UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Analisi prestazionale del protocollo HTTP/2

Relatore: Prof. Nicola Zingirian

Laureando: Riccardo Di Bella
Matricola 2000163

ANNO ACCADEMICO 2022 – 2023

Data di laurea: 29 settembre 2023

Abstract

La tesi analizza il protocollo HTTP/2 il cui obiettivo di design è stato quello di ottimizzare l'esperienza di navigazione web, consentendo un utilizzo più efficiente delle risorse di rete rispetto a quanto permesso dal protocollo HTTP/1.1. Questa ottimizzazione è particolarmente importante nello scenario attuale, nel quale i trasferimenti web coinvolgono, per ciascuna pagina web scaricata, numerosissime micro-transazioni, ovvero anche centinaia di richieste di contenuti di piccole dimensioni (spesso poche decine di kilobyte).

La tesi si concentra sull'analisi del protocollo HTTP/2, esaminando le complessità aggiuntive nell'implementazione e le prestazioni in diverse condizioni d'uso. Il funzionamento del protocollo viene esaminato in dettaglio, mettendo in luce le differenze rispetto a HTTP/1.1 e le caratteristiche che lo distinguono, come il multiplexing (cioè la capacità di gestire contemporaneamente più flussi di dati) su una singola connessione TCP, l'uso di framing binari e la compressione degli header.

Attraverso lo sviluppo di un server web sperimentale in linguaggio C che supporta la comunicazione in formato HTTP/2, sono state condotte misurazioni per confrontarne le prestazioni con quelle del suo predecessore. I risultati evidenziano miglioramenti significativi quando sono possibili richieste concorrenti, mentre si registra un deciso peggioramento delle prestazioni in caso di poche richieste concorrenti.

La tesi fornisce un’analisi dettagliata del protocollo HTTP/2, esamina le sue complessità implementative e confronta le sue prestazioni con quelle di HTTP/1.1 in varie situazioni d'uso, dimostrando come HTTP/2 possa portare notevoli vantaggi nell'ottimizzazione delle risorse di rete quando si tratta di gestire micro-transazioni e richieste concorrenti durante il download delle pagine web.

Indice

1. Introduzione

Nel mondo del protocollo HTTP, la gestione delle connessioni TCP è stata oggetto di significative evoluzioni nel corso degli anni.

Inizialmente, nelle prime versioni dell'HTTP, note come 0.9 e 1.0, ogni transazione corrispondeva esattamente ad una singola connessione TCP. Questo significava che ogni volta che veniva effettuata una richiesta HTTP veniva aperta una connessione TCP separata e questa veniva chiusa immediatamente dopo aver completato la risposta HTTP. Questo approccio, noto oggi come modalità "Connection close", richiedeva ai client web di stabilire e terminare connessioni individuali per ogni oggetto incorporato in una pagina web, come ad esempio pagine HTML, icone, multimedia, frame, record JSON, codice JavaScript, ed altro ancora. Di conseguenza, i client web hanno già da allora adottato la strategia, ancora in uso, di aprire più connessioni TCP concorrenti, una per ogni transazione, al fine di sovrapporre le latenze di rete e applicative. Tuttavia, per evitare di sovraccaricare il sistema del server web con un eccessivo numero di connessioni TCP aperte simultaneamente, i client hanno sempre mantenuto un numero limitato di connessioni concorrenti aperte, un "pool" di alcune connessioni (tipicamente 6), limitando di conseguenza il numero di transazioni contemporanee.

Per migliorare le prestazioni di ciascuna connessione TCP nel "pool" di connessioni aperte HTTP/1.1 ha introdotto inizialmente la modalità "Connection keep-alive" come impostazione predefinita. Questa modalità consentiva a diverse transazioni di essere eseguite in sequenza sulla stessa connessione TCP. Grazie a questa modalità, a parte la prima transazione che apriva una nuova connessione, le successive non subivano l'inefficienza dovuta alla latenza di stabilizzazione della connessione e alla fase di "TCP slow-start", in quanto entravano immediatamente nella fase di evitamento della congestione. Tuttavia, le transazioni serializzate in sequenza su una singola connessione TCP non erano in grado di eliminare il "Head of Line" (HoL) blocking a livello applicativo, dovuto alle latenze applicative: stream lenti nella produzione o nel rendering, sia lato server che lato client, riducevano la larghezza di banda della connessione, aumentando così il tempo di accodamento della connessione nell'attesa di acquisire una connessione TCP disponibile dalla "pool". Questo avveniva come effetto del "controllo del flusso" a livello di trasporto, anche quando non si verificava alcuna congestione.

Per superare questo ostacolo, il protocollo HTTP/2 ha introdotto la possibilità per diverse transazioni di utilizzare la stessa connessione TCP in modo concorrente invece che consecutivo. In HTTP/2, l'applicazione segmenta ogni flusso di dati in frame binari e li invia in un unico flusso TCP. Di conseguenza, l'applicazione segmenta il flusso e intercala i pacchetti di diversi flussi serializzandoli in un flusso TCP. Tuttavia, questa complessa architettura non è riuscita a eliminare il "HoL Blocking" a livello di trasporto, poiché la perdita occasionale o la consegna fuori sequenza di un pacchetto di flusso di dati causa un rallentamento agli altri flussi che condividono la stessa connessione.

Questo lavoro esamina l'incremento della complessità nella gestione degli stream dei server HTTP/2 attraverso l'implementazione di un server sperimentale, sviluppato in linguaggio C. Tale server gestisce in modo simultaneo diversi flussi su una singola connessione e analizza gli overhead associati a questa operazione.

La tesi è organizzata come segue:

Nella sezione 2 esaminiamo il protocollo HTTP/2 in dettaglio. Descriviamo la sua struttura generale e i principi fondamentali, tra cui il multiplexing ed il framing dei dati. Esploriamo vari tipi di frame utilizzati nel protocollo HTTP/2, come i frame DATA, HEADERS, PRIORITY, e molti altri. Discutiamo anche la struttura di base delle richieste e delle risposte, nonché il ruolo fondamentale di HPACK nella compressione delle intestazioni. Inoltre, approfondiamo il concetto di Server Push, il suo funzionamento, le potenzialità e il suo stato attuale di supporto.

Nella sezione 3, entriamo nel dettaglio sull'implementazione di un server sperimentale per HTTP/2. Spieghiamo il processo di generazione dei certificati TLS e presentiamo lo schema di funzionamento del web server. Analizziamo il "Main Loop" e le funzionalità chiave come il "Connection Handler", il "Request Handler", e il "Push Handler". Inoltre, esaminiamo la gestione delle connessioni TLS e le comunicazioni non bloccanti. Approfondiamo anche il ruolo di HPACK nella compressione e decompressione delle informazioni. Qui discutiamo anche delle limitazioni e delle non conformità del server sperimentale rispetto al protocollo HTTP/2 e di come implementare un client web compatibile con questo protocollo, inclusi i cambiamenti da effettuare rispetto al server.

La sezione 4 è dedicata all'analisi delle prestazioni. Valutiamo l’impatto del dimensionamento del payload dei frame DATA, il trasferimento di singoli file, il traffico tipico delle pagine web e confrontiamo le prestazioni in base al numero di richieste e all'uso di più connessioni. Esaminiamo anche i tempi di caricamento delle pagine web utilizzando un web browser reale. Infine, esploriamo anche sperimentalmente la diffusione del protocollo HTTP/2 nel panorama attuale, considerando le sue implicazioni e l'adozione in corso.

In tutta questa tesi si fa riferimento all’RFC 9113 [1], in cui viene definito il protocollo HTTP/2.

Il codice del web server, dei client e degli script utilizzati per le misure sono disponibili a questo indirizzo: https://github.com/riccardodibella/tesi-http2

2. Analisi del protocollo HTTP/2

2.1. Descrizione generale

Il protocollo HTTP/2 modifica il modo in cui HTTP viene trasmesso in rete. Vengono mantenute le altre caratteristiche di HTTP (metodi, status codes e il resto della semantica del protocollo) ma si cerca di avere un utilizzo più efficiente delle risorse di rete rispetto a quanto possibile con HTTP/1.1.

Uno dei problemi principali che hanno guidato lo sviluppo del nuovo protocollo è quello di poter trasferire contemporaneamente più risorse tra server e client, superando le limitazioni che le vecchie versioni del protocollo impongono. In HTTP/1.1 il parallelismo viene raggiunto parzialmente grazie alla creazione di più connessioni TCP verso lo stesso host, al pipelining delle richieste e alla fusione di risorse diverse all'interno di un unico file per poterle trasferire con un'unica richiesta con le tecniche come lo spriting (unione di più immagini in un'immagine più grande) e l'inclusione del maggior numero di risorse possibili in un unico file javascript o css. [7] Sebbene queste tecniche portino ad un miglioramento delle prestazioni della navigazione web, nessuna di queste riesce a risolvere completamente il problema del trasferimento di più risorse in parallelo minimizzando le inefficienze nell'utilizzo della rete.

Come descritto nella sezione 2.2 "Multiplexing", HTTP/1.1 non permette di gestire contemporaneamente più richieste sulla stessa connessione TCP. Per questo motivo, una pratica diffusa tra i browser è quella di mantenere dalle 4 alle 8 connessioni aperte con ogni server in modo da poter dividere le richieste tra tutte le connessioni. [9] Questa tecnica porta ad un miglioramento solo parziale delle prestazioni in quanto le pagine web moderne possono dover effettuare anche centinaia di richieste prima di poter completare il rendering [10], mentre con così poche connessioni per ogni server si possono avere solo un numero limitato di richieste in elaborazione in ogni momento. Questa tecnica va inoltre a penalizzare le altre applicazioni attive in Internet che hanno un comportamento più corretto utilizzando solo una connessione per il proprio funzionamento. [11] Infine, l'apertura di più connessioni non permette di usufruire appieno delle potenzialità offerte dai meccanismi di controllo di congestione del protocollo TCP, venendo penalizzati dallo "slow start" di ogni connessione.

HTTP/2 è un protocollo binario per ridurre il numero di byte di overhead associati ad ogni frame e facilitare il parsing. [11] I vantaggi di avere un protocollo testuale come le precedenti versioni di HTTP sarebbero stati comunque ridotti dall'utilizzo di comunicazioni crittografate con TLS, in quanto nessun dato relativo al nuovo protocollo verrebbe trasmesso in rete in modo leggibile. L'analisi dei frame binari è comunque resa possibile tramite strumenti di debugging come Wireshark.

Salvo dove specificato diversamente, tutti i valori numerici sono da considerarsi senza segno e i loro byte seguono il network order.

2.2. Multiplexing

HTTP/2 permette di condividere la stessa connessione TCP tra più richieste tramite la creazione di stream virtuali indipendenti tra di loro all'interno della connessione.

Grazie a questo meccanismo si risolve il problema dell'HTTP Head of line blocking. Con le versioni precedenti di HTTP le risposte delle richieste devono essere inviate nello stesso ordine in cui le richieste sono state spedite dal client: con questo modello FIFO una richiesta con tempi di elaborazione e di invio della risposta lunghi impedisce che altre richieste possano essere servite per la stessa connessione. HTTP/1.1 ha introdotto il pipelining delle richieste per risolvere almeno parzialmente il problema: viene consentito da parte del client l'invio di più richieste sulla stessa connessione senza aspettare di ricevere risposta in modo che queste possano essere elaborate contemporaneamente dal server; questo sistema non risolve completamente il problema dell'Head of line blocking in quanto le risposte devono comunque essere inviate in ordine. Anche a causa di difficoltà di gestione di questo meccanismo da parte di intermediari come i proxy, al momento il pipelining delle richieste HTTP/1.1 non è supportato in nessun browser moderno. [11][12][13][14]

In HTTP/2 ad ogni richiesta è associato uno stream e i dati di richiesta e risposta sono inviati in un formato binario che permette di associarli allo stream e quindi alla richiesta corretti. In questo modo le risposte possono essere inviate nella connessione senza rispettare l'ordine di arrivo delle richieste e evitando il problema dell'Head of line blocking per le richieste HTTP. Come discusso più avanti (sez. 2.12) non viene affrontato il problema dell'head of line blocking a livello di connessione TCP.

I frame devono essere processati nell'ordine in cui sono ricevuti per garantire che entrambi gli endpoint mantengano una visione sincronizzata dello stato della connessione.

Ogni stream è identificato mediante un numero intero. Sia il client che il server possono creare nuovi stream: agli stream creati dal client viene associato in ordine crescente un numero dispari, a quelli aperti dal server un numero pari. Lo stream 0 viene riservato per comunicazioni che riguardano lo stato di tutta la connessione. Se in una connessione si esauriscono gli stream identifier disponibili per la creazione di nuovi stream (da 1 a 2^{31}-1) da parte di un endpoint è necessario creare una nuova connessione per comunicazioni successive.

Ogni stream ha un ciclo di vita indipendente dagli altri, descritto nella sezione 2.7.

2.3. Framing

L'unità di base di cui si compone la comunicazione è il frame, una struttura binaria di lunghezza variabile.

2.3.1. Struttura generale di un frame

Esistono diversi tipi di frame, ciascuno con uno scopo e una struttura diversa. Tutti i tipi sono accomunati dal formato della parte iniziale del loro layout, detta header, che contiene informazioni di carattere generale sul frame.

L’header di un frame è composto dai seguenti campi:

- Length (3 byte): indica la lunghezza in byte del payload del frame (i 9 byte del frame header non sono inclusi in questo conteggio).
- Type (1 byte): indica il tipo del frame, determinandone il formato del payload e la semantica.
- Flags (1 byte): ciascun bit può essere associato a un flag, con un significato che può variare a seconda del tipo di frame.
- Stream identifier (4 byte): indica il numero dello stream a cui il frame è associato. Solo gli ultimi 31 bit possono essere utilizzati, mentre il primo bit è riservato (messo a 0 per i frame inviati e ignorato in ricezione).

Possono poi essere presenti un numero variabile di byte costituenti il payload, con struttura e dimensione dipendenti dal tipo di frame e dal valore del campo Length.

La dimensione massima del payload di un frame è stabilita dal parametro SETTINGS_MAX_FRAME_SIZE secondo il meccanismo descritto alla sezione 2.3.6 riguardante i frame SETTINGS.

I flag definiti nell’RFC 9113 sono i seguenti:

Nome flag	Posizione	Tipi di frame
PRIORITY	0x20	HEADERS
PADDED	0x08	DATA, HEADERS, PUSH_PROMISE
END_HEADERS	0x04	HEADERS, PUSH_PROMISE, CONTINUATION
END_STREAM	0x01	DATA, HEADERS
ACK	0x01	SETTINGS, PING

Tutti i bit del campo Flags non utilizzati per un tipo di frame devono essere lasciati a 0 per i frame inviati e ignorati per i frame ricevuti.

Negli RFC 9113 e 9218 sono definiti i seguenti tipi di frame (viene riportato tra parentesi il corrispondente codice del campo Type). Eventuali estensioni del protocollo possono definire altri tipi di frame e un endpoint deve ignorare tutti i frame di tipi sconosciuti o non supportati.

2.3.2. DATA (0x00)

Un frame DATA può essere utilizzato per inviare qualsiasi sequenza di byte associata ad uno stream. Frame di questo tipo possono essere usati, ad esempio, per trasportare il corpo di un messaggio HTTP, sia per la richiesta che per la risposta.

Per i frame DATA sono definiti i flag PADDED e END_STREAM. Il flag PADDED segnala che ai dati trasportati dal frame è stato applicato un padding: la dimensione di tale padding è specificata nel primo byte del payload. Questo primo byte (presente solo se il flag PADDED è settato) è seguito

dai dati da trasportare e il frame è completato dall'eventuale padding composto da byte fissati a 0. Il flag END_STREAM indica che il peer non invierà altri frame riguardanti la transazione che sta avvenendo nello stream, che passa allo stato "closed" o "half-closed (local)" per il peer che invia il frame, a seconda dello stato in cui si trova la transazione HTTP.

Salvo quanto negoziato da eventuali estensioni al protocollo, questo tipo di frame è l'unico soggetto a controllo di flusso (sez. 2.8 "Flow-control").

2.3.3. HEADERS (0x01)

I frame HEADERS vengono usati per aprire uno stream e trasportare una porzione di dati relativi agli header HTTP di una richiesta.

Semanticamente, un frame di questo tipo può corrispondere alla sezione di header o di trailer di un messaggio HTTP/1.1 oppure a una loro parte.

Per i frame HEADERS sono definiti flag PRIORITY, PADDED, END_HEADERS e END_STREAM.

Il flag PADDED ha lo stesso significato di quanto descritto per i frame DATA: il primo byte del payload del frame contiene la lunghezza in byte del padding applicato alla fine del frame, che deve essere composto di soli zeri.

Se il flag PRIORITY è attivato, i 6 byte iniziali del payload (dopo l'eventuale byte che indica la lunghezza del padding) vanno interpretati nello stesso modo del payload di un frame PRIORITY.

Se il flag END_HEADERS di un frame HEADERS non è attivo, il frame deve essere seguito da uno o più frame CONTINUATION nello stesso stream, l'ultimo dei quali deve avere il proprio flag END_HEADERS attivato. I frame di questa sequenza composta da un frame HEADERS e uno o più frame CONTINUATION devono essere inviati in modo contiguo sulla connessione: non possono essere intervallati da frame di altro tipo o appartenenti ad altri stream. Questo meccanismo si rende necessario quando un singolo blocco di header HTTP ha dimensione maggiore alla dimensione massima di un singolo frame (impostata con il setting SETTINGS_MAX_FRAME_SIZE).

Il flag END_STREAM ha uno scopo simile a quello definito per i frame DATA: viene usato da un peer per segnalare che il blocco di header che inizia con quel frame è la conclusione del messaggio HTTP. A differenza di quello che avviene per i frame DATA, un frame HEADERS con il flag END_STREAM attivo può essere seguito da altri frame CONTINUATION sullo stesso stream se il suo flag END_HEADERS non è attivo. In questo caso, la trasmissione sullo stream può considerarsi conclusa solo quando si riceve l'ultimo frame CONTINUATION, segnalato dal suo flag END_HEADERS.

