UNIVERSITÀ DEGLI STUDI DI PERUGIA

Dipartimento di Matematica e Informatica

CORSO DI LAUREA TRIENNALE IN INFORMATICA

Tesi di Laurea

HTTP/3: Il Protocollo del Prossimo Decennio

Laureando

Matteo Digiorgio

Relatore

Prof. Sergio Tasso

Anno Accademico 2019 - 2020

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

In questa tesi di laurea viene trattato principalmente il protocollo HTTP, dalla sua nascita fino al protocollo che ci troveremo ad usare in futuro (in alcuni casi lo utilizziamo già oggi!).

Nel **Capitolo 1** vedremo da dove è nato HTTP, in che modo funzionano le sue vecchie versioni e cosa volevano migliorare di versione in versione partendo dalla semplice e basilare pubblicazione del 1991 fino alla standardizzazione. Inoltre si può vedere un incipit di sintassi HTTP che comunque fa parte della sua base su cui ancora oggi ci si appoggia.

Nel **Capitolo 2** si parla di HTTP/2, il protocollo che ha "rivoluzionato" HTTP. Basandosi su un protocollo creato da Google, HTTP/2 è la sua standardizzazione con alcune differenze. La sua pubblicazione è avvenuta circa 15 anni dopo la standardizzazione di HTTP, portando dei miglioramenti che erano irraggiungibili con HTTP/1.1. Discutere di HTTP/2 era necessario dato che il suo successore ci si basa quasi completamente.

Nel **Capitolo 3** si parla di HTTP/3, il protocollo che fa un passo avanti ad HTTP/2. Ancora in fase di bozza, HTTP/3 trova la sua forza e innovazione sopra QUIC, un nuovo protocollo, anch'esso in fase di bozza, che va a posizionarsi nella pila ISO/OSI al posto di TLS. Ma novità che differenzia HTTP/3 dai suoi predecessori è che lavora sopra UDP, il che va a cambiare la "classica" pila HTTP + TLS + TCP che è stata utilizzata per molti anni, in un nuovo approccio del tipo HTTP + QUIC + UDP. In questo capitolo si potrà vedere cosa questi cambiamenti comportano.

Capitolo 1
La nascita e l'evoluzione del protocollo HTTP

1.1 La storia e la sintassi di HTTP

Nel 1898, mentre lavorava al CERN, Tim Berners-Lee scrisse una proposta per costruire un sistema ipertestuale sopra l'Internet. Inizialmente si chiamava Mesh (maglia), e successivamente fu rinominata in World Wide Web durante la sua implementazione nel 1990. Era costruito sopra gli esistenti protocolli TCP e IP, consisteva in quattro elementi costrittivi:

- Un formato testuale per rappresentare i documenti ipertestuali, l'HyperText Markup Language (HTML)
- Un protocollo semplice per lo scambio di questi documenti, l'HyperText Transfer Protocol (HTTP)
- Un client che mostri i documenti, il primo Web browser chiamato WorldWideWeb
- Un server che permette l'accesso ai documenti, una versione prematura di httpd

Questi quattro blocchi furono completati per la fine del 1990, e il primo server funzionante si trovava fuori il CERN all'inizio del 1991. Il post pubblicato a metà dello stesso anno da Tim Berners-Lee[1] ad oggi è considerato l'inizio ufficiale del World Wide Web come progetto pubblico [1].

COS'È E DOVE SI POSIZIONA IL PROTOCOLLO HTTP

HTTP (**Hypertext Transfer Protocol**), come suggerisce il nome, in origine il era stato concepito per trasferire documenti ipertestuali (documenti che contengono

[1] https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt

collegamenti verso altri documenti). Abbastanza in fretta gli sviluppatori si sono resi conto che il protocollo aveva un enorme potenziale e avrebbe potuto trasportare altri tipi di file (tipo un'immagine), quindi la parte dell'acronimo Hypertext non è più adeguato ma per quanto si è diffuso HTTP è troppo tardi per rinominarlo.

HTTP dipende da una connessione di rete affidabile, di solito fornita da TCP/IP, che a sua volta è costruita su una connessione fisica (Ethernet, Wi-FI, e così via). Dato che i protocolli di comunicazione sono suddivisi in livelli, ogni livello può concentrarsi su quello che sa fare meglio. HTTP non si concentra sui compiti dei livelli sottostanti, ad esempio su come la connessione di rete viene effettuata. Infatti le applicazioni HTTP devono prestare attenzione a come gestire gli errori di rete o le disconnessioni, il protocollo stesso non tiene conto di questi compiti.

La Figura 1.1 mostra il modello **Open System Interconnection** (OSI), un modello concettuale spesso usato per descrivere questo approccio a livelli e HTTP si posiziona al settimo ed ultimo livello della pila [2].

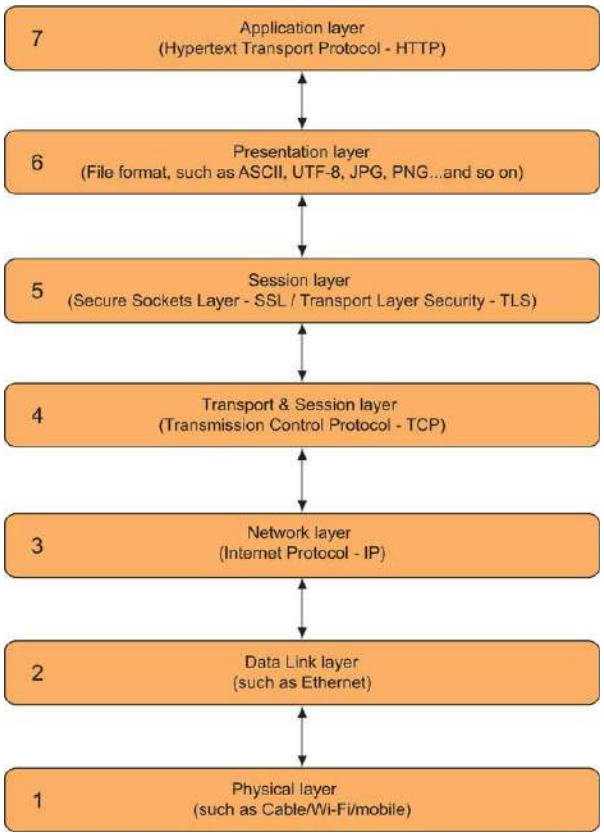

Figura 1.1: Struttura del modello OSI

1.1.1 HTTP/0.9

In origine HTTP era stato concepito per essere un protocollo molto semplice. Infatti una richiesta è composta da una singola linea di codice che inizia con l'unico metodo esistente **`GET`** seguito dal percorso della risorsa.

```
GET /mypage.html
```

Anche la risposta che viene generata è molto semplice.

```
<HTML>
A very simple HTML page
</HTML>
```

Come si può vedere la risposta è un singolo documento ipertestuale (nessun header o alcun metadata, solo HTML). Dopo ogni richiesta la connessione tra server e client viene chiusa [1].

1.1.2 HTTP/1.0

Com'è possibile intuire HTTP/0.9 è limitato sotto molti punti di vista. Tant'è che dopo la sua uscita in molti hanno iniziato a lavorare su possibili estensioni per renderlo versatile. La versione finale di HTTP/1.0 ha aggiunto alcune caratteristiche chiave tra cui:

- L'informazione della versione del protocollo viene inviata in ogni richiesta (HTTP/1.0 viene aggiunta nella stessa linea del metodo `GET`)
- Nella prima linea della risposta viene visualizzato un codice di stato a tre cifre che permette anche al browser di capire il risultato della richiesta e di adattare il suo comportamento di conseguenza
- L'aggiunta di un header al protocollo (opzionale), sia da parte della richiesta che della risposta, permette la trasmissione di metadata
- Con l'aiuto del sopracitato header è possibile trasmettere altri documenti che non siano semplici file HTML (grazie al campo **Content-Type** nell'header)
- Due metodi sono stati aggiunti al già conosciuto `GET`: **`HEAD`** e **`POST`**

```
GET /mypage.html HTTP/1.0
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)

200 OK
Date: Tue, 15 Nov 1994 08:12:31 GMT
Server: CERN/3.0 libwww/2.17
Content-Type: text/html
<HTML>
A page with an image
  <IMG SRC="/myimage.gif">
</HTML>
```

Successivamente per ricevere l'immagine viene effettuata una seconda richiesta.

```
GET /myimage.gif HTTP/1.0
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)

200 OK
Date: Tue, 15 Nov 1994 08:12:32 GMT
Server: CERN/3.0 libwww/2.17
Content-Type: text/gif
(image content)
```

Anche con HTTP/1.0 dopo ogni richiesta la connessione tra server e client viene chiusa.

Tutte queste novità sono state introdotte con un approccio del tipo try-and-see (prova e vedi che succede) durante gli anni tra il 1991 e il 1995. Nel Novembre del 1996 fu pubblicato un documento[2] con le caratteristiche discusse in questa sezione che fu chiamato HTTP/1.0 ma che comunque non era ancora definibile una standardizzazione [1] [2].

1.1.3 HTTP/1.1

In parallelo allo sviluppo e ai miglioramenti di HTTP che poi sono stati pubblicati come HTTP/1.0, dal 1995 si stava già lavorando ad una standardizzazione effettiva di HTTP. La prima standardizzazione è stata HTTP/1.1, pubblicata all'inizio del 1997[3], che ha chiarito le ambiguità e introdotto numerosi miglioramenti:

[2] https://tools.ietf.org/html/rfc1945
[3] https://tools.ietf.org/html/rfc2068

- Una connessione può essere riutilizzata, risparmiando tempo per riaprirla più volte per visualizzare le risorse incorporate
- È stato aggiunto il pipelining, che consente di inviare una seconda richiesta prima che la risposta per la prima sia completamente trasmessa, abbassando la latenza della comunicazione
- Sono supportate anche le risposte a pezzi (chunked)
- Sono stati introdotti tre ulteriori header utili per il controllo della cache: ETag, Cache-control e Vary
- La negoziazione del contenuto, inclusa la lingua, la codifica o il tipo, è stata introdotta e consente ad un client e un server di concordare il contenuto più adeguato da scambiare
- Grazie all'header Host, la possibilità di ospitare diversi domini allo stesso indirizzo IP ora consente la colocation del server

Un miglioramente significativo è stato di certo quello di abilitare una connessione persistente (che non si chiude ad ogni risposta). Miglioramento necessario dato che con il passare degli anni il web era diventato sempre più ricco di contenuti multimediali e la chiusura della connessione ad ogni risposta risultava dispendioso.

Il problema in HTTP/1.0 è stato risolto con un nuovo header HTTP **`Connection:`** e grazie al valore **`Keep-Alive`** permette di tenere aperta la connessione.

```
GET /page.html HTTP/1.0
Connection: Keep-Alive
```

Il server risponde come al solito, ma se supporta questa funzionalità, include un header **`Connection: Keep-Alive`** nella risposta.

```
HTTP/1.0 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Connection: Keep-Alive
Content-Type: text/html
Content-Length: 12345
```

```
Server: Apache

<!doctype html>
<html>
<head>
...etc.
```

Questa risposta dice al client che può inviare un'altra richiesta sulla stessa connessione.

HTTP/1.1 di default permette una connessione aperta e di fatto non bisogna specificare nell'header di tenerla aperta. Se il server vuole chiudere la connessione, per qualsiasi ragione, deve includere nella risposta un header HTTP **`Connection: close`** [1] [2].

```
HTTP/1.1 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Connection: close
Content-Type: text/html; charset=UTF-8
Server: Apache

<!doctype html>
<html>
<head>
...etc.
Connection closed by foreign host.
```

1.2 Introduzione ad HTTPS

I messaggi HTTP vengono mandati attraverso l'internet non crittografati e di conseguenza possono essere letti da chiunque mentre viene indirizzato verso la sua destinazione. L'internet, come suggerisce il nome, è una rete di computer, non un sistema point-to-point. L'internet non permette nessun controllo su come i messaggi vengono instradati, e tu, come un utente dell'internet, non hai idea come molti terzi vedranno i tuoi messaggi dato che vengono mandati dal tuo **internet service provider** (ISP) verso le compagnie di telecomunicazione e ad altri terzi. Dato che HTTP è un messaggio di testo normale, i messaggi possono essere intercettati, letti, e persino alterati in viaggio.

HTTPS è la versione sicura di HTTP che cifra i messaggi in transito con l'utilizzo del protocollo **Transport Layer Security** (TLS), anche se è spesso conosciuto con il nome della sua versione precedente, **Secure Sockets Layer** (SSL).

Per essere più specifico, SSL, inventato da Netscape, ha avuto tre versioni: SSLv1 (versione interna che non è mai stata rilasciata al pubblico), SSLv2 (prima versione rilasciata nel 1995) e SSLv3 (che risolveva alcune insicurezze di SSLv2 pubblicato l'anno successivo). Dato che SSL era inventato da Netscape, non era uno standard formale di Internet. Quindi SSL fu standardizzato in TLS, che era abbastanza simile a SSLv3 anche se incompatibile. Spesso accade che quando ci si trova davanti ad una standardizzazione, anche con il passaggio alla nuova tecnologia in molti continuino a riferirsi alla nuova con il vecchio acronimo. Per questo motivo tutt'oggi in molti si riferiscono a TLS chiamandolo ancora SSL.

HTTPS ha aggiunto tre importanti concetti ai messaggi HTTP:

- **Crittografia** — I messaggi non possono essere letti da terzi mentre sono in transito
- **Integrità** — I messaggi non possono essere alterati durante il transito, dato che l'intero messaggio crittografato viene firmato digitalmente e la firma viene verificata crittograficamente prima della decrittografia
- **Autenticazione** — Il server è quello con cui intendevi parlare

HTTPS funziona con l'utilizzo della crittografia a chiave pubblica, che permette ai server di fornire chiavi pubbliche, in un formato di certificati digitali, quando gli utenti si connettono per la prima volta. Come viene mostrato in Figura 1.2, il browser cripta i messaggi usando la chiave pubblica del server, che a sua volta solo lui potrà decifrare con la sua chiave privata. Questo sistema permette di comunicare in modo sicuro con un sito web senza dover conoscere una chiave privata condivisa, cruciale in un sistema come l'internet, dove nuovi siti web e utenti vanno e vengono ogni secondo.

