove si evidenzia

un utilizzo di WebSocket dal carattere ludico, ma che nasconde un impiego di diverse tecnologie, oltre al protocollo in questione, del tutto singolare.

Il sistema viene definito dagli autori un “force sense sharing network game”, un gioco dove agli utenti viene chiesto di applicare una forza su un dispositivo tattile situato in posizioni remote rispetto al server centrale. I diversi dispositivi, una volta collezionate ed elaborate le varie interazioni con l’utente, possono interagire con questi attraverso un motore fisico, in modo da presentare una forza opposta che si contrappone all’utente a seconda della game-logic.



(a)Top view



(b)Front view

FIGURA 8 DISPOSITIVO REMOTO

FIGURA 9 SCREENSHOT DELL'APPLICAZIONE

Il protocollo WebSocket viene quindi utilizzato in maniera intensiva nella comunicazione tra i dispositivi, rappresentati in Figura 8, e il server centrale che si occuperà di collezionare le informazioni remote, elaborarle e quindi calcolare le diverse forza da presentare agli utenti.

L’intero progetto viene diviso in quattro parti fondamentali, due lato server e due lato client:

1. **Server Side**:
    a. **Real-time OS**: che si occupa per primo della stabilita della comunicazione in real-ttime. Al di sopra viene posto il web server Node.js;
    b. **Node.js**: web server contenente principalmente due moduli:
        i. La controparte server della connessione WebSocket;
        ii. Il motore fisico, implementato quasi interamente con MATLAB/Simulink.
2. **Client Side**:

a. **Google Chrome**: riadattamento del browser per gli scopi specifici del sistema, integrato in esso vi è WebSocket;
b. **mbed**: comune framework per l'IoT, è il punto finale di tutta la connessione nonchè il nodo con cui si interfaccia il dispositivo fisico. Due sono i principali compiti:
    i. **Stabilization Control**
    ii. **Haptic device contol**

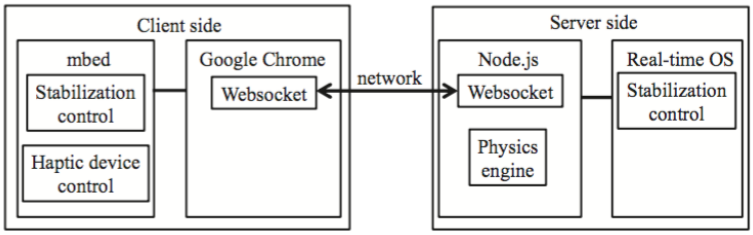


FIGURA 10 ENVIRONMENT DEL SISTEMA

Gli autori infine lasciano aperte strade di ricerca per eventuali altri environment per il sistema, viene citato ad esempio l'utilizzo di UDP come layer di trasporto al posto di TCP, con ovviamente sostituzione di TCP sull'application layer.

2.5.2 UTILIZZO DI WEBSOCKET PER LA PUBBLICAZIONE DI INFORMAZIONI METEREOLOGICHE

Un altro interessante caso di studio su WebSocket è studiato in [21] dove alcuni ricercatori dell'università di Nanjing, China, hanno sviluppato uno speciale framework di monitoraggio meteorologico utilizzando il protocollo WebSocket come intermediario tra i sensori che rilevano i dati fisici e le pagine remote di monitoraggio. In particolare il protocollo viene adottato per l'implementazione di un Middleware Server che possa convertire i dati metereologici in informazioni compatibili per l'applicazione, dopodiché possa spedirli verso i vari client.

L'intero framework può essere suddiviso in quattro moduli elementari:

1. **Wireless sensing**: primo layer che si occupa della ricezione dai sensori metereologici sparsi geograficamente in zone di interesse. Queste informazioni possono essere di diverse tipologie come ad esempio: temperatura, umidità, luce, velocità e direzione del vento, precipitazioni, etc…;
2. **Middleware layer**: il core dell'intero sistema. Si preoccupa di ricevere i dati e di spedirli con connessione TCP per la comunicazione e con WebSocket verso la pagina web. Grazie a quest'ultimo protocollo si ottiene anche l'aggiornamento in real-time delle informazioni direttamente sulla pagina navigata dall'utente;
3. **Network layer**: si occupa esclusivamente di fare da ponte tra i due layer vicini;
4. **Application layer**: il layer dell'applicazione vera e propria, che serve pagine web utilizzando HTTP per le parti statiche e WebSocket per le parti dinamiche che necessitano un aggiornamento frequente.

Lo studio si concentra prevalentemente sul MWServer, la cui architettura prevede 8 componenti:

1. Instructions and query module;
2. Meta-info module;
3. State-info module;
4. FTP-info module;
5. Bacis-info display module;
6. Routing-info module;
7. Data-parsing module
8. Service management module.

Grazie alle varie componenti appena elencate il nodo è in grado quindi di fornire due particolari funzioni:

1. Collection, parsing, mining, integration e storage dei dati metereologici forniti dalla rete di sensori;
2. Ottimizzare l'intera rete, in modo da prolungarne il lifecycle.

Gli autori concludono poi con alcune valutazioni finale sul progetto, tra cui il fatto di poter rendere l'applicazione real-time grazie a WebSocket e di poter, sempre grazie al protocollo, dividere i contenuti a seconda della frequenza con cui questi vengono aggiornati, servendo quelli statici tramite HTTP e quelli real-time con WebSocket.

2.5.3 STUDIO SUL CONSUMO ENERGETICO DI WEBSOCKET E REST SU PIATTAFORME MOBILI

Nella ricerca scientifica [22] portata a termine da tre studiosi tedeschi vengono indagate le differenze di consumi energetici su piattaforme mobili tra due tecnologie per alcuni versi differenti, ma che spesso vengono utilizzate con scopi simili: REST e WebSocket. Al di là del protocollo WebSocket, in largo analizzato in precedenza, verrà ora presentato il protocollo, o per meglio dire *la tecnica* (dal momento che non si tratta di un protocollo vero e proprio), REST e successivamente il confronto tra i due.

2.5.3.1 REST

L'architettura REST[1] è stata sviluppata dopo l'HTTP Object model e ha cinque punti cardine:

1. Le risorse remote hanno un identificato univoco, un id, in forma di URI;
2. Le risorse sono servite cross-platform ma mantengono un proprio stato al di là di queste;
3. I metodi per interagire con un servizio RESTful sono gli HTTP method (GET, POST, PUT, DELETE, HEAD e OPTIONS) dal momento che ogni interazione viene effettuata tramite un'HTTP request/response;
4. Ogni risorsa deve essere servita in formato standard: XML o JSON;

[1] REST: Representational State Transfer

5. La comunicazione è stateless, ogni stato viene catturato client-side sotto forma di risorsa remota.

La semplicità di questa tecnologia è stata indubbiamente il principale punto di forza, tanto da averla resa oramai uno standard-de-facto commerciale. Importante sottolineare come la differenza principale con WebSocket sia che con REST il server è in grado di inviare dati solamente se questi vengono richiesti dal client. L'utilizzo di request-polling è quindi l'unico metodo per un client di un servizio RESTful per ricevere aggiornamenti.

2.5.3.2 Confronto tra REST e WebSocket

L'ultimo punto citato è sicuramente uno dei più importanti per quanto riguarda il consumo energetico: il fatto di dover effettuare un continuo polling verso il server impegna il client maggiormente rispetto a una connessione sempre aperta come si può avere con WebSocket. Viene inoltre analizzato l'overhead delle due tecnologie: mentre REST utilizza uno scambio di request/response HTTP, ognuno con un header dai 200 ai 2048 byte (in media 700-800), una connessione WebSocket utilizza HTTP solo durante l'handshake iniziale e per il resto scambia singoli frame, con header di dimensioni nettamente minori, 2-14 byte. Infine vengono fatte valutazioni sulla rete che servirà l'applicazione, a seconda della tecnologia diverse tipologie di nodi potrebbero creare problemi: mentre REST, grazie alla propria natura intrinseca, raramente presenta problemi causati dalla rete, WebSocket necessita di un'attenzione e di un supporto maggiore, ad esempio con nodi come proxy-server, load-balancer o firewalls.

2.5.3.3 Studio Energetico tra REST e WebSocket

L'esperimento effettuato presenta 12 casi differenti, presenti in Figura 11:

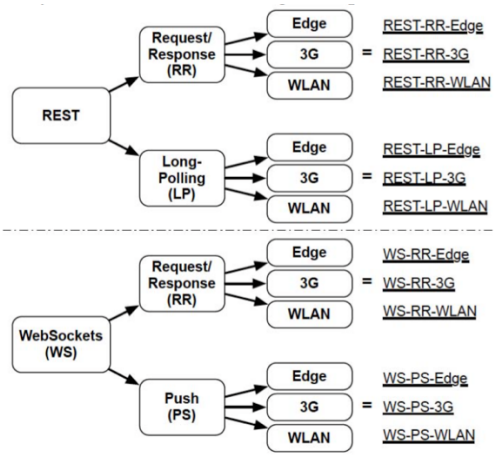


Figura 11 Sistemi Analizzati

Dove la simmetria di test è completa ad eccezione del fatto, dovuto alla natura delle due tecnologie, che per REST viene testato il LP (long-polling) mentre per WebSocket la controparte: PS (push).

Lo studio viene effettuato utilizzando Jetty 9 come WebServer, jwamp per il framework WAMP e Jersey framework per rendere il servizio RESTful. La parte client viene implementata invece tramite un'applicazione iOS su iPhone.

L'esperimento mette in luce il fatto che su una connessione IEEE 802.11 (WLAN) non vi sono differenze interessanti, i due protocolli hanno performance simili. Alcune differenze si riescono ad apprezzare invece con connessioni EDGE e 3G, dove WebSocket è sicuramente più performante e causa un consumo energetico minore. Un punto sicuramente a sfavore di REST, per quanto riguarda il consumo energetico, è una delle politiche di risparmio energetico di iPhone secondo la quale il modulo per la connettività viene spento e riacceso dopo la trasmissione, mentre invece viene ovviamente lasciato attivo per una connessione aperta, questo gioca a favore di WebSocket. L'analisi si conclude infine con l'osservazione che l'overhead delle due tecnologie, cosi come la differenza di connessione 3G o EDGE, non comporta differenze apprezzabili in fatto di consumo energetico.

2.5.4 Sistema WebSocket Proxy per dispositivi mobili

Uno degli scopi principali di molte ricerche riguardanti HTTP è quello di abbattere latenze e ritardi nelle risposte da parte del server. Si è studiato [13] come la maggior parte degli utenti non riesca a tollerare un ritardo superiore ai 2 secondi di caricamento della pagina web e un incremento di 100ms nel tempo di caricamento di una pagina web di un sito di ecommerce possa ridurre le vendite dell'1%. Ovviamente è valido anche il contrario, Google ha dichiarato di aver incrementato le vendite tramite ad del 20% riducendo il tempo di caricamento di 500ms.