La sequenza di byte trasportata dal payload di un frame HEADERS (tolti i byte usati per la segnalazione della priorità, la lunghezza del padding e il padding stesso) e il payload di tutti i frame CONTINUATION successivi costituisce un unico blocco di header HTTP codificato con HPACK come descritto alla sezione 2.5.

2.3.4. PRIORITY (0x02)

Questo tipo di frame viene utilizzato per modificare la priorità di uno stream.

La distribuzione della priorità dei diversi stream viene usata per segnalare l'importanza relativa tra gli stream ed esprimere una preferenza sulla quantità di risorse da allocare per la gestione delle richieste degli stream.

Questo schema di priorità si basa su una struttura ad albero per le dipendenze tra gli stream. Per ogni nodo, viene assegnato un peso relativo (costituito da un numero intero) a ciascuno degli stream che dipendono dal suo completamento. Il peso associato a ogni nodo indica in che proporzione assegnare risorse a quello stream rispetto al totale delle risorse disponibili per il nodo genitore.

La priorità di uno stream può essere definita alla sua apertura con un campo nel payload del frame HEADERS e modificata in ogni altro momento con un frame PRIORITY, che può essere inviato anche in stream chiusi o in stato di idle.

Questo metodo di gestione delle priorità, definito alla sezione 5.3 dell'RFC 7540, è stato deprecato nel più recente RFC 9113 perché definisce uno schema molto generale ma che ha visto un supporto limitato da parte di molti server a causa della varietà delle modalità di segnalazione da parte dei client e della difficoltà di unire informazioni riguardo alle preferenze del client a altre informazioni già note al server riguardanti, ad esempio, la struttura di una pagina HTML. [3]

All'interno dell'RFC 9218 è definito uno schema più semplice di segnalazione delle priorità basato sia sul nuovo header HTTP "Priority" che su frame di tipo PRIORITY_UPDATE. Nello stesso RFC viene definito un setting (SETTINGS_NO_RFC7540_PRIORITIES) da utilizzare in HTTP/2 per segnalare che verrà ignorato il meccanismo di segnalazione della priorità descritto nell'RFC 7540. Maggiori dettagli su questo schema alternativo sono riportati nella sezione riguardante i frame di tipo PRIORITY_UPDATE.

Lo standard HTTP/2 consente ai server di ignorare le segnalazioni di priorità da parte del client e usare qualsiasi altro algoritmo per stabilire in che modo allocare le risorse di rete, ma questo può portare a performance subottimali per quanto riguarda i tempi di caricamento delle pagine. [3]

Una possibilità offerta dalle segnalazioni di priorità fornite dal client può essere quella di privilegiare il caricamento di immagini ed elementi che sono visibili nella porzione di pagina che l'utente sta visualizzando. [8]

Nel web server descritto nella sezione 3 "Implementazione" di questa tesi vengono ignorate tutte le informazioni riguardo alle priorità inviate dai client: i frame HEADERS e DATA vengono inviati nell'ordine in cui vengono generati, inviando però sempre tutti i frame HEADERS disponibili prima di inviare un frame DATA.

2.3.5. RST_STREAM (0x03)

Un frame RST_STREAM viene usato per segnalare un errore in uno stream, imponendo al ricevitore di non inviare altri frame (con l'eccezione di PRIORITY) in quello stream.

La dimensione del payload è fissa a 4 byte, necessari per contenere il codice dell'errore da segnalare.

Un frame di questo tipo deve essere associato a uno stream: per segnalare condizioni di errore che riguardano la connessione nella sua interezza deve essere usato un frame GOAWAY.

Per maggiori dettagli si veda la sezione 2.9 "Gestione degli errori".

2.3.6. SETTINGS (0x04)

I frame SETTINGS vengono usati per modificare alcuni parametri di funzionamento del protocollo HTTP/2.

Un peer invia un frame SETTINGS per fissare il valore di uno o più parametri (detti anche "setting") identificati tramite un codice, mentre un peer che riceve un frame SETTINGS contenente delle impostazioni deve confermare di averle ricevute e applicate. Il valore delle impostazioni non viene negoziato: ogni peer stabilisce il valore di un parametro, che verrà usato per le comunicazioni all'interno della connessione in cui il frame SETTINGS viene inviato. I due endpoint possono stabilire valori differenti per lo stesso parametro e ciascuno deve rispettare il valore imposto dall'altro.

Un peer può modificare più volte il valore di un setting durante la comunicazione: fino a quando non viene inviato un frame SETTINGS che lo modifica esso assume il proprio valore di default, da quel momento in poi viene considerato valido sempre l'ultimo valore comunicato per quel setting.

Nell'RFC 9113 sono definiti setting per impostare la massima dimensione in termini di memoria della tabella per la decodifica degli header (SETTINGS_HEADER_TABLE_SIZE, sez 2.5 "HPACK"), per consentire o bloccare la ricezione di risorse tramite push (SETTINGS_ENABLE_PUSH, sez 2.6 "Server Push"), per porre un limite al numero di stream aperti in un dato istante (SETTINGS_MAX_CONCURRENT_STREAMS), per impostare la dimensione iniziale di tutte le finestre di flow-control (SETTINGS_INITIAL_WINDOW_SIZE, sez 2.8 "Flow-control"), per impostare la dimensione massima di un frame (SETTINGS_MAX_FRAME_SIZE, sez. 2.3 "Framing") e per impostare la dimensione massima di un blocco di header (SETTINGS_MAX_HEADER_LIST_SIZE, sez 2.5 HPACK).

Con l'RFC 9218 è stato aggiunto il setting SETTINGS_NO_RFC7540_PRIORITIES, descritto nella sezione 2.3.4.

Tutti i setting sconosciuti o non gestiti devono essere ignorati.

Per i frame SETTINGS è definito il flag ACK: in seguito alla ricezione di un frame SETTINGS con questo flag impostato a 0 un peer deve inviare un frame SETTINGS con il flag ACK attivato per notificare al peer che le modifiche delle impostazioni sono state ricevute.

Ciascun peer deve inviare un frame SETTINGS all'inizio della connessione come descritto nella sezione 2.10 "Creazione della connessione".

2.3.7. PUSH_PROMISE (0x05)

Un frame PUSH_PROMISE viene usato per iniziare il push di una risorsa da parte del server, secondo la procedura descritta alla sez. 2.6 “Server Push”.

Per questo tipo di frame sono definiti i flag PADDED e END_HEADERS, entrambi con funzione analoga a quella per i frame HEADERS: se il flag PADDED è attivato il primo byte del payload contiene la lunghezza in byte del padding finale del frame, mentre se il flag END_HEADERS è a 0 ci saranno uno o più frame CONTINUATION immediatamente successivi al frame PUSH_PROMISE.

Successivamente all’eventuale byte della lunghezza del padding, i primi 4 byte del payload contengono lo stream identifier dello stream in cui verrà inviata la risposta alla richiesta che si sta inviando con il frame PUSH_PROMISE.

2.3.8. PING (0x06)

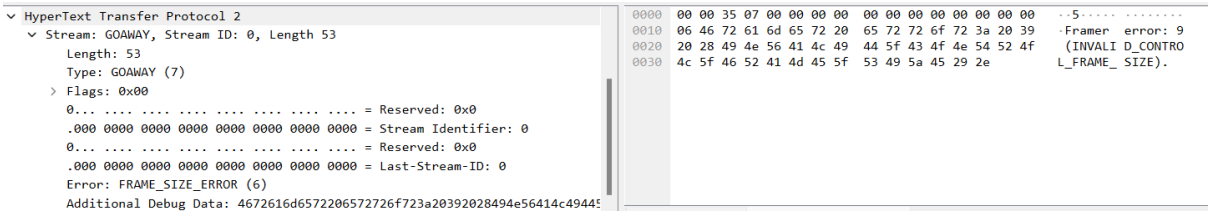
I frame ping vengono utilizzati da un peer per valutare lo stato di una connessione in idle e misurare il round trip delay minimo.

Per effettuare questa misurazione, un endpoint invia sulla connessione un frame PING con flag ACK impostato a 0 e un payload di 8 byte arbitrario. Il ricevente di questo frame deve inviare in risposta un altro frame PING con il flag ACK impostato a 1 e un payload identico a quello del frame ricevuto.

2.3.9. GOAWAY (0x07)

Un frame GOAWAY indica una condizione di errore riguardante la connessione.

Il payload di un frame GOAWAY contiene sempre lo stream identifier dell’ultimo stream processato dall’endpoint che sta inviando il frame e un codice di errore. Oltre a questi due campi, ciascuno lungo 4 byte, possono essere inserite altre informazioni di debug, ad esempio sotto forma di stringa, per permettere di identificare più facilmente il problema.



Per maggiori dettagli si veda la sezione 2.9 “Gestione degli errori”.

2.3.10. WINDOW_UPDATE (0x08)

I frame WINDOW_UPDATE vengono usati per incrementare la dimensione della finestra di controllo di flusso per uno stream o per l’intera connessione.

Il payload di un frame di questo tipo ha dimensione fissa di 4 byte, che ad eccezione del primo bit (riservato) contengono il valore dell'incremento da applicare alla dimensione della finestra di flow-control desiderata.

Per maggiori informazioni si veda la sezione 2.8 "Flow-control".

2.3.11. CONTINUATION (0x09)

Un frame CONTINUATION viene usato per inviare una parte di un blocco di header codificato con HPACK di un messaggio HTTP.

Frame di questo tipo possono essere inviati solo dopo un frame HEADERS, PUSH_PROMISE o CONTINUATION a patto che quel frame non abbia il flag END_HEADERS attivo.

Per questo tipo di frame è definito il flag END_HEADERS: se un frame CONTINUATION non ha questo flag attivo deve essere seguito da un altro frame CONTINUATION nello stesso stream.

2.3.12. PRIORITY_UPDATE (0x10)

Il tipo di frame PRIORITY_UPDATE viene usato all'interno dell'Extensible Prioritization Scheme descritto nell'RFC 9218 [3], a cui si fa riferimento in tutta questa sezione.

Invece di organizzare le richieste in una struttura ad albero con un peso assegnato ad ogni ramo, questo schema di segnalazione delle priorità si basa su due parametri per ogni richiesta: un livello di urgenza (valore numerico intero con valori da 0 a 7 in ordine decrescente di priorità) e un flag per indicare se la risposta può essere processata in modo incrementale o meno.

I frame PRIORITY_UPDATE vengono inviati nello stream 0 e riportano nei primi 4 byte del payload lo stream identifier dello stream sulla cui priorità si sta intervenendo.

La parte restante del payload contiene il valore dei parametri codificato secondo lo standard "Structured Field Values", che nel caso di questo schema di segnalazione produce stringhe del tipo "u=5, i" (livello 5 di urgenza, flag incrementale attivato). I caratteri della stringa vengono riportati secondo la codifica ASCII.

Il parametro di urgenza può essere utilizzato da un client per indicare la precedenza con cui vuole che le risorse associate alle richieste vengano inviate, aspettandosi che un server che utilizza questo schema di segnalazione delle priorità invii tutte le risposte per le richieste più urgenti che ricevute in quella connessione prima di servire le altre con urgenza inferiore.

Il livello di urgenza più alto è associato al valore 0 e quello minimo al valore 7, mentre il valore di default è 3. Il livello 7 (cioè il livello a priorità più bassa) dovrebbe essere riservato a richieste che non hanno nessun collegamento con un'interazione dell'utente, come il download di aggiornamenti software o altre operazioni da svolgere in background.

Il flag "incremental" viene attivato se il client ritiene di poter beneficiare da un arrivo progressivo della risorsa richiesta.

L’attivazione di questo flag può essere utile ad esempio in caso si stia richiedendo un’immagine jpg progressiva o un documento HTML: nel primo caso un’immagine può essere visualizzata all’interno della pagina con una risoluzione più bassa quando non è stata consegnata completamente, nel secondo la presenza di headers per il preloading delle risorse all’inizio del documento HTML può anticipare l’invio di nuove richieste per le risorse collegate. Il valore di default di questo parametro è 0.

Al momento della redazione di questa tesi nessun browser web utilizza questo schema di segnalazione di priorità per il protocollo HTTP/2, anche in caso il server includa il setting SETTINGS_NO_RFC7540_PRIORITIES impostato a 1 nel primo frame inviato.

2.4. Struttura base request/response

Una semplice transazione HTTP è iniziata dal client, che invia un frame HEADERS in cui sono codificati tutti gli header e pseudo-header che compongono la richiesta. Ad esempio, la risorsa interessata dalla richiesta è specificata con lo pseudo-header “:path” e il metodo è specificato tramite il pseudo-header “:method”.

Se la richiesta si esaurisce con l’invio del frame HEADERS, in questo frame viene attivato il flag END_STREAM per segnalare la conclusione dell’invio della richiesta da parte del client.

A questo punto, il server invia nello stesso stream in cui il client ha inviato il frame della richiesta un frame HEADERS che codifica il blocco di header HTTP della risposta. In particolare, lo status code viene riportato nello pseudo-header “:status”.

Se il server non intende inviare nessun corpo per la risposta viene attivato il flag END_STREAM nel frame HEADERS, altrimenti vengono inviati altri frame DATA nello stesso stream, fino a quando non viene inviato un frame con il flag END_STREAM attivato.

2.5. HPACK

Oltre all’RFC 9113, in questa sezione si fa riferimento all’RFC 7541. [2]

HPACK è l’algoritmo di compressione per gli header dei messaggi HTTP utilizzato nel protocollo HTTP/2. Questo algoritmo mira a risolvere uno dei problemi di efficienza di HTTP/1.1: la ridondanza degli header tra i diversi messaggi.

Sebbene per le risposte questo problema possa essere marginale in quanto la dimensione del payload può superare di molto quella degli header, per quanto riguarda le richieste (soprattutto senza payload, come nel metodo GET) il dover ripetere gran parte degli header identici in ogni richiesta (ad esempio i cookie, che possono avere dimensione anche nell’ordine dei KB) può rappresentare un overhead significativo. [15]

Questa tecnica di compressione si basa sulla costruzione di una tabella di header persistente all’interno dello stato della connessione, che viene modificata durante la comunicazione tra i peer e in cui vengono aggiunte le nuove coppie chiave-valore degli header che si prevede verranno ripetute in richieste successive man mano che vengono incontrate. In ogni messaggio, se sia il nome che il

valore di un header o solo il suo nome sono presenti all'interno della tabella possono essere codificati come un indice all'interno della tabella stessa invece che come literal.

In HTTP/2 non è presente la status line di richiesta e risposta, quindi le informazioni in essa contenute vengono codificate all'interno del blocco degli header sotto forma di "pseudo headers", header particolari che iniziano con il carattere ":". Per le richieste sono definiti gli pseudo headers ":method" (metodo della richiesta), ":path" (risorsa) e ":authority" (identità del server) mentre per le risposte è definito lo pseudo header ":status" (status code).

La tabella degli header si compone di due parti: la prima è una tabella statica standardizzata, mentre la seconda ha lunghezza variabile e viene costruita durante la comunicazione.

La tabella statica si compone di 61 header ed è stata popolata con gli header che possono incontrare più di frequente durante la comunicazione: sono presenti alcuni pseudo headers di metodi, status codes e percorsi più comuni codificati come chiave e valore, oltre a altri header di cui è specificato solo il nome (riguardanti ad esempio l'identità del server e il controllo della cache).

Durante lo scambio di messaggi HTTP tra i due peer, per ogni coppia chiave-valore è possibile scegliere se aggiungerla alla tabella dinamica o meno. Per motivi di sicurezza, è possibile imporre che un header non venga mai aggiunto alla tabella dinamica, neanche se la comunicazione viene inoltrata attraverso altri intermediari prima di arrivare a destinazione.

Per trasmettere un nuovo header non presente all'interno della tabella, si può inviarlo come sequenza di caratteri ascii o utilizzare un codice di Huffman statico definito nello standard.

2.6. Server Push

Il Server Push è uno strumento che HTTP/2 mette a disposizione dei server per inviare ai client risorse per cui si anticipa il client dovrà effettuare una richiesta in futuro, prima che il client si renda conto di aver bisogno di quelle risorse.

Lo schema di funzionamento consentito da questa funzionalità costituisce una variazione rispetto al modello di base del protocollo HTTP. Tradizionalmente, tutte le transazioni HTTP vengono iniziate dal client che effettua la richiesta, in risposta alla quale il server invia una risposta con l'eventuale contenuto richiesto. [16] Con Server Push, invece, il server invia attivamente delle risorse al client senza che ci sia mai stata una richiesta per quella risorsa: come discusso più avanti, il server crea delle richieste fittizie a cui associare le risorse inviate.

2.6.1. Funzionamento

All'interno di uno stream associato a una richiesta del client, il server può inviare un frame PUSH_PROMISE per iniziare l'invio di una risorsa tramite Push. Un frame PUSH_PROMISE contiene due elementi principali: i dettagli di una "richiesta tipo" che dovrebbe essere fatta per ricevere la risorsa aggiuntiva che si sta inviando tramite Push come risposta e l'identifier dello stream in cui la risorsa aggiuntiva verrà inviata.

La struttura del blocco di header inviato con un frame PUSH_PROMISE (e eventuali frame CONTINUATION successivi) è la stessa di un frame HEADERS che compone una richiesta:

codificato con HPACK (sez. 2.5), deve contenere almeno gli pseudo-headers “:method”, “:path”, “:scheme” e “:authority”.

Lo stream identifier per l’invio della risorsa specificato in un frame PUSH_PROMISE deve appartenere allo spazio degli stream a disposizione del server (numeri pari) e deve corrispondere ad uno stream ancora nello stato “idle”.

L’invio della risorsa è identico a quello che avverrebbe in risposta a una richiesta del client: si compone di frame HEADERS e DATA come descritto alla sez. 2.4 “Struttura base request/response”.

Se un client riceve un frame PUSH_PROMISE per una risorsa già presente nella propria cache, può scegliere di annullare l’invio della risorsa da parte del server inviando un frame RST_STREAM sullo stream in cui la risorsa dovrà essere inviata.

Se un client non desidera ricevere alcuna risorsa tramite Server Push, viene messo a disposizione il setting SETTINGS_ENABLE_PUSH per disabilitare questa funzionalità.