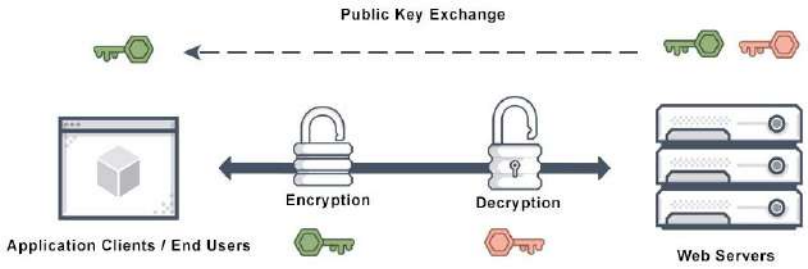

Figura 1.2: Cifratura asimmetrica

I certificati digitali vengono rilasciati e firmati digitalmente da varie **certificate authorities** (CAs) di fiducia del browser, per questo motivo è possibile autenticare che la chiave pubblica è per il server a cui ci si vuole connettere. Bisogna chiarire che una connessione HTTPS (verificabile nel browser dalla presenza del lucchetto vicino all'URL) garantisce che la connessione è criptata ma non garantisce che il server sia affidabile.

HTTPS è basato su HTTP ed è quasi senza soluzione di continuità con HTTP stesso. Di default è hostato su una porta diversa (porta 443 invece della porta 80 di HTTP), ed ha un diverso schema URL (https:// invece di http://), ma fondamentalmente non altera il modo in cui HTTP viene usato dal punto di vista della sintassi o formato del messaggio ad eccezione della cifratura e decifratura [2].

Secondo il Web Almanac[4], nel 2020 si è arrivati ad avere tra il 73% e il 77% di siti web che vengono serviti su HTTPS.

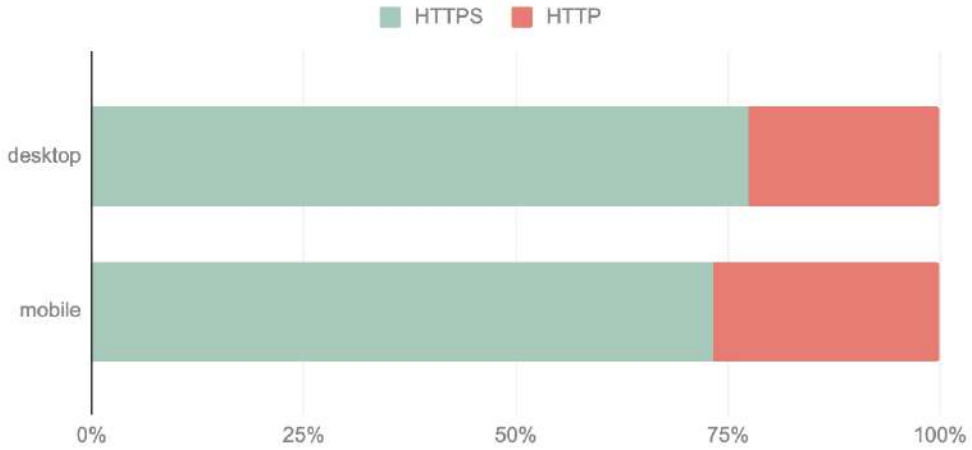

Figura 1.3: Utilizzo di HTTPS per i siti web

[4] https://almanac.httparchive.org/en/2020/security

Ma bisogna anche far notare che le richieste HTTPS che girano sono effettivamente di più in percentuale. Infatti sempre nel 2020 circa l'86% delle richieste erano richieste HTTPS. E come si può notare in Figura 1.4, in solo tre anni, dal 2017 al 2020 c'è stato un incremento di 50 punti percentuali (dal 36% all'86%).

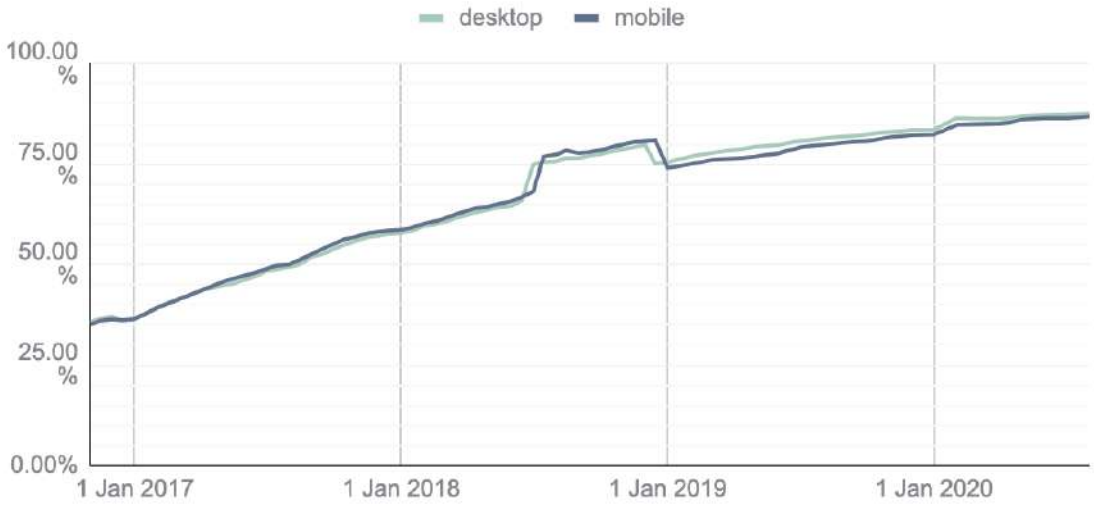

Figura 1.4: Percentuale di richieste che utilizzano HTTPS

FUNZIONAMENTO DI HTTPS

Quando il client si collega ad un HTTPS server, attraversa una fase di negoziazione (**TLS handshake**). In questa fase, il server fornisce la chiave pubblica, il client e il server si accordano sul metodo di cifratura da utilizzare, e poi i due negoziano una chiave di cifratura condivisa da usare durante la sessione. Al termine della sessione la chiave viene scartata. In questo modo se la chiave venisse scoperta, non si potrebbe comunque decifrare le sessioni passate.

Dopo che la sessione HTTPS è stata avviata, vengono scambiati messaggi HTTP standard. Il client e il server cifrano i messaggi prima di mandarli per poi decifrarli all'arrivo, ma alla vista di un web developer medio o server manager, non sono notabili delle differenze tra HTTPS e HTTP dopo la configurazione. Tutto avviene trasparentemente a meno che non si vanno a vedere i messaggi grezzi mandati attraverso la rete. HTTPS avvolge le richieste e le risposte HTTP di fatto non sostituendolo con un altro protocollo [2].

Capitolo 2
HTTP/2

Dato che molte caratteristiche di HTTP/2 si ritrovano in HTTP/3, prima di iniziare a parlare di quest'ultimo e andare a spiegare il perché sarà il protocollo del prossimo decennio bisogna capire in che situazione il web si trova oggi e capire il funzionamento del protocollo utilizzato da più del 50% dei siti web su internet.[5]

2.1 La strada verso la HTTP/2

HTTP non è cambiato radicalmente dal 1999, ovvero da quando HTTP/1.1 è diventato uno standard. In quel periodo si stava lavorando ad una versione aggiornata (HTTP-NG), che avrebbe cambiato completamente il metodo di funzionamento di HTTP, ma fu abbandonato nel 1999. Quel cambiamento radicale sarebbe stato troppo complesso e non ci sarebbe stato modo di introdurlo nel mondo reale [2].

2.1.1 SPDY

Nel 2009 Google ha annunciato che stava lavorando ad un nuovo protocollo chiamato SPDY (non è un acronimo e si pronuncia "speedy"). Stavano sperimentando su questo protocollo in condizioni di laboratorio e avevano visto dei risultati eccellenti, fino al 64% di miglioramento nel tempo di caricamento della pagina. L'esperimento era stato eseguito su copie delle 25 top pagine web.

SPDY è stato costruito sopra HTTP, ma fondamentalmente non cambiava il protocollo, più o meno come HTTPS che avvolgeva HTTP senza modificarlo. I metodi e i concetti di header di HTTP continuano ad esistere anche in SPDY. SPDY lavorava ad un livello inferiore ed agli sviluppatori web, proprietari di server, ed agli utenti, l'uso di SPDY era quasi trasparente. Ogni richiesta HTTP veniva semplicemente convertita in una richiesta SPDY, mandata al server, e convertita

[5] https://w3techs.com/technologies/details/ce-http2

nuovamente. Addizionalmente SPDY veniva implementato solo sopra HTTPS, che permetteva di nascondere la struttura e il formato del messaggio ai terzi che si trovavano di passaggio tra il client e il server. Tutte le reti esistenti, router, switch, e altre infrastrutture, potevano gestire i messaggi SPDY senza alcun cambiamento e senza nemmeno sapere che stavano gestendo dei messaggi SPDY invece di messaggi HTTP/1. Essenzialmente SPDY era retrocompatibile e poteva essere introdotto con modifiche e rischi minimi, che è indubbiamente il motivo principale del perché ha avuto successo a differenza di HTTP-NG.

Mentre HTTP-NG provava a sistemare molti problemi di HTTP/1, l'obiettivo principale di SPDY era quello di affrontare le limitazioni di performance di HTTP/1.1. Ha introdotto alcuni concetti importanti per affrontare questi limiti:

- **Multiplexing** — Richieste e risposte usavano una singola connessione TCP e venivano suddivise in pacchetti interlacciati raggruppati in flussi separati
- **Prioritizzazione delle richieste** — Per evitare l'aggiunta di un nuovo problema di performance mandando tutte le richieste allo stesso momento, veniva introdotto il concetto di prioritizzazione delle richieste
- **Compressione delle intestazioni HTTP** — Da molto tempo il corpo del messaggio HTTP veniva compresso, ma adesso anche l'header poteva essere compresso

Era impossibile introdurre queste funzionalità con il protocollo richiesta-e-risposta basato sul testo, come era stato per tutto quel tempo fino all'uscita di SPDY, un protocollo binario. Questo cambiamento ha permesso alla connessione singola di gestire piccoli messaggi, che insieme formavano messaggi HTTP più grandi, similmente a TCP che rompe i messaggi HTTP in pacchetti TCP più piccoli. SPDY ha implementato i concetti di TCP al livello HTTP in modo da permettere che molteplici messaggi HTTP possano viaggiare allo stesso momento.

È importante citare anche una importante funzionalità aggiuntiva come quella del **server push** che permetteva al server di richiamare risorse extra. Ad esempio se il client richiede la home page, server push può fornire il file CSS necessario, in

risposta alla richiesta. In questo modo si evita di dover richiedere successivamente il file CSS e quindi si risparmia nelle prestazioni del viaggio di andata e di ritorno.

SPDY ha riscosso un gran successo arrivando nel 2018 ad essere utilizzato dal 9.1% dei siti web. Con l'avvento di HTTP/2 che l'ha sostituito c'è stato un drastico calo fino a diventare inutilizzato.[6] HTTP/2 infatti, in fase di prima bozza (Novembre 2012), era basato fortemente su SPDY e poi nei successivi due anni è stato leggermente modificato e migliorato. Fino ad esser pubblicato come nuovo standard nel Maggio 2015[7] [2].

2.2 Caratteristiche di HTTP/2

HTTP/2 è stato creato principalmente per risolvere i problemi di performance in HTTP/1, e la nuova versione si differiva con l'aggiunta dei seguenti concetti:

- Protocollo binario invece che testuale
- Multiplexing invece di connessione sincrona
- Flow control
- Prioritizzazione dello stream
- Compressione dell'header
- Server push

La maggior parte di queste aggiunte non vanno in conflitto su come il protocollo viene "mandato sul cavo" tra client e server. Ad un alto livello, dove molti programmatori web avranno a che fare con la semantica di HTTP, HTTP/2 non è molto diverso dal suo predecessore. Infatti ha gli stessi metodi (`GET`, `POST`, `PUT` e cosí via), utilizza la stessa struttura URL, gli stessi codici di risposta (`200`, `404`, `301`, `302`), e quasi lo stesso header HTTP. In breve HTTP/2 è un modo più efficiente per fare le stesse richieste HTTP.

[6] https://w3techs.com/technologies/details/ce-spdy
[7] https://tools.ietf.org/html/rfc7540

Nonostante ciò, HTTP/2 dovrebbe portare a sviluppare siti web in modo diverso. Allo stesso modo in cui una forte comprensione di HTTP/1 ha portato a molte ottimizzazioni, una forte comprensione di HTTP/2 consente agli sviluppatori di ottimizzare e rendere i siti web più veloci [2].

2.2.1 Binario invece di testuale

Una delle differenze maggiori tra HTTP/1 e HTTP/2 è che quest'ultimo è binario, protocollo packet-based (basato sui pacchetti), a differenza di HTTP/1 che è interamente text-based (basato sul testo). I protocolli text-based sono più semplici per gli umani ma molto più difficili da analizzare per i computer. Questa situazione era accettabile quando HTTP era nato e i siti web erano molto semplici con poche richieste e risposte se non addirittura una singola richiesta con una singola risposta. Per il web moderno questo approccio si stava rendendo sempre più limitante col passare del tempo.

Con il protocollo text-based, le richieste devono essere inviate e le risposte ricevute per intero prima di poter elaborare un'altra richiesta. Nel complesso, HTTP ha funzionato in questo modo negli ultimi 20 anni, anche se sono stati apportati dei piccoli miglioramenti. HTTP/1.0 introdusse l'HTTP body binario, per esempio, dove immagini ed altri media potevano essere mandati in risposta, e poi HTTP/1.1 introdusse il pipelining e la codifica in blocchi. Un problema di queste due ultimi implementazioni era il blocco head-of-line (HOL) per cui il messaggio in cima alla coda impedisce l'invio di risposte successive che potrebbero essere pronte, per non parlare del fatto che il pipelining non era ben supportato nel mondo reale.