Una delle principali cause del ritardo sulla comunicazione web è dovuto al protocollo sottostante, TCP, molto spesso facendo riferimento a "TCP acknowledgment delay". Il fatto che ogni pacchetto abbia bisogno del rispettivo ACK da parte del destinatario, uno dei principi cardine di TCP, rallenta in ampia misura la comunicazione.

Un'alternativa analizzata in [23] da alcuni studiosi giapponesi presenta una soluzione proxy-server con il compito di tramutare il traffico HTTP/TCP in traffico WebSocket, specializzata per dispositivi mobili. Dal momento che lo standard WebSocket non definisce altro che l'header dei singoli frame si può utilizzare per spedire qualsiasi tipo di contenuto all'interno del body.

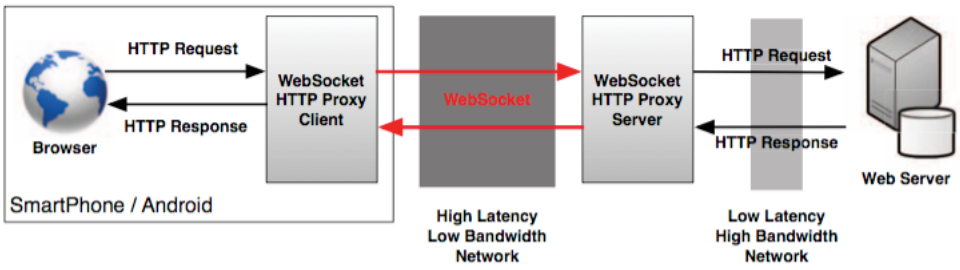


FIGURA 12 STRUTTURA DELLA SOLUZIONE

Come viene rappresentato in Figura 12 il nodo WebSocket proxy risiede client-side in modo che il traffico HTTP venga convertito immediatamente prima di lasciare il dispositivo. Dopodiché attraversando la rete ad alta latenza sotto forma di traffico WebSocket i dati raggiungono il nodo server, vengono riconvertiti in dati HTTP,

processati da un normale server ed inviati nuovamente al client seguendo il percorso inverso.

I risultati dell'esperimento mettono in risalto come effettivamente il guadagno sia netto. Trasformando il traffico HTTP/TCP in dati WebSocket si ha un miglioramento della latenza pari al 169% in un ambiente simulato e del 137% in un ambiente reale. Viene mostrato in Figura 13 la relazione Throughput/delay modificando quest'ultimo da 5 ms a 300ms in una rete simulata.

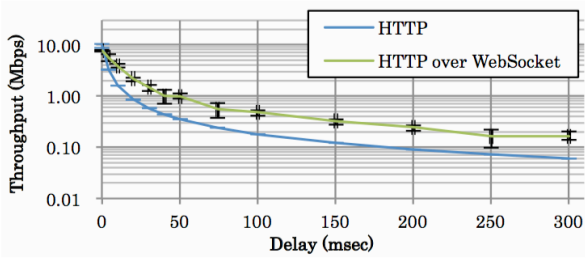


FIGURA 13 THROUGHPUT VS DELAY

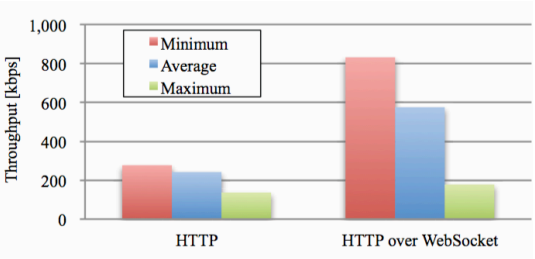


FIGURA 14 HTTP VS HTTP SU WEBSOCKET

Mentre in Figura 14 viene mostrato come in una rete reale, con gestore 3g NTT DoCoMo, si abbia un miglioramento più modesto, ma comunque importante.

3 ANALISI E COMPARAZIONE QUANTITATIVA DEL PROTOCOLLO HTTP

Dopo avere studiato e analizzato i protocolli e relativi documenti scientifici, raccolti nei capitoli precedenti, ci si vuole concentrare ora su un'implementazione personale con l'obbiettivo di valutare le varie performance che questi protocolli possono garantire. A tal scopo verrà ora illustrata l'architettura del progetto, distinguendo le varie sezioni dell'applicazione. Nello schema in Figura 15 sono state raccolte le categorie con le quali il progetto verrà suddiviso; la matrice sarà quindi composta da due parti parallele, la prima vede il confronto tra la tecnica a polling con HTTP/1.1 e HTTP/2, la seconda invece vede in analisi la tecnica a Push, utilizzando per HTTP/1.1 la tecnologia WebSocket mentre per HTTP/2 la tecnologia HTTP/2 Push, feature innovativa propria del protocollo.

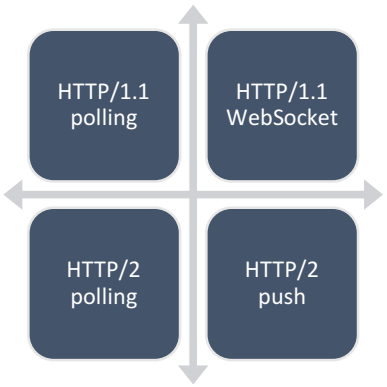


FIGURA 15 SEMANTICHE DI PROGETTO

Appare evidente che incrociando i due protocolli HTTP/2 e WebSocket il quarto caso sarebbe dovuto essere HTTP/2 con WebSocket. Al momento della stesura del presente documento di tesi questa combinazione non è stata ancora standardizzata, nonostante sia stata fatta nell'Agosto del 2014 una proposta di RFC [24] scaduta poi nel Febbraio 2015. Si è voluto quindi inserire al posto della combinazione HTTP/2 con WebSocket

una delle feature più interessanti di HTTP/2, vale a dire la Push proattiva di risorse remote da parte del Server verso il Client.

Per effettuare i test quantitativi sui due protocolli si è voluto suddividere l'intero progetto in due sub applicazioni, una di IM analizzata nel paragrafo 3.1 sviluppata per testare le seguenti casistiche:

- HTTP/1.1 con tecnica di polling;
- HTTP/2 con tecnica di polling;
- HTTP/1.1 con WebSocket con tecnica a push;

Il progetto è poi stato completato da una seconda applicazione analizzata nel paragrafo 3.2, di dimensione ridotta rispetto alla precedente, sviluppata con Node.js per testare l'ultima casistica rimasta:

- HTTP/2 con tecnica Push.

3.1 ARCHITETTURA DELL'APPLICAZIONE IM

L'intero progetto Java è stato sviluppato utilizzando diversi strumenti. Per la parte con HTTP e tecnica a polling e HTTP/1.1 con WebSocket è stato utilizzato il linguaggio Java versione 1.8 con il supporto dell'IDE Eclipse JavaEE for Web Developers, MARS (release di Giugno 2015). Per la parte di studio di HTTP/2 push invece è stato utilizzato Node.js versione 4.2.4. Per quanto riguarda i test con browser desktop sono stati utilizzati tre tra i più comuni browser commerciali: Google Chrome Version 51.0.2704.103, Apple Safari versione 9.1.1 e Mozilla Firefox versione 42.0.

Per quanto riguarda il WebContainer la scelta è ricaduta su Jetty versione 9.3, grazie alla sua natura open-source e facilmente customizzabile, nonché al fatto di essere WebSocket-friendly. Per la parte di Node.js invece è stato utilizzato un modulo esterno, molnarg/node-http2 [25] per interagire con HTTP/2, nonché altri moduli classici come `fs` e `path`.

Per valutare al meglio le performance dei due protocolli si sono indagate diverse tipologie di applicazioni, ognuna con differenti caratteristiche per mettere in risalto peculiarità differenti. La scelta finale è ricaduta su un'applicazione di IM[1] (quanto "istantanea" sarà proprio oggetto d'analisi) dove vengono generati messaggi distribuiti in seguito in broadcast a tutti gli utenti in ascolto, costituendo così un unico paradigma di comunicazione possibile, uno a molti.

L'applicazione presenta alcune semplificazioni che possono essere interpretate come spunti per sviluppi futuri, raccolti successivamente nel capitolo: Possibili sviluppi progettuali futuri. Alcuni di questi possono essere:

[1] IM: messaggistica istantanea

- le conversazioni sono tutte broadcast, ogni messaggio inviato da un utente autenticato viene recapitato a tutti gli utenti in quel momento online. Non sono possibili comunicazioni personali user-to-user;
- il processo di autenticazione, dal momento che non rappresenta il core-process del progetto, ignora l'inserimento della password, processando solamente l'username inserito, in modo da riconoscere e distinguere il singolo utente durante il flow all'interno dell'applicazione.

L'architettura dell'applicazione viene suddivisa in due macro-moduli: uno lato client e uno lato server, distinzione caratteristica di questo contesto di studio. Ognuno di questi interagisce con la controparte ed è composto da alcune componenti che svolgono azioni elementari. Una rappresentazione schematica è data nella Figura 16 e nella Figura 17, che analizzano in maniera schematica l'architettura dell'applicazione, facendo una prima distinzione Client Side e Server Side e successivamente una divisione più particolare dei vari sottomoduli.

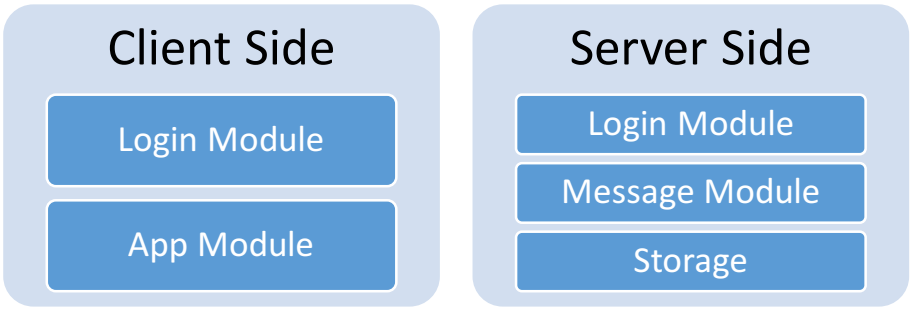


FIGURA 16 ARCHITETTURA DELL'APPLICAZIONE SENZA WEBSOCKET

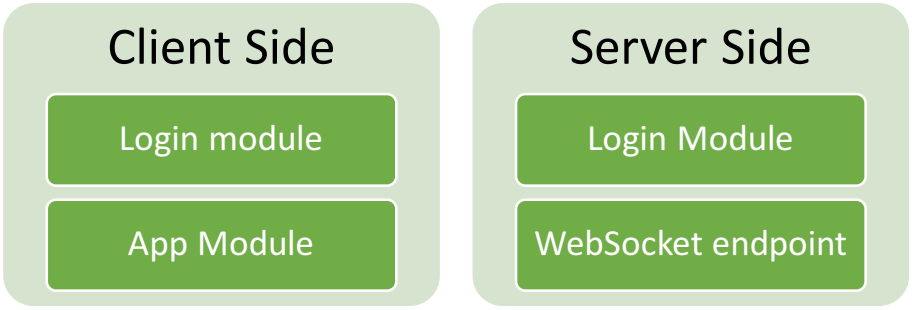


FIGURA 17 ARCHITETTURA DELL'APPLICAZIONE CON WEBSOCKET

Verranno ora analizzati i diversi moduli distinguendo tra moduli client-side e moduli server-side.