2.6.2. Potenzialità

A livello teorico, il Server Push può migliorare di molto i tempi di caricamento di alcune pagine in alcune condizioni.

Per un caso d’uso di web browsing, per capire quanto una pagina possa trarre beneficio da questo strumento o da altri strumenti analoghi si può valutare l’albero delle dipendenze della pagina web, ovvero in che modo sono collegate le risorse della pagina web e in che ordine vengono richieste. Un esempio di albero delle dipendenze di una semplice pagina web potrebbe essere il seguente [17]:

```
C:\wwwroot\index.html
├─┬ /css\homepage.css
│ └── /img\homepage-hero.jpg
├── /favicon.ico
└── /bootstrap-4.0.0-dist\css\bootstrap.min.css
```

In questo esempio, i due file css e l’icona vengono richiesti nella pagina HTML e l'immagine hero è richiesta da un file css. Immaginando che il browser faccia le richieste per i due file css solo dopo aver finito il download e il parsing del file html e che richieda l’immagine solo dopo aver elaborato completamente il file css che la richiede, uno strumento come il server push può ridurre il tempo di caricamento della pagina inviando tutte e 4 le risorse aggiuntive assieme al file html: in questo modo, invece di avere un tempo di caricamento minimo di 3 RTT (1 per la pagina HTML + 1 per i file css + 1 per l’immagine) si può ridurre a 1 RTT (tutte le risorse vengono inviate in seguito alla prima richiesta).

Si osserva che, in ambiente controllato, i benefici del Server Push in termini di riduzione nel tempo di caricamento della pagina aumentano linearmente con la profondità dell’albero delle dipendenze e con il tempo di latenza tra client e server. [18]

Un altro vantaggio dell'utilizzo di questa funzionalità può essere la mitigazione del problema dello slow start della connessione TCP: la larghezza di banda disponibile in una connessione è inizialmente limitata dal protocollo e viene aumentata man mano che i dati vengono scambiati sulla connessione. In uno scenario in cui la risorsa richiesta dal client deve essere generata al momento con un'operazione di lunga durata, l'invio di risorse collegate durante la generazione della risorsa "principale" può dare modo alla connessione di aumentare la velocità di invio, assorbendo i round trip dello slow start con risorse il cui invio non è urgente (non possono essere utilizzate finché la risorsa richiesta non viene generata). [19]

2.6.3. Supporto

Nonostante in caso di utilizzo corretto il Server Push possa portare a dei miglioramenti nei tempi di caricamento delle pagine web, questo sistema si è rivelato difficile da usare con efficacia per i gestori dei siti web.

Questo meccanismo, infatti, può essere utile solo fino a quando una risorsa non è già presente nella cache del client e quindi, in presenza di strategie di caching efficaci, può velocizzare solo la prima visita di una pagina web, almeno per quanto riguarda gli asset statici. Costituiscono uno spreco di risorse di rete, quindi, sia l'invio di risorse che sono già presenti all'interno della cache HTTP del browser, che l'invio di risorse che non verranno mai richieste durante la visita della pagina web, ad esempio perché vengono utilizzate solo in seguito ad un evento. Si osserva che un utilizzo scorretto di questo tipo può portare ad un peggioramento delle prestazioni di navigazione. [20]

Uno strumento per mitigare il problema dell'ignoranza sullo stato della cache del client da parte del server potrebbe essere quello del cache digest, il cui sviluppo è però stato abbandonato. [21]

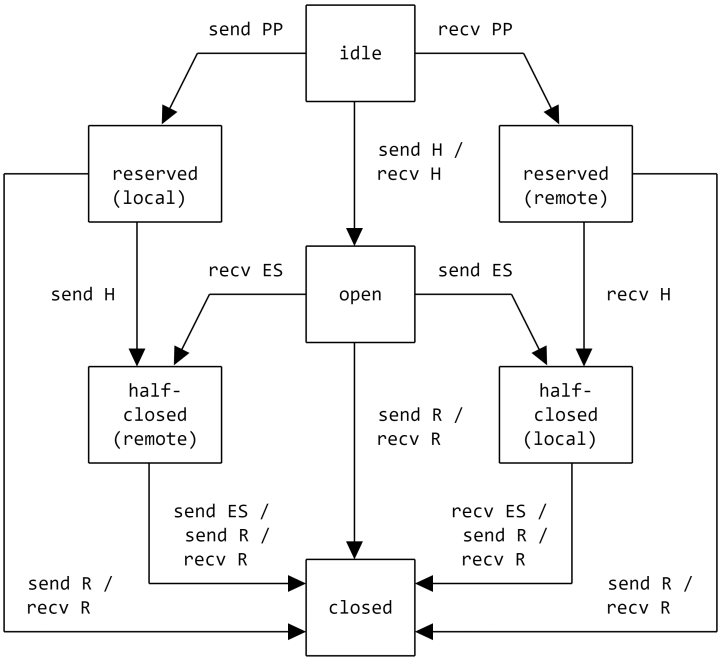
Per motivazioni di uso scorretto dello strumento con conseguente aumento nei tempi di caricamento e complessità aggiuntiva nella gestione delle risorse inviate con Push (che vengono conservate in una cache in memoria separata da quella delle richieste HTTP) e grazie a un uso molto limitato di questa funzionalità, Server Push è stato disabilitato nelle versioni più recenti di Chrome e degli altri browser della stessa famiglia.

Vengono proposti degli strumenti alternativi, come l'utilizzo del codice informativo 103 Early Hints durante la generazione della risorsa richiesta per segnalare al browser altre risorse da iniziare a scaricare oppure il tag <link> con l'attributo rel="preload" da inserire all'interno delle pagine HTML (utile quando l'invio della pagina inizia prima che venga generata completamente), che delegando la decisione dello scaricamento al client riescono a sfruttare al meglio la cache. [22]

Oltre a Chrome e gli altri browser basati su Chromium, anche il web server nginx ha rimosso il supporto a questa funzionalità. [23] Il browser Mozilla Firefox, invece, consente ancora l'utilizzo del Server Push.

2.7. Ciclo di vita degli stream

Per ogni stream è definito un insieme di stati e di transizioni tra di loro, schematizzato nella figura seguente:



Tutti gli stream si trovano inizialmente nello stato “idle”. Da questo stato, se viene inviato o ricevuto un frame HEADERS si passa allo stato “open”. Gli stream identificati da un numero dispari possono essere aperti inviando un frame HEADERS solo dal client, i rimanenti solo dal server.

Quando un peer vuole segnalare di aver finito l’invio di frame in uno stream aperto, invia un frame HEADERS o DATA con il flag END_STREAM attivato: se l’altro peer non ha ancora inviato un frame con lo stesso flag attivato sullo stream si passa allo stato “half-closed (local)” per chi ha inviato il frame con END_STREAM e “half-closed (remote)” per l’altro endpoint. Quando anche il secondo endpoint invia un frame HEADERS o DATA con il flag END_STREAM attivato lo stream passa allo stato “closed”.

Quando viene usato lo strumento del Server Push (sezione 2.6), con l’invio di un frame PUSH_PROMISE l’id dello stream che si riserva per l’invio futuro della risorsa quello stream passa allo stato “reserved (local)” per chi invia il frame e “reserved (remote)” per chi lo riceve. Da questo stato, quando un frame HEADERS viene inviato dal server lo stream passa rispettivamente nello stato “half-closed (local)” per il server e “half-closed (remote)” per il client. In un frame riservato con PUSH_PROMISE, infatti, non è consentito l’invio di dati associati ad una richiesta (con HEADERS e DATA) da parte del client.

In tutti gli stati ad eccezione di “idle” è possibile per entrambi gli endpoint inviare un frame RST_STREAM e terminare ogni comunicazione all’interno di quello stream.

Gli stream devono essere aperti a partire dallo stato “idle” in ordine crescente di stream identifier: quando viene aperto uno stream, tutti gli altri stream disponibili all’apertura da parte di quel peer con identifier minore dello stream appena aperto passano automaticamente allo stato “closed”. Questa transizione non è riportata nello schema.

2.8. Flow-control

Nel protocollo HTTP/2 è prevista una forma di controllo di flusso per permettere ad un peer di controllare la quantità di dati che possono essere inviati dall'altro endpoint. Questo meccanismo può essere utile per proteggere ricevitori con risorse limitate e porre un limite alla quantità di memoria che un endpoint è disposto ad utilizzare per una connessione.

Il flow-control viene gestito con un sistema di credito: ogni peer definisce tramite i frame SETTINGS il credito iniziale in numero di byte che l'altro peer può inviare sulla connessione e in ogni nuovo stream. Questo credito viene consumato da ogni byte di payload dei frame soggetti a controllo di flusso che l'altro peer invia sulla connessione e sullo stream. Un peer non può mai inviare su uno stream un frame soggetto a flow-control con un payload di dimensione maggiore al credito disponibile per quello stream o al credito disponibile per la connessione. La quantità di byte soggetti a controllo di flusso che si possono inviare prende il nome di flow-control window.

Il credito per la connessione e per i singoli stream può essere aumentato tramite i frame WINDOW_UPDATE: ogni frame di questo tipo specifica un incremento nella dimensione della finestra di flow-control che va ad essere aggiunto al credito disponibile in uno stream se lo stream identifier del frame WINDOW_UPDATE è diverso da 0, altrimenti si aggiunge alla flow-control window di connessione. L'incremento specificato in un frame di questo tipo deve essere maggiore di zero.

La dimensione iniziale delle finestre di flow-control può essere impostata con il setting SETTINGS_INITIAL_WINDOW_SIZE, se questa impostazione non viene specificata viene utilizzato il valore di default 65535. Questo valore viene applicato alla connessione e a tutti i nuovi stream. Quando si modifica il valore di questa impostazione è necessario aggiornare il credito disponibile per tutti i frame aperti e per la connessione: il nuovo credito (C) viene calcolato a partire da nuova dimensione iniziale (N), dimensione iniziale precedente (O) e credito precedente (P) come $C = N + (P - O)$.

Diminuendo la dimensione iniziale della finestra di flow-control è possibile raggiungere valori negativi di credito: questa non è una situazione anomala e deve essere correttamente gestita da un peer evitando di inviare frame soggetti a controllo di flusso fino a quando il credito disponibile sullo stream e sulla connessione non torna positivo.

Non è possibile per un peer disabilitare completamente il meccanismo di controllo di flusso per i frame in ricezione. Un effetto simile si può raggiungere inviando un frame WINDOW_UPDATE in seguito alla ricezione di ogni frame soggetto a flow-control, ripristinando ogni volta la dimensione totale della finestra di controllo di flusso. Sono necessari due frame WINDOW_UPDATE separati per ripristinare la flow-control window sia per la connessione che per lo stream.

Sebbene sia possibile inviare frame WINDOW_UPDATE per dimensioni arbitrarie maggiori di 0, è consigliato per un ricevitore di frame DATA evitare di inviare gli incrementi alla dimensione delle finestre di flow-control utilizzando valori molto piccoli: questa tecnica di controllo di flusso può soffrire infatti del problema della "Silly Window Syndrome", che si verifica quando i dati vengono inviati dividendoli in blocchi molto piccoli, ciascuno con un overhead associato (in questo caso costituito dai frame header). [24] In HTTP/2 questo fenomeno può avere l'effetto di inviare

molti frame DATA ciascuno con payload di dimensione ridotta, peggiorando le prestazioni di utilizzo del canale a causa dei 9 byte di header di ogni frame data ripetuti ad ogni frame inviato.

Tra i tipi di frame definiti nell'RFC 9113 gli unici soggetti a controllo di flusso sono i frame DATA, estensioni al protocollo HTTP/2 possono modificare questo comportamento previa negoziazione.

2.9. Gestione degli errori

Il protocollo HTTP/2 mette a disposizione due tipi di frame diversi per segnalare condizioni di errore al peer, distinguendo tra errori che riguardano lo stato di tutta la connessione (frame GOAWAY, sez 2.3.9) e errori limitati ad uno stream specifico (frame RST_STREAM, sez 2.3.5).

Entrambi i tipi di frame riportano un codice di errore e i frame GOAWAY possono contenere anche informazioni di debug aggiuntive all'interno del payload. É definito un codice di errore particolare (NO_ERROR) per segnalare condizioni di terminazione naturale della connessione.

I frame GOAWAY contengono lo stream identifier dell'ultimo stream che è stato considerato dall'endpoint. In questo modo, il ricevitore del frame ha la certezza che richieste inviate su stream successivi a quello specificato potranno essere inviate senza controindicazioni in una nuova connessione, anche se queste non sono idempotenti.

2.10. Creazione della connessione

Il protocollo HTTP/2 può essere utilizzato con connessioni crittografate nello stesso modo con cui il protocollo HTTP/1.1 viene usato tradizionalmente in HTTPS.

Dato che la connessione TLS che deve trasportare la comunicazione di HTTP/2 viene effettuata alla stessa porta di una creata da un client che intende utilizzare esclusivamente HTTP/1.1 viene utilizzato il meccanismo di negoziazione ALPN (Application-Layer Protocol Negotiation), un'estensione del protocollo TLS: questo strumento consente di utilizzare l'infrastruttura già esistente per il traffico HTTPS in modo trasparente per tutti gli altri livelli di rete, senza aumentare i tempi di latenza o richiedere round trip aggiuntivi. [5][25]

Questo metodo di negoziazione è stato creato per le necessità specifiche di HTTP/2 ma può essere utilizzato per negoziare l'utilizzo arbitrari protocolli a livello applicazione trasportati con TLS. [5]

Durante l'handshake TLS il client invia una lista di protocolli supportati per la comunicazione in quel canale, ciascuno identificato da una sequenza di caratteri ("h2" per HTTP/2, "http/1.1" per HTTP/1.1). Il server seleziona poi uno dei protocolli proposti dal client: se non c'è sovrapposizione tra l'insieme dei protocolli proposti dal client e quelli supportati dal server l'handshake fallisce. Se uno dei due peer non supporta ALPN o la sequenza di caratteri "h2" non è presente nella lista proposta dal client può essere usato come fallback il protocollo HTTP/1.1.

Una volta stabilita la connessione e negoziato con successo l'utilizzo del protocollo HTTP/2, il client invia il proprio Connection Preface, composto da una sequenza di 24 byte predefinita seguita da un frame SETTINGS, opzionalmente vuoto. La sequenza di 24 byte, chiamata anche "Magic" (ad esempio in Wireshark), è la seguente: "PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n". Il Connection

Preface che il server deve inviare all'inizio della connessione è invece composto solo da un frame SETTINGS, opzionalmente vuoto.

2.11. HTTP/2 in chiaro

All'interno dell'RFC 7540 è definito un metodo per negoziare l'uso del protocollo HTTP/2 a partire da una connessione HTTP in chiaro: in una richiesta HTTP/1.1, un client disposto proseguire la connessione con HTTP/2 può segnalarlo tramite l'utilizzo dell'header Upgrade, specificando come identificatore del protocollo il token "h2c" (h2 cleartext).

Questo meccanismo non ha mai avuto nessun supporto su larga scala ed è stato deprecato nel più recente RFC 9113.

Il protocollo consente comunque un funzionamento in chiaro senza bisogno di una connessione TLS crittografata mantenendo esattamente gli stessi meccanismi previsti per HTTP/2 con lo schema https, a partire dall'invio del connection preface da parte del client. Non viene però definito nessun meccanismo di negoziazione, quindi il client deve avere una conoscenza pregressa riguardo al fatto che sulla porta a cui si sta effettuando la connessione sia attivo un server HTTP/2. In ogni caso, grazie alla presenza del connection preface all'inizio della comunicazione da parte del client, se la connessione viene effettuata erroneamente con un server HTTP/1.1 questi interpreterà i byte ricevuti come una richiesta malformata e tipicamente non continuerà nell'elaborazione degli altri frame inviati dal client sulla connessione. [4]

2.12. Sviluppi successivi: HTTP/3

Il protocollo HTTP/3 è stato sviluppato per risolvere alcuni dei problemi legati all'utilizzo del TCP per il protocollo HTTP/2, utilizzando invece per il livello di trasporto il protocollo QUIC basato su UDP.

Il nuovo protocollo presenta molte similarità con quello studiato in questa tesi: sono presenti meccanismi di framing e di compressione degli header leggermente modificati, mentre la gestione degli stream di dati è delegata a QUIC. [26]

Grazie all'utilizzo di UDP per il trasporto degli stream si risolve il problema dell'Head of Line blocking delle connessioni TCP: con HTTP/2, se un pacchetto viene perso o rallentato questo va a impattare le prestazioni di tutti gli stream della connessione, anche se il pacchetto conteneva dati che riguardano solamente uno stream. Il protocollo QUIC, invece, permette di consegnare i dati associati ai diversi stream in modo indipendente tra di loro, evitando che la perdita di un pacchetto possa creare un ritardo per richieste non collegate.

Il problema del HoL blocking di TCP è particolarmente accentuato in caso di connessioni trafficate o instabili [27], come osservato anche nella sezione 4.5 riguardante le performance al variare del numero di connessioni.

3. Implementazione

3.1. Generazione dei certificati

Per la costruzione di una connessione sicura tra un client e il web server di esempio utilizzato per i test è necessario disporre di un certificato TLS ritenuto attendibile dal client.

Per la generazione di questo certificato vengono utilizzati gli automatismi definiti nel protocollo ACME [6], che mette a disposizione diverse modalità per dimostrare di avere il controllo sull'entità (dominio o indirizzo IP) per cui si vuole generare il certificato.