HTTP/2 passa ad un protocollo binario completo, in cui i messaggi HTTP vengono suddivisi e inviati in frame ben definiti (2.4 HTTP/2 frames). Tutti i messaggi HTTP/2 utilizzano la codifica in blocchi come standard e quindi non è necessario impostarla. Questi frame sono simili ai pacchetti TCP che sono alla base della maggior parte delle comunicazioni HTTP. Quando si ricevono tutti i frame, il messaggio HTTP può essere ricostruito. Nonostante per certi versi sia simile a TCP, HTTP/2 è ancora solidamente stratificato sopra TCP (sebbene Google abbia sperimentato la sostituzione di TCP con QUIC così avendo una implementazione più leggera di HTTP/2, 3.1 QUIC). L'utilizzo di TCP al disotto è fondamentale per

garantire l'arrivo dei messaggi, in ordine, senza dover gestire questo problema nel livello applicativo.

La rappresentazione binaria di HTTP/2 avviene per l'invio e la ricezione dei messaggi, ma i messaggi stessi sono simili ai vecchi messaggi HTTP/1. Il framing binario al di solito è gestito da client di livello inferiore o da librerie (browser web o server web). Come menzionato prima, gli applicativi di alto livello come JavaScript non hanno bisogno di sapere come i messaggi vengono mandati, e possono, per la maggior parte delle volte, trattare una connessione HTTP/2 come una connessione HTTP/1.1. Ovviamente può essere utile capire e addirittura vedere i frame HTTP/2 per fare debugging su errori inaspettati [2].

2.2.2 Multiplexing invece di connessione sincrona

HTTP/1 è un protocollo sincrono a singola richiesta e risposta. Il client manda un messaggio HTTP/1 e il server manda indietro una risposta HTTP/1. Ovviamente nel web moderno risulta incredibilmente inefficiente dato che un sito è spesso composto da centinaia di risorse. La soluzione alternativa più utilizzata in HTTP/1 era aprire connessioni multiple o mandare meno richieste ma più grandi invece di molte piccole richieste, ma entrambe le soluzioni portavano i loro problemi e inefficienze. La Figura 2.1 mostra come tre connessioni TCP possono essere usate per mandare e ricevere tre richieste HTTP/1 in parallelo. Da notare che la Richiesta 1 non viene mostrata perché solo dopo la richiesta iniziale sono necessarie risorse multiple, successivamente richieste in parallelo (Richieste 2 - 4).

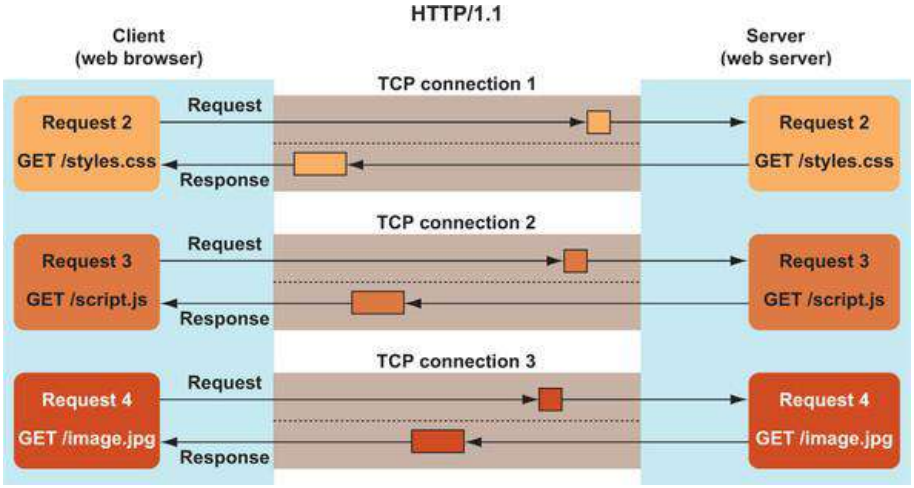

Figura 2.1: Richieste HTTP/1 multiple in parallelo con connessioni TCP multiple

HTTP/2 permette multiple richieste nello stesso momento su una singola connessione TCP, usando diversi stream per ogni richiesta o risposta HTTP. Questo concetto di multiple richieste indipendenti allo stesso momento è stato possibile passando al framing binario, dove ogni frame ha un indicatore di stream (stream ID). Il ricevente può ricostruire l'intero messaggio quando tutti i frame dello stesso stream sono stati ricevuti.

I frame sono la chiave che permette di mandare messaggi multipli allo stesso momento. Ogni frame è etichettato in modo da poter risalire al suo stream, e ciò permette a due, tre o cento messaggi di poter essere mandati o ricevuti allo stesso tempo sulla stessa connessione multiplex, al contrario delle sei connessioni HTTP/1 parallele consentite dalla maggior parte dei browser. La Figura 2.2 mostra le stesse tre richieste della Figura 2.1 ma le richieste vengono mandate una dopo l'altra sulla stessa connessione (simile alla pipeline di HTTP/1), e le risposte vengono rimandate indietro ma mescolate (impossibile utilizzando pipeline di HTTP/1).

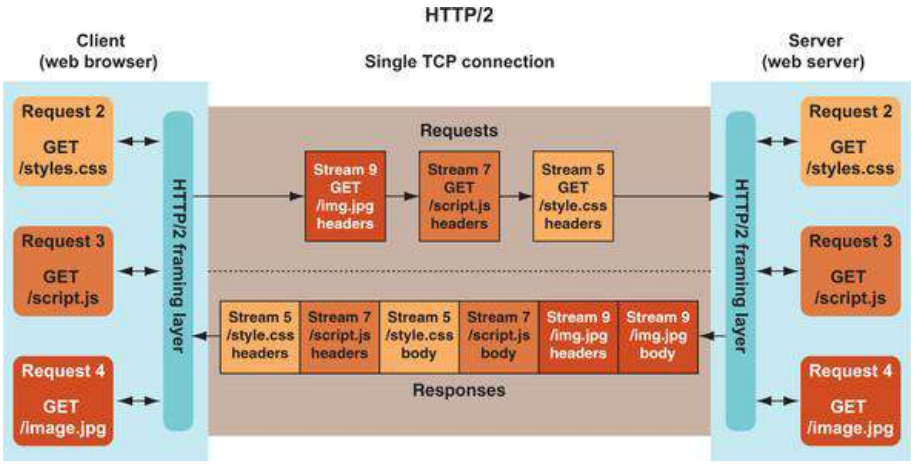

Figura 2.2: Tre risorse richieste su una connessione HTTP/2 multiplex

Capire la Figura 2.2 è la chiave per capire HTTP/2. Se si comprende il concetto e il perché è diverso da HTTP/1, si è sulla buona strada per comprendere HTTP/2 [2].

2.2.3 Prioritizzazione dello stream e flow control

Prima di HTTP/2, HTTP era un protocollo a singola richiesta/risposta, quindi non c'era bisogno di priorizzare. Il client (di solito il web browser) decideva la priorità fuori l'HTTP, decidendo con quale ordine mandare i messaggi, usando come limite

quello di HTTP/1 (sei connessioni tipicamente). Questa priorizzazione richiedeva prima risorse critiche (HTML, render blocking CSS, e JavaScript critico) e successivamente elementi non bloccanti (immagini e JavaScript asincrono). Le richieste venivano messe in coda, in attesa di una connessione HTTP/1 libera, e la coda gestita dal browser decideva la priorità.

Adesso HTTP/2 ha un limite di richieste molto più alto (di solito 100 stream attivi), quindi non è più necessario che il browser metta in coda le risorse che possono essere mandate immediatamente. In questo modo però si potrebbe arrivare a sprecare la larghezza di banda su risorse con priorità inferiore (come immagini) e quindi far sembrare che la pagina si carichi più lentamente su HTTP/2. La prioritizzazione dello stream è necessaria in modo che le risorse critiche possono essere inviate con una priorità più alta. Viene implementata dal server mandando più messaggi con alta priorità piuttosto che messaggi con bassa priorità quando una coda di frame è in attesa di essere inviata. Inoltre la prioritizzazione dello stream permette un maggior controllo rispetto ad HTTP/1, dove le connessioni separate sono indipendenti. In HTTP/1 tutte le risorse devono essere mandate con la stessa priorità sulle sei connessioni separate, oppure ad esempio le prime cinque possono essere inviate mentre la sesta viene trattenuta. Con HTTP/2 tutte e sei le richieste possono essere inviate con la priorità che le appartiene, e questa priorità viene utilizzata per decidere la quantità di risorse da allocare per l'invio di ciascuna risposta.

Il flow control (controllo del flusso) è un'altra necessaria conseguenza dovuta all'utilizzo di stream multipli sulla stessa connessione. Se il ricevente non riesce a processare i messaggi in arrivo alla stessa velocità del mittente che li sta mandando, si ottiene un backlog (un arretrato), che dovrà essere bufferizzato ed eventualmente si arriverà ad una perdita di pacchetti, di conseguenza ci sarà la necessità di un rinvio dei pacchetti persi. TCP permette la limitazione della connessione in uno scenario di questo tipo a livello di connessione, ma HTTP/2 lo fa avvenire al livello dello stream. Per esempio, in una pagina web con un video in diretta, se il video viene messo in pausa dall'utente può essere prudente mettere in pausa il download solo per quello stream HTTP/2 e non per altre risorse usate dal sito, che dovrebbero essere scaricate alla massima capacità sugli altri stream [2].

2.2.4 Compressione dell'Header

Gli Header HTTP vengono usati per mandare informazioni aggiuntive circa le richieste e le risposta tra client e server, e viceversa. C'è molta ripetizione negli header di solito, dato che spesso sono uguali per ogni risorsa. Di seguito alcuni header che vengono mandati ad ogni richiesta e spesso ripetono valori già mandati in precedenza:

- **Cookie** — Vengono mandati ad ogni richiesta al dominio. I cookie header possono diventare particolarmente grandi (fino a qualche KB con un limite imposto dai browser di 4KB per cookie) e spesso sono necessari solo per l'HTML
- **User-Agent** — Tipicamente questo header indica il tipo di browser e device che si sta utilizzando. Non cambia mai durante la sessione, ma nonostante ciò viene inviato con ogni richiesta
- **Host** — Utilizzato per qualificare completamente l'URL a cui si sta facendo la richiesta e rimane sempre lo stesso per ogni richiesta allo stesso host
- **Accept** — Definisce il formato della risposta che ci si aspetta (formati immagine accettabili che il browser sa come visualizzare e cosí via). Dato che i formati supportati da un browser in genere non cambiano senza un aggiornamento del browser, questo header ovviamente cambia in base al tipo di richiesta (immagine, documento, font e cosí via), ma rimane la stessa per ogni istanza di questi tipi
- **Accept-Encoding** — Definisce il formato di compressione. Tipicamente gzip (al momento della scrittura viene usato dal 95.5% dei siti che utilizzano compressione), deflate (0.7%), ed è in aumento anche l'utilizzo di Brotli per i browser che lo supportano (3.9%).[8] Come per l'header Accept, questo header non cambia durante la sessione

HTTP/1 consente la compressione dei body HTTP (sarebbe esattamente l'header Accept-Encoding), ma non consente la compressione degli header HTTP.

[8] https://w3techs.com/technologies/details/ce-compression

HTTP/2 introduce il concetto di compressione dell'header ma utilizza una tecnica diversa dalla compressione che già avveniva per il body. Questa scelta permette l'utilizzo della compressione di richieste incrociate e per prevenire alcuni problemi di sicurezza con algoritmi utilizzati per la compressione dei body HTTP [2].

2.2.5 Server push

Un altra importante differenza tra HTTP/1 e HTTP/2 è che quest'ultimo ha introdotto il concetto di server push, che permette al server di rispondere ad una richiesta con più di una risposta e quindi a tutti gli effetti rispondere mandando risorse che il client non ha richiesto esplicitamente.

Ad oggi il server push è stato dichiarato deprecato da Google. Dopo circa sei anni dalla pubblicazione di HTTP/2, server push viene ancora usato raramente. Negli scorsi tre anni 2018 - 2020, circa il 99.96% delle connessioni HTTP/2 create da Chrome non hanno utilizzato il server push. Questo numero indica la mancanza di sforzi da parte degli operatori dei server per aumentare l'utilizzo del server push, dovuto probabilmente alla complessità per una implementazione efficiente. Inoltre dai client vengono utilizzati meno del 40% dei push ricevuti, rispetto al 63,51% di due anni fa. Gl'altri push non sono validi, non vengono mai abbinati alle richieste o sono già presenti nella cache (il che rende il push inutile) [3].

2.3 Come viene stabilita una connessione HTTP/2

Dato che HTTP/2 è molto diverso da HTTP/1 al livello di connessione, il client web browser e il server hanno bisogno di poter parlare e capire HTTP/2 per poterlo usare. Dato che nella connessione sono comprese due parti indipendenti, è necessario un processo che permetta ad ogni parte di essere d'accordo e disposta ad utilizzare HTTP/2.