3.1.1 CLIENT-SIDE

Per quanto riguarda l'applicazione client-side l'architettura è composta principalmente da due moduli: un modulo utilizzato per l'autenticazione dell'utente e un modulo per l'applicazione vera e propria. In entrambi i casi viene fatto uso di alcune librerie esterne in modo da facilitare il processo di sviluppo laddove non fosse di interesse per lo studio. Per la parte grafica è stato utilizzato Twitter Bootstrap per la sua semplicità d'utilizzo ed

estrema flessibilità anche in campo mobile, per il set di icone invece è stato utilizzato Font-Awesome. Per la parte javascript è stato utilizzato esclusivamente il framework jQuery laddove si necessitava di aggiornare la pagina in risposta a particolari eventi (caso di HTTP con WebSocket).

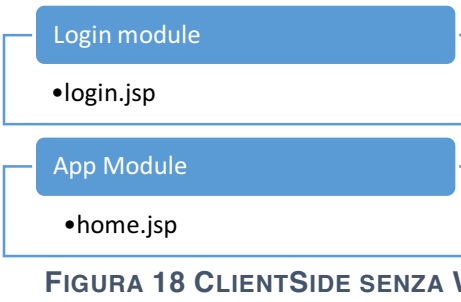


FIGURA 18 CLIENTSIDE SENZA WS

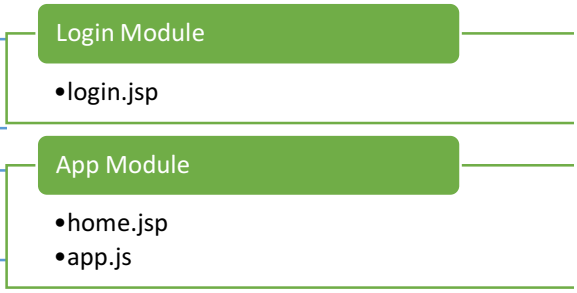


FIGURA 19 CLIENTSIDE CON WS

Verranno ora illustrati nel dettaglio i due moduli client-side.

3.1.1.1 CLIENT-SIDE LOGIN MODULE

Il modulo di login lato client prevede una sola componente: una pagina in formato JSP che viene presentata all'utente all'accesso all'applicazione. Per far si che il file venga presentato come pagina iniziale si è inserito nella lista dei Welcome File nel file di configurazione WEB-INF/web.xml il nome del file:

```
<welcome-file-list>
      <welcome-file>index.html</welcome-file>
      <welcome-file>login.jsp</welcome-file>
</welcome-file-list>
```

All'utente viene quindi chiesto di inserire username e password in un form. Eseguendo tale operazione vengono inviate le informazioni tramite HTTP Resquest con Method POST, in modo da salvaguardare la sicurezza per le credenziali appena inserite, all'endpoint remoto sul server.

Nell'intestazione della pagina oltre ad alcune informazioni di carattere logistico viene indicato il protocollo usato, questa caratteristica va sottolineato che non viene hard-coded durante lo sviluppo ma viene utilizzata una feature JSP per la stampa dinamica, in modo che venga espresso dinamicamente ad ogni accesso.

FIGURA 20 SCREENSHOT DEL MODULO LOGIN_MOD

Per quanto riguarda le distinzioni dovute alla semantica del progetto questo modulo rimane invariato in tutti e quattro i casi, sia utilizzando HTTP/1.1 che HTTP/2 e sia con WebSocket che senza. Lo screenshot in Figura 20 è stato tratto dal caso di studio HTTP/1.1 con WebSocket.

3.1.1.2 CLIENT-SIDE APP MODULE

Una volta autenticato l'utente prosegue nel flow dell'applicazione accedendo alla pagina di home, dove ha luogo la logica vera e propria dell'applicazione. Questo modulo è formato da una sola componente nel caso si lavori senza WebSocket e da due componenti nel caso sia presente invece il protocollo WebSocket.

Sempre grazie a JSP è possibile stampare a video il numero di utenti online e i messaggi attualmente salvati online. La differenza principale sta nella distinzione tra i due casi di utilizzo: con o senza WebSocket.

Nel caso il protocollo WebSocket non venga utilizzato l'aggiornamento della pagina viene effettuato seguendo una delle seguenti strategie:

- **Manuale**: Refresh manuale dell'intera pagina;
- **Polling**: Refresh automatico tramite javascript dell'intera pagina;
- **Polling con Ajax**: Refresh automatico solamente della lista dei messaggi.

La scelta seguirà la seconda strategia: verrà effettuato da parte del pool di client un refresh completo dell'intera pagina a una certa frequenza, questo permetterà di osservare anche le performance di HTTP/1.1 confrontandole con quelle di HTTP/2 in casi limiti di stress dell'ambiente.

Nel caso venga utilizzato WebSocket non si mette in atto alcuna strategia: alla ricezione da parte del client-websocket-endpoint di un messaggio si aggiorna dinamicamente il DOM HTML della pagina. Interessante sottolineare come in questo caso, a differenza del precedente, vengono utilizzate due tecnologie: jQuery per semplificare l'aggiornamento dinamico e JSON per la trasmissione e la codifica dei messaggi scambiati con il server. Infine viene precisato che le tipologie di messaggi scambiati con il server in questo caso sono tre:

1. **User-message**: ricezione di un messaggio standard, si andrà ad aggiornare la lista dei messaggi;
2. **User-online**: ricevuto quando un nuovo utente si connette, si andrà ad aggiornare il numero degli utenti online;
3. **Log-message**: messaggio di log, si andrà ad aggiornare la lista dei messaggi di log, presente a fianco dei messaggi normali.

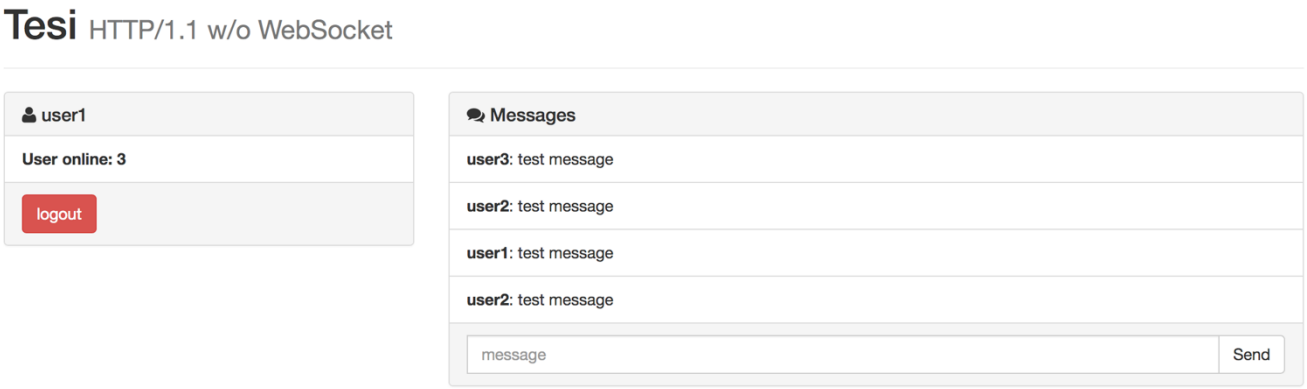


FIGURA 21 SCREENSHOT DEL MODULO APP_MOD

In Figura 21 viene presentato uno screenshot dell'applicazione, dove l'utente si è autenticato come "user1" e sono stati ricevuti messaggi da altri utenti online (per i quali è stato effettuato l'accesso con altri browser).

3.1.2 SERVER-SIDE

L'architettura server-side dell'applicazione presenta i corrispettivi endpoint dei moduli client-side più un terzo modulo di Storage nel caso non sia presente il protocollo WebSocket o un terzo modulo come endpoint della connessione WebSocket nel caso il protocollo sia presente.

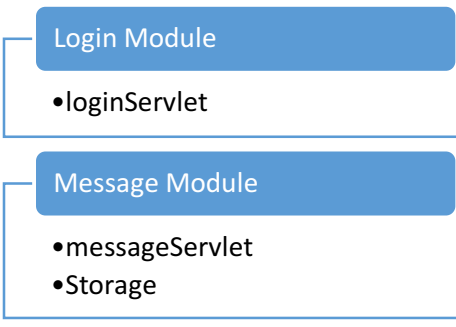


FIGURA 22 SERVERSIDE SENZA WS

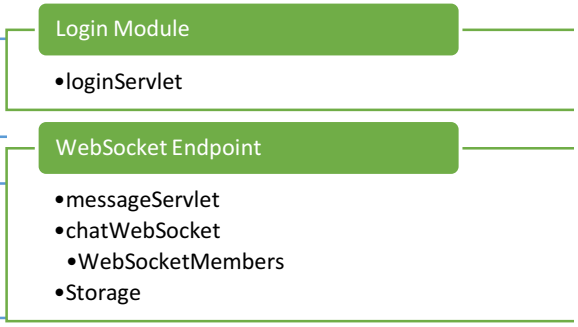


FIGURA 23 SERVERSIDE CON WS

3.1.2.1 SERVER-SIDE LOGIN MODULE

Il modulo di Login Server-side è semplicemente la controparte del modulo studiato in precedenza. Si preoccupa di ricevere l'username dell'utente ed effettua l'autenticazione. Come verrà sottolineato in seguito nei possibili sviluppi futuri dell'applicazione, in un contesto di produzione, ovviamente questo modulo dovrà verificare che l'utente sia

registrato, controllare la password inserita ed autenticarlo o meno. Sempre inerente a questo modulo un ulteriore possibile sviluppo futuro potrebbe riguardare la registrazione di nuovi utenti, ma per la natura di questo progetto di tesi sarebbe stato di poco interesse.

Una volta ricevute le credenziali dell'utente viene impostato il valore dell'username nella sessione della comunicazione HTTP, in modo poi da poterlo utilizzare nei moduli successivi. Il comportamento di questo modulo rimane invariato sia nel caso sia presente WebSocket sia nel caso non sia presente.

3.1.2.2 Server-side Message Module

Il modulo MESSAGE_module è caratteristico esclusivamente dell'applicazione che non utilizza il protocollo WebSocket. La funzione principale di questo modulo è di rispondere a HTTP POST Request effettuate dal client tramite una Servlet, `messageServlet`, contenenti il messaggio inviato salvando il contenuto nell'apposita componente Storage.

Il componente Storage è stato implementato utilizzando il pattern Singleton, in modo da mantenere una sola istanza durante tutta l'applicazione. Al suo interno vengono salvati i messaggi scambiati e gli utenti in che hanno effettuato l'accesso. Un possibile sviluppo futuro, necessario in un scenario di produzione, è sicuramente l'implementazione di un database che vada a sostituire in maniera più performante, sicura e scalabile questa componente.