Per la generazione dei certificati di questo web server è stata utilizzata la "DNS-01 challenge", una procedura che permette di dimostrare di controllare il record DNS a cui il server sarà associato. In questa challenge, l'ente certificatore fornisce una stringa da inserire in un record TXT associato al nome del dominio da certificare. Quando questa stringa viene inserita e la sua presenza viene verificata dall'ente certificatore (chiamato anche CA: "Certification Authority") viene emesso il certificato. [28]

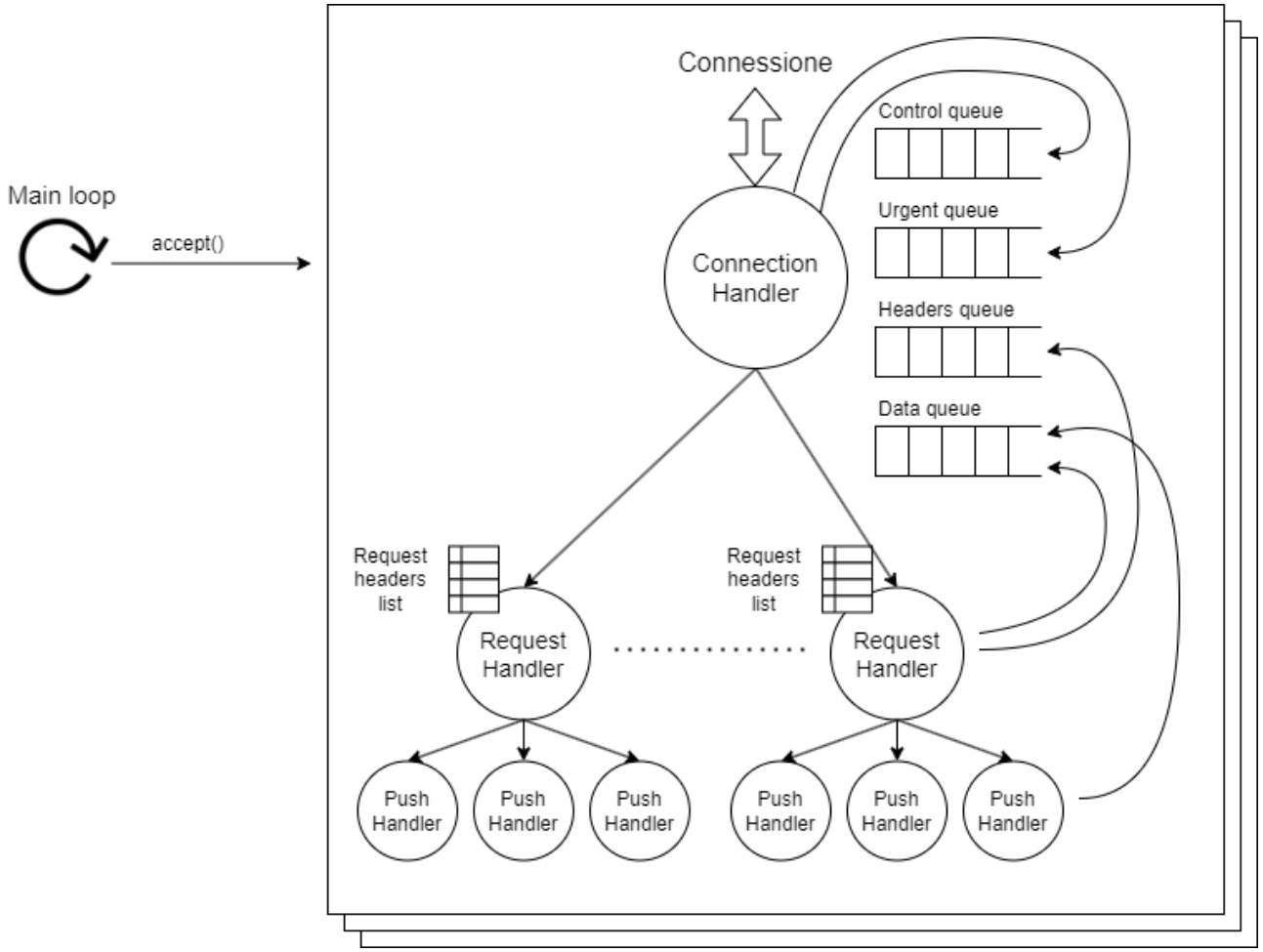
L'ente certificatore utilizzato è Let's Encrypt mentre il provider DNS con cui è stato registrato un dominio per il server è Duck DNS, in quanto entrambi questi servizi mettono a disposizione delle API gratuite per svolgere questa procedura.

Concretamente, la gestione del certificato è stata effettuata tramite una versione modificata del client ACME "Dehydrated", che grazie alla libreria Openssl automatizza la sequenza di richieste da inviare all'ente certificatore e al provider DNS e genera le chiavi private per il certificato TLS ottenuto al termine della challenge DNS. [29][30]

Dopo aver creato automaticamente un account presso Let's Encrypt, il client ACME invia una richiesta per la creazione del certificato specificando i dati da inserire nel certificato tra cui il Common Name, ovvero il dominio per cui si vuole generare il certificato. Nella risposta della CA è presente un URL a cui accedere per avere le informazioni sulla challenge da completare, tra cui un URL per ottenere il token e un altro per richiedere la verifica del completamento della challenge. Dopo aver ottenuto il token della challenge, il client richiede al provider DNS (in questo caso Duck DNS) tramite le API messe a disposizione la creazione di un record TXT associato al dominio con il contenuto che la CA deve verificare. A questo punto, viene segnalato il completamento della challenge a Let's Encrypt e, quando la verifica è completata, si ottiene il certificato firmato.

3.2. Schema di funzionamento web server

Il funzionamento del web server di esempio sviluppato per le misurazioni può essere schematizzato così:



Questo server si organizza in base a quattro categorie di processo principali: Main Loop, Connection Handler, Request Handler e Push Handler.

3.2.1. Main Loop

Questo è il processo principale in esecuzione nel server. Oltre alla configurazione iniziale del contesto SSL (che verrà applicata a tutte le connessioni crittografate) vengono accettate le connessioni entranti per la porta configurata, ciascuna delle quali verrà gestita da un processo Connection Handler dedicato.

3.2.2. Connection Handler

Ogni processo Connection Handler è associato ad una connessione con un client. La prima operazione che viene fatta per ogni connessione è l'handshake TLS, durante il quale viene selezionato il protocollo applicativo da utilizzare tra HTTP/1.1 e HTTP/2.

Se viene negoziato l'utilizzo di HTTP/1.1, il Connection Handler risponde sequenzialmente alle richieste che gli vengono inviate dal client, fino alla sua disconnessione. In questo caso viene

utilizzato esattamente un processo per connessione, senza fare uso di altri processi Request Handler o Push Handler. Il corpo delle risposte inviate con HTTP/1.1 utilizza il Chunked Transfer Encoding con una dimensione massima dei chunk di 8192 byte.

Per la gestione di una connessione HTTP/2, invece, dopo l'invio del Connection Preface del server (sezione 2.10 "Creazione della connessione") si entra in un ciclo in cui, fino alla ricezione di un frame GOAWAY o alla disconnessione del client, si gestisce lo scambio di frame in maniera non bloccante come descritto nella sezione 3.4 "Comunicazioni non bloccanti".

In particolare, quando viene ricevuto un blocco di uno o più frame che può essere elaborato, se è un frame HEADERS (con tutti gli eventuali frame CONTINUATION) viene decodificato e la lista degli headers HTTP viene gestita da un nuovo processo "Request Handler", altrimenti il frame viene gestito direttamente dal Connection Manager.

3.2.3. Request Handler

Un Request Handler viene creato in seguito alla completa ricezione e alla decodifica di una richiesta HTTP. Questo processo si occupa di interpretare gli header, verificare se la risorsa richiesta esiste e preparare i frame della risposta da inviare.

La risorsa da inviare è individuata grazie allo pseudo-header ":path". Se non è presente un file al percorso relativo desiderato viene generata una risposta 404 senza nessun frame DATA, altrimenti il frame HEADERS della risposta ha status code (pseudo-header ":status") 200 e vengono generati i frame DATA per la risposta.

I frame HEADERS e DATA generati per la risposta vengono inviati in due code IPC distinte (Headers queue e Data queue) come descritto alla sezione 3.4 "Comunicazioni non bloccanti".

Successivamente all'invio dei frame HEADERS ma prima di iniziare la lettura del file e la conseguente generazione dei frame DATA, se il client acconsente all'utilizzo del Server Push e questa funzionalità è attivata sul server vengono generati i frame PUSH_PROMISE e HEADERS per le risorse da inviare tramite PUSH. Le eventuali risorse aggiuntive da inviare sono configurate in modo specifico per ogni risorsa che il client può richiedere. Per ogni risorsa aggiuntiva da inviare, dopo l'invio nella coda "Headers queue" dei frame PUSH_PROMISE e HEADERS per quel push viene creato un processo Push Handler per l'invio dei frame DATA per quella risorsa.

3.2.4. Push Handler

Un processo di questo tipo viene creato per l'invio di frame DATA relativi ad una risorsa aggiuntiva da inviare tramite Server Push.

Data una richiesta del client, oltre al processo Request Handler relativo alla risposta di quella richiesta possono essere utilizzati molto processi Push Handler, uno per ogni risorsa da inviare tramite Push.

3.3. Gestione della connessione TLS

La crittografia nelle connessioni utilizzate dal server è stata gestita utilizzando la libreria Openssl.

Per la creazione di una connessione TLS crittografata è prima necessario costruire un oggetto SSL_CTX (detto anche contesto SSL [31]), che contiene alcuni parametri di configurazione per la creazione delle connessioni. [32]

In questo server la creazione del SSL_CTX viene effettuata all'inizio dell'avvio del server, prima di iniziare ad accettare connessioni da parte dei client.

Per la creazione di un SSL_CTX per un server web HTTP/2 sono necessari alcuni passaggi, eseguiti con il codice seguente:

```
SSL_METHOD* method = TLS_server_method();
SSL_CTX* ssl_ctx = SSL_CTX_new(method);
SSL_CTX_set_min_proto_version(ssl_ctx, TLS1_VERSION);
SSL_CTX_use_certificate_file(ssl_ctx,HTTPS_CERTIFICATE_PATH,SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ssl_ctx,HTTPS_PRIVATE_KEY_PATH,SSL_FILETYPE_PEM);
SSL_CTX_set_alpn_select_cb(ssl_ctx, &select_alpn_callback, NULL);
SSL_CTX_set_mode(ssl_ctx, SSL_MODE_ENABLE_PARTIAL_WRITE);
```

In questo frammento di codice sono stati rimosse le parti per la gestione di eventuali errori durante la creazione e la configurazione.

HTTPS_CERTIFICATE_PATH e HTTPS_PRIVATE_KEY_PATH sono parametri di configurazione per individuare il certificato e la chiave privata per la connessione HTTPS generati come descritto alla sezione 3.1.

La struttura SSL_METHOD usata per la creazione di SSL_CTX è usata per indicare il tipo di connessione da creare (in questo caso TLS) e se deve rappresentare un client o un server. [32]

Dopo l'ottenimento del contest SSL, vengono applicate alcune configurazioni, nell'ordine:

- Impostazione del protocollo minimo che può essere usato per la crittografia: viene scelto come protocollo minimo il TLS 1.0, seguendo le linee guida di Openssl. [33]
- Configurazione del file con il certificato pubblico e la relativa chiave privata per il server.
- Impostazione della callback per la selezione del protocollo applicativo con ALPN (descritta in seguito).
- Abilitazione delle scritture parziali per avere un comportamento di SSL_write analogo a quello della funzione write della libreria standard C, ovvero permettendo di scrivere solo una parte dei byte specificati nella connessione per ogni esecuzione della funzione.

Come già descritto nella sezione 2.10 "Creazione della connessione", il protocollo ALPN permette di selezionare un protocollo applicativo tra una lista di opzioni fornite dal client. Con la libreria Openssl, la selezione del protocollo viene fatta all'interno di una callback chiamata durante l'handshake (select_alpn_callback nella porzione di codice precedente). In questa callback, una lista dei protocolli accettabili per il client formattata in questo modo viene passata come parametro:

```
uint8_t alpn_protos_array[] = { 2, 'h', '2', 8, 'h', 't', 't', 'p', '/', '1', '.', '1' };
```

Per restituire il protocollo selezionato sono messi a disposizione due parametri in uscita: uno per indicare la lunghezza della stringa selezionata e un altro per un puntatore ai suoi caratteri.

Dopo aver fatto l'accept per una connessione entrante, sono necessarie alcune azioni aggiuntive per rendere la connessione crittografata:

- Creare un oggetto SSL che rappresenta la connessione con il client a partire dalle impostazioni del contesto SSL (funzione SSL_new).
- Associazione del file descriptor ritornato dalla accept all'oggetto SSL appena ottenuto (SSL_set_fd).
- Completamento dell'handshake TLS nella connessione con il client (SSL_accept).

A questo punto, tramite SSL_get0_alpn_selected si ottiene il protocollo di livello applicazione selezionato durante l'handshake, gestendo la comunicazione in base al protocollo negoziato:

```
SSL_get0_alpn_selected(ssl, (const unsigned char **) &alpn_data_ptr, &t);
if(t == 2 && alpn_data_ptr[0] == 'h' && alpn_data_ptr[1] == '2'){
        // Token h2: HTTP/2 negoziato
}else{
        // Se non è h2 la connessione viene trattata come HTTP/1.1
}
```

Una volta completate queste operazioni, lettura e scrittura sulla connessione si possono effettuare tramite SSL_read e SSL_write in modo analogo a quanto viene fatto tradizionalmente con le chiamate di sistema read e write, sostituendo al socket l'oggetto SSL.

3.4. Comunicazioni non bloccanti

Dallo schema alla sezione 3.2 si può vedere che un processo Connection Handler si occupa sia della lettura che della scrittura dei frame sulla connessione TLS.

Seguendo lo schema di funzionamento si ha che la risposta per ogni richiesta viene preparata indipendentemente, e si rende quindi necessario un meccanismo per leggere i frame ricevuti sulla connessione e inviare i frame per tutte le risposte contemporaneamente. Per ottenere questa funzionalità i frame da inviare vengono condivisi tra i vari Request Handler e Push Handler e il singolo Connection Handler attraverso un meccanismo a scambio di messaggi: in questo modo si evitano problematiche legate all'accesso concorrente da parte di più processi ad una stessa regione di memoria (che ci sarebbero state con un approccio di condivisione dei frame da inviare basato su un buffer di memoria condiviso). Questo scambio di messaggi utilizza i socket UNIX e per separare i socket di diverse connessioni viene usato il pid del Connection Handler come parte del nome dei socket.

I messaggi inviati al Connection Handler sono divisi in quattro categorie, andando a formare quindi quattro code: Control queue, Urgent queue, Headers Queue e Data queue. La divisione tra queste quattro code si rende necessaria per la diversa natura dei messaggi che possono essere inviati.

Nella Control queue vengono inviati messaggi di controllo (non associati a nessun frame) per richiedere un'azione al Connection Handler. Come descritto nella sezione 3.6.3 riguardo alla costruzione dei frame PUSH_PROMISE, questa coda viene usata per inviare una richiesta per ottenere il primo stream identifier inutilizzato nello spazio degli stream apribili dal server.

La Urgent queue è stata creata per permettere di inviare dei frame con priorità alta senza doverli accodare assieme ai frame costituenti le risposte delle richieste ricevute. Esempi di questi frame possono essere i frame PING o SETTINGS con il flag ACK attivato.

Le ultime due code, Headers queue e Data queue, vengono usate per inviare i frame relativi alle risposte per le richieste inviate dal client. La coda Headers può essere utilizzata per inviare frame HEADERS e PUSH_PROMISE (con gli eventuali frame CONTINUATION successivi), mentre la Data queue viene usata per i frame di tipo DATA. Grazie a questa divisione, il controllo di flusso a livello di connessione viene applicato solo alla Data queue e non alle altre. Concretamente, dato che i messaggi vengono letti uno alla volta dalla coda e non vengono estratti fino a che il relativo frame viene inviato, se nella connessione non è disponibile una finestra di flow-control abbastanza ampia per inviare il primo frame nella Data queue questa viene bloccata fino a quando questa situazione non cambia, e vengono inviati solo frame appartenenti alle code Urgent e Headers (nessuno di questi frame è infatti soggetto a controllo di flusso).

Per evitare situazioni di blocco, tutte le letture dalla connessione e dalle code dei messaggi sono non bloccanti. In questo modo, nel ciclo principale del Connection Handler si resta in attesa fino a quando non è possibile effettuare una lettura su almeno uno dei cinque socket (uno per la connessione e quattro per le code dei messaggi). Quando su almeno uno di questi cinque socket è possibile effettuare una lettura, si effettuano dei tentativi non bloccanti di lettura su ciascuno dei cinque socket con un ordine predefinito, fino a quando non si legge un frame intero dalla connessione o non si riceve un messaggio.

Il primo socket da cui viene tentata una lettura è quello associato alla connessione con il client, poi nell'ordine i tentativi di lettura vengono fatti sulla Control queue, sulla Urgent queue, sulla Headers queue e sulla Control queue. In questo modo, eventuali frame urgenti come PING o RST_STREAM possono essere interpretati subito senza dover aspettare di inviare tutte le risposte in coda.

Per ottenere letture e scritture non bloccanti, il socket a cui l'oggetto SSL è associato è stato reso non bloccante attivando i flag O_ASYNC e O_NONBLOCK tramite il comando fcntl.

Le letture e le scritture non bloccanti in questa connessione crittografate vengono effettuate con le stesse funzioni utilizzabili con socket bloccanti (SSL_read e SSL_write) ma vanno gestiti codici di errori aggiuntivi legati al fatto che l'operazione non possa essere completata senza bloccaggio (SSL_ERROR_WANT_READ e SSL_ERROR_WANT_WRITE).

3.5. Gestione della compressione con HPACK

Come già spiegato, HPACK viene usato per ottenere una lista di headers HTTP (coppie chiave-valore) a partire da una sequenza di byte ottenuta unendo il payload di un frame HEADERS e quello di tutti gli eventuali frame CONTINUATION che lo seguono, oltre che per il procedimento inverso (costruire un frame HEADERS a partire da una lista di header HTTP).

3.5.1. Decodifica

In questo server, per ogni connessione la decodifica con HPACK viene effettuata nel Connection Handler, durante la gestione di una lista di frame HEADERS e CONTINUATION ricevuti. Lo stato

della compressione può essere associato quindi a questo unico processo, salvando come variabili globali la dimensione corrente della tabella dinamica e un puntatore per l'accesso a questa tabella.