Il passaggio ad HTTPS era stato l'ultimo simile cambiamento, che era stato possibile grazie ad un nuovo schema URL ed inoltre era servito su una porta diversa. Questo cambiamento permise una chiara separazione dei protocolli e perciò una

chiara indicazione di quale protocollo si stava usando. Però ci sono diversi aspetti negativi nel passaggio ad un nuovo schema:

- Fino a che l'adozione rimane universale, l'URL di default deve rimanere http:// oppure https://. Aggiungere un nuovo schema come fece HTTPS, richiederebbe un reindirizzamento per utilizzare HTTP/2, che però comporterebbe lentezza (il principale obiettivo che HTTP/2 avrebbe dovuto risolvere)
- I siti dovrebbero cambiare i link per adattarsi al nuovo schema. Sebbene il problema dei link interni può essere risolto utilizzando i link relativi, i link esterni hanno bisogno di includere l'intero URL, incluso lo schema. Per questo motivo l'adozione di HTTPS è stata spesso complicata, in parte, dalla necessità per i siti di aggiornare tutti gli URL al nuovo schema
- Con le infrastrutture di rete esistenti ci sono problemi di incompatibilità (dato che i firewall bloccano tutto ciò che non arriva sulle porte standard)

Per questi motivi e per rendere la transizione ad HTTP/2 senza soluzione di continuità, HTTP/2 non usa un nuovo schema ma utilizza dei metodi alternativi per stabilire la connessione. La specifica di HTTP/2[9] fornisce tre modi per instaurare una connessione (più una quarta che è stata aggiunta successivamente):

- HTTPS negotiation
- Header HTTP Upgrade
- Prior knowledge

HTTP/2 è disponibile su HTTP non crittografato (conosciuto come h2c), e su HTTPS crittografato (h2). Praticamente tutti i siti web supportano HTTP/2 solo su HTTPS (h2), l'opzione 1 viene usata per la negoziazione di HTTP/2 [2].

[9] https://tools.ietf.org/html/rfc7540#section-3

2.3.1 HTTPS Negotiation

Le connessioni HTTPS passano attraverso una fase di negoziazione del protocollo per stabilire la connessione, dato che hanno bisogno di accordarsi sul protocollo SSL/TLS, cifratura, e altre varie impostazioni da usare prima che la connessione venga stabilita e prima che vengano scambiati messaggi HTTP. Questa fase è flessibile, cosí permettendo l'introduzione di nuovi protocolli HTTPS e cifrature per poi essere usati solo se client e server raggiungono un accordo [2].

HTTPS HANDSHAKE

Usare HTTPS significa usare SSL/TLS per cifrare una connessione HTTP standard, che sia HTTP/1 o HTTP/2. Durante il TLS handshake avviene lo scambio di una chiave condivisa attraverso una cifratura asimmetrica per poi passare ad una cifratura simmetrica possibile con l'utilizzo della chiave condivisa. Tutto ciò avviene all'inizio della connessione come si vede in Figura 2.3 per impostare la cifratura della connessione (TLSv1.2, il più utilizzato al momento). Questo handshake cambia leggermente con il nuovo standard TLSv1.3 (3.1.1 Connessione).

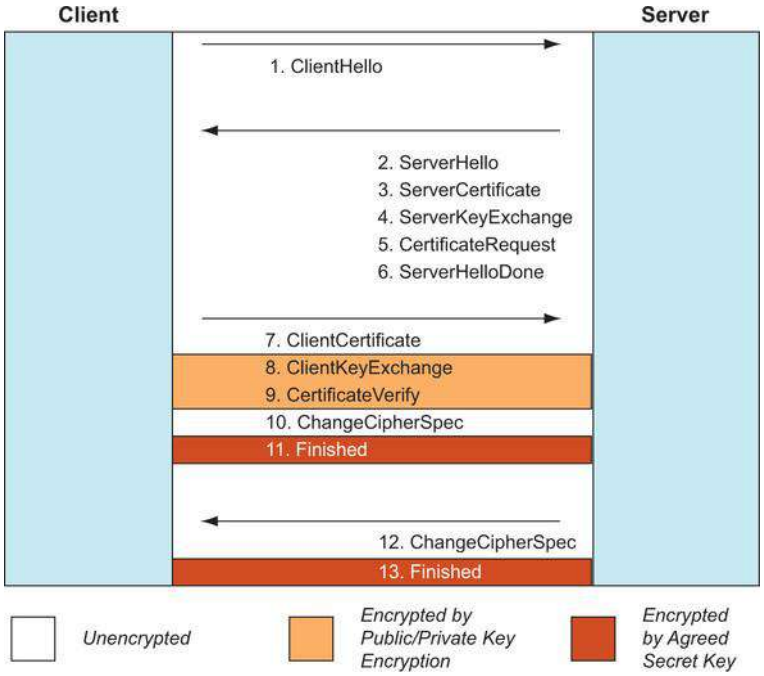

Figura 2.3: HTTPS handshake

L’handshake coinvolge quattro serie di messaggi:

- Il client invia un messaggio **ClientHello** specificando le sue capacità crittografiche. Questo messaggio viene mandato non cifrato perché il metodo di cifratura non è stato ancora accordato
- Il server invia indietro un messaggio simile, **ServerHello**, scegliendo il protocollo HTTPS (come TLSv1.2) basandosi su ciò che il client supporta. Inoltre manda il cifrario che utilizzerà per questa connessione (ad esempio ECDHE-RSA-AES128-GCM-SHA256), di nuovo basandosi sulle indicazione del messaggio ClientHello e quindi su cosa il client supporta. Successivamente fornisce al client il certificato del server HTTPS (**ServerCertificate**). I dettagli della chiave segreta dipendo dalla crittografia selezionata (**ServerKeyExchange**) e dall’eventuale necessità di un certificato HTTPS del client (**CertificateRequest**, non necessario per la maggior parte dei siti web). Ed alla fine il server dice che ha finito (**ServerHelloDone**)
- Il client verifica il certificato del server e invia il suo certificato se richiesto (**ClientCertificate**). In seguito invia i dettagli della sua chiave segreta (**ClientKeyExchange**). Questi dettagli vengono inviati crittografati con la chiave che si trova all’interno del certificato del server, in questo modo solo il server con la sua chiave privata potrà decifrarlo. Se il certificato del client viene richiesto, un messaggio **CertificateVerify** viene mandato, firmato con la chiave privata del client in modo da dimostrare la proprietà del certificato. Il client utilizza i dettagli di **ServerKeyExchange** e **ClientKeyExchange** per definire una chiave simmetrica e successivamente invia un messaggio **ChangeCipherSpec** per informare il server che la cifratura è iniziata. Infine invia un messaggio **Finished** cifrato
- Il server a sua volta cambia la sua connessione con una cifrata simmetricamente e invia un messaggio **Finished** cifrato

Dopo tutti questi passaggi, la sessione HTTPS è avviata, e tutte le comunicazioni future sono protette con la chiave concordata. Dopo che questa

connessione HTTPS è avvenuta con successo, i messaggi HTTP futuri non hanno bisogno di passare attraverso questa negoziazione. Inoltre future connessioni possono saltare alcuni di questi passaggi se si riutilizza la chiave che è stata usata l'ultima volta, attraverso il processo conosciuto come **session resumption** [2].

APPLICATION-LAYER PROTOCOL NEGOTIATION

ALPN[10] aggiunge una estensione sia al messaggio **ClientHello**, dove i client possono informare quali protocolli applicativi supporta, ed anche nel messaggio ServerHello, dove i server possono confermare quali protocolli applicativi usare dopo la negoziazione HTTPS.

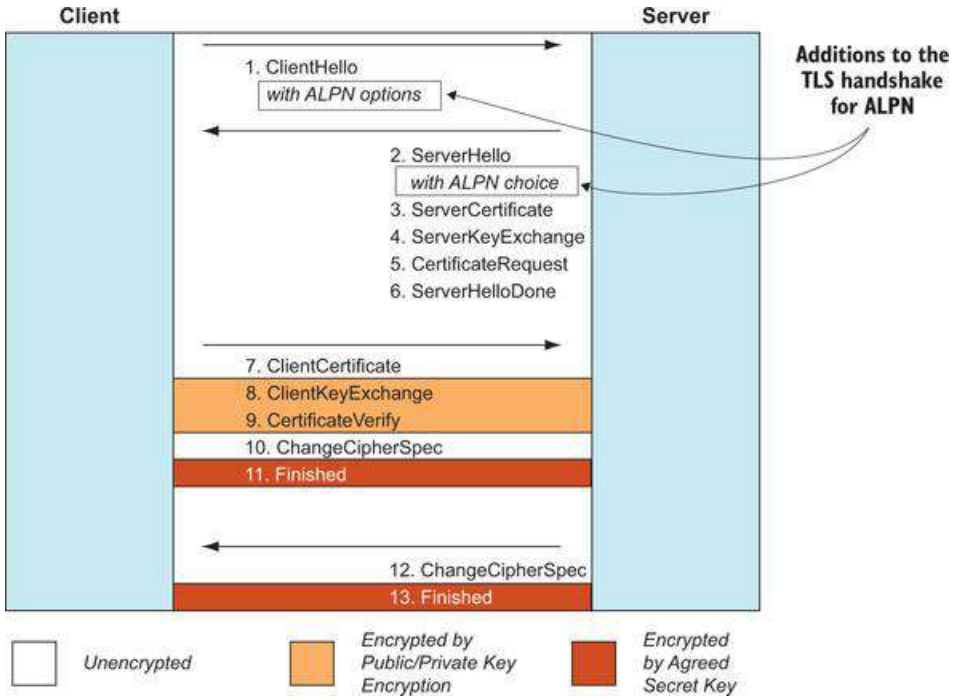

Figura 2.4: HTTPS handshake con ALPN

ALPN è semplice e può essere utilizzato per accordarsi se utilizzare o meno HTTP/2 per i messaggi di negoziazione HTTPS esistenti senza aggiungere ulteriori round trip, reindirizzamenti o altri ritardi di aggiornamento. Inoltre ALPN può essere utilizzato per altri protocolli oltre a HTTP/2, tant'è che quando ALPN nacque nel

[10] https://tools.ietf.org/html/rfc7301

2014 includeva sole le prime tre versioni di HTTP: HTTP/0.9, HTTP/1.0, e HTTP/1.1. Successivamente con l'uscita di HTTP/2 venne registrato [2].

2.3.2 Header HTTP Upgrade

Un client può richiedere un upgrade di una esistente connessione HTTP/1.1 con una HTTP/2 mandando una header HTTP **Upgrade**. Questo header dovrebbe esser utilizzato solo per connessioni HTTP non criptate (h2c). Connessioni HTTP/2 criptate HTTPS (h2) non dovrebbero usare questo metodo per negoziare HTTP/2 e dovrebbero usare ALPN che fa parte della negoziazione HTTPS.

L'esempio successivo mostra come funziona l'header Upgrade.

```
GET / HTTP/1.1
Host: www.example.com
Upgrade: h2c
```

Un server che non comprende HTTP/2 può rispondere con un messaggio HTTP/1.1, come se l'header Upgrade non fosse stato inviato

```
HTTP/1.1 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Connection: Keep-Alive
Content-Type: text/html
Server: Apache

<!doctype html>
<html>
<head>
...etc.
```

Invece un server che comprende HTTP/2, invece di ignorare la richiesta di upgrade, può rispondere con un **`HTTP/1.1 101`**, informado che cambiarà protocollo

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

Successivamente il server passa immediatamente ad HTTP/2, inviando un frame `SETTING` e quindi inviando la risposta al messaggio originale in formato HTTP/2 [2].

2.3.3 Prior knowledge

Il terzo ed ultimo modo per instaurare una comunicazione HTTP/2 funziona se il client sa già che il server comprende HTTP/2. In questo caso il client può iniziare a "parlare" HTTP/2 immediatamente, saltando qualsiasi richiesta di upgrade. Ci sono vari modi con cui il client sa se il server comprende HTTP/2, uno di questi consiste nel utilizzo del HTTP Alternative Services (2.3.4 HTTP Alternative Services).

Questa opzione è la più rischiosa perché presupponendo che il server parli HTTP/2 i client devono avere cura di gestire in modo appropriato eventuali messaggi di rifiuto nel caso la presupposizione risultasse errata. Questo metodo dovrebbe essere utilizzato solo quando si ha il controllo sia del client che del server [2].

2.3.4 HTTP Alternative Services

Un quarto modo, non incluso nella specifica originale di HTTP/2, è attraverso l'utilizzo di HTTP Alternative Services[11], aggiunto come uno standard separato dopo il rilascio di HTTP/2. Questo standard permette al server di informare il client usando HTTP/1.1 (attraverso un header HTTP **Alt-Svc**) che la risorsa richiesta è disponibile in un'altra posizione (come un'altra porta del server) usando un protocollo diverso. Questo protocollo potrà essere utilizzato per iniziare una comunicazione HTTP/2 con il prior knowledge citato precedentemente (2.3.3 Prior knowledge). Inoltre HTTP Alternative Services non serve solo in questo caso ma può essere utilizzato sopra una connessione HTTP/2 in caso si voglia cambiare connessione (tipo cambiare con una più vicina al client o una meno occupata) [2].