3.1.2.3 Server-side WebSocket Endpoint

Il modulo che svolge la funzione di endpoint per WebSocket è presente ovviamente solo nella versione che utilizza il protocollo. È composto da due differenti componenti, entrambi strettamente legati tra di loro, ma legati anche alla libreria fornita da Jetty per l'implementazione del protocollo Server-side.

Il primo componente viene definito come nella versione parallela, senza WebSocket, appena descritta ma la funzione è completamente differente. MessageServlet si preoccupa esclusivamente di estendere la classe fornita dalla libreria di supporto `WebSocketServlet` e di effettuare alcune configurazioni iniziali, tra cui la classe che si occuperà realmente del traffico, nel nostro caso `ChatWebSocket`.

Il secondo componente, `ChatWebSocket`, riceve in ingresso i messaggi inviati dagli utenti in formato JSON, ne fa il parsing e manda il contenuto in broadcast a tutti i client in quel momento presenti online. Per fare questo la classe implementa i quattro metodi caratteristici del protocollo: `onConnect`, `onMessage`, `onClose` e `onError`. Questo modulo per controllare gli utenti online utilizza in maniera combinata il componente Storage, che ha comportamenti simili a quelli nel MESSAGE_mod, e il componente WebSocketMembers per controllare quali tra i client che hanno eseguito l'accesso (e che quindi hanno una sessione autenticata) hanno effettivamente aperto una connessione WebSocket.

3.2 Architettura dell'applicazione HTTP/2 Push

A differenza di quella Java, l'applicazione che utilizza Node.js è estremamente più semplice e ridotta, dal momento che è dedicata solamente a un quarto dell'intera semantica del progetto (si veda Figura 15). Il principale obbiettivo in questo caso sarà testare la possibilità con HTTP/2 di effettuare la push di risorse dal Server al Client, in maniera proattiva, senza quindi la necessità da parte di quest'ultimo di richiederle.

Per poter interagire con il livello di trasporto dedicato a HTTP/2 è stato utilizzato il modulo Node disponibile su GitHub, Molnarg/node-http2 [25]. Questo modulo, come si vedrà in seguito nell'esposizione dei moduli del progetto, garantisce, un'API comoda per interagire con HTTP/2, molto simile, come anche indicato dall'autore, all'API HTTPS di Node.js.

Saranno esposti nei seguenti paragrafi una valutazione su una forzatura del protocollo attuata durante lo sviluppo del progetto e un'analisi dettagliata dei moduli che compongono l'applicazione: modulo client e modulo server.

3.2.1 Forzatura del protocollo

La feature HTTP/2 push, come si può studiare nel Capitolo 1 di questo elaborato di tesi, è stata intesa per l'invio di risorse statiche verso la cache del client. Si intende quindi in questo caso risorse come file CSS o script Javascript, non risorse dinamiche come i messaggi di chat appartenenti alla semantica dell'applicazione precedentemente esposta.

Per meglio testare il protocollo e la potenzialità in analisi si è voluto però stressare la feature portandola il più vicino possibile alla semantica di WebSocket e quindi a un'applicazione simile alla precedente. Per fare questo l'applicazione server quando riceve la richiesta da parte del client di una determinata pagina html effettua un certo numero abbastanza elevato di Push Promise, in modo da riservarsi abbastanza "spazio di lavoro" in seguito. Dopodiché, anche attendendo istanti in modo da simulare la generazione dinamica dei contenuti, procede con l'invio delle risorse verso il client tramite semplici file di testo con il contenuto desiderato. Una volta terminato l'invio delle risorse, cioè quando sono terminate le Push Promise effettuate precedentemente, si conclude la comunicazione con l'invio di un semplice file HTML, per rispettare l'effettiva semantica del protocollo. A questo punto il client può procedere richiedendo nuovamente l'endpoint desiderato scatenando di nuovo tutto l'iter. L'intero processo è rappresentato in Figura 24.

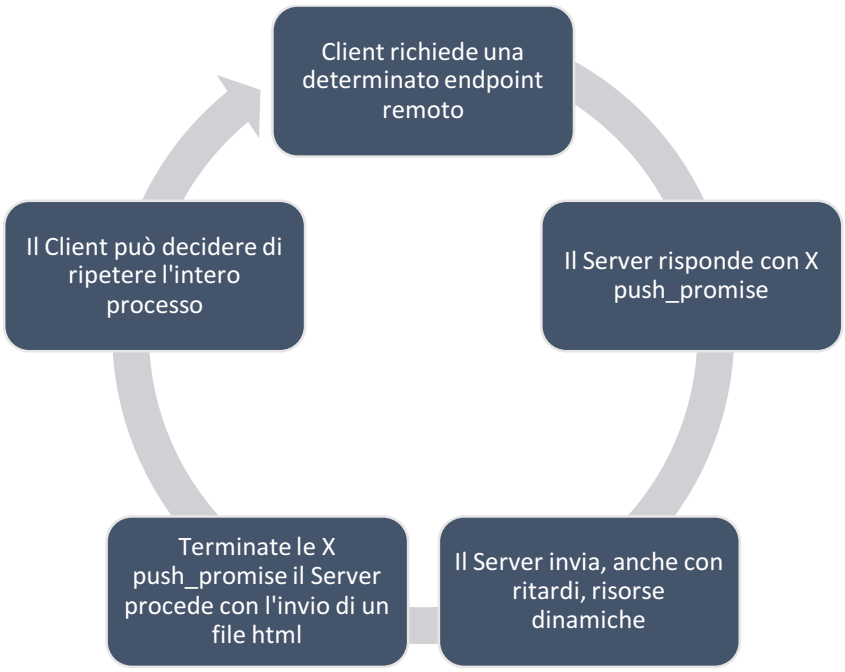


FIGURA 24 SEMANTICA DELL'APPLICAZIONE

Appare evidente come l'intero processo sia una forzatura del protocollo rispetto allo scopo per cui è stato creato, ma è stato scelto di procedere in questo modo per poter analizzare e testare il protocollo sotto stress.

3.2.2 ARCHITETTURA DEL MODULO SERVER

Dopo aver avviato il server, specificando gli endpoint della chiave e del certificato, per stabilire una connessione consistente con SSL, viene registrata una callback `onRequest(request, response)` che verrà scatenata ogni volta che il client effettuerà una request. Come già descritto in precedenza il Server, dopo aver definito quale sarà il file HTML e il numero di Push Promise da inviare, elementi definibili anche a runtime, procede con l'invio delle Promise grazie al seguente frammento di codice:

```
response.writeHead(200);
// sending PUSH_COUNT push promises
var pushes = []
for(i = 0; i < PUSH_COUNT; i++)  {
      var messageEndpoint = "/message_" + i + ".txt";

      pushes[i] = response.push(messageEndpoint);
}
```

Come si può notare dal codice viene dichiarato un ciclo nel quale vengono stabiliti gli endpoint dei singoli messaggi (per semplificare il caso saranno tutti messaggi di testo). A questo punto viene sfruttata l'API offerta dal modulo Molnarg/node-HTTP/2 e vengono inviate le promise, salvando il riferimento in un array.

Successivamente si procede riutilizzando l'array appena definito in modo da inviare effettivamente i messaggi remoti verso il Client, andando così a salvare semplici file di

testo nella cache del client con un contenuto con funzione di placeholder. Di seguito la relativa parte di codice:

```
for(i = 0; i < PUSH_COUNT; i++)  {

      var message = (new Date).getTime();

      console.log(i + " - Writing data: " + message);
      pushes[i].end(message);
}
```

Nell'esempio sopra riportato all'interno di ogni messaggio viene inserito l'istante di creazione, utile eventualmente in caso si vogliano effettuare test sui tempi di risposta.

Infine la callback `onRequest` si conclude inviando un file HTML d'esempio verso il client, per rispettare la semantica di comunicazione del protocollo:

```
filename = path.join(__dirname, 'res', htmlFile);

if (
      (filename.indexOf(__dirname) === 0) &&
      fs.existsSync(filename) &&
      fs.statSync(filename).isFile()) {

      response.writeHead(200);
      var fileStream = fs.createReadStream(filename);
      fileStream.pipe(response);
      fileStream.on('finish', response.end);

} else {
      response.writeHead(404);
      response.end();
}
```

Nel caso il file sia presente nel filesystem del Server viene inviato, in caso contrario viene semplicemente inviato una response con Status Code 404, quest'ultima parte non è comunque particolarmente interessante ai fini dell'applicazione.

3.2.3 Architettura del Modulo Client

La controparte del modulo Server appena descritto è ovviamente il modulo Client, che si preoccupa di effettuare in un primo momento una request verso l'endpoint indicato al momento dell'invocazione e successivamente di gestire le varie push effettuate dal Server.

Sempre grazie al modulo Molnarg/node-http/2 la gestione delle callback è estremamente semplificata. Viene fornita un'API per gestire l'arrivo al Client di una response e di una push, in modo da poter definire callback scatenabili in questi differenti istanti.

La callback scatenata all'arrivo della response vera e propria si preoccupa semplicemente di creare un file locale con il contenuto appena ricevuto. Il seguente frammento di codice rappresenta un eventuale punto di partenza:

```
request.on('response', function(response) {
  mkdirp(path.dirname(destFilename), function (err) {
    if (err)  {
      console.error(err)
    } else {
     response.pipe(fs.createWriteStream(destFilename)).on('fini
sh', function() {
           finish()
      });
    }
  });
});
```

Una volta terminata la ricezione del file viene chiamata la funzione `finish()` che si preoccupa di controllare se tutte le Push Promise siano state ricevute ed eventualmente conclude il processo.

La seconda callback invece viene scatenata alla ricezione di un Frame Push. Il seguente frammento di codice rappresenta un'eventuale politica attuabile alla ricezione:

```
request.on('push', function(pushRequest) {

     var filename = path.join(__dirname, '/client-cache/' +
pushRequest.url);

     push_promises.push(pushRequest.url)
     pushRequest.on('response', function(pushResponse) {
          mkdirp(path.dirname(filename), function (err) {
               if (err) console.error(err)
               else {
                    pushResponse.pipe(fs.createWriteStream(fil
               ename)).on('finish', finish);
               }
          });
     });

});
```

Per prima cosa alla ricezione di una Push Promise viene creato l'endpoint locale sul filesystem, in un'apposita cartella `/client-cache/` dove verrà salvato il contenuto della Push, dopodichè si registra un'ulteriore callback che verrà scatenata all'effettiva ricezione della Push (nella presente simulazione i file di testo generati Server-side). Anche in questo caso alla chiusura della pipe di ricezione della push viene scatenata la funzione `finish()` in modo che sia possibile controllare se tutte le risorse sono state ricevuto e in tal caso sia possibile concludere il processo.