La tabella dinamica viene gestita come lista concatenata, salvando per ogni entry il nome, il valore e la entry successiva. In questo modo, dato che gli inserimenti nella tabella dinamica vengono fatti all'inizio della lista, possono essere fatti immediatamente. Le operazioni di rimozione, ricerca e calcolo della dimensione della tabella, invece, sono fatte ogni volta scorrendo tutta la lista. In condizioni normali di utilizzo questo non dovrebbe costituire un overhead importante a livello di prestazioni, soprattutto in confronto ad altri difetti nell'implementazione del processo di decodifica che portano a ritardi significativi, ma in caso di connessioni longeve con dimensioni negoziate per tabella dinamica molto grandi anche i ritardi portati da questa componente possono diventare significativi.

Il parsing della sequenza di byte codificata avviene iterativamente: in base al contenuto del primo byte non elaborato viene identificato il tipo di header ricevuto e i byte successivi vengono interpretati di conseguenza. Il numero di byte per ogni header è variabile.

Il primo byte non elaborato può contenere un'indicazione riguardo ad un aggiornamento della dimensione della tabella dinamica: in quel caso, i byte successivi al primo possono codificare la nuova dimensione della tabella.

In caso di header puramente indicizzato (sia nome che valore presenti all'interno della tabella) un numero variabile di byte successivi al primo viene utilizzato per codificare l'indice in cui l'header si trova all'interno della tabella.

Per tutti gli altri tipi di header, il nome può essere indicizzato o codificato come stringa (come descritto di seguito) mentre il valore è sempre descritto come stringa. L'aggiunta di tali header alla tabella avviene solo per gli header di tipo "Literal header field with incremental indexing", mentre i tipi "without indexing" e "never indexed" sono trattati allo stesso modo non aggiungendoli alla tabella dinamica.

Ogni stringa può essere codificata direttamente con i propri caratteri ASCII oppure compressa attraverso un codice di Huffman definito all'interno dello standard HPACK.

Per la decodifica di una sequenza di bit codificata con Huffman, in questo server viene seguita la seguente procedura:

- Viene letto un bit alla volta dalla stringa da decodificare
- Il bit letto viene convertito in un carattere ('0' o '1') e viene aggiunto in coda ad una stringa contenente tutti i bit ancora da decodificare letti fino a quel momento
- Quella stringa viene confrontata per uguaglianza con tutte le stringhe corrispondenti alla codifica di Huffman di ogni carattere definite nello standard. Se viene trovata una sequenza di bit uguale a quella corrente il carattere a cui è associata viene aggiunto alla stringa decodificata, altrimenti si ricomincia il procedimento leggendo il bit successivo

Questa operazione di decodifica è molto dispendiosa: per ogni bit all'interno della stringa da decodificare viene effettuata una ricerca confrontando carattere per carattere la sequenza di '0' e '1' corrente con tutte le 256 stringhe della tabella di decodifica. Concretamente, si sono osservati tempi

di circa 10 ms per la decodifica di un blocco di header per una richiesta fatta con un web browser. Questi tempi si riducono a circa 1 ms dalla seconda richiesta in poi, in quanto gran parte degli header che vengono inviati sono già presenti all'interno della tabella dinamica.

Questa latenza non ci sarebbe stata con un'implementazione di questa operazione di decodifica di Huffman tramite una ricerca ad albero, in cui ogni bit della sequenza di byte da decodificare viene considerato solo una sola volta per spostarsi all'interno dell'albero di ricerca invece di richiedere per ogni bit fino a 256 confronti di uguaglianza tra stringhe di lunghezza variabile.

L'implementazione di una ricerca ad albero di questo tipo è però più complicata rispetto all'utilizzo del confronto tra stringhe e i tempi di latenza osservati non sono stati considerati tali da richiedere una riscrittura di questa procedura di decodifica per gli scopi di questa tesi.

Va comunque tenuto presente che questa latenza di decodifica è presente per tutte le richieste HTTP/2 ricevute da questo server e andrebbe quindi considerata nel valutare le prestazioni del protocollo.

Si nota, inoltre, che sebbene per una condizione di navigazione web normale questa latenza sia limitata a pochi millisecondi, in casi limite con blocchi di header delle dimensioni di molti KB si possono avere tempi di decodifica di qualche secondo.

3.5.2. Codifica

In questo server, la procedura di codifica HPACK viene utilizzata per la creazione dei blocchi HEADERS e PUSH_PROMISE per le risposte.

Per i blocchi HEADERS, l'unica informazione che viene specificata è lo status code (pseudo-header ":status"), quindi tutti i frame di questo tipo generati dal server hanno un unico byte di payload, in cui è codificato l'indice dello status code corretto. In particolare, possono essere generate risposte con status code 400 (in caso di richiesta malformata, ad esempio in cui non viene specificato nessun ":path"), 404 (file specificato non esistente) e 200 (file trovato).

Per la codifica del blocco di header in un frame PUSH_PROMISE, tutti gli header vengono codificati come literal ASCII, senza aggiungerli alla tabella dinamica di HPACK.

Ogni header è quindi codificato così:

- Il primo byte segnala che si sta inviando un "Literal header field without indexing", quindi un header in cui sia il nome che il valore vengono codificati come literal e non vengono aggiunti alla tabella
- Il byte successivo indica la lunghezza in byte del nome. Il primo bit messo a 0 in questo byte indica che i byte sono codificati come caratteri ASCII e non con il codice di Huffman
- I seguenti byte contengono i caratteri ASCII del nome dell'header

Il valore dell'header viene codificato subito dopo con la stessa modalità del nome.

3.6. Costruzione di una risposta

3.6.1. Frame HEADERS

Come già spiegato nella sezione sull'implementazione della codifica HPACK, in questo server gli unici frame HEADERS generati contengono soltanto un header, per lo pseudo-header ":status".

Ogni frame HEADERS ha quindi un payload della dimensione di 1 byte.

In questo byte il primo bit viene impostato a 1 per segnalare che sia il nome che il valore dell'header verranno letti dalla tabella di HPACK. I restanti 7 bit contengono l'indice all'interno della tabella in cui si trovano gli status code da inviare: indice 8 per lo status code 200, 12 per lo status 400 e 13 per lo status 404, secondo quanto definito nell'RFC 7541. [2]

Lo stream identifier inserito nel frame HEADERS è lo stesso della richiesta a cui è associato.

In ogni frame HEADERS viene attivato il flag END_HEADERS, mentre per gli status 400 e 404 viene attivato anche END_STREAM.

3.6.2. Frame DATA

Vengono forniti dei parametri di configurazione per impostare una dimensione massima dei frame DATA da inviare e per scegliere se riempire sempre fino alla dimensione massima questi frame.

Se si sceglie di non forzare il riempimento dei frame DATA, per il payload di ciascun frame verrà effettuata solo una lettura (con la chiamata a sistema read) per il file specificato nella richiesta. In questo modo, dato che per rilevare la fine del file da leggere è necessario che la read legga 0 byte, l'ultimo frame DATA inviato per la risposta avrà un payload di dimensione 0 e il flag END_STREAM attivato.

Volendo forzare sempre il raggiungimento della dimensione massima del payload, invece, vengono effettuate delle chiamate read fino a quando non si hanno abbastanza dati da inviare o il frame non è riempito completamente. In questo caso, a meno che il file non abbia una dimensione che è multiplo di quella dei frame da inviare, il flag END_STREAM sarà attivato nell'ultimo frame che trasporta i dati del file.

Valutando le prestazioni del server al variare di questi due parametri di configurazione (sezione 4.1 "Dimensione payload frame DATA"), per tutti i test la dimensione massima dei frame è stata fissata a 4096 byte, senza forzare il riempimento del frame.

3.6.3. Frame PUSH_PROMISE

In un frame PUSH_PROMISE, che va inviato nello stesso stream in cui è stata ricevuta la richiesta a cui è associato, vanno inseriti l'id del nuovo stream in cui la risorsa aggiuntiva verrà inviata e gli header della richiesta "fittizia" per cui la risorsa aggiuntiva che si vuole inviare è la risposta.

Il compito dell'assegnazione degli stream identifier per gli stream aperti dal server viene assegnato al Connection Handler, che è l'unico processo che riesce a tenere traccia del ciclo di vita e della storia della comunicazione in una connessione. Secondo il meccanismo di comunicazione tra

processi già descritto nella sezione sulle comunicazioni non bloccanti, viene inviato al Connection Handler un messaggio nella Control queue, contenente il pid del Request Handler da cui si sta inviando la richiesta. Il Request Handler resta quindi in attesa di ricevere un messaggio su un socket UNIX con il proprio pid nel nome, in modo da non avere conflitti con altri processi che dovessero fare la stessa operazione contemporaneamente. Il Connection Manager tiene traccia degli stream identifier che rilascia, inviando al Request Handler il primo numero pari non ancora assegnato.

Gli pseudo-header che vengono inviati per ciascuna richiesta fittizia sono ":method" (fissato a "GET"), ":scheme" (fissato a "https"), ":authority" (che contiene il nome dns con cui si accede al server) e ":path" (per il percorso relativo della risorsa). Tutti questi header sono codificati usando i loro caratteri ASCII e senza inserirli nella tabella dinamica di HPACK, come descritto nella sezione sull'implementazione del processo di codifica.

3.7. Limitazioni e non conformità

Durante la realizzazione del server web di esempio utilizzato per i test sono stati trascurati alcuni aspetti, portando sia a limiti prestazionali che non conformità con lo standard HTTP/2.

Come già descritto nella sezione riguardante i frame di tipo PRIORITY, in questo server vengono ignorati tutti i tipi di segnalazione di priorità inviati dal client, sia per il vecchio schema definito nell'RFC 7541 [4] che per quello nuovo dell'RFC 9218. [3] Sebbene questo non possa portare ad un peggioramento delle prestazioni per quanto riguarda la velocità del trasferimento cumulativa di tutte le risorse richieste, in questo modo non si usufruisce di uno strumento che potrebbe migliorare sensibilmente l'esperienza utente durante una navigazione web, in quanto risorse di calcolo e di rete vengono "sprecate" per risorse che non devono essere trasferite urgentemente per dare dei tempi di risposta rapidi.

Altri limiti alle prestazioni di questo server sono portati dai tempi di latenza della decodifica di HPACK, soprattutto per la prima richiesta effettuata sulla connessione, e per il mancato utilizzo della compressione degli header per tutti i messaggi HTTP in uscita. Entrambi questi aspetti sono già stati discussi nella sezione 3.5 riguardante la gestione dell'implementazione di HPACK.

Sebbene il server abbia un comportamento corretto per richieste inviate da client con un comportamento normale come i browser web più diffusi o lo strumento da riga di comando utilizzato per fare le misurazioni (h2load), la sua gestione di molti casi limite non è conforme allo standard HTTP/2. Alcuni esempi di situazioni critiche non rilevate o non gestite correttamente sono errori nel dimensionamento dei frame, transizioni non consentite tra gli stati degli stream ed errori durante la decodifica con HPACK: in tutti questi casi il server si comporta ignorando queste situazioni, senza segnalare all'altro peer nessun tipo di errore.

Un altro aspetto in cui il server non è conforme allo standard è quello che prevede di imporre dei limiti fissi per tutte le componenti di lunghezza variabile che possono essere inviate dal client. Ogni endpoint, infatti, dovrebbe ad esempio stabilire un limite di lunghezza degli header oltre al quale far fallire la decodifica e segnalare un errore, per proteggersi da attacchi ad esaurimento di risorse. Nessuno di questi aspetti di sicurezza è stato preso in considerazione per questo server.

Questo server, infine, considera solo i meccanismi di controllo di flusso a livello di connessione, ignorando completamente le limitazioni che possono essere imposte dall'altro peer a livello di stream. Non si sono però mai osservate situazioni generate da client tradizionali in cui questo comportamento può costituire un problema.

3.8. Realizzazione di un client web HTTP/2

3.8.1. Componenti minime

Gran parte delle componenti necessarie per un client web sono già state realizzate per la scrittura del server, come la gestione della compressione con HPACK, il parsing dei frame e la gestione di alcuni tipi di frame (come PING e SETTINGS).

La scrittura dei frame HEADERS può essere effettuata con la stessa tecnica di codifica dei frame PUSH_PROMISE del server, scrivendo tutti gli headers come literal ASCII non compressi senza utilizzare la tabella dinamica.

Per un client web che debba effettuare solamente una richiesta al server possono essere usati i meccanismi di lettura e scrittura bloccanti usati di default, senza dover ricorrere ad uno schema di scambio di messaggi tra processi e alle letture non bloccanti come fatto per il server.

Sono però necessarie alcune parti aggiuntive non presenti nel server, relative alla connessione con il server e all'invio dei frame WINDOW_UPDATE per consentire la ricezione di payload arbitrari nella risposta.

3.8.2. Modifiche rispetto al server

Oltre alle operazioni effettuate normalmente da un client web per connettersi ad un server, ad esempio la risoluzione dell'indirizzo ip a partire dal nome di dominio (funzione gethostbyname) e la creazione della connessione TCP (chiamata a sistema connect), sono necessari dei passaggi aggiuntivi per avere una negoziazione corretta di HTTP/2 con TLS.

In modo analogo a quanto visto per il server, le prime operazioni da svolgere per configurare la connessione sono l'ottenimento del metodo da usare per il client (TLS_client_method) e la creazione del SSL_CTX (SSL_CTX_new).

Con SSL_CTX_set_verify è possibile scegliere il comportamento da tenere in caso di errori nella verifica del certificato fornito dal server: con SSL_VERIFY_PEER viene fatto fallire immediatamente l'handshake in caso di errori durante la verifica, mentre con SSL_VERIFY_NONE l'handshake viene completato in ogni caso. [34]

Va poi costruita la lista dei protocolli da inviare durante la negoziazione ALPN secondo lo stesso formato definito per la callback di selezione del server, impostandola per le connessioni da creare con la seguente chiamata:

```
SSL_CTX_set_alpn_protos(ssl_ctx, alpn_protos_array, sizeof(alpn_protos_array));
```

Prima dell'esecuzione dell'handshake TLS con SSL_connect, in alcuni casi è necessario impostare il nome del server a cui ci si sta connettendo tramite la funzione SSL_set_tlsext_host_name. [35]

Quando l'handshake TLS è completato, l'eventuale protocollo a livello applicativo negoziato da ALPN è ottenibile nello stesso modo in cui lo è nel server.

Rispetto al server, una volta negoziato l'utilizzo di HTTP/2 è differente anche la gestione dei Connection Preface. Un client, infatti, deve come prima cosa inviare i 24 byte del magic seguiti da un frame SETTINGS, gestendo il Connection Preface inviato dal server che consiste solamente in un frame SETTINGS.

Un altro meccanismo non presente nel web server di esempio che deve essere aggiunto per avere un client funzionante deve essere quello dell'invio dei frame WINDOW_UPDATE in seguito alla ricezione di frame DATA. Una semplice soluzione potrebbe essere quella di inviare un frame WINDOW_UPDATE in risposta alla ricezione di ogni frame DATA con payload di almeno un byte, ma questa strategia soffre delle problematiche già descritte nella sezione 2.8 "Flow-control".

4. Misurazioni e analisi

4.1. Dimensione payload frame DATA

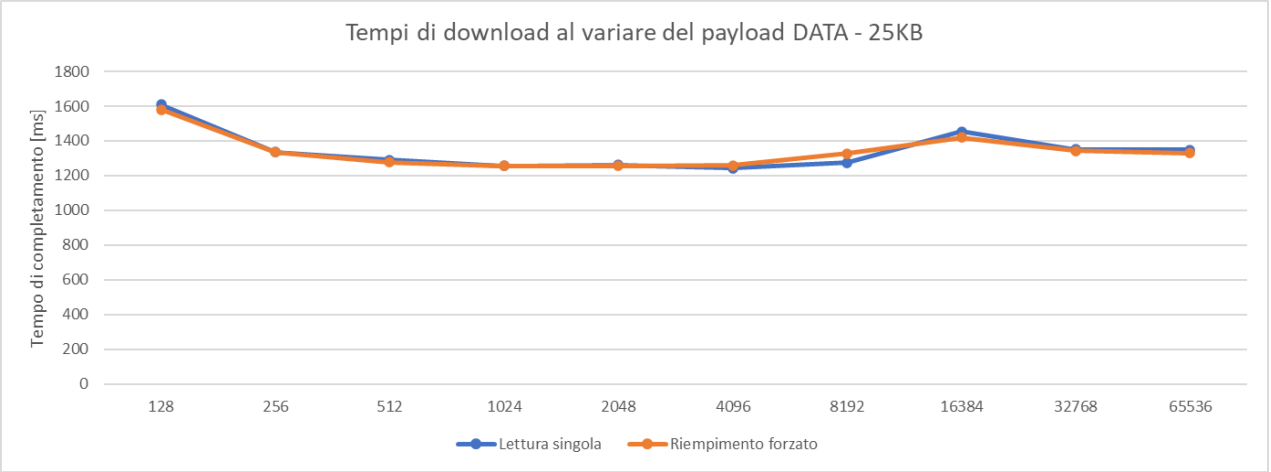
Come descritto nella sezione 3.6.2 del capitolo riguardante l'implementazione del web server, ogni peer può scegliere autonomamente come dimensionare i frame DATA da inviare.

In questa sezione si analizza come variano le prestazioni del server web al variare della dimensione massima dei frame DATA costruiti. Viene valutato inoltre se forzare sempre il raggiungimento della dimensione massima del payload finchè sono presenti dati a sufficienza per farlo può influire sui risultati rispetto all'invio di ogni risultato delle letture da file in un frame DATA a sé.

Per ogni valore dei parametri da valutare, il test consiste nel misurare il tempo di completamento di 100 sessioni, in ciascuna delle quali vengono effettuate 100 richieste simultanee per una risorsa delle dimensioni di 25 KB, senza porre un limite al numero massimo di stream aperti contemporaneamente. Viene registrata poi la media dei tempi di completamento di ogni sessione per una data combinazione di parametri.