2.3.5 La "magic" string di HTTP/2

Il primo messaggio che deve esser mandato in una connessione HTTP/2 (non importa quale metodo si utilizzi per stabilirla) è il messaggio di prefazione HTTP/2, o detta anche la "magic" string. Questo messaggio viene inviato dal client come primo

[11] https://tools.ietf.org/html/rfc7838

messaggio in una connessione HTTP/2. È una sequenza di 24 ottetti ed ha questo aspetto in notazione esadecimale:

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0
```

Questa sequenza si traduce nel seguente messaggio in ASCII:

```
PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
```

A prima vista può sembrare un messaggio strano, ma non a caso, è quasi un messaggio in stile HTTP/1:

```
PRI * HTTP/2.0↵
↵
SM↵
↵
```

Il metodo HTTP è **`PRI`** (invece di `GET` o `POST`), la risorsa è **`*`** (tutto), e la versione di HTTP è `HTTP/2.0`. Successivamente un doppio ritorno (↵ quindi non è presente un header request) seguito dal corpo della richiesta **`SM`**.

L'intenzione di questo messaggio senza senso serve per quando un client cerca di parlare HTTP/2 con un server che non lo parla. Un server di questo tipo tenta di analizzare il messaggio come farebbe con un qualsiasi altro messaggio HTTP e non riesce perché non riconosce il metodo senza senso (`PRI`) o la versione HTTP (`HTTP/2.0`) e dovrebbe rifiutare il messaggio. Dal punto di vista del server invece, il server sa che il client parla HTTP/2 ma invece di inviare la magic string deve inviare un frame `SETTINGS` come primo messaggio (che può essere vuoto) [2].

2.4 HTTP/2 frames

Quando una connessione HTTP/2 viene stabilita, si può iniziare ad inviare messaggi HTTP/2. I messaggi HTTP/2 sono composti da frame di dati che vengono inviati su stream di una singola connessione di tipo multiplex. I frame sono un concetto di

basso livello che molti web developer non hanno bisogno di conoscere, ma vale sempre la pena comprendere gli elementi costruttivi di una tecnologia. Molti errori possono essere risolti guardando la comunicazione al livello dei frame HTTP/2, quindi vedere la connessione da quel punto di vista ha usi pratici oltre che teorici [2].

2.4.1 Formato dei frame HTTP/2

Prima di iniziare a vedere un esempio reale dei frame che vengono inviati quando si visita una pagina web, può essere utile capire come è costruito un frame HTTP/2. Ogni frame è composto da un header con lunghezza fissa (Tabella 2.1), seguito dal payload (carico).

Tabella 2.1: Formato dei frame HTTP/2

Campo	Lunghezza	Descrizione
Length	24 bit	Lunghezza del frame, esclusi tutti i campi header descritti in questa tabella con una dimensione massima di 2^{24} - 1 ottetti; limitato dal SETTING_MAX_FRAME_SIZE, che di default ha la dimensione di 2^{24} - 1 ottetti
Type	8 bit	Attualmente, 14 tipi di frame:[a] ■ DATA (0x0) ■ HEADERS (0x1) ■ PRIORITY (0x2) ■ RST_STREAM (0x3) ■ SETTINGS (0x4) ■ PUSH_PROMISE (0x5) ■ PING (0x6) ■ GOAWAY (0x7) ■ WINDOW_UPDATE (0x8) ■ CONTINUATION (0x9) ■ ALTSVC (0xa), aggiunto con RFC 7838[b] ■ (0xb), usato in passato[c] ■ ORIGIN (0xc), aggiunto con RFC 8336[d]
Flags	8 bit	Flag specifici del frame
Reserved Bit	1 bit	Attualmente non in uso e deve essere impostato a 0
Stream Identifier	31 bit	Un numero intero senza segno di 31 byte che identifica il frame

[a] https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml
[b] https://tools.ietf.org/html/rfc7838
[c] https://github.com/httpwg/http-extensions/pull/323
[d] https://tools.ietf.org/html/rfc8336

Il fatto che i frame siano definiti in modo così esplicito è ciò che rende HTTP/2 un protocollo binario. I frame HTTP/2 sono diversi dai messaggi di testo HTTP/1 di lunghezza variabile, che dovevano essere analizzati cercando interruzioni di riga e spazi, un processo inefficiente e soggetto ad errori. Il formato molto più rigoroso e ben definito dei frame HTTP/2 consente un'analisi più semplice e messaggi più piccoli, poiché è possibile utilizzare codici particolari (come `0x01` per specificare che il frame è di tipo **`HEADER`**).

Il campo **`Length`** è autoesplicativo. Nella prossima sezione (2.4.2 Esamina del flusso di messaggi HTTP/2) ci sarà un approfondimento del campo **`Type`** in dettaglio. Il campo **`Flags`** è specifico in base al tipo del frame. Attualmente il campo **`Reserved Bit`** non viene utilizzato. Ed anche il campo **`Stream Identifier`** è autoesplicativo [2].

2.4.2 Esamina del flusso di messaggi HTTP/2

Il modo più semplice per capire i frame è vederne un utilizzo nel mondo reale. Il seguente esempio è stato ottenuto usando nghttp per connettersi a www.facebook.com.

```
$ nghttp -va https://www.facebook.com | more
[  0.043] Connected
The negotiated protocol: h2
[  0.107] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
[  0.107] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=10420225)
[  0.107] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[  0.107] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=3>
          (dep_stream_id=0, weight=201, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=5>
          (dep_stream_id=0, weight=101, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=7>
          (dep_stream_id=0, weight=1, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=9>
          (dep_stream_id=7, weight=1, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=11>
          (dep_stream_id=3, weight=1, exclusive=0)
[  0.107] send HEADERS frame <length=43, flags=0x25, stream_id=13>
```

```
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
          ; Open new stream
          :method: GET
          :path: /
          :scheme: https
          :authority: www.facebook.com
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.28.0
[  0.138] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
[  0.138] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=10420224)
[  0.257] recv (stream_id=13) :status: 200
[  0.257] recv (stream_id=13) x-xss-protection: 0
[  0.257] recv (stream_id=13) pragma: no-cache
[  0.257] recv (stream_id=13) cache-control: private, no-cache, no-store,
must-revalidate
[  0.257] recv (stream_id=13) x-frame-options: DENY
[  0.257] recv (stream_id=13) strict-transport-security: max-age=15552000;
preload
[  0.257] recv (stream_id=13) x-content-type-options: nosniff
[  0.257] recv (stream_id=13) expires: Sat, 01 Jan 2000 00:00:00 GMT
[  0.257] recv (stream_id=13) set-cookie: fr=0m7urZrTka6WQuSGa..BaQ42y.61.A
AA.0.0.BaQ42y.AWXRqgzE; expires=Tue, 27-Mar-2018 12:10:26 GMT; Max-Age=7776
000; path=/; domain=.facebook.com; secu
re; httponly
[  0.257] recv (stream_id=13) set-cookie: sb=so1DWrDge9fIkTZ7e-i5S2To; expi
res=Fri, 27-Dec-2019 12:10:26 GMT; Max-Age=63072000; path=/; domain=.facebo
ok.com; secure; httponly
[  0.257] recv (stream_id=13) vary: Accept-Encoding
[  0.257] recv (stream_id=13) content-encoding: gzip
[  0.257] recv (stream_id=13) content-type: text/html; charset=UTF-8
[  0.257] recv (stream_id=13) x-fb-debug: yrE7eqv05dkxF8R1+i4VlIZmUNInVI+AP
DyG7HCW6t7NCEtGkIIRqJadLwj87Hmhk6z/N3O212zTPFXkT2GnSw==
[  0.257] recv (stream_id=13) date: Wed, 27 Dec 2017 12:10:26 GMT
[  0.257] recv HEADERS frame <length=517, flags=0x04, stream_id=13>
          ; END_HEADERS
          (padlen=0)
          ; First response header
<!DOCTYPE html>
<html lang="en" id="facebook" class="no_js">
<head><meta charset="utf-8" />
...etc.
[  0.243] recv DATA frame <length=1122, flags=0x00, stream_id=13>
...
[  0.243] recv DATA frame <length=2589, flags=0x00, stream_id=13>
...
[  0.264] recv DATA frame <length=13707, flags=0x00, stream_id=13>
...
[  0.267] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=33706)
[  0.267] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=33706)
...
[416.688] recv DATA frame <length=8920, flags=0x01, stream_id=13>
          ; END_STREAM
[417.226] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
          (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

Parte dell'output è stato tagliato, in particolare molti frame **`DATA`** sono stati tagliati. Anche con molti tagli questo output può sembrare complesso e per questo verrà analizzato a pezzi in questo capitolo.

Per prima cosa, una volta instaurata la connessione, avviene la negoziazione di HTTP/2 su HTTPS (h2). Dato che nghttp non mostra l'impostazione di HTTPS o la magic string di HTTP/2, si riceve immediatamente il frame **SETTING** [2].

```
$ nghttp -v https://www.facebook.com | more
[0.043] Connected
The negotiated protocol: h2
[0.107] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
```

SETTINGS FRAME

Il SETTINGS frame (**0x4**) è il primo frame che deve essere inviato dal server ed anche dal client (dopo la magic string di HTTP/2). Il frame è costituito da un payload vuoto o da più coppie campo/valore, come mostrato nella Tabella 2.2.

Tabella 2.2: Formato del frame SETTINGS

Campo	Lunghezza	Descrizione
Identifier	16 bit	Sei impostazioni sono definite nella specifica e in seguito ne è stata aggiunta una ulteriore: ■ SETTINGS_HEADER_TABLE_SIZE (0x1) ■ SETTINGS_ENABLE_PUSH (0x2) ■ SETTINGS_MAX_CUNCURRENT_STREAMS (0x3) ■ SETTINGS_INITIAL_WINDOW_SIZE (0x4) ■ SETTINGS_MAX_FRAME_SIZE (0x5) ■ SETTINGS_MAX_HEADER_LIST_SIZE (0x6) ■ Non assegnato (0x7) ■ SETTINGS_ENABLE_CONNECT_PROTOCOL (0x8), aggiunto con RFC 8441[a]
Value	32 bit	Valori di default: ■ 4096 ottetti ■ 1 ■ Infinito ■ 65535 ottetti ■ 16384 ottetti ■ Infinito ■ 0

[a] https://tools.ietf.org/html/rfc8441

Il frame SETTINGS definisce solo un flag che può essere impostato nel frame header comune: **ACK** (0x1). Viene impostato il flag a 0 all'inizio della comunicazione; viene successivamente impostato su 1 per un riconoscimento dall'altra parte della comunicazione (ricevente). Se si riceve il frame con il flag a 1, allora non si dovrebbero più mandare altri frame settings.

Adesso con queste conoscenze si può analizzare meglio il primo messaggio dell'esempio fatto ad inizio sezione:

```
[0.107] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
```

Il frame SETTINGS che è stato ricevuto ha un payload con lunghezza di 30 ottetti, con nessun flag impostato (quindi non è un frame di acknowledgement, ACK), ed ha come stream ID 0. Stream ID 0 è riservato per i messaggi di controllo (frame SETTINGS e frame WINDOW_UPDATE), quindi è corretto per il server l'utilizzo dello stream 0 per inviare questo frame SETTINGS.

Successivamente si ricevono i setting veri e propri, in questo caso cinque come viene anche anticipato (niv=5). Analizzando individualmente le impostazioni dei setting si può notare che:

1. Facebook utilizza un SETTING_HEADER_TABLE_SIZE di 4096 ottetti. Questa impostazione viene utilizzata per la compressione HPACK dell'header HTTP.
2. Inoltre Facebook utilizza un SETTING_MAX_FRAM_SIZE di 16384 ottetti, in questo modo il client (nghttp) non deve inviare alcun payload piu grande su questa connessione.

3. Dopo Facebook imposta il SETTINGS_MAX_HEADER_LIST_SIZE a 131072 ottetti, in modo da non permettere al client di inviare header non compressi piu grandi di quanto impostato.
4. Facebook imposta il SETTINGS_MAX_CONCURRENT_STREAMS a 100 stream (come discusso nella sezione 2.2.3 Prioritizzazione dello stream e flow control).
5. Infine Facebook imposta il SETTINGS_INITIAL_WINDOW_SIZE a 65536 ottetti. Questa impostazione viene utilizzata per il flow control.

Vale la pena fare delle osservazioni su quanto successo in questo frame. Per cominciare, i setting possono essere inviati in qualsiasi ordine. Inoltre molti di questi setting hanno utilizzato le impostazioni di default, quindi in realtà il server avrebbe potuto mandare questo frame SETTINGS ridotto:

```
[0.107] recv SETTINGS frame <length=18, flags=0x00, stream_id=0>
          (niv=3)
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
```

Non c'è nulla di male nell'essere più espliciti sui valori che si vuole utilizzare. Da notare che Facebook non imposta il SETTINGS_ENABLE_PUSH. Questa impostazione permette al server di inviare inviare al client risorse non richieste ma di cui avrà bisogno, quindi sta al client utilizzare questo setting. È importante per il client disattivarlo se non supporta HTTP/2 push o non vuole abilitarlo.

Tornando all'esempio, analizzando i successivi 3 frame SETTINGS:

```
[0.107] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[0.107] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
```

Dopo il primo frame SETTINGS (già discusso), il client manda un frame SETTINGS con un paio di setting. Successivamente il client manda un acknowledgment SETTINGS frame, con il flag **ACK** (0x01) impostato, con lunghezza 0 e con nessun setting da inviare (niv=0). Un po' più sotto nel flusso di frame si trova l'acknowledgement del server al frame SETTINGS del client, con la stessa impostazione.

```
[0.138] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
```

Questo esempio mostra che esiste un periodo dove una parte ha inviato un frame SETTINGS ma non ha ricevuto alcun acknowledgment. In questo lasso di tempo i setting ricevuti non possono essere utilizzati e verranno impiegati quelli di default. Poiché tutte le implementazioni HTTP/2 devono essere in grado di elaborare i valori di default, non si incorre in errori di alcun tipo [2].

WINDOW_UPDATE FRAME

Il server inoltre ha inviato un frame WINDOW_UPDATE:

```
[0.107] recv WINDOW_UPDATE frame <length=4, flags=0x00,
stream_id=0>
          (window_size_increment=10420225)
```

Il frame WINDOW_UPDATE (0x8) viene utilizzato per il flow control, per limitare la quantità di dati che possono essere inviati in modo da evitare di sovraccaricare il ricevitore. Sotto HTTP/1, può viaggiare solo una richiesta alla volta per connessione TCP. Se il client inizia a sovraccaricarsi con i dati, ferma di processare i pacchetti TCP; poi il flow control di TCP (simile a quello di HTTP/2) entra in gioco e rallenta l'invio dei dati fino a che il ricevitore non è in grado di poterne gestire di più. In HTTP/2 ci sono stream multipli sulla stessa connessione, quindi non si può dipendere dal flow control di TCP ma bisogna implementare il proprio metodo di rallentamento del flusso.

Il valore iniziale per il window size lo si trova nel frame SETTTINGS, e il frame WINDOW_UPDATE viene utilizzato per incrementare questa finestra. Quest'ultimo è un frame molto semplice, senza alcun flag e con solo un valore (e il bit riservato), come mostrato nella Tabella 2.3.

Tabella 2.3: Formato del frame WINDOW_UPDATE

Campo	Lunghezza	Descrizione
Reserved Bit	1 bit	Non viene utilizzato
Window Size Increment	31 bit	Numero di ottetti che possono essere inviati prima che il successivo frame WINDOW_UPADATE viene ricevuto

Il WINDOW_UPDATE viene applicato allo stream dato, oppure se lo stream scelto risulta lo stream 0, le modifiche vengono applicate all'intera connessione HTTP/2.