3.3 Architettura del client di test IM

Di fondamentale importanza all'interno dell'intero progetto è indubbiamente la parte di test. È stato creato un apposito client per attuare in maniera automatica e precisa test sulle performance dei protocolli per quanto riguarda l'implementazione Java, sfruttando l'applicazione precedentemente descritta. Il client di test fa un uso intensivo di alcune

librerie fornite da Jetty create appositamente per interfacciarsi con la relativa controparte server-side.

Anche in questo caso il client di test è formato da diversi moduli, rappresentati in maniera schematica in Figura 25, che interagiscono tra di loro per portare a termine il test. Risulta interessante sottolineare come data la diversa natura delle applicazioni, a seconda che venga utilizzato o meno WebSocket, i test stessi seguiranno strade differenti. Ciononostante, per uniformità, si è cercato di mantenere inalterato il punto di partenza e il punto di chiusura dell'analisi, lasciando all'automatismo dello stesso client la libertà di percorrere la strada opportuna.

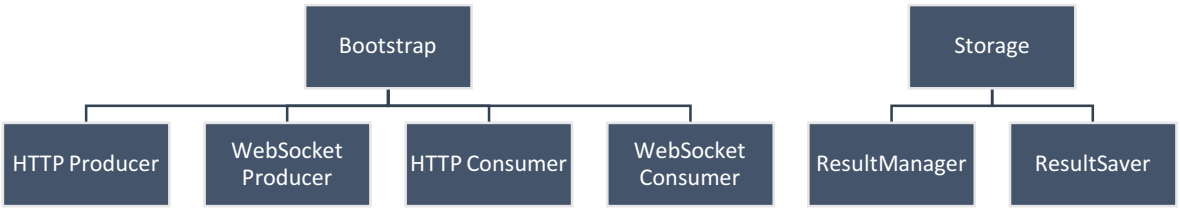


FIGURA 25 ARCHITETTURA DEL CLIENT DI TEST

Verranno ora analizzati singolarmente tutti i moduli seguendo un approccio top-down, partendo quindi dalle macro categorie iniziali e andando sempre più nel dettaglio dei vari moduli.

3.3.1 BOOTSTRAP

Il modulo di Bootstrap è, come suggerisce il nome, il punto di partenza dell'intero client. È comune a tutti i test ed è il punto di configurazione di tutte le impostazioni riguardanti il test. In questo modulo vengono inizializzati i vari client, distinguendo se si tratta di un test solo su HTTP (HTTP/1.1 o HTTP/2) o di HTTP e WebSocket, lanciati e recuperati i risultati in fase finale, una volta completato il test tramite il modulo di Storage vengono salvati su disco i risultati e i metadata.

Le impostazioni che si possono attuare in questa classe sono le seguenti:

- **Protocollo di test**: a scelta tra una delle tre combinazioni: HTTP/1.1, HTTP/1.1 con WebSocket, HTTP/2;
- **Numero di client produttori**: il numero di client che si preoccuperà di inviare messaggi verso il server;
- **Numero di client consumatori**: il numero di client che si preoccuperà di ricevere i messaggi dal server, la semantica di questo comportamento è differente a seconda che si tratti di un client consumatore HTTP o di un client consumatore HTTP + WebSocket;
- **Dimensione del Payload**: nel caso venga utilizzata la combinazione HTTP/1.1 con WebSocket è possibile anche stabilire un particolare fattore moltiplicativo,

chiamato Incremental Payload, utile a definire l'incremento del payload ad ogni messaggio inviato;

- **Intervalli di attesa e numero di messaggi per intervallo**: per implementare una frequenza di invio variabile da parte dei client produttori viene fornita una serie di intervalli e un numero di messaggi per ogni intervallo. Per rendere più chiaro il comportamento viene presentato un esempio: qualora il numero di messaggi per intervallo fosse 10 e gli intervalli fossero identificati da [2000, 1000, 500] espressi in millisecondi ogni produttore si preoccuperà di inviare inizialmente 10 messaggi con tempo di attesa tra un messaggio e il successivo di 2000 millisecondi, successivamente altri 10 con tempo di attesa 1000 e infine con tempo di attesa 500. Il tempo totale calcolabile per questa parte di test è quindi: 10*2000 + 10*1000 + 10*500 = 35000 millisecondi;
- **Tempo di aggiornamento del client consumatore**: importante valore per quanto riguarda i test con il solo protocollo HTTP, quindi con tecnologia a polling. Rappresenta il tempo tra un aggiornamento di pagina e il successivo nel polling dei client consumatori, attuato per verificare aggiornamenti.
- **Filename per i risultati**: posizione sul filesystem locale dove al termine dell'esecuzione verranno salvati i risultati del test e i relativi metadata;
- **Endpoint remote**: url remoto sul server per effettuare login, invio di messaggi, connessione alla WebSocket.

Infine questo modulo si preoccupa di calcolare anche il tempo totale trascorso tra l'avvio del test e il suo completamento. I moduli core dell'intero client di test sono i client produttori e consumatori.

3.3.2 Client Produttori

Il modulo dei i client produttori è caratteristico per la funzione di inviare messaggi verso il server, rispettando le impostazioni fornite nel modulo precedente, Bootstrap. Viene implementato con un thread esterno che si preoccupa di agire in modo asincrono rispetto al resto dell'applicazione.

Seguendo in maniera schematica lo svolgimento delle azioni del thread vengono ora analizzate in maniera sequenziale:

1. Il client inizializza gli HttpClient veri e propri, classe di utility fornita da la libreria Jetty `jetty-client.jar`, distinguendo i diversi casi a seconda del protocollo selezionato;
2. Viene effettuato il login sul server, inviando una POST verso l'endpoint specificando l'username desiderato;
3. Il core dell'intero ciclo di vita del thread avviene quando effettuando un ciclo vengono inviati messaggi verso il server, a seconda che sia un client HTTP o un client HTTP+WS il metodo è differente, ma la semantica rimane invariata;
4. Successivamente il client produttore segnala la presenza di un messaggio ancora da confermare a tutti i client consumatori, che a loro volta lo salveranno in una particolare lista, analizzata successivamente;

5. Terminato l'invio dei vari messaggi il client conclude il proprio ciclo vita effettuando il logout dal server e la chiusura delle varie risorse precedentemente aperte.

FIGURA 26 CICLO DI OPERAZIONI PRINCIPALI DEL CLIENT PRODUTTORE

3.3.3 CLIENT CONSUMATORI

Il modulo che si occupa di ricevere e misurare il tempo di ricezione dei singoli messaggi è il client consumatore. Questo particolare modulo ha un comportamento differenziato a seconda che si tratti di un consumatore HTTP o HTTP e WebSocket, dal momento che la semantica dei due consumatori è differente vale la pena procedere differenziando i due casi.

3.3.3.1 CLIENT CONSUMATORE HTTP

Sia nel caso il protocollo sia HTTP/1.1 sia nel caso sia HTTP/2 l'unico modo che il client consumatore ha di ricevere i messaggi è attraverso un'operazione di polling. Una volta avviato quindi dal modulo padre procede seguendo passo passo le seguenti operazioni:

1. Inizializzazione degli HTTPClient, come nel caso dei client produttori;
2. Effettua il login sul server;
3. Effettua una GET iniziale sulla homepage dell'applicazione salvando in locale la response;
4. A intervalli regolari il client procede simulando un aggiornamento della pagina, semplicemente praticando ulteriori GET. Qualora la response fosse differente da quella salvata in precedenza il client dichiara ricevuti tutti i messaggi precedentemente notificati dal produttore. Salvando i tempi di risposta calcolati e l'overhead accumulato il client conclude il proprio lavoro;
5. Il ciclo di vita di questo modulo si conclude come il precedente effettuando il logout dal server e la successiva chiusura delle varie risorse aperte.

FIGURA 27 CICLO DI OPERAZIONI PRINCIPALI CLIENT HTTP CONSUMATORE

Per quanto riguarda il tempo di risposta, questo viene calcolato facendo la differenza tra gli istanti in un singolo messaggio viene confermato dal consumatore e notificato dal produttore. Per l'overhead invece il client ad ogni aggiornamento di pagina tiene traccia dei dati effettivamente ricevuti e li somma a una variabile locale, alla ricezione di un messaggio salva nel risultato di test il valore fino a quel momento ottenuto e successivamente lo azzera per prepararsi a successive conferme.

3.3.3.2 CLIENT CONSUMATORE WEBSOCKET

Per quanto riguarda il consumatore WebSocket la successione delle operazioni svolte è simile al precedente, ma semanticamente differente in alcuni punti. Per raggiungere il proprio obbiettivo questo modulo fa uso di una classe di utility, fornita da una libreria Jetty, che serve esclusivamente come endpoint WebSocket.

Si procede ora in maniera sequenziale analizzando passo passo ogni operazione effettuata.

1. Come per i casi precedenti anche in questo caso è necessario effettuare l'inizializzazione dell'HTTPClient e del WebSocketCient, saranno questi i protagonisti dell'intera operazione;
2. Una volta effettuato il login il WebSocketClient si preoccupa di connettersi all'endpoint WS remoto. Per fare questo utilizza la classe di utility citata in precedenza, l'unico compito di questa classe è quello di gestire la connessione con l'endpoint remoto e alla ricezione di un messaggio notificare il Client;
3. In seguito il client si mette in attesa della notifica di un nuovo messaggio, con la ricezione di questa procede con la conferma calcolando come nel caso precedente il tempo di risposta. Per quanto riguarda l'overhead invece ci si limita esclusivamente a calcolarlo rispetto al singolo messaggio dal momento che la natura della comunicazione in questo caso è intrinsecamente differente rispetto al precedente;

4. Il ciclo di vita si conclude anche in questo come i precedenti, con il logout e la chiusura delle varie risorse.

Una differenza sostanziale tra le due tipologie di client consumatori è che mentre con il client consumatore HTTP vengono confermati in una volta sola tutti i messaggi in attesa, con il client WebSocket viene confermato un singolo messaggio per volta.

3.3.4 STORAGE

L'ultimo modulo, utilizzato da tutti i precedenti, è il modulo di Storage. Questo ha il compito di tenere in memoria tutti i risultati del test durante l'esecuzione e una volta terminata di salvare su disco i metadata del test in formato testo e i risultati veri e propri in formato csv. La scelta di quest'ultimo formato è dovuta al fatto che facilita l'importazione in Excel dei risultati per l'analisi tramite grafici.

L'operazione di salvataggio dei risultati durante l'esecuzione dei test viene attuato seguendo una strategia a matrice. Ogni messaggio ha una propria riga, dove il contenuto dei messaggi viene utilizzato come chiave primaria, mentre ogni client consumatore ha una propria colonna. In questo modo si ottiene una tabella con tutti i risultati dei tempi di risposta per ogni messaggio e una tabella con i vari overhead.

In fase di chiusura, quando i risultati vengono salvati su disco, viene calcolata la media tra i vari valori dei client, in questo modo si ha per ogni messaggio il tempo di risposta medio e l'overhead medio tra i vari client, ottenendo una tabella di sole tre colonne.