Sono stati scelti questi valori per ogni sessione in base alla struttura tipica di una pagina web: come analizzato nei report di HTTP Archive, la dimensione totale delle risorse richieste in una pagina web è di circa 2.2MB e per ogni pagina vengono fatte mediamente tra le 80 e le 120 richieste (queste statistiche possono variare a seconda della quantità di siti web considerati e del periodo analizzato, ma l'ordine di grandezza rimane lo stesso). [10] Si ha quindi che la dimensione media delle risorse richieste per il caricamento di una pagina è di circa 25 KB, e per simulare un traffico simile a quello della navigazione web si misura il tempo di completamento di 100 richieste.

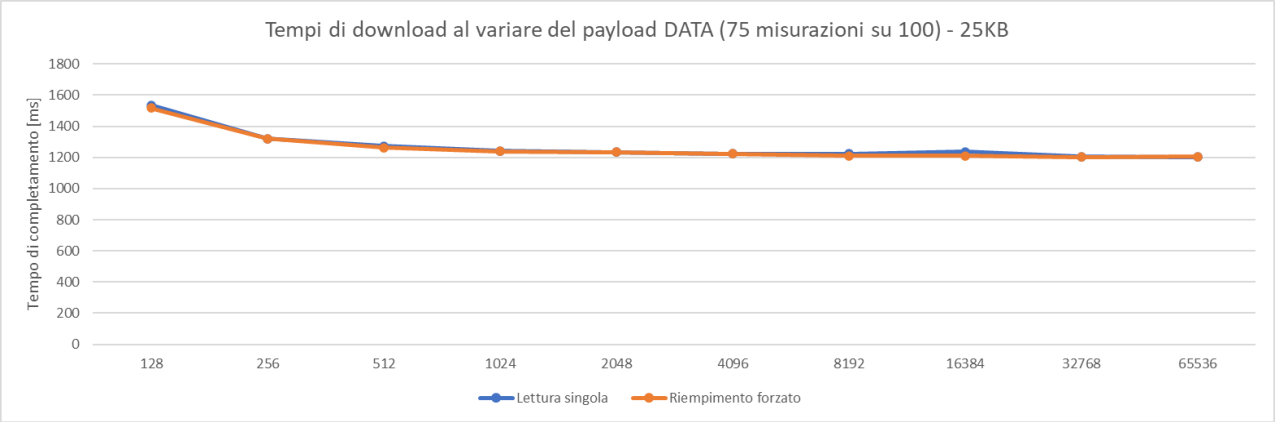
L'andamento medio dei tempi di completamento osservato facendo variare la dimensione del payload come potenza di 2 è il seguente:



Si osservano i valori più alti per dimensioni di frame molto piccole, a causa di un un maggiore contributo dei frame header sul totale dei byte inviati. All'aumentare della dimensione massima del payload le prestazioni migliorano, fino a raggiungere il tempo di download minimo per frame di 4096 KB.

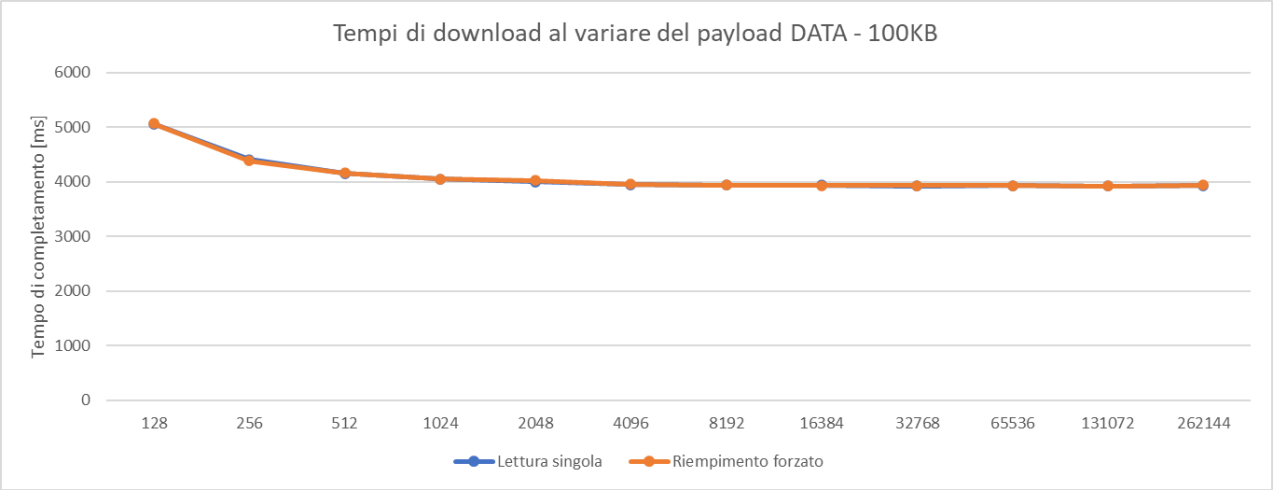
Dopo questo minimo, i tempi di completamento salgono per valori di 8 KB e 16 KB, per poi stabilizzarsi per dimensioni di frame maggiori.

Restringendo l'analisi alle 75 misurazioni con tempi inferiori per ogni valore dei parametri (sul totale di 100 misurazioni) non si osserva nessun picco per valori maggiori di 8 KB, quindi l'aumento dei tempi di caricamento osservato nel grafico precedente può essere imputato a ritardi che non sono presenti per ogni misurazione, dovuti al traffico della rete.



Per tutti i valori della dimensione dei frame, l'andamento che si ha con riempimento forzato segue molto da vicino quello osservato con la lettura singola.

In base a quanto osservato, per tutti gli altri test il server HTTP/2 descritto in questa tesi viene configurato con una dimensione massima del payload dei frame DATA massima di 4096 byte, senza forzare il riempimento di ogni frame.



Ripetendo l'analisi con richieste per una risorsa da 100KB, si osserva un andamento molto simile a quello del grafico precedente, con la differenza che la curva discendente dei tempi di completamento si stabilizza per valori più grandi della dimensione dei frame. Questo suggerisce che la dimensione migliore dei frame DATA potrebbe dipendere anche dalle dimensioni della risorsa da inviare.

I dati relativi al riempimento forzato seguono lo stesso andamento di quelli con la lettura singola, analogamente a quello che si verifica per 25 KB.

Uno dei risultati più rilevanti legati all'efficienza del protocollo HTTP/2.0 è il fatto che la prestazione rilevata in termini di capacità a livello di trasporto (Bulk Capacity) ottenuta dividendo l'intero volume dei dati per il tempo di scaricamento complessivo subisce una variazione ridotta (20%), anche se non trascurabile, al variare della dimensione media della risorsa di un fattore 4, come mostrato in tabella

Dimensione risorsa [KB]	Num. accessi	Volume totale [KB]	Durata [ms]	Bulk Capacity [MB/s]
25.00	100	2500.00	1244.00	2.01
100.00	100	10000.00	3924.40	2.54

Registrando le risposte con status code 200 che si ottengono effettuando delle richieste ad alcuni siti web in produzione, si osserva che la soluzione più comune per il dimensionamento dei frame DATA è quella di utilizzare una dimensione massima di 8192 byte o, meno di frequente, 16384 byte, in entrambi i casi consentendo l'emissione di frame che non raggiungono la dimensione massima anche se esistono altri dati da inviare.

Questo è coerente con quanto è già noto per il dimensionamento dei chunk nell'ambito del Chunked Transfer Encoding usato in HTTP/1.1, in cui 8KB costituisce lo standard di fatto per la dimensione dei blocchi. [36]

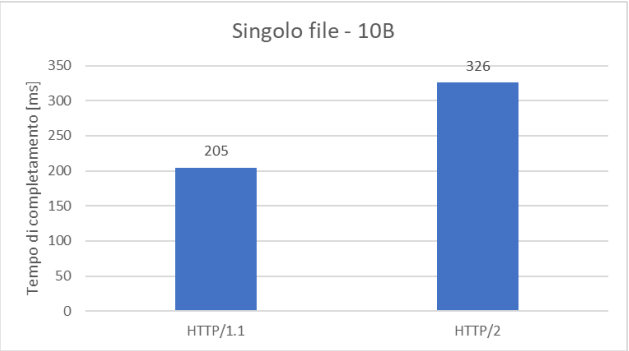
4.2. Trasferimento singolo file

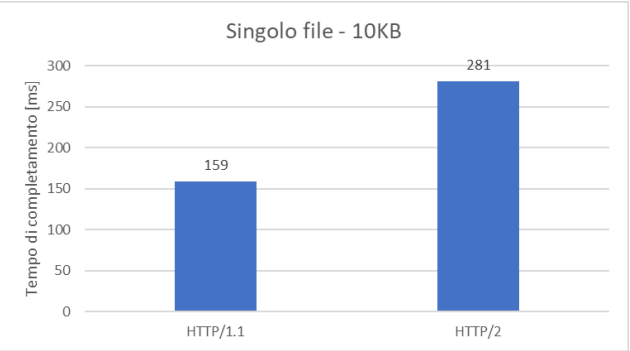
Per confrontare le prestazioni del protocollo HTTP/2 con quelle di HTTP/1.1 per quanto riguarda il tempo di completamento di una sola richiesta, viene effettuato più volte il download di una risorsa con ciascuno dei due protocolli da analizzare. Viene eseguita questa operazione per risorse di dimensioni diverse (10 B, 10 KB e 10 MB), registrando i tempi di completamento di ciascuna richiesta.

Tutte le richieste vengono fatte sequenzialmente creando per ciascuna di esse una nuova connessione al server: in questo modo si può confrontare l'impatto delle operazioni iniziali da svolgere in una connessione HTTP/2, che influiscono sul tempo di completamento di una singola richiesta ma possono essere "ammortizzate" in caso di connessioni utilizzate tra più richieste.

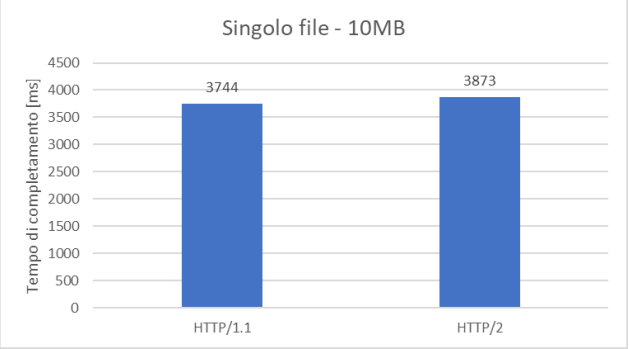
Le dimensioni massime dei frame DATA di HTTP/2 e dei chunk di HTTP/1.1 sono state impostate entrambe a 4096 byte, per permettere un confronto del funzionamento dei due protocolli in condizioni di lavoro identiche.

Per ogni tipo di richiesta vengono fatte 200 misurazioni, scartando le 50 più alte di ogni gruppo per eliminare occasionali tempi sproporzionatamente elevati che potrebbero modificare le statistiche raccolte. Per ciascuno dei gruppi con le 150 misurazioni rimanenti viene valutata la media.





Si osservano tempi di completamento significativamente più alti utilizzando il protocollo HTTP/2 rispetto ad HTTP/1.1 per il trasferimento di file di piccole dimensioni (10B e 10KB). Questo è dovuto ai tempi aggiuntivi e alla maggiore complessità per il setup iniziale della connessione a livello applicativo e la sua gestione con il nuovo protocollo. Queste fonti di ritardo non possono essere controbilanciate dalla compressione degli header, che potrebbe costituire l'unico vantaggio possibile del nuovo protocollo rispetto al setup sperimentale sviluppato, ma che potrebbe portare comunque ad una riduzione di un numero assolutamente trascurabile di bytes trasmessi.



Ovviamente, per il trasferimento di un file di dimensioni maggiori (10MB), invece, i due protocolli si rivelano sostanzialmente equivalenti, con una differenza nei tempi di download inferiore al 4%.

In tutti e tre i confronti si osserva un aumento di circa 120÷130 ms nei tempi di download di un singolo file passando da HTTP/1.1 a HTTP/2 a prescindere dalle dimensioni del file stesso. Questa differenza ha un impatto relativo maggiore per risorse più piccole, mentre costituisce solo una piccola variazione nei tempi di download di file più grandi.

4.3. Traffico tipico pagine web

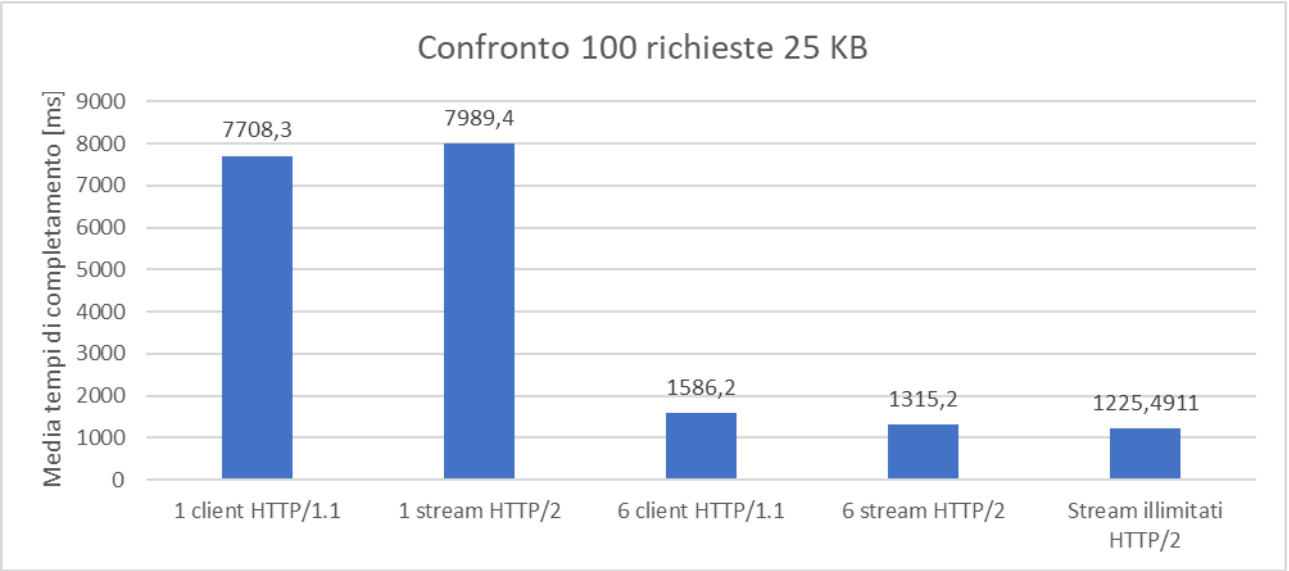
Come già spiegato nella sezione 4.1 "Dimensione payload frame DATA", una pagina web può effettuare tipicamente un centinaio di richieste prima di terminare il caricamento, ciascuna delle quali con una risposta delle dimensioni medie dell'ordine dei 25 KB.

Si confrontano quindi le prestazioni dei due protocolli in questa situazione tipica della navigazione web, misurando il tempo di completamento di 100 richieste per risorse di 25 KB.

Per il protocollo HTTP/1.1 vengono misurate le performance considerando un solo client e 6 client in parallelo, mentre per HTTP/2 si effettua la misura facendo variare il numero massimo di stream aperti contemporaneamente (1, 6 e illimitato). In questo modo, oltre a simulare la condizione

normale in cui si trovano i browser moderni (circa 6 client per HTTP/1 [9], stream illmitati per HTTP/2) si possono confrontare anche i tempi fissando il numero di trasferimenti contemporanei allo stesso valore.

Ogni misurazione viene fatta 100 volte e viene valutata la media dei tempi di completamento.



Si osservano prestazioni sostanzialmente equivalenti (differenza tra i tempi sotto al 4%) tra i due protocolli per il caso singolo client/singolo stream, in quanto il funzionamento per una richiesta alla volta del protocollo HTTP/2 è molto simile a quello di HTTP/1.1.

Nel confronto tra 6 client e 6 stream emerge un miglioramento nelle prestazioni del nuovo protocollo rispetto al predecessore, con una riduzione nella media dei tempi osservati del 17%.

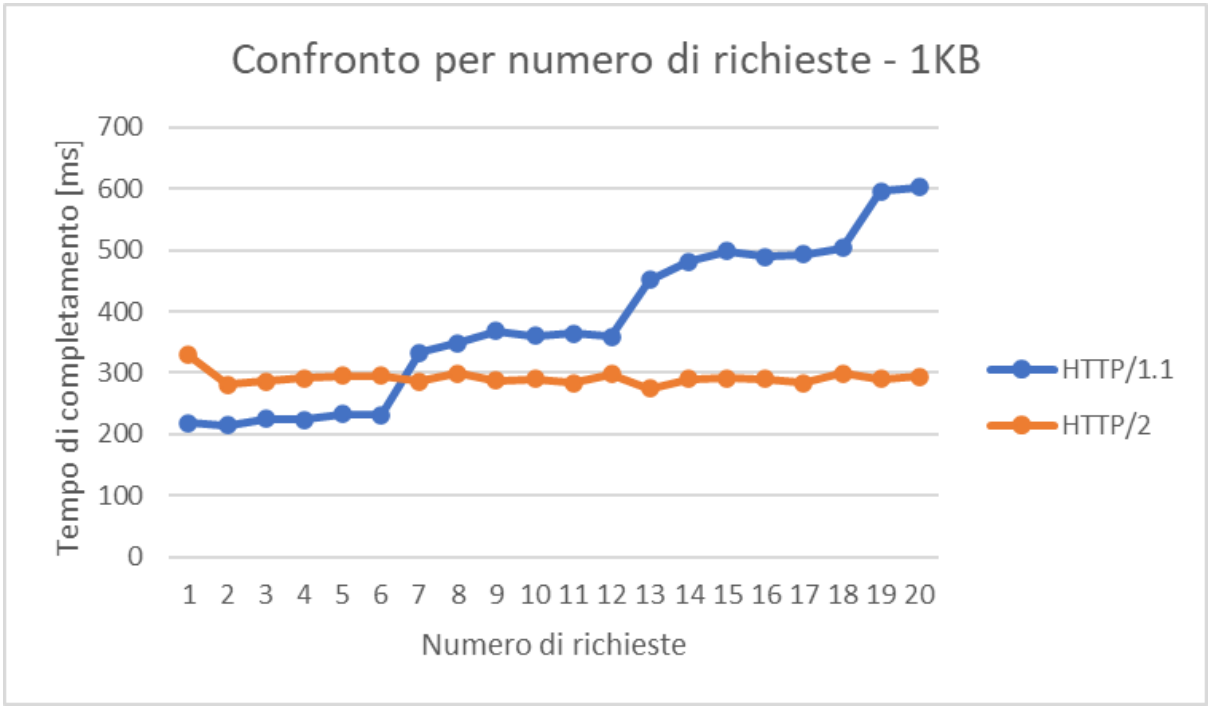
Il miglioramento delle prestazioni è ancora maggiore non ponendo un limite al numero degli stream per il protocollo HTTP/2, con una media dei tempi di completamento di circa il 23% inferiore rispetto ad HTTP/1.1 con 6 client.