Il flow control di HTTP/2 si applica solo ai frame DATA. Tutti gli altri tipi di frame possono continuare ad essere inviati anche se la finestra del flow control è stata utilizzata. Questo impedisce che i messaggi importanti (come lo stesso WINDOW_UPDATE) vengano bloccati da grandi frame DATA [2].

PRIORITY FRAME

I frame successivi sono parecchi frame PRIORITY:

```
[0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=3>
          (dep_stream_id=0, weight=201, exclusive=0)
[0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=5>
          (dep_stream_id=0, weight=101, exclusive=0)
[0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=7>
          (dep_stream_id=0, weight=1, exclusive=0)
[0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=9>
          (dep_stream_id=7, weight=1, exclusive=0)
[0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=11>
          (dep_stream_id=3, weight=1, exclusive=0)
```

La gestione della priorità degli stream può diventare complessa ma in breve il funzionamento del frame è alquanto semplice. Alla base è sufficiente sapere che ci

sono alcune richieste (come l'HTML iniziale, CSS critico, JavaScript critico) a cui si può dare una priorità rispetto ad altre (come immagini o JavaScript asincrono non critico). Nel frame `PRIORITY`, come mostrato in Tabella 2.4, sono presenti i tre campi: `Stream Dependency`, `Weigh` ed `E (Exclusive)`. Il primo specifica la dipendenza della risorsa, il secondo il suo peso e il terzo l'esclusività di accesso alla dipendenza. Utilizzando questi campi il server può decidere in che ordine il client deve elaborare le risorse [2].

Tabella 2.4: Formato del frame `PRIORITY`

Campo	Lunghezza	Descrizione
E (Exclusive)	1 bit	Indica se lo stream è esclusivo
Stream Dependency	31 bit	Indica a quale stream questo header dipende
Weight	8 bit	Il peso (ponderazione) dello stream

HEADERS FRAME

Dopo le prime impostazioni si arriva al nocciolo del protocollo e viene fatta una richiesta HTTP/2 che viene inviata in un frame `HEADERS` (`0x1`):

```
[0.107] send HEADERS frame <length=43, flags=0x25, stream_id=13>
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
          ; Open new stream
          :method: GET
          :path: /
          :scheme: https
          :authority: www.facebook.com
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.28.0
```

Se si ignora le prime righe del frame header, il resto dovrebbe sembrare familiare ad una richiesta HTTP/1. Per ricordare, una richiesta HTTP/1 è composta dalla combinazione della first line (metodo e versione del protocollo) e dal header `Host` obbligatorio (seguito da altri header HTTP se presenti):

```
GET / HTTP/1.1↵
Host: www.facebook.com↵
```

In HTTP/2 invece di avere uno specifico tipo di header per le richieste o una diversa first line nel frame `HEADERS`, **tutto** viene inviato come header, e sono stati creati anche dei nuovi **psedutoheaders** (che iniziano con i due punti) per definire le varie parti della richiesta HTTP:

```
:method: GET
          :path: /
          :scheme: https
          :authority: www.facebook.com
```

Notare che lo pseudoheader **`:authority`** ha sostituito l'header `Host` di HTTP/1.1. Gli pseudoheaders di HTTP/2 sono rigorosamente definiti e non si possono aggiungere dei propri pseudoheaders come avveniva per gli headers HTTP.[12]

```
:barry: value
```

È necessario attenersi ai normali header HTTP, senza i due punti iniziali, per tutti gli header specifici dell'applicativo.

```
barry: value
```

Gli pseudoheader possono essere creati solo in caso di nuove specifiche, come è già accaduto con lo pseudoheader **`:protocol`**.[13] L'aggiunta di un nuovo pseudoheader necessita anche di un nuovo parametro `SETTINGS` per indicare il supporto da parte di client e server (`SETTINGS_ENABLE_CONNECT_PROTOCOL`).

[12] https://tools.ietf.org/html/rfc7540#section-8.1.2
[13] https://tools.ietf.org/html/rfc8441#section-3

Tabella 2.5: Formato del frame HEADERS

Campo	Lunghezza	Descrizione
Pad Length	8 bit (opzionale)	Un campo opzionale che indica la lunghezza del campo Padding
E (Exclusive)	1 bit	Indica se lo stream è esclusivo
Stream Dependency	31 bit	Indica a quale stream questo header dipende
Weight	8 bit	Il peso (ponderazione) dello stream
Header Block Fragment	Lunghezza del frame meno gli altri campi in questa tabella	Gli header della richiesta (inclusi gli psdeudoheader)
Padding	Indicato dal campo Pad Length	Impostato a 0 per ogni byte rimpieto

Alcuni di questi campi (Stream Dependency, Weight ed E) sono stati discussi nella sezione PRIORITY FRAME. I campi Pad Length e Padding vengono aggiunti per motivi di sicurezza per consentire facoltativamente di nascondere la lunghezza effettiva del messaggio. Il campo Header Block Fragment è dove tutti gli header (inclusi gli pseudoheader) vengono inviati. Questo campo non è in chiaro, come potrebbe far intendere l'output mostrato prima. In realtà questo campo viene compresso e criptato (HPACK), ma in questo caso nghttp lo ha decompresso e decriptato mostrandone il contenuto.

Il frame HEADERS definisce anche quattro flag che possono essere impostate nel header frame comune:

- END_STREAM (0x1) è impostato se nessun altro frame segue questo frame HEADERS (come un frame DATA per una richiesta POST). In modo un po' controintuitivo, i frame CONTINUATION sono esenti da questa restrizione; sono considerati continuazioni del frame HEADERS piuttosto che frame aggiuntivi e sono controllati dal flag END_HEADERS

- END_HEADERS (0x4) indica che tutti gli header HTTP sono contenuti in questo frame e non sono seguiti da un frame CONTINUATION con header aggiuntivi
- PADDED (0x8) viene impostato quando si utilizza il riempimento. Questo flag significa che i primi 8 bit del frame DATA indicano quanto padding è stato aggiunto alla fine del frame HEADERS
- PRIORITY (0x20) indica che i campi E, Stram Dependency e Weight sono impostati in questo frame

Adesso riguardando all'output del frame HEADER visto ad inizio sezione:

```
[0.107] send HEADERS frame <length=43, flags=0x25, stream_id=13>
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
          ; Open new stream
          :method: GET
          :path: /
          :scheme: https
          :authority: www.facebook.com
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.28.0
```

Vari flag vengono impostati, che combinati compongono il valore esadecimale di 0x25 e che nghttp2 mostra nella riga sottostante. Con i flag END_STREAM (0x1) ed END_HEADERS (0x4) si indica che questo frame contiene la richiesta completa e che non è presente un frame DATA. Il flag PRIORITY (0x20) indica che ci sono delle priorità in questo frame. Inoltre si può notare che questo frame dipende dallo stream 11, quindi gli viene impostata la priorita appropriata e un peso di 16. Nghttp nota che questo stream è nuovo e quindi elenca i vari pseudoheader HTTP ed i vari header delle richieste HTTP.

Le risposte HTTP vengono anche inviate con un frame HEADERS sullo stesso stream, come si vede in questo esempio:

```
[  0.257] recv (stream_id=13) :status: 200
[  0.257] recv (stream_id=13) x-xss-protection: 0
[  0.257] recv (stream_id=13) pragma: no-cache
```

```
[  0.257] recv (stream_id=13) cache-control: private, no-cache, no-store,
must-revalidate
[  0.257] recv (stream_id=13) x-frame-options: DENY
[  0.257] recv (stream_id=13) strict-transport-security: max-age=15552000;
preload
[  0.257] recv (stream_id=13) x-content-type-options: nosniff
[  0.257] recv (stream_id=13) expires: Sat, 01 Jan 2000 00:00:00 GMT
[  0.257] recv (stream_id=13) set-cookie: fr=0m7urZrTka6WQuSGa..BaQ4Ay.61.A
AA.0.0.BaQ42y.12345678; expires=Tue, 27-Mar-2018 12:10:26 GMT; Max-Age=7776
000; path=/; domain=.facebook.com; secu
re; httponly
[  0.257] recv (stream_id=13) set-cookie: sb=so11234567890TZ7e-i5S2To; expi
res=Fri, 27-Dec-2019 12:10:26 GMT; Max-Age=63072000; path=/; domain=.facebo
ok.com; secure; httponly
[  0.257] recv (stream_id=13) vary: Accept-Encoding
[  0.257] recv (stream_id=13) content-encoding: gzip
[  0.257] recv (stream_id=13) content-type: text/html; charset=UTF-8
[  0.257] recv (stream_id=13) x-fb-debug: yrE7eqv05dkxF8R1+1234567890nVI+AP
DyG7HCW6t7NCEtGkIIRqJadLwj87Hmhk6z/N3O212zTPFXkT2GnSw==
[  0.257] recv (stream_id=13) date: Wed, 27 Dec 2017 12:10:26 GMT
[  0.257] recv HEADERS frame <length=517, flags=0x04, stream_id=13>
          ; END_HEADERS
          (padlen=0)
          ; First response header
```

La prima risposta è lo pseudoheader **`:status`**, che da come risposta solo il codice a tre cifre (`200`). Questo pseudoheader è seguito da vari header HTTP (ovviamente non vengono mandati in chiaro ma si utilizza HPACK per rendere gli header più sicuri e leggeri). In maniera un po' confusa nghttp da prima il payload del frame piuttosto che i dettagli. Infatti in fondo a questa lista di header si trovano i dettagli del frame dove il flag `END_HEADERS` (`0x4`) segnala che l'intera risposta è entrata nel frame [2].

DATA FRAME

Dopo il frame `HEADERS`, il frame `DATA` (`0x0`) viene utilizzato per inviare i corpi dei messaggi (bodies). In HTTP/1 il body veniva inviato nella risposta dopo gli header HTTP. In HTTP/2, i dati viaggiano su messaggi separati dato che sono un tipo di messaggio diverso. Grazie a questa separazione di risposte in più frame, è possibile avere stream multiplexati sulla stessa connessione.

I frame `DATA` sono semplici, contengono tutti i dati necessari: codifica UTF-8, gzip, codice HTML, byte che compongono un'immagine JPEG o qualsiasi altra cosa. L'header principale contiene già il campo lunghezza quindi nel frame `DATA` non viene richiesto. Come per il frame `HEADERS`, il frame `DATA` consente l'uso del padding per oscurare la dimensione del messaggio per motivi di sicurezza. Quindi il formato del frame `DATA` è alquanto semplice come mostrato in Tabella 2.6.

Tabella 2.6: Formato del frame DATA

Campo	Lunghezza	Descrizione
Pad Length	8 bit (opzionale)	Un campo opzionale che indica la lunghezza del campo Padding
Data	31 bit	I dati
Padding	Indicato dal campo Pad Length	Impostato a 0 per ogni byte rimpieto

Il frame DATA definisce due flag che possono essere impostate nel header frame comune:

- END_STREAM (0x1) è impostato se questo frame è l'ultimo nello stream
- PADDED (0x8) viene impostato quando si utilizza il riempimento. Significa che i primi 8 bit del frame DATA vengono utilizzati per indicare quanto padding è stato aggiunto alla fine del frame.

In questo esempio molto testo è stato tagliato per motivi di spazio ma al posto di **...etc** ci dovrebbero essere altri dati come all'inizio dell'esempio:

```
<!DOCTYPE html>
<html lang="en" id="facebook" class="no_js">
<head><meta charset="utf-8" />
...etc.
[  0.243] recv DATA frame <length=1122, flags=0x00, stream_id=13>
...etc.
[  0.243] recv DATA frame <length=2589, flags=0x00, stream_id=13>
...etc.
[  0.264] recv DATA frame <length=13707, flags=0x00, stream_id=13>
...etc.
[  0.267] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=33706)
[  0.267] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=33706)
...etc.
[416.688] recv DATA frame <length=8920, flags=0x00, stream_id=13>
```

Qui si può vedere come il codice HTML è stato inviato in vari frame DATA, e man mano che il client processa i frame, manda al server i frame WINDOW_UPDATE, permettendo al server di mandare più dati [2].