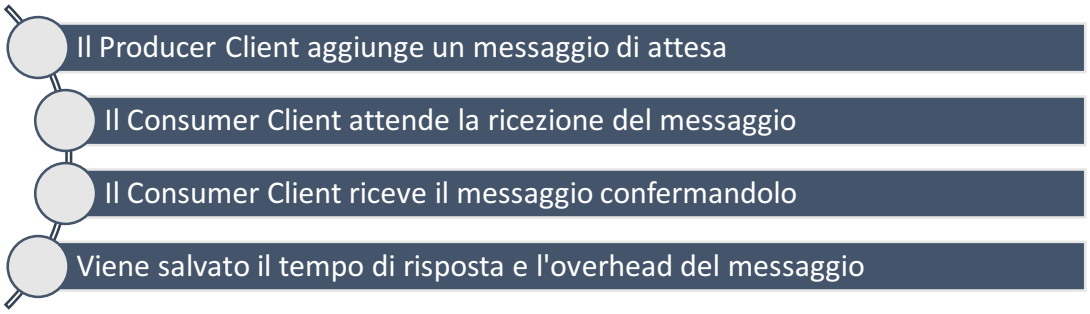


FIGURA 28 SEMANTICA DI TEST PER UN SINGOLO MESSAGGIO

3.4 ANALISI DEI TEST

Lo sviluppo dei moduli precedentemente analizzati porta in maniera diretta allo svolgersi dei test quantitativi e alla relativa analisi dei risultati. I vari test sono stati eseguiti su un notebook personale, Apple MacBook Pro (versione Late 2013) con CPU Intel Core i5 da 2,4 GHz e memoria RAM da 8 GB 1600 MHz. Il sistema operativo di supporto è stato quello nativo originale del notebook, macOS, con l'utilizzo di macchine virtuali, eseguite con Parallels versione 11, con Debian Linux. La scelta per la configurazione di rete

utilizzata è ricaduta in una offerta da Parallels: configurazione a Bridge tra la macchina Host e la VM[1].

Come presentato all'inizio di questo capitolo le tipologie di test e di analisi dei protocolli sono differenti:

- HTTP/1.1 con tecnica di Polling;
- HTTP/2 con tecnica di Polling;
- HTTP/1.1 con WebSocket e tecnica di Push;
- HTTP/2 con tecnica Push nativa.

Seguendo quindi questo schema si sono svolti in maniera differenziata i test, utilizzando le applicazioni dedicate. I risultati ottenuti verranno ora analizzati.

3.4.1 SEMANTICA A POLLING

Si è iniziato prendendo in esame la tecnica a livello implementativo più semplice: la tecnica a polling. In questo caso un determinato numero di Client produttori invierà messaggi verso il Server e un determinato numero di Client consumatori si preoccuperà di riceverli effettuando misurazioni diversificate.

Di fondamentale importanza in questo caso è precisare alcune delle variabili più importanti in ingresso al sistema:

- **Frequenza di invio dei messaggi**: per tutti i test si è mantenuto fisso questo valore, rappresentato dalla sequenza: {2000, 1000, 500, 200, 100, 90, 80, 70, 60, 50, 40, 35, 30, 25, 20, 15, 10, 5}. I valori rappresentano i millisecondi di attesa tra un invio e il successivo, il numero di messaggi inviati sarà fisso per semplificare il numero di variabili in gioco. In uno dei primi test effettuati ad esempio sono stati inviati 10 messaggi attendendo 2000 millisencondi dopo ogni invio, successivamente altri 10 con un attesa di 1000 millisencondi, etc…;
- **Tempo di refresh dei consumatori**: data la semantica dell'applicazione anche questa variabile è di fondamentale importanza, rappresenta infatti il tempo di attesa tra un refresh e il successivo da parte dei clienti consumatori.

In uscita al sistema vi sono due differenti parametri analizzati in seguito: l'overhead della comunicazione e la ratio dei messaggi ricevuti.

3.4.1.1 OVERHEAD

Data la semantica a polling risulta evidente come all'aumentare della frequenza di invio dei messaggi da parte dei client produttori, mantenendo fisso il tempo di refresh dei client consumatori, l'overhead della comunicazione aumenti: mantenendo fissa la dimensione del singolo messaggio i byte della comunicazione variano, questo è causato da molteplici fattori come ad esempio la dimensione dell'header dei singoli frame scambiati. Si sono effettuati diversi test a tal proposito e verranno ora analizzati i due principali che mettono in risalto le differenze tra HTTP/1.1 e HTTP/2.

[1] VM: virtual machines, macchine virtuali

Nel primo test effettuato sono stati utilizzati 2 Client produttori e 2 Client consumatori, valori ridotti per testare il sistema senza stressarlo. Il tempo di refresh dei consumatori è rimasto fisso a 1000 millisencondi e le frequenze di invio dei messaggi sono le stesse presentate in precedenza. Il grafico ottenuto è rappresentato in Figura 29.

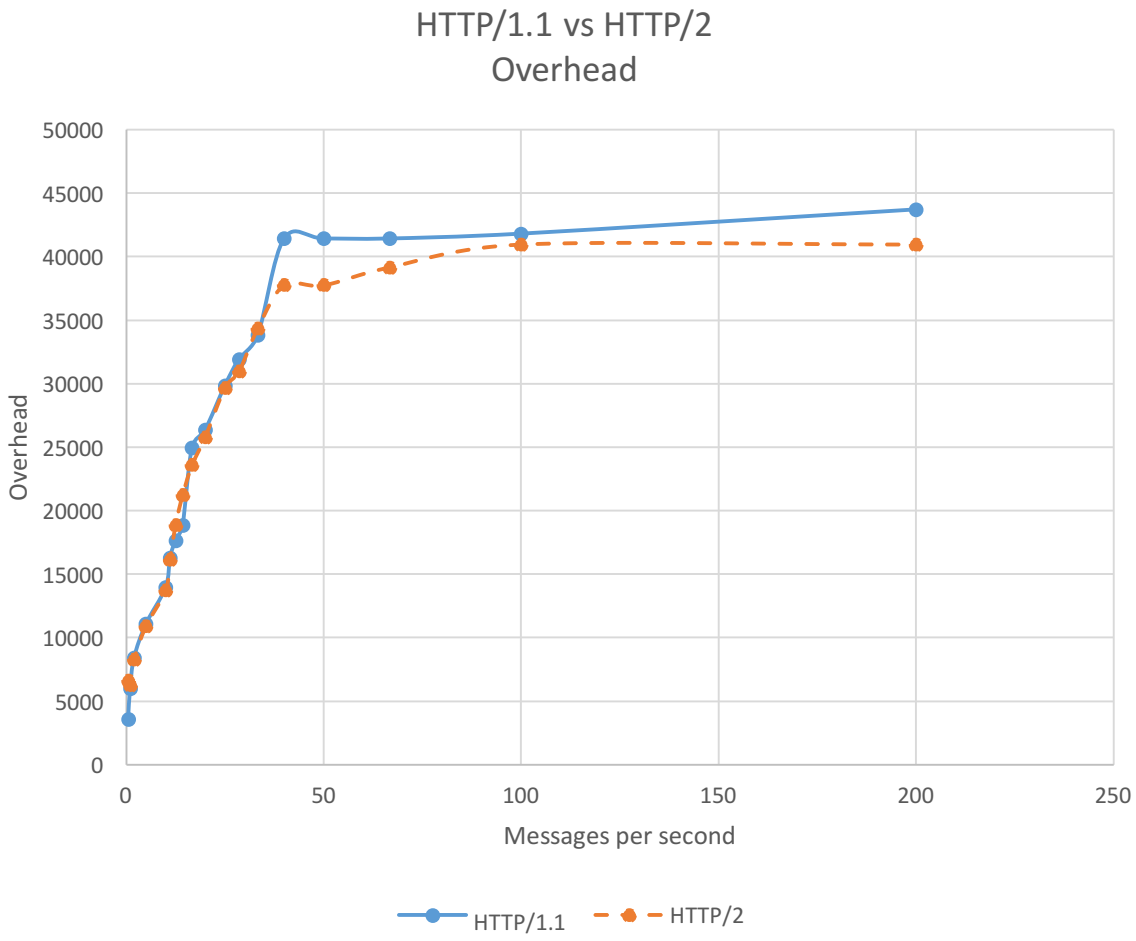


FIGURA 29 OVERHEAD SEMANTICA A POLLING - 2 CONSUMATORI E 2 PRODUTTORI

Come si può analizzare dal grafico l'andamento dell'overhead è strettamente esponenziale in un primo momento andando in seguito ad appiattirsi su un asintoto posto a 40000 byte di overhead. Il grafico inoltre non presenta particolari differenze tra i due protocolli, entrambi arrivano a un livello di saturazione intorno ai 50 messaggi per secondi inviati.

Una differenza tra i due protocolli si può invece apprezzare aumentando il numero di client produttori e consumatori da 2 a 10 e lasciando invariate le altre variabili in ingresso, i risultati dei test in questo caso sono rappresentate in Figura 30.