4.4. Confronto per numero di richieste

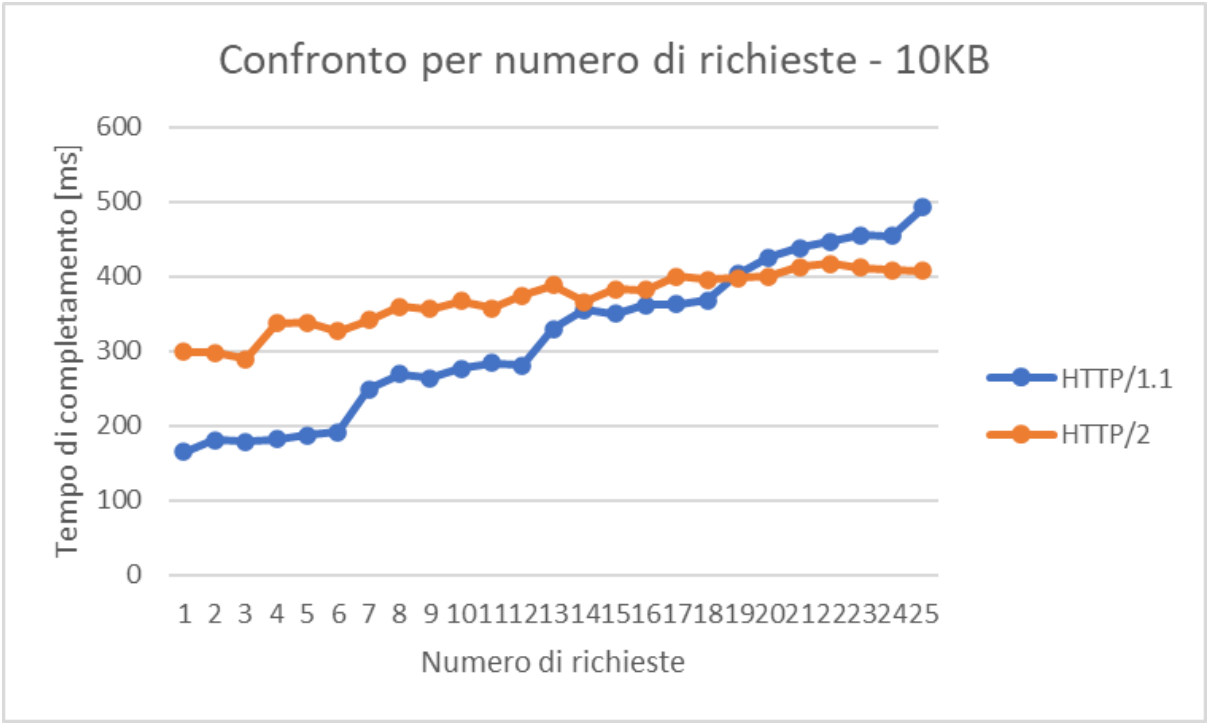
Nella sezione precedente si è osservato un miglioramento delle prestazioni offerte da HTTP/2 rispetto ad HTTP/1.1, mentre valutando il trasferimento di un singolo file la situazione è invertita, avendo tempi di download inferiori con il protocollo più vecchio.

In questa analisi si cerca di misurare la soglia oltre la quale il protocollo HTTP/2 ha prestazioni migliori rispetto al suo predecessore.

Per risorse di 1, 10 e 100 KB si vanno a confrontare i tempi di completamento di un numero crescente di richieste sia per un singolo client HTTP/2 senza limitare il numero di stream che per 6 client HTTP/1.1, situazione tipica in un contesto di web browsing. [9]

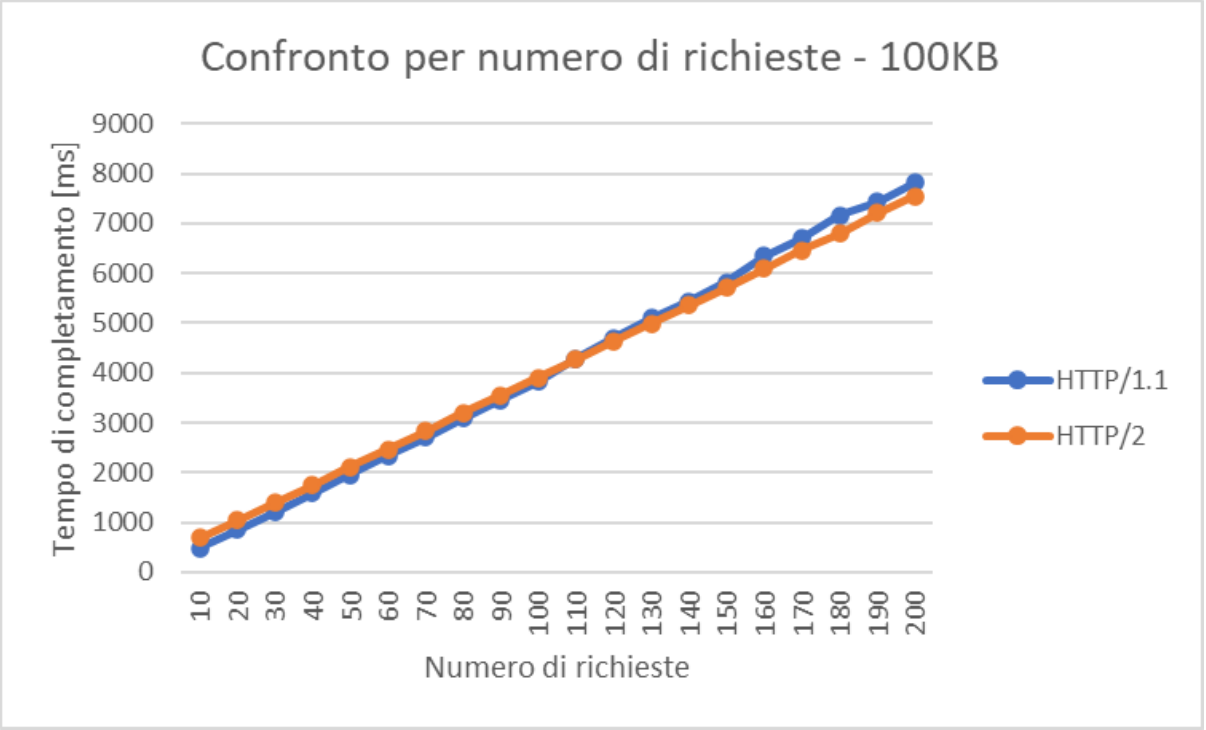


Per il trasferimento di risorse di 1KB si ha un miglioramento delle prestazioni da 7 richieste in poi. Il tempo di servizio di questo tipo di richieste è infatti costante per quanto riguarda il protocollo HTTP/2 (almeno considerando un numero di richieste relativamente basso come nell'intervallo di questa misurazione), mentre dato che il numero di richieste HTTP/1.1 in elaborazione è limitato a 6 ogni volta che si supera un multiplo di 6 come numero di richieste si ha uno "scalino" nel tempo di completamento finale dovuto a un ulteriore sequenza di richiesta/risposta sequenziale rispetto a tutti i gruppi di 6 richieste simultanee precedenti.



Per risorse di 10 KB la soglia di miglioramento delle prestazioni di HTTP/2 rispetto al predecessore è a circa 20 richieste. Nonostante gli "scalini" nei tempi di completamento dopo ogni multiplo di 6

siano ancora osservabili (anche se meno nitidamente rispetto al caso precedente), il punto in cui il nuovo protocollo inizia ad avere performance migliori viene spostato in avanti perchè l'effetto dell'aumento del numero di richieste non è più trascurabile sui tempi di completamento di HTTP/2.

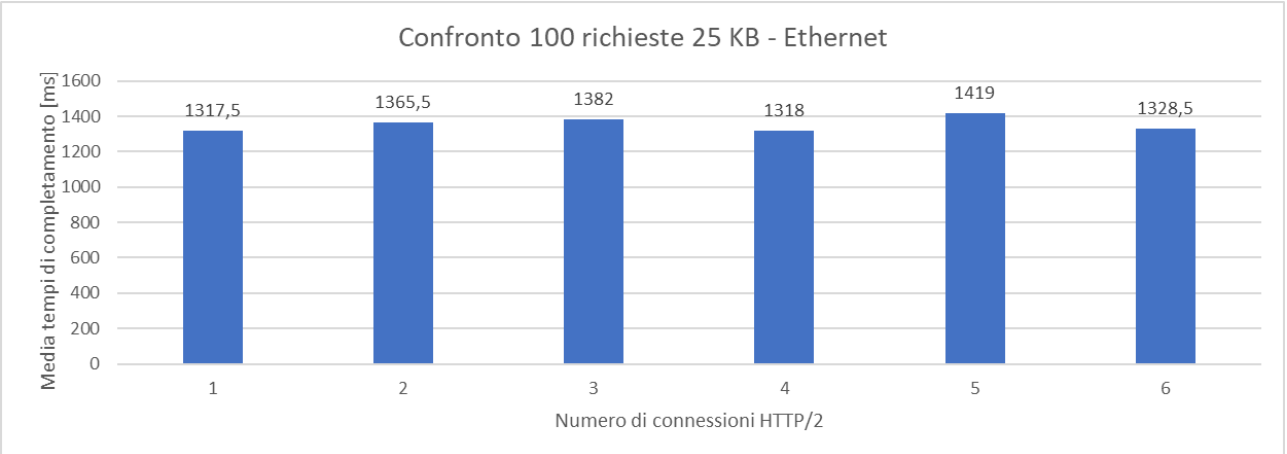


Aumentando la dimensione delle risorse trasferite a 100 KB ciascuna, si misura un miglioramento nelle prestazioni di HTTP/2 solo oltre le 110 richieste, rimanendo comunque con tempi sostanzialmente identici tra i due protocolli fino circa a quando si raggiungono le 170 richieste.

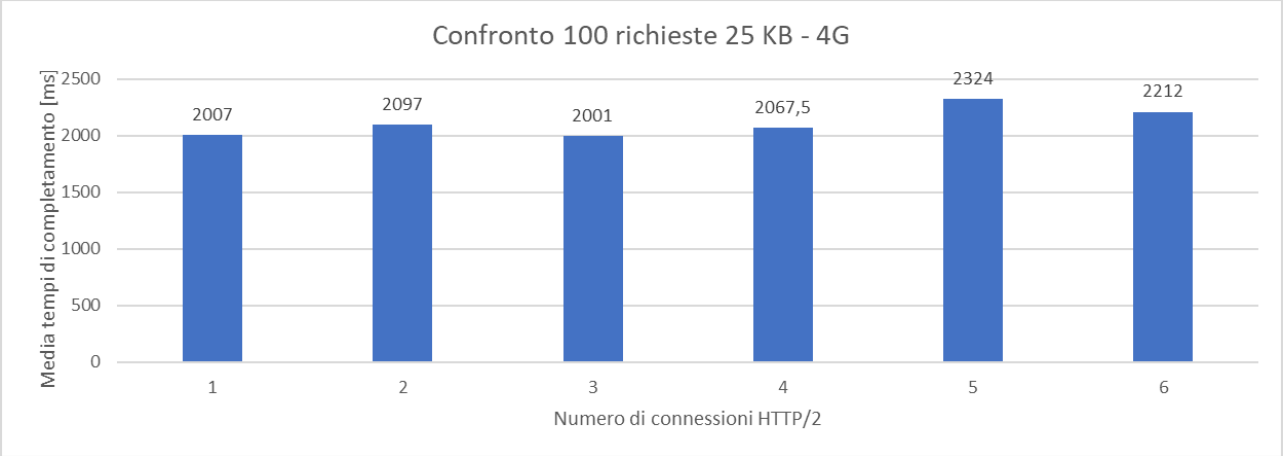
4.5. Utilizzo di più connessioni

Uno degli obiettivi di sviluppo di HTTP/2 è quello di avere un protocollo in grado di utilizzare una sola connessione TCP per gestire più richieste allo stesso tempo, senza necessitare della creazione di molte connessioni per raggiungere buone performance.

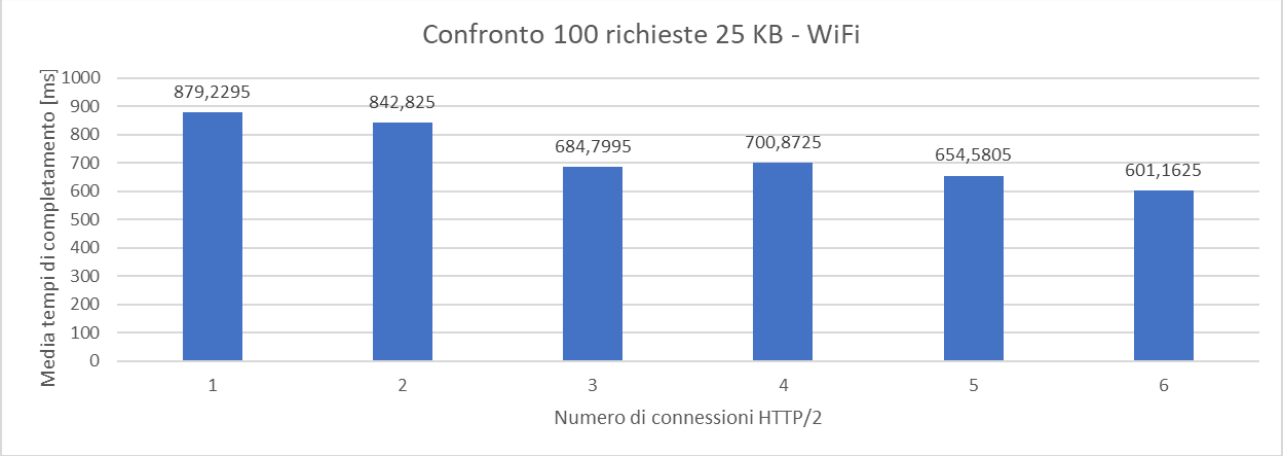
Si misurano i tempi di completamento di 100 richieste per una risorsa da 25 KB, facendo variare il numero di connessioni da 1 a 6.



In una connessione cablata domestica non si osserva nessun miglioramento delle prestazioni dovuto all'aumento del numero di connessioni: le differenze tra i tempi osservati possono essere attribuite a della variabilità normale nei tempi di trasferimento e non mostrano un andamento discendente all'aumentare delle connessioni.



Anche con una connessione 4G in un'area non affollata si osservano risultati simili: i tempi di completamento rimangono sostanzialmente costanti al variare del numero di connessioni.



Per una connessione WiFi in un edificio affollato, invece, si osserva un deciso miglioramento nei tempi di completamento all'aumentare del numero di connessioni.

Questo miglioramento delle prestazioni nel caso di connessioni in ambiente affollato rispetto ad una connessione domestica cablata può essere spiegato grazie al fenomeno dell'Head of Line blocking a livello di protocollo TCP: la perdita di un pacchetto blocca la trasmissione su tutti gli stream in una connessione, rendendola di fatto inutilizzabile finché il pacchetto non viene ritrasmesso.

L'utilizzo di più connessioni HTTP/2 può quindi essere utile in questi contesti di reti instabili o molto trafficate [37], portando però alle stesse considerazioni che potevano essere fatte per l'utilizzo di più connessioni da parte di HTTP/1.1, tra cui una riduzione dello sfruttamento dei meccanismi di flow-control di TCP e la penalizzazione di altre applicazioni che utilizzano solo una connessione (sez. 2.2 "Multiplexing").

Il problema dell'Head of Line blocking a livello del protocollo di trasporto è stato affrontato nel protocollo HTTP/3, come descritto alla sezione 2.12.

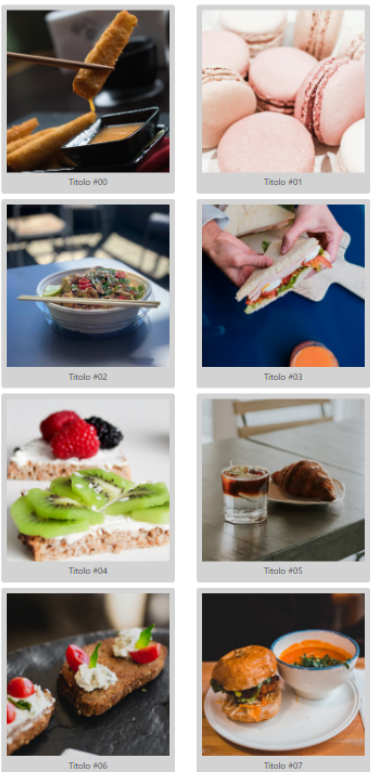
4.6. Tempi di caricamento pagine

In questa sezione vengono confrontate le prestazioni di HTTP/1.1 e HTTP/2 nel caricamento di diverse pagine web.

Le pagine usate per queste misurazioni sono le seguenti:

File singolo (10 B)
File singolo (100 B)
File singolo (1 KB)
File singolo (10 KB)
File singolo (25 KB)
File singolo (100 KB)
File singolo (1 MB)
File singolo (10 MB)
1 immagine
5 immagini
10 immagini
50 immagini
100 immagini
1 immagine (fetch)
5 immagini (fetch)
10 immagini (fetch)
50 immagini (fetch)
100 immagini (fetch)

Una homepage contenente un'immagine hero e una lista statica di collegamenti



Una lista di card, ciascuna con un titolo ed un'immagine diversa

Sono stati scelti questi due formati di pagine per simulare una semplice homepage di un sito web (file index.html, sulla sinistra) e un listino elettronico con una lista di prodotti (sulla destra).