GOAWAY FRAME

Alla fine dei messaggi dell'esempio iniziale si trova il frame GOAWAY (0x7):

```
[417.226] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
(last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

Questo frame viene utilizzato per interrompere la connessione perché non ci sono più messaggi da inviare o perché si è verificato un errore grave. Il formato del frame GOAWAY viene mostrato nella tabella 2.7.

Tabella 2.6: Formato del frame GOAWAY

Campo	Lunghezza	Descrizione
Reserved Bit	1 bit	Non viene utilizzato
Last-Stream_ID	31 bit	L'ultimo ID stream elaborato in entrata, per consentire al client di sapere se uno stream avviato di recente è stato perso
Error Code	32 bit	Il codice di errore, nel caso in cui il frame GOAWAY sia stato inviato a causa di un errore: ■ NO_ERROR (0x0) ■ PROTOCOL_ERROR (0x1) ■ INTERNAL_ERROR (0x2) ■ FLOW_CONTROL_ERROR (0x3) ■ SETTINGS_TIMEOUT (0x4) ■ STREAM_CLOSED (0x5) ■ FRAME_SIZE_ERROR (0x6) ■ REFUSED_STREAM (0x7) ■ CANCEL (0x8) ■ COMPRESSION_ERROR (0x9) ■ CONNECT_ERROR (0xa) ■ ENHANCE_YOUR_CALM (0xb) ■ INADEQUATE_SECURITY (0xc) ■ HTTP_1_1_REQUIRED (0xd)
Additional Debug Data	Resto della lunghezza del frame (opzionale)	Formato non definito, specifico per l'implementazione

Questo frame non definisce alcun flag. Adesso riguardando l'esempio del frame `GOAWAY` si può notare come la connessione sia stata chiusa nel modo più pulito possibile. L'ultimo stream ID ricevuto è `0` quindi l'ultimo stream non era uno stream avviato del server. Non ci sono stati errori (`NO_ERROR` [`0x0`]) e non ci sono dati debug aggiuntivi [2].

Capitolo 3
HTTP/3

HTTP/2 è stato concepito per migliorare le inefficienze del protocollo HTTP, principalmente permettendo una singola connessione multiplex. Sotto HTTP/1.1 la connessione era ampiamente sottoutilizzata, perché poteva essere utilizzata solo per una risorsa alla volta. Se c'erano dei ritardi nella risposta di una richiesta, la connessione veniva bloccata e non utilizzata. HTTP/2 consente di utilizzare la connessione per più risorse, quindi altre risorse possono utilizzare la stessa connessione.

Oltre a prevenire connessioni sprecate, HTTP/2 fornisce prestazioni migliorate, perché le connessione HTTP sono di per se inefficienti. C'è un costo per creare una connessione HTTP, altrimenti non ci sarebbe alcun vantaggio nell'utilizzo del multiplexing. I costi non sono dovuti al HTTP stesso, ma alle due tecnologie sottostanti utilizzate per creare la connessione: TCP e TLS per fornire HTTPS.

TCP è un protocollo apparentemente semplice ma è molto più complesso di quanto la maggior parte delle persone creda. Come per HTTP/1.1, TCP presenta alcune inefficienze alla base e alcuni utenti potrebbero iniziare a percepirle solo ora poiché vengono affrontate nei protocolli di livello superiore come HTTP, dovuto anche alle richieste per sito in continuo aumento sul web.

Il protocollo è ancora in evoluzione, anche se piuttosto lentamente. Sebbene vengano create continuamente nuove opzioni e algoritmi per il congestion control, i browser vengono aggiornati per sfruttare queste funzionalità, se disponibili, ci vuole del tempo prima che entrino negli stack di rete dei server. Per molte ragioni alcuni si chiedono se TCP dopotutto sia il protocollo sottostante corretto per HTTP e se un nuovo protocollo sia la strada da percorre, un protocollo progettato da zero per le esigenze attuali e future di HTTP (la specifica di TCP risale a più di trent'anni fa). Uno di questi protocolli è QUIC [2].

3.1 QUIC

QUIC (pronunciato quick) è un nuovo protocollo inventato ed utilizzato tutt'oggi da Google basato su UDP, che mira a sostituire TCP e altre parti del tradizionale stack HTTP per affrontare molte delle inefficienze. HTTP/2 ha introdotto alcuni concetti simili a TCP (come i pacchetti e il flow control), ma QUIC porta questi concetti al livello successivo e sostituisce TCP.

QUIC è stato creato con queste caratteristiche in mente:

- Tempi di creazione della connessione drasticamente ridotti
- Congestion control migliorato
- Multiplexing senza il blocco HOL (Head of Line)
- Forward error correction
- Migrazione della connessione

Le prime tre caratteristiche sono autoesplicative. Le ultime due sono delle aggiunte particolarmente interessanti.

Forward error correction (FEC) cerca di ridurre la necessità di ritrasmissione dei pacchetti. Ogni pacchetto che viene inviato include anche dati sufficienti degli altri pacchetti in modo che un pacchetto mancante possa essere ricostruito senza doverlo ritrasmettere. Ogni pacchetto UDP contiene più payload di quanto sia strettamente necessario, perché tiene conto del potenziale di pacchetti persi che possono essere ricreati più facilmente in questo modo. Quindi bisogna considerare il Forward Error Correction come un sacrificio in termini di "dati per pacchetto UDP" che possono essere inviati, ma il vantaggio è non dover ritrasmettere un pacchetto perso, il che richiederebbe molto più tempo (il destinatario deve confermare un pacchetto mancante, richiedilo di nuovo e attendere la risposta).

La migrazione della connessione mira a ridurre il sovraccarico della configurazione della connessione consentendo a una connessione di spostarsi tra le reti. Sotto TCP, la connessione è collegata all’indirizzo IP e alla porta su entrambe le parti (client e server). La modifica dell’indirizzo IP richiede la creazione di una nuova connessione TCP. Questo andava bene quando TCP è stato inventato perché era

improbabile che gli indirizzi IP cambiassero durante la durata di una sessione. Ora, con più reti (cablate, wireless e mobile), questa situazione non può essere data per scontata. QUIC ti consente di avviare la sessione a casa tramite Wi-Fi e poi passare a una rete mobile senza dover riavviare la sessione. Inoltre si dovrebbe poter utilizzare entrambe le connessioni contemporaneamente per una connessione QUIC tramite una tecnica chiamata **multipath** [2].

È bene specificare che Forward error correction e multipath non sono utilizzabili nella versione attuale di QUIC ma sono in fase di sperimentazione. Sono stati posticipati sia per permettere l'uscita di QUICv1 più in fretta, sia perché non portavano grandi migliorie in performance al tempo.

Sebbene UPD non sia un protocollo di trasporto sicuro, QUIC aggiunge uno strato sopra UDP che introduce l'affidabilità. Offre ritrasmissione dei pacchetti, congestion control e altre funzionalità altrimenti già presenti in TCP. I dati inviati tramiti QUIC da un endpoint appariranno prima o poi all'altra estremità, a condizione che la connessione venga mantenuta. Nella Figura 3.1 viene mostrato lo stack di rete HTTP/2 a sinistra e lo stack di rete QUIC a destra, quando viene utilizzato HTTP come trasporto [4].

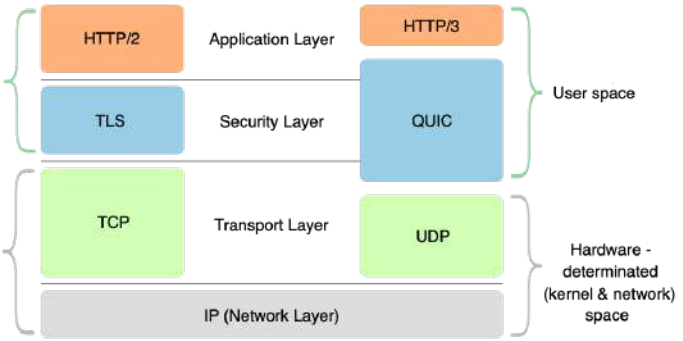

Figura 3.1: Dove si posiziona QUIC nello stack HTTP

3.1.1 Connessione

Una connessione QUIC è una singola conversazione tra due endpoint QUIC. La creazione della connessione combina la negoziazione della versione con le negoziazioni dell'algoritmo crittografico e del trasporto al fine di ridurre la latenza di setup della connessione. Per inviare effettivamente i dati su tale connessione, è necessario creare e utilizzare uno o più stream.

Ogni connessione possiede una serie di identificatori di connessione, detti ID di connessione, ognuno dei quali può essere utilizzato per identificarla. Gli ID vengono selezionati in modo indipendente dagli endpoint. La funzione principale di questi ID è garantire che le modifiche nell'indirizzamento a livelli di protocollo inferiori (UDP, IP e inferiori) non causino la consegna dei pacchetti ad un endpoint sbagliati. Sfruttando l'ID di connessione, le connessioni possono quindi migrare tra indirizzi IP e interfacce di rete in modi che TCP non potrebbe mai fare. Ad esempio, la migrazione permette ad un download in corso di passare da una connessione di rete cellulare a una Wi-Fi più veloce quando l'utente si avvicina ad una rete Wi-Fi utilizzabile e viceversa.

Subito dopo che il primo pacchetto ha inizializzato la connessione, viene inviato un frame crittografico che inizia a configurare l'handshake per il livello sicuro. Il livello di sicurezza utilizzato è TLSv1.3. Non è possibile rinunciare ad utilizzare TLS in una connessione QUIC. Il protocollo è progettato per essere difficile da manomettere dai middle-box (intermediari).

Per ridurre il tempo necessario per stabilire una nuova connessione, un client che si è precedentemente connesso a un server può memorizzare nella cache determinati parametri della connessione precedente e successivamente impostare una connessione 0-RTT (0 Round Trip Time) con il server. Ciò consente al client di inviare i dati immediatamente, senza attendere il completamente di un handshake [4].

Ora si tratterà la differenza tra il 0-RTT di TCP + TLS e il 0-RTT di QUIC. In Figura 3.2 viene mostrata una connessione TCP + TLSv1.3. Da notare che non è stata presa in considerazione la possibilità di utilizzare il TCP Fast Open ma è stato utilizzato il 0-RTT di TLSv1.3. Il TCP Fast Open (TFO) permetteva, durante una riconnessione, di saltare un round trip dell'handshake. Fino ad oggi è stato utilizzato raramente, anche se sulla carta prometteva un miglioramento in performance del 4 - 40%. Anche se il TFO probabilmente non vedrà mai la luce, tuttavia, come per molte altre tecnologie, l'implementazione originale non è ciò che conta di più ma l'idea è la parte significativa. Nel caso di TFO, il concetto di miglioramento delle prestazioni dietro esso andrà comunque a vantaggio degli utenti per anni poiché influenzerà le tecnologie future come ad esempio HTTP/3 [5] [6].

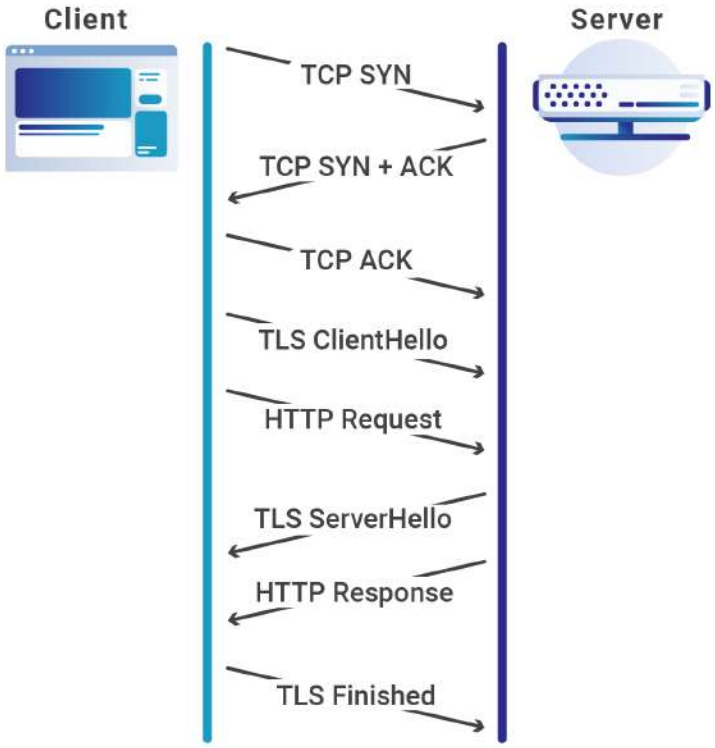

Figura 3.2: Richiesta HTTP su TCP+TLS (con 0-RTT)

Analizzando la Figura 3.2 si può notare che non è presente un vero e proprio 0-RTT come ce lo si aspetta. Questo perché il 0-RTT si riferisce solo al handshake TLS e quindi bisogna comunque aspettare l'handshake TCP prima di poter inviare la prima richiesta.

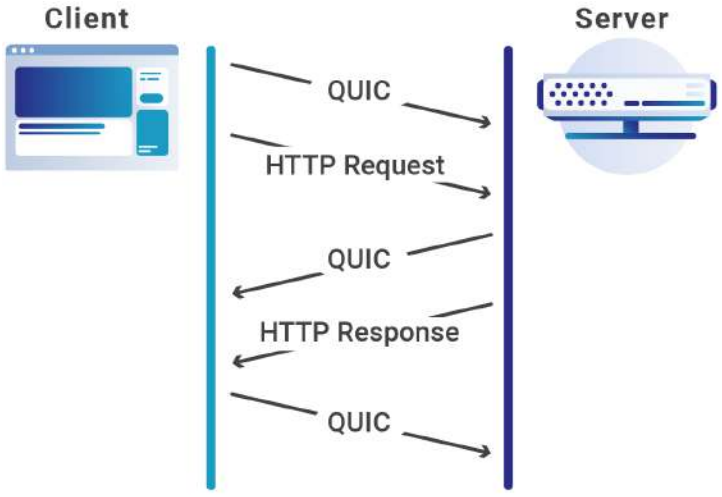

Figura 3.3: Richiesta HTTP su QUIC (con 0-RTT)

Adesso mettendo a confronto la Figura 3.2 con la Figura 3.3, si può notare come con QUIC si ha un 0-RTT effettivo. Ovviamente nel caso in cui non sia stata effettuata una connessione con il server in precedenza, ci sarà bisogno di una fase di negoziazione che quindi aggiungerà un round trip [5].

3.1.2 Stream

Gli stream in QUIC possono essere:

- Stream unidirezionali trasportano i dati in una direzione. Dal creatore dello stream al suo peer. Di stream unidirezionali ne esistono per ora di due tipi come viene discusso più avanti in questa sezione
- Stream bidirezionali consentono di inviare i dati in entrambe le direzioni. Questi stream vengono anche chiamati Request Stream

Entrambi i tipi di stream possono essere creati da entrambi gli endpoint, possono inviare contemporaneamente dati interlacciati con altri stream e possono essere annullati. Per inviare dati tramite una connessione QUIC, vengono utilizzati uno o più stream.

Su ogni stream viene effettuato un flow control, consentendo ad un endpoint di limitare l'impegno di memoria e di applicare la back pressure (contropressione). Anche sulla creazione degli stream viene applicato il flow control, permettendo ad ogni peer di dicharare il massimo numero di stream che è disposto ad accettare.

Gli stream vengono identificati da un numero intero senza segno a 62 bit, denominato stream ID. I due bit meno significativi vengono utilizzati per identificare il tipo di stream (unidirezionale o bidirezionale) e il creatore dello stream. Il bit meno significativo (`0x1`) dello stream ID identifica il creatore dello stream. I client avviano gli stream con numero pari (bit meno significativo impostato su `0`); i server avviano gli stream con numero dispari (bit impostato su `1`). Il secondo bit meno significativo (`0x2`) dello stream ID distingue tra stream unidirezionali e bidirezionali. Gli stream unidirezionali hanno sempre il bit impostato a `1`, mentre quelli bidirezionali hanno il bit impostato a `0`.

QUIC consente ad un numero arbitrario di stream di funzionare contemporaneamente. Un endpoint limita il numero di stream in entrata attivi contemporaneamente limitando la soglia specificando lo stream ID massimo (dato che gli stream non sono numeri random ma partono da `0` e si incrementano di `1` ad ogni stream nuovo). Questa impostazione è specifica per ogni endpoint a cui la si applica.

Gli endpoint utilizzano gli stream, ovviamente, per inviare e ricevere i dati. Questo dopotutto è il loro scopo finale. Gli stream sono un'astrazione ordinata dello stream di byte. Gli stream separati, tuttavia, non sono necessariamente forniti nell'ordine con cui sono partiti.

Il multiplexing degli stream ha un effetto significativo sulle prestazioni delle applicazioni se le risorse allocate agli stream hanno una corretta priorità (come ad esempio avveniva con HTTP/2). QUIC stesso non fornisce frame per lo scambio di informazioni di prioritizzazione. Si basa invece sulla ricezione delle informazioni della priorità dall'applicazione che utilizza QUIC. Ci sono state delle critiche sul modello di prioritizzazione di HTTP/2 che affermavano che fosse eccessivamente complesso e non utilizzato ed implementato da molti server HTTP/2. Per ora la prioritizzazione in HTTP/3 è stata rimossa dalla specifica principale ed è in lavorazione per una specifica separata [4].

TIPI DI STREAM UNIDIREZIONALI

Gli stream unidirezionali, in entrambe le direzioni, vengono utilizzati per una serie di scopi. Lo scopo è indicato dal tipo di stream, che viene inviato come un numero intero di lunghezza variabile all'inizio dello stream. Il formato e la struttura dei dati che seguono questo numero intero sono determinati dal tipo di stream.