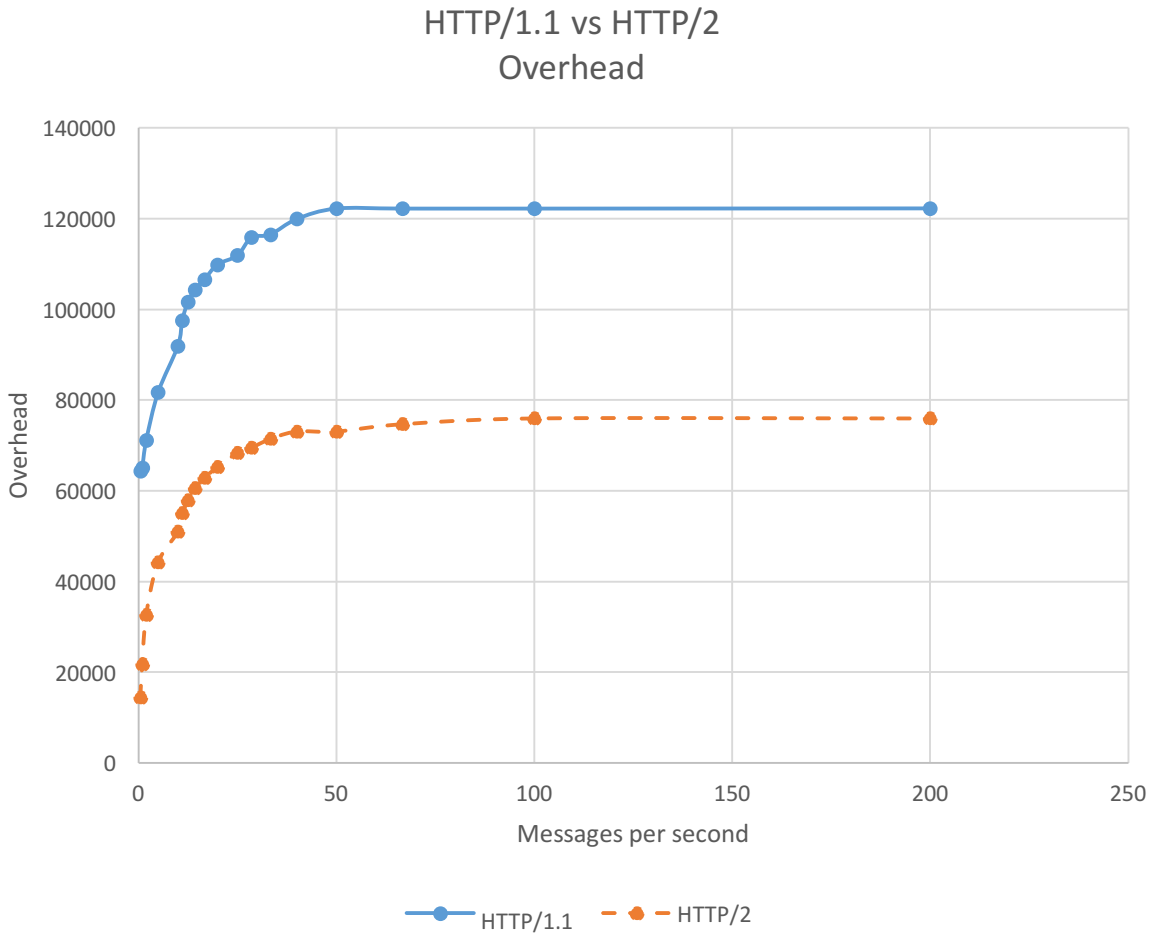


FIGURA 30 OVERHEAD SEMANTICA A POLLING - 10 CONSUMATORI E 10 PRODUTTORI

In questo caso la differenza tra i due protocolli è evidente. Mentre HTTP/2 tende a saturare a 80000 byte HTTP/1.1 satura a 120000 byte. In questo caso HTTP/2 presenta performance nettamente migliori rispetto alla controparte, HTTP/1.1, comportamento sicuramente influenzato dal fatto che HTTP/2 tra le varie feature che lo contraddistinguono dal predecessore presenta anche il fatto di comprimere l'header grazie a tecnologia HPACK (si veda il Capitolo 1 per maggiori informazioni), questa possibilità snellisce sicuramente la comunicazione, rendendo HTTP/2 nettamente più performante di HTTP/1.1 in caso di una comunicazione composta da numerosi messaggi.

3.4.1.2 RATIO MESSAGGI RICEVUTI

Oltre alla misurazione dell'overhead è fondamentale anche la misurazione della ratio dei messaggi ricevuti, cioè di quanti messaggi inviati sono effettivamente ricevuti, dal momento che le frequenze di invio raggiungono valori alti che possono stressare il sistema e quindi compromettere l'effettiva ricezione di alcuni messaggi. Per questa semantica verrà presentata solamente una casistica, parallela all'ultima del paragrafo precedente, con quindi 10 Client produttori e 10 Client consumatori. I risultati ottenuti sono rappresentati in Figura 31.

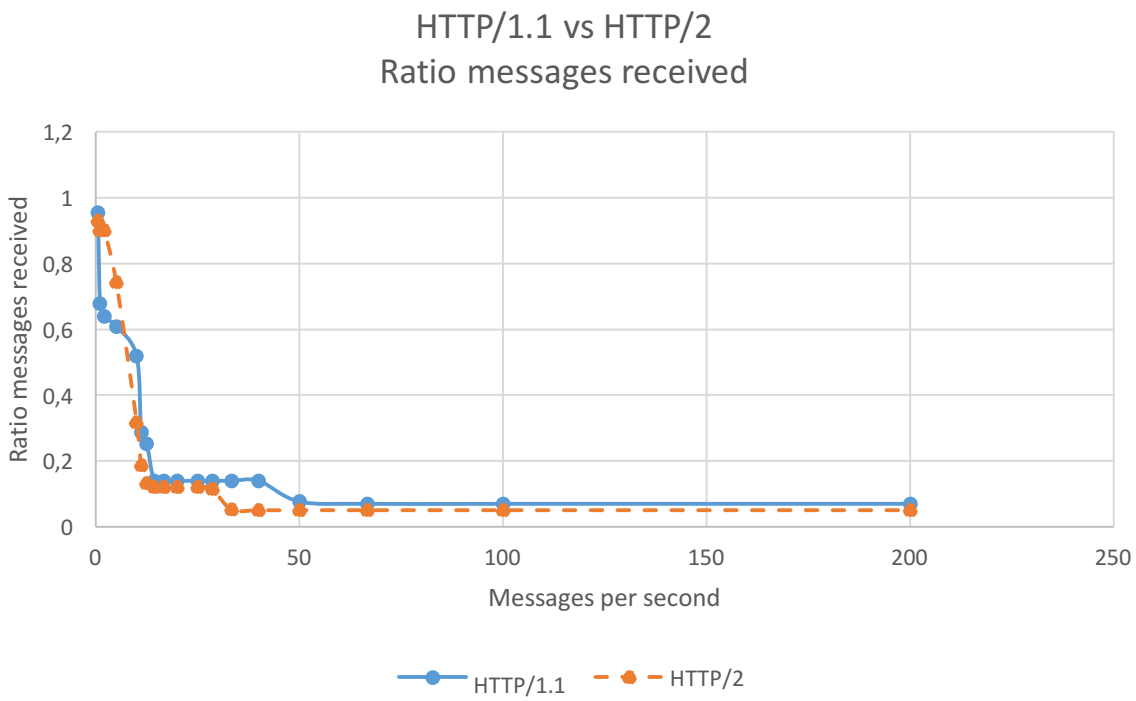


FIGURA 31 RATIO MESSAGGI RICEVUTI CON SEMANTICA A POLLING

In entrambi i casi, sia con HTTP/1.1 che con HTTP/2 sia ha un degrado molto simile, praticamente totale al raggiungimento della frequenza 50 messaggi per secondo. Da ricordare durante la valutazione del grafico il numero di client: 10 produttori e 10 consumatori e il modesto ambiente di sviluppo dove sono stati effettuati i test.

3.4.2 SEMANTICA A PUSH

I test che prevedono semantica a Push, differendo per logica di utilizzo dalla precedente, pongono un'evidente predisposizione alla misurazione del tempo di risposta dei messaggi. Quanto tempo quindi impiega un singolo messaggio, dall'invio dal Client produttore, ad essere ricevuto dal Client consumatore.

In questo caso, per differente semantica applicativa, si è mantenuto un singolo Client produttore e un singolo Client consumatore, ponendo invece, differentemente dal caso precedente, una nuova variabile in ingresso: il payload dei messaggi.

Durante l'invio dei messaggi il Client produttore si preoccuperà quindi di aumentare il payload del singolo messaggio di un determinato fattore, chiamato in fase di test Incremental Payload, e successivamente di procedere con l'invio. Verranno analizzati ora quindi i risultati ottenuti.

3.4.2.1 TEMPO DI RISPOSTA

Come nei casi precedenti anche in questo si è cercato di mantenere più variabili simili tra i due ambienti di test, quindi in questo caso HTTP/1.1 con WebSocket e HTTP/2 Push. Il risultato, di notevole interesse per il caso di studio, è stato l'incapacità di portare l'Incremental Payload per il caso con WebSocket al di sopra di 1000 mentre con HTTP/2 Push si è arrivati senza difficoltà a 100000, una differenza quindi di due fattori 10. Questo

risulta senz'altro significativo, ma prevedibile pensando a come sono stati intesi i due protocolli al momento della standardizzazione: WebSocket rappresenta un protocollo per la comunicazione a livello utente, con messaggi quindi di dimensioni ridotte, HTTP/2 Push invece è la feature della nuova versione di HTTP intesa per l'invio di risorse, quindi file, remoti, abile dunque all'invio di *messaggi* (i byte del singolo file) anche molto più elevati. I risultati ottenuti quindi, lasciando l'Incremental Payload ai valori precedentemente citati, sono rappresenta in Figura 32.

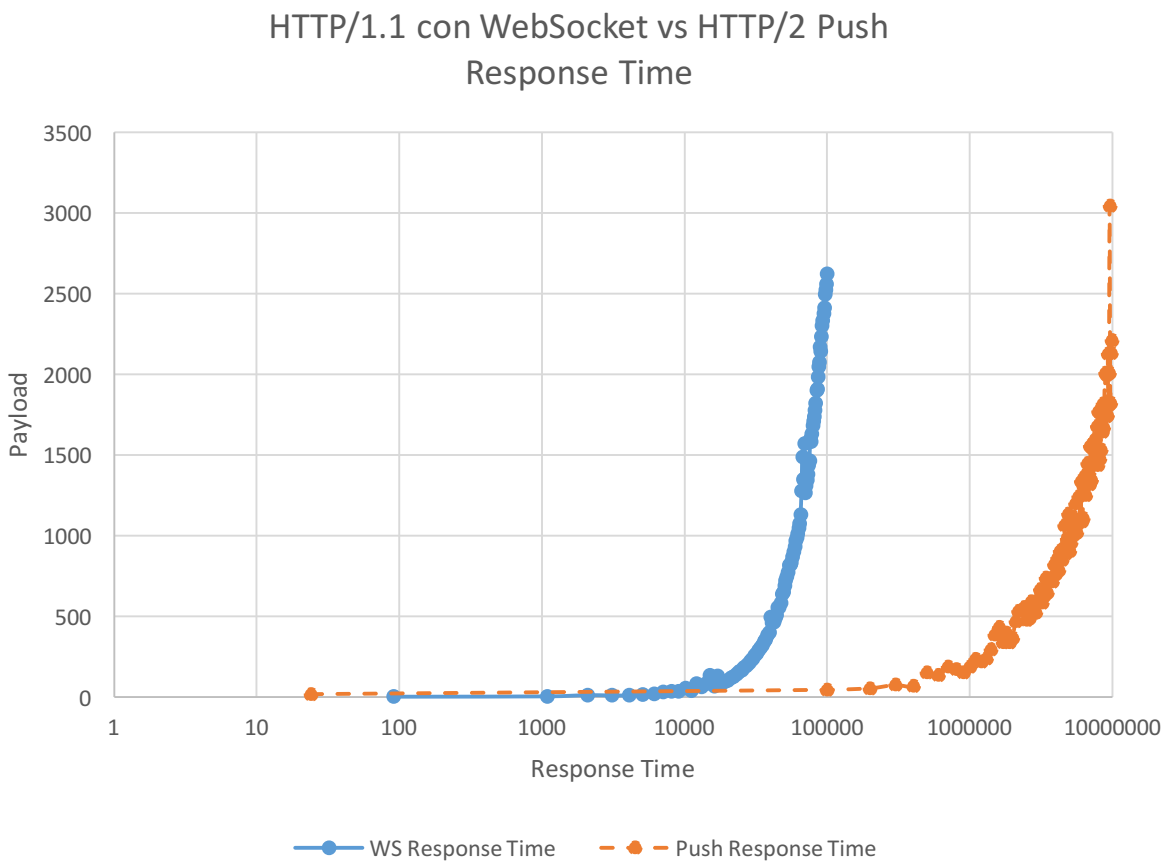


FIGURA 32 TEMPO DI RISPOSTA CON SEMANTICA PUSH

Risulta evidente notare dal grafico come all'aumentare del payload dei messaggi WebSocket tenda a degenerare molto prima rispetto ad HTTP/2 Push, la differenza come si può leggere nell'asse logaritmico del grafico è di due fattori 10.