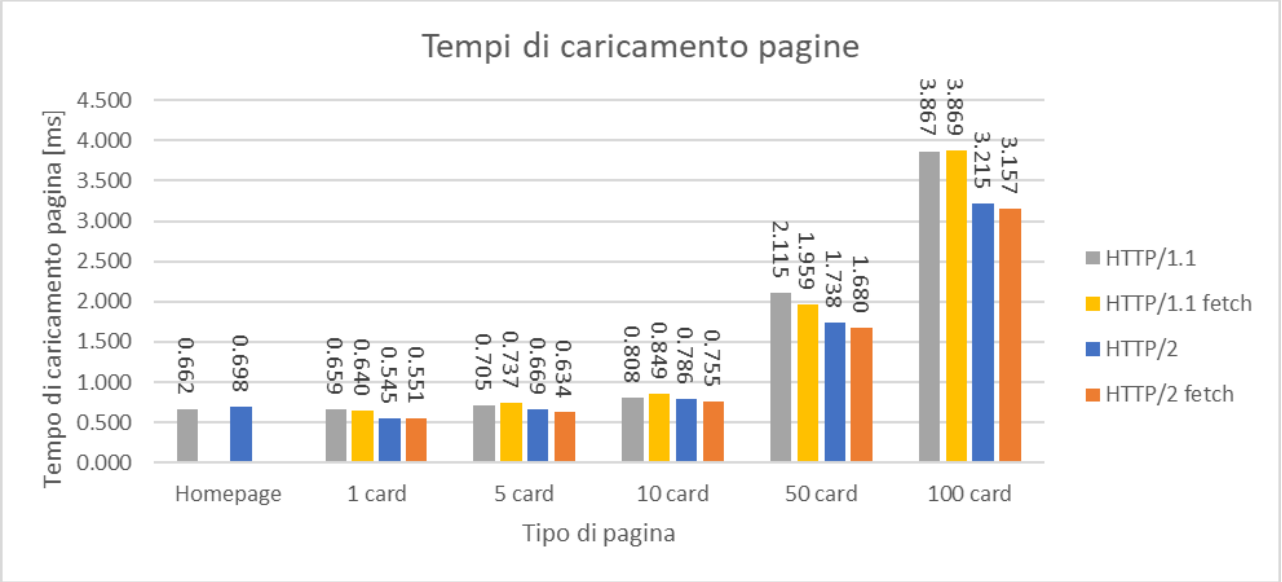
Sono state create pagine con liste di card di lunghezza diversa, per poter studiare i vantaggi dell'utilizzo di HTTP/2 al variare del numero di immagini in una pagina.

Per ogni pagina composta da una lista di immagini, esiste anche una sua variazione in cui le immagini da caricare non sono specificate all'interno del file html ma sono contenute in una lista caricata in modo asincrono (con la funzione fetch di Javascript). Con questa variazione si cerca di misurare se esiste una differenza tra i due protocolli per l'aumento dei tempi di caricamento portata dall'aggiunta di un round trip aggiuntivo per questa richiesta fatta con fetch precedente al caricamento delle immagini.

In tutte le pagine vengono caricati anche un piccolo file css per gli stili specifici della pagina, un'icona da mostrare nella barra delle schede vicino al titolo e l'intero file css della libreria

Bootstrap (utilizzata comunemente per la gestione degli stili in una pagina, facilitando ad esempio il dimensionamento delle componenti per diverse dimensioni dello schermo).

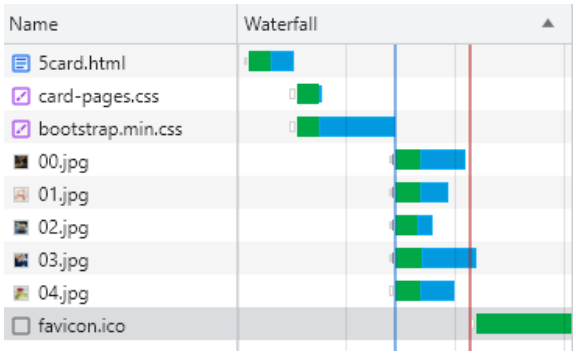
Per tutte le misurazioni in questa sezione viene utilizzato Google Chrome, registrando il tempo di fine del caricamento ("Finish") della pagina indicato nella scheda "Network" degli strumenti sviluppatore. [38]

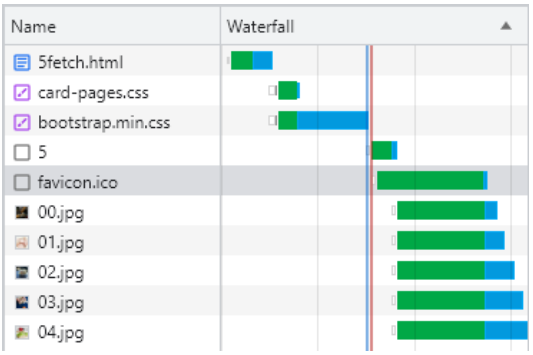


L'unico caso in cui non si osserva un netto miglioramento con l'utilizzo di HTTP/2 è nell'homepage, in cui si misura una differenza di circa il 5% tra i tempi di caricamento.

Per tutte le pagine con le card delle immagini, invece, il miglioramento delle prestazioni è significativo, mantenendosi a un valore di circa il 17%. Sebbene questo si traduca in un guadagno di pochi millisecondi per le pagine di dimensioni inferiori, all'aumentare del numero di immagini si arriva ad una differenza di più di mezzo secondo su un totale di circa 4.

Non si misura una grande differenza tra le versioni delle pagine caricate con e senza fetch. In alcuni casi si può osservare un tempo leggermente inferiore per la versione con fetch, riconducibile al modo in cui Google Chrome gestisce il caricamento dell'icona della pagina: il download di questa risorsa, che viene considerato all'interno del tempo "Finish" utilizzato per queste misurazioni, viene iniziato solo quanto tutto il resto degli elementi del DOM sono stati scaricati, quindi in tutte le pagine costruite senza fetch si ha una richiesta successiva al completamento di tutte quelle delle immagini delle card per scaricare l'icona. Con una pagina costruita con fetch, invece, l'evento di completato caricamento del DOM avviene prima della richiesta asincrona per ottenere la lista delle immagini, quindi in parallelo alla richiesta fatta con fetch viene inviata questa richiesta per l'icona, evitando di doverla fare in coda allo scaricamento di tutte le immagini. Di seguito un esempio di distribuzione dei tempi di download con HTTP/2 con e senza fetch.





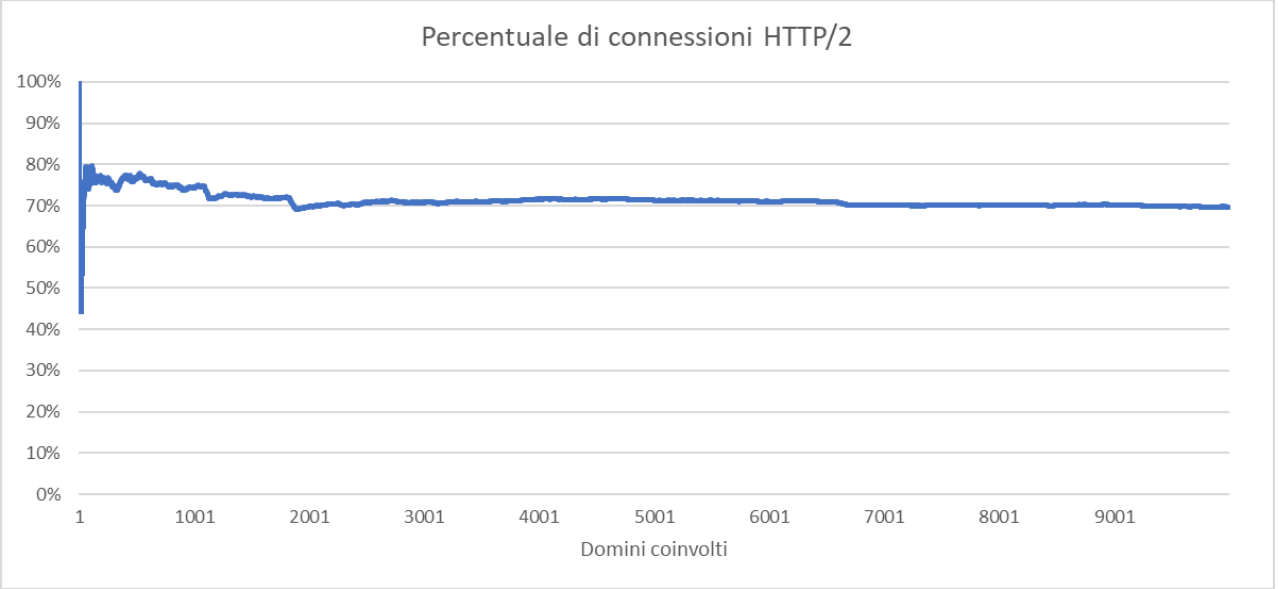
4.7. Diffusione del protocollo HTTP/2

Per valutare la diffusione del protocollo HTTP/2, sono stati condotti test di connessione verso una selezione di domini tra i più frequentati, registrando il protocollo applicativo selezionato per ciascuna connessione.

Nel contesto di questa misurazione, è stato utilizzato un programma derivato da un client web HTTP/2, da noi sviluppato con alcune restrizioni. Sono state rimosse tutte le funzionalità relative all'invio delle richieste e alla ricezione delle risposte, mantenendo solo la procedura di stabilimento della connessione verso un dominio specifico. Per ogni dominio testato, lo stato risultante è stato registrato in un file, e questo stato poteva rappresentare la selezione tra due protocolli applicativi possibili (HTTP/1.1 e HTTP/2), oppure un codice di errore (ad esempio, impossibilità di risolvere il nome del dominio, fallimento della connessione TCP, errore nell'handshake TLS o altro errore generico).

La scelta dei domini da testare è stata basata sulla lista di popolarità di Cisco Umbrella, che tiene conto del numero di richieste DNS per ogni dominio verso i server DNS di Cisco Umbrella.[39][40] Questa lista ordina i domini in base alle richieste DNS, senza considerare il tipo di servizio associato a ciascun dominio. I domini utilizzati sono stati i primi 10.000 della lista "Top 1 million" (aggiornata al 11 settembre 2023), senza restrizioni legate ai Top Level Domain.

A causa dell'eterogeneità della lista, durante il processo di misurazione è stato necessario escludere manualmente alcuni domini per i quali l'handshake non aveva esito e la chiamata a SSL_connect rimaneva in sospeso. Inoltre, considerando la mole di dati da raccogliere, il numero massimo di tentativi per stabilire la connessione TCP è stato ridotto al fine di minimizzare i tempi di attesa in caso di connessioni non riuscite.



In sintesi, su un totale di 10.000 domini testati, solo 7.176 di essi hanno avuto successo nella negoziazione del protocollo applicativo tra HTTP/1.1 e HTTP/2. Tra tutte le connessioni riuscite, il protocollo HTTP/2 è stato selezionato nel 69,7% dei casi. Limitando l'analisi ai primi 1.000 domini, questa percentuale sale al 72,3%, mentre restringendola ai primi 500 domini si raggiunge il 76,7% di connessioni HTTP/2 sul totale.

Questi risultati sono coerenti con quanto riportato nel report "State of the web" di HTTP Archive, datato 1 agosto 2023, il quale indica che la percentuale di richieste effettuate utilizzando il protocollo HTTP/2 durante la raccolta dei dati è del 68,5% per gli user agent desktop e del 68,3% per gli user agent mobile. [10]

5. Conclusioni

Le analisi condotte in questa tesi hanno fatto emergere alcuni vantaggi ed alcune criticità del protocollo HTTP/2 rispetto al suo predecessore HTTP/1.1.

Come spiegato nella sezione 2.1, uno degli obiettivi che hanno guidato lo sviluppo del nuovo protocollo è stato il miglioramento del trasferimento di più risorse tra client e server, in modo più veloce, efficiente e rispettoso delle risorse di rete rispetto a quanto possibile con HTTP/1.1. È in questo contesto di più trasferimenti simultanei, infatti, che si sono osservati i maggiori miglioramenti delle prestazioni legati all'utilizzo di HTTP/2: con un numero sufficientemente elevato di richieste si hanno miglioramenti per tutte le dimensioni delle risorse da trasferire, anche se i benefici diminuiscono per trasferimenti di risorse grandi in quanto in quelle condizioni la maggior parte dei tempi di trasferimento è causata dalla velocità limitata della rete.

Confrontando i due protocolli in una situazione tipica per il contesto della navigazione web si possono fare alcune considerazioni: come discusso più avanti, le prestazioni di HTTP/2 per situazioni di trasferimenti sequenziali sono peggiori di quelle di HTTP/1.1, ma passando ad un livello di parallelismo più alto seppur limitato (6 connessioni per HTTP/1.1, altrettanti stream per HTTP/2) si hanno già miglioramenti di prestazioni, anche senza utilizzare appieno il parallelismo potenzialmente illimitato del nuovo protocollo. Non ponendo limiti al grado di parallelismo delle richieste da effettuare, inoltre, si osservano prestazioni ancora migliori. Questo confronto conferma che il parallelismo realizzato attraverso il multiplexing di più stream in un'unica connessione TCP è più efficace di quanto si può raggiungere mantenendo i flussi di dati divisi in diverse connessioni, in quanto si riescono a sfruttare appieno i meccanismi di controllo della congestione di TCP e la penalizzazione dello "slow start" ha un impatto minore.

Se con molte richieste da poter parallelizzare HTTP/2 è più efficiente, il protocollo HTTP/1.1 si dimostra migliore in caso si vogliano effettuare delle richieste singole al server: la complessità per la gestione di questo protocollo è molto inferiore e si possono evitare tutte le operazioni che con HTTP/2 vanno svolte all'inizio della connessione, come l'invio delle impostazioni e l'inizializzazione di tutte le componenti necessarie a mantenere lo stato della connessione. In situazioni di questo tipo, l'effetto della complessità aggiuntiva introdotta con il nuovo protocollo non può essere controbilanciato dalla compressione degli header, e altre funzionalità come il framing delle richieste e il multiplexing di più stream in una connessione non portano a nessun beneficio.

Grazie ad un'analisi in diverse condizioni di rete, inoltre, emergono le limitazioni legate all'utilizzo del protocollo TCP: in condizioni di rete instabile o trafficata si può osservare il fenomeno dell'Head of Line Blocking, con il quale un ritardo nella ricezione di un pacchetto per uno stream porta ad un ritardo nella consegna dei frame in tutti gli stream. In queste situazioni di rete, quindi, la presenza di più connessioni caratteristica dell'utilizzo di HTTP/1.1 può portare un beneficio concreto. Questo problema ha guidato il design del nuovo protocollo HTTP/3, che mantiene gran parte delle caratteristiche di HTTP/2 ma utilizza l'UDP a livello di trasporto, permettendo di trasmettere e ricevere i frame di stream diversi in modo indipendente tra di loro.

Complessivamente, il protocollo HTTP/2 può migliorare concretamente l'esperienza di navigazione web e il trasferimento contemporaneo di più risorse rispetto ad HTTP/1.1, soprattutto con una connessione di rete stabile. Alla gestione di questo protocollo è però associata una complessità maggiore, che può non essere giustificata se è necessario effettuare solo poche richieste.

Bibliografia

1. https://www.rfc-editor.org/rfc/rfc9113
2. https://www.rfc-editor.org/rfc/rfc7541
3. https://www.rfc-editor.org/rfc/rfc9218
4. https://www.rfc-editor.org/rfc/rfc7540
5. https://www.rfc-editor.org/rfc/rfc7301
6. https://www.rfc-editor.org/rfc/rfc8555
7. https://http2-explained.haxx.se/en/part3
8. https://http2-explained.haxx.se/en/part6
9. https://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/
10. https://httparchive.org/reports/state-of-the-web
11. https://http2.github.io/faq/
12. https://en.wikipedia.org/wiki/Head-of-line_blocking
13. https://stackoverflow.com/a/45583977
14. https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x
15. https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/
16. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
17. https://github.com/brillout/website-dependency-tree
18. https://arxiv.org/pdf/2207.05885.pdf
19. https://blog.yoav.ws/posts/being_pushy/
20. https://groups.google.com/a/chromium.org/g/blink-dev/c/K3rYLvmQUBY/
21. https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cache-digest-05
22. https://developer.chrome.com/blog/removing-push/
23. http://nginx.org/en/CHANGES
24. https://www.extrahop.com/company/blog/2016/silly-window-syndrome/
25. https://medium.com/geekculture/exploring-application-layer-protocol-negotiation-alpn-c47b5ec3b419

26. https://blog.cloudflare.com/http-3-from-root-to-tip/

27. https://www.cloudflare.com/it-it/learning/performance/what-is-http3/

28. https://letsencrypt.org/it/docs/challenge-types/

29. https://www.splitbrain.org/blog/2017-08/10-homeassistant_duckdns_letsencrypt

30. https://github.com/dehydrated-io/dehydrated

31. https://wiki.openssl.org/index.php/Simple_TLS_Server

32. https://www.openssl.org/docs/manmaster/man7/ssl.html

33. https://www.openssl.org/docs/man3.0/man3/SSL_CTX_new.html

34. https://www.openssl.org/docs/man1.0.2/man3/SSL_CTX_set_verify.html

35. https://github.com/openssl/openssl/issues/7147#issuecomment-419621673

36. https://asktom.oracle.com/pls/apex/asktom.search?tag=oracle-block-size

37. https://dl.acm.org/doi/pdf/10.1145/3178876.3186181

38. https://stackoverflow.com/questions/30266960/website-response-time-difference-between-load-and-finish

39. https://docs.umbrella.com/deployment-umbrella/docs/point-your-dns-to-cisco

40. http://s3-us-west-1.amazonaws.com/umbrella-static/index.html