```
Unidirectional Stream Header {
  Stream Type (i),
}
```

Per ora sono definiti due tipi di stream: Control stream e Push stream. Ci sarebbero altri tipi di stream ma sono riservati [7].

CONTROL STREAM

In uno stream Control (`Stream Type (i) = 0x0`) i dati sono esattamente i frame HTTP/3 dove vengono approfonditi nella sezione 3.2.3 Gli stream QUIC e HTTP/3.

Ogni lato **deve** avviare un singolo stream di controllo all'inizio della connessione e inviare il proprio frame SETTINGS come primo frame dello stream.

Poiché il contenuto dello stream di controllo viene utilizzato per gestire il comportamento di altri stream, gli endpoint **dovrebbero** fornire sufficiente credito per il flow control al fine di impedire che lo stream di controllo del peer venga bloccato.

Viene utilizzata una coppia di stream unidirezionali anziché un singolo stream bidirezionale. Ciò consente a entrambi i peer di inviare i dati non appena sono in grado. A seconda che 0-RTT sia disponibile sulla connessione QUIC, il client o il server potrebbero essere in grado di inviare i dati dello stream uno prima dell'altro [7].

PUSH STREAM

Il server push è uno strumento opzionale che è stato introdotto in HTTP/2 ma, come già detto, ad oggi il suo utilizzo è raro e spesso ne risulta controproducente l'utilizzo. Comunque dato che il push stream è stato implementato e il server push è utilizzabile in HTTP/3, di seguito verrà analizzata la base che si trova in questo stream.

Uno stream Push oltre ad avere l'identificativo del tipo (`Stream Type (i) = 0x1`), nel formato è presente anche il `Push ID`. I dati rimanenti di questo stream sono i frame HTTP/3, e le risposte HTTP alla promessa di push seguite da una risposta HTTP finale [7]. Il formato dell'header push è il seguente:

```
Push Stream Header {
  Stream Type (i) = 0x01,
  Push ID (i),
}
```

3.1.3 Sicurezza

L'attuale comunicazione web è protetta nella misura del possibile da TLS, ma ciò lascia comunque una discreta quantità di metadati visibile a terzi. In particolare, tutto l'header TCP è visibile e ciò può far trasparire una quantità significativa di informazioni. Ad esempio è possibile utilizzare le informazioni negli header TCP per rilevare quale video si sta guardando su Netflix.[14] Gli header TCP possono anche essere manomessi da terze parti, mentre il client e il server che comunicano non se ne rendono conto.

La crittografia e la privacy sono fondamentali per il design di QUIC. Tutte le connessioni QUIC sono protette da manomissioni e interruzioni, e la maggior parte degli header non è nemmeno visibile a terzi. Si può notare come nella Figura 3.4 laddove la maggior parte delle specifiche del trasporto è in chiaro e solo i dati sono criptati, QUIC (Figura 3.5) cripta quasi tutto ed applica protezione dell'integrità [8].

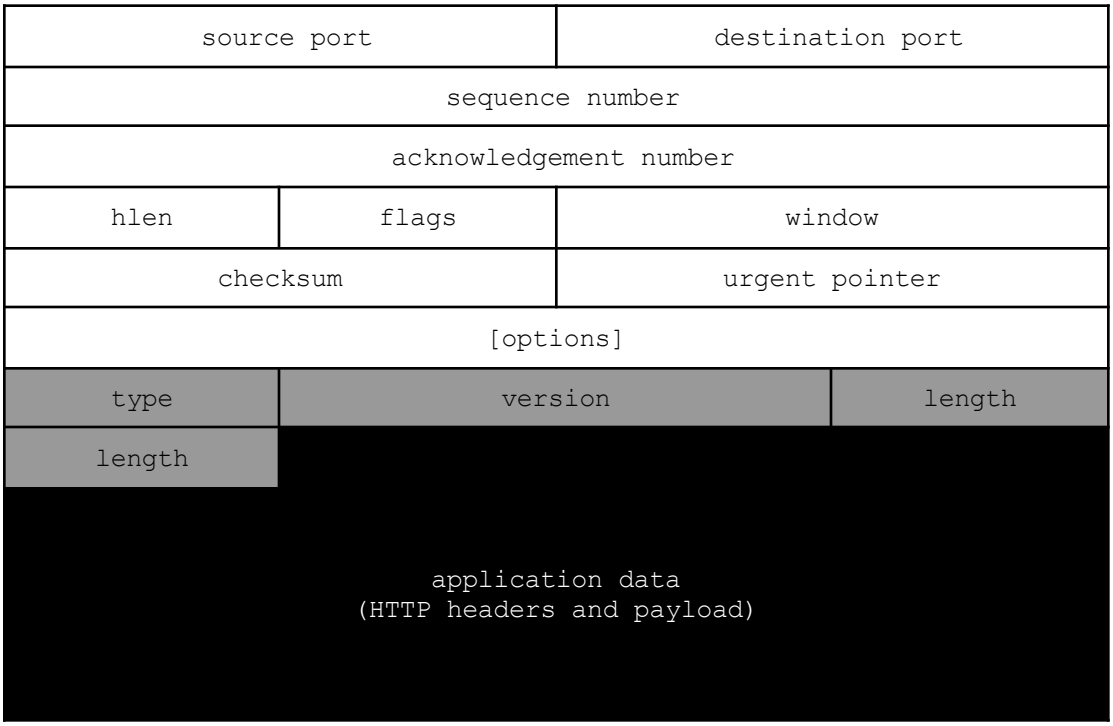

Figura 3.4: HTTP con TLS/TCP

[14] https://dl.acm.org/doi/10.1145/3029806.3029821

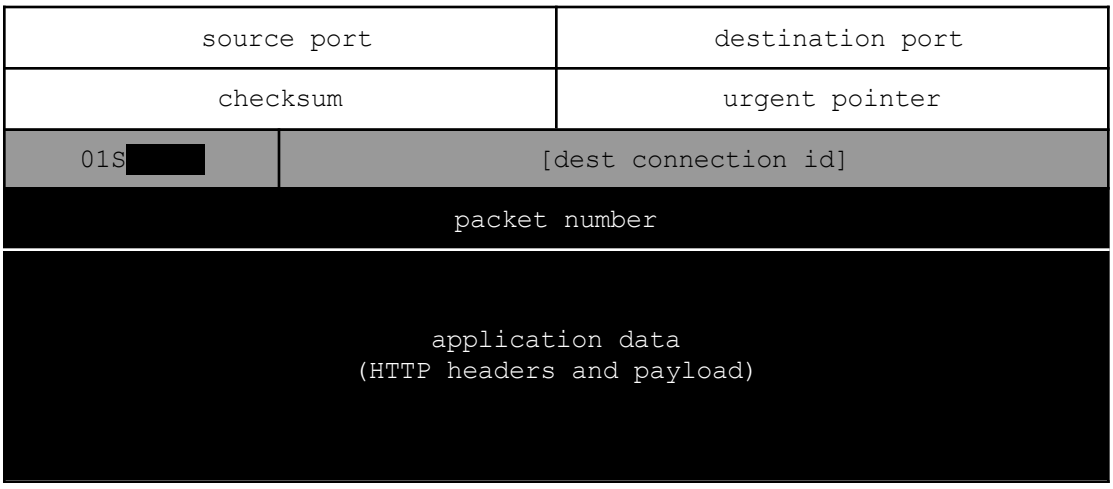

Figura 3.5: HTTP con QUIC

3.1.4 Spin Bit

Una delle discussioni più lunghe all'interno del gruppo di lavoro di QUIC riguardava un singolo bit: lo Spin Bit. I sostenitori di questo bit insistono sul fatto che è necessario che gli operatori e le persone sul percorso tra due endpoint QUIC siano in grado di misurare la latenza. Chi è contro sostiene che potrebbe essere un problema data la potenziale fuga di informazioni. Ad oggi è presente come strumento opzionale nella bozza di QUIC.[15]

Entrambi gli endpoint, il client e il server, mantengono un valore di spin, `0` o `1`, per ciascuna connessione QUIC e impostano il bit di spin sui pacchetti che vengono inviati e questo bit rimane invariato per tutta la durata del round trip. In breve il bit cambia ad ogni round trip. L'effetto è quindi un impulso di uno e zero, in quel campo, che gli osservatori possono monitorare [4].

3.1.5 User Space

Ritornando alla Figura 3.1, bisogna specificare che QUIC rientra nello user space (spazio utente). L'implementazione di un protocollo di trasporto nello spazio utente aiuta a consentire una rapida iterazione del protocollo, poiché è relativamente facile

[15] https://tools.ietf.org/html/draft-ietf-quic-transport-34

far evolvere il protocollo senza la necessità che il client e il server aggiornino il kernel del loro sistema operativo per distribuire nuove versioni di QUIC.

Un effetto ovvio dell'implementazione di un nuovo protocollo di trasporto nello spazio utente è che possiamo aspettarci di vedere molte implementazioni indipendenti [4].

3.2 HTTP/3

In maniera simile a quanto successe per HTTP/2, introdotto per trasportare HTTP in binario, HTTP/3 introduce una nuovo modo per inviare HTTP sulla rete. HTTP mantiene gli stessi paradigmi e concetti che già aveva in precedenza. Ci sono header e body, con richieste e risposte. Ci sono anche cookie e cache. Ciò che cambia in sostanza in HTTP/3 è il modo in cui i bit vengono inviati da una parte all'altra della comunicazione [4].

3.2.1 Schema e prima connessione

HTTP/3 verrà eseguito utilizzando gli URL `HTTPS://`. Il mondo è pieno di questi URL ed è stato ritenuto impraticabile e addirittura irragionevole introdurre un altro schema URL per HTTP/3. Proprio come era successo per HTTP/2.

La complessità aggiuntiva nella situazione HTTP/3 è tuttavia che, laddove HTTP/2 era un modo completamente nuovo di trasportare HTTP via cavo, era comunque ancora basato su TLS e TCP come lo era HTTP/1. Il fatto che HTTP/3 venga eseguito su QUIC cambia un po' le cose in alcuni aspetti importanti.

Gli URL `HTTP://` detti legacy (vecchi, obsoleti), che sono "clear-text", verranno lasciati così come sono e man mano che si procede in un futuro con trasferimenti più sicuri, probabilmente verranno utilizzati sempre meno frequentemente. Le richieste a tali URL semplicemente non riceveranno alcun upgrade a HTTP/3.

La prima connessione ad una risorsa o host non ancora visitato tramite un URL `HTTPS://` sarà probabilmente stabilita via TCP (eventualmente in parallelo con un tentativo di connessione QUIC). L'host potrebbe essere un legacy server che non

supporta QUIC, o ci potrebbe essere un middle box in mezzo alla connessione che ostacola il successo di una connessione QUIC.

Un client e server moderni probabilmente negozierebbero HTTP/2 al primo handshake. Quando viene stabilita la connessione e il server risponde ad una richiesta HTTP effettuata dal client, il server può comunicare al client la disponibilità e preferenza per HTTP/3 [4].

3.2.2 Bootstrap via Alternative Service

L'header alternate service (servizio alternativo) e il frame HTTP/2 `ALT-SVC` corrispondente, ovviamente non sono stati creati specificatamente per QUIC o HTTP/3. Fanno parte di un meccanismo già progettato e creato (2.3.4 HTTP Alternative Service) che permette ad un server di dire a un client: "Guarda, eseguo lo stesso servizio su QUESTO HOST usando QUESTO PROTOCOLLO su QUESTA PORTA".[16]

Un client che riceve una risposta di questo genere viene avvertito che se lo supporta e vuole connettersi a quell'altro host in parallelo e in background, può farlo utilizzando il protocollo specificato (in questo caso HTTP/3), e se ha successo, passare a quest'ultimo invece di continuare la connessione stabilita all'inizio.

Se la connessione iniziale utilizza HTTP/2 o anche HTTP/1, il server può rispondere e dire al client che può riconnettersi e provare HTTP/3. Questa connessione può avvenire sullo stesso host o su un altro che sa come servire il contenuto sul protocollo prescelto. L'informazione fornita all'interno dell'header `Alt-Svc` è dotata di una data di scadenza (ma, max-age), che permette la redirezione dei client verso host