HTTP/2 risulta quindi evidentemente preferibile se si intende scambiare messaggi (di qualsiasi natura: user-level o risorse statiche) di dimensioni elevate, avendo una degenerazione del tempo di risposta, conseguente al payload, molto più elevata rispetto al caso con WebSocket. Questo comportamento, come nel caso precedente, è sicuramente in parte influenzato dalla compressione dell'Header di HTTP/2, nonché dalle ulteriori numerose ottimizzazioni apportate rispetto ad HTTP/1.1: basti pensare al fatto che i frame a livello di rete non sono più in forma testuale ma in formato binario, questo ottimizza la comunicazione e rende il protocollo evidentemente più performante rispetto al predecessore.

D’altro canto è importante ricordare che, come è stato annotato in precedenza, l’utilizzo di HTTP/2 Push per questa semantica è risultata una forzatura del protocollo, essendo questo studiato per lo scambio di risorse statiche e non avendo un’interfaccia semplice come WebSocket per quanto riguardo lo scambio di messaggi user-level.

3.5 POSSIBILI SVILUPPI PROGETTUALI FUTURI

A fronte di valutazioni sui singoli test vale la pena fare alcune considerazioni su possibili spunti di lavoro futuri. Indubbiamente l’applicazione sviluppata per effettuare i test non è production-ready, si dovrebbe ad esempio sviluppare un sistema di autenticazione e di registrazione utenti più solido, oltre che l’implementazione di un storage sicuro e persistente, tramite ad esempio database. La semantica dell’applicazione stessa, potrebbe essere estesa distinguendo più stanze di chat multi utente piuttosto che una singola o anche la possibilità di creare conversazioni private user-to-user. Per quanto riguarda i messaggi scambiati si potrebbe valutare inoltre la possibilità di poter permettere lo scambio di contenuti non solo testuali, ma anche multimediali, come immagini, video o audio.

Infine per quanto riguarda la scelta dei protocolli a seconda del progetto che si vuole portare a termine si potrebbe preferire un supporto differente: se ad esempio si verifica spesso la condivisione di contenuti multimediali che potrebbero causare un forte ritardo nell’aggiornamento della pagina è preferibile utilizzare HTTP/2 con polling, se si vuole invece privilegiare la caratteristica real-time per un applicazione di messaggistica pura, prevalentemente testuale, è preferibile l’utilizzo di HTTP/1.1 con WebSocket.

Gli ultimi test analizzati inoltre mettono in luce caratteristiche differenti per quanto riguarda il tempo di risposta, con semantica a Push, di HTTP/1.1 con WebSocket e HTTP/2 Push. A seconda del livello di complessità a cui si vuole portare l’applicazione si possono prendere in esame soluzioni ibride che prevedono l’utilizzo di entrambi i protocolli generando sinergie tra i due e sfruttando le peculiarità offerte.

CONCLUSIONI

L'obbiettivo principale di questo progetto di tesi comprendeva lo studio e l'utilizzo dei protocolli HTTP/2 e WebSocket nonché l'analisi delle performance basate su questi. Per quanto riguarda HTTP/2, essendo tra i due il protocollo di standardizzazione più recente, si è cercando di focalizzare l'attenzione, durante tutto il percorso di ricerca, principalmente sulle feature più innovative rispetto alle precedenti versioni, come ad esempio la possibilità di effettuare Push di risorse in maniera proattiva dal Server al Client. Considerando WebSocket invece, essendo un protocollo ormai più maturo, si è ricercato uno studio più incentrato sul confronto con HTTP/2, analizzando e studiando eventuali punti in comune, similitudini e differenze.

Per entrambi i protocolli successivamente si è concentrato lo studio sullo stato dell'arte, come quindi questi due protocolli vengono utilizzati in ambienti industriali o nella ricerca accademica. Questo passaggio è stato di fondamentale importanza in quanto ha portato l'intero progetto di tesi verso un'implementazione personale e originale, sfruttando i due protocolli in maniera parallela, analizzando contesti d'utilizzo simili ma con tecnologie sottostanti differenti. Le due tipologie di applicazione, l'implementazione Java e quella con Node.js, hanno inoltre permesso di studiare i protocolli sfruttando differenti librerie e framework commerciali maturi per un livello di production-ready, come Jetty e il modulo Molnarg/node-http2. In entrambe le applicazioni si è ricercato lo studio delle peculiarità delle librerie, sfruttando le feature più moderne legate ai due protocolli. Grazie a queste tecnologie si è potuto mettere a confronto, tramite test concreti quantitativi, il comportamento dei due protocolli analizzando importanti aspetti, interessanti anche a livello industriale, come overhead, ratio dei messaggi ricevuti e tempo di risposta.

Il presente elaborato di tesi è solamente un primo passo nella ricerca legata ai due protocolli, durante tutto il percorso portato avanti si sono studiati i due protocolli in maniera parallela e mai incrociata, dal momento che durante la stesura del presente documento non vi sono standardizzazioni di supporto ad HTTP/2 con WebSocket, questo è sicuramente il primo spunto per sviluppi futuri. Eventuali altri punti di partenza per ricerche sui due protocolli potrebbero focalizzare l'attenzione su particolari feature, ad esempio potrebbe risultare interessante una ricerca su algoritmi e tecnologie per effettuare la Push HTTP/2 di risorse statiche verso il client in maniera dinamica.

BIBLIOGRAFIA

[1] T. Berners-Lee, «Tim Berners-Lee's proposal,» [Online]. Available: http://info.cern.ch/Proposal.html.

[2] W. W. W. Foundation, «History of the Web,» [Online]. Available: http://webfoundation.org/about/vision/history-of-the-web/.

[3] Google Inc., «Google Developers,» 27 May 2015. [Online]. Available: https://developers.google.com/speed/spdy/.

[4] F. L. -. T. Crunch, «Google Starts Fading Out SPDY Support In Favor Of HTTP/2 Standard,» 9 September 2015. [Online]. Available: http://techcrunch.com/2015/02/09/google-starts-fading-out-spdy-support-in-favor-of-http2-standard/.

[5] WG, IETF HTTP, «HTTP/2 Frequently Asked Questions,» [Online]. Available: https://http2.github.io/faq/#what-are-the-key-differences-to-http1x.

[6] R. Fielding, U. Irvine, J. Gettys, Compaq/W3C e e. alii, «HTTP 1.1 - RFC2616,» June 1999. [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4.4.

[7] R. Murphey, «rmurphey/chrome-http2-log-parser,» 22 December 2015. [Online]. Available: https://github.com/rmurphey/chrome-http2-log-parser.

[8] R. Peon, I. Google, H. Ruellan e C. CRF, «HPACK: Header Compression for HTTP/2 - RFC 7541,» Internet Engineering Task Force (IETF), May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7541.

[9] P. Deutsch e A. Enterprises, «DEFLATE - RFC 1951,» May 1996. [Online]. Available: https://tools.ietf.org/html/rfc1951.

[10] Wikipedia, «CRIME,» May 2015. [Online]. Available: https://en.wikipedia.org/wiki/CRIME.

[11] D. A. Huffman, «A Method for the Construction of Minimum-Redundancy Codes,» 1952. [Online]. Available: http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf.

[12] H. De Saxcé, I. Oprescu e Y. Chen, «Is HTTP/2 Really Faster Than HTTP/1.1,» *IEEE Global Internet Symposium,* 2015.

[13] Y. Elkhatib, G. Tyson e M. Welzl, «Can SPDY really make the web faster?,» *IFIP,* 2014.

[14] Y. Chow, K. Song, Y. Bai e K. Levy, «Design and Implementation of a Cloud-based Cross-Platform Mobile Health System with HTTP/2,» *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops,* 2013.

[15] C. Mueller, S. Lederer, C. Timmerer e H. Hellwagner, «Dynamic adaptive Streaming over HTTP/2,» *IEEE Computer Society,* p. 6, 2013.

[16] S. Wei e V. Swaminathan, «Cost effective video streaming using Server Push over HTTP/2,» *IEEE 16th International Workshop on Multimedia Signal Processing (MMSP),* 22-24 September 2014.

[17] J. Lawson, «WebSockets: A Guide,» 2 April 2012. [Online]. Available: http://buildnewgames.com/websockets/.

[18] W3Schools, «HTML URL Encoding Reference,» [Online]. Available: http://www.w3schools.com/tags/ref_urlencode.asp.

[19] A. Dennis, «Will WebSocket survive HTTP/2?,» 14 March 2016. [Online]. Available: http://www.infoq.com/articles/websocket-and-http2-coexist.

[20] K. Kawase, T. Mijoshi e K. Terashima, «Development of Multilateral Tele-control Game Using WebSocket and Physics Engine,» *IEEE/SICE International Symposium on System Integration (SII),* 11-13 December 2015.

[21] L. Yao, D. Xu, W. Zhang e J. Zhou, «Using WebSocket-based Technology to Build Real-Time Meteorological Wireless Sensor Network Information Publishing Platform,» *IEEE - 10th International Conference on Natural Computation,* 2014.

[22] V. Herwig, R. Fischer e P. Braun, «Assesment of REST and WebSocket in regards to their Energy Consuption for mobile Applications,» *IEEE International Conference on Intelligent Data Acquisition and Advance Computing System,* 24-26 September 2015.

[23] H. Nakajima, M. Isshiki e Y. Takefuji, «WebSocket Proxy System for Mobile Devices,» *IEEE 2nd Global Conference on Consumer Electronics,* 2013.

[24] Y. Hirano e G. Inc, «WebSocket over HTTP/2,» IETF.org, 12 August 2014. [Online]. Available: https://tools.ietf.org/html/draft-hirano-httpbis-websocket-over-http2-01.

[25] G. Molnár e N. Hurley, «GitHub - molnarg/node-http2,» April 2016. [Online]. Available: https://github.com/molnarg/node-http2.

[26] B. M. Leiner, D. D. Clark, V. G. Cerf e e. alii, «Internet Society - Brief History of the Internet,» [Online]. Available: http://www.internetsociety.org/sites/default/files/Brief_History_of_the_Internet.pdf.

[27] Google, Inc, «QUIC,» [Online]. Available: https://www.chromium.org/quic.

[28] C. Allison e H. Bakri, «How the Web Was Won - Keeping the Computer Networking Curriculum Current with HTTP/2,» *IEEE,* 2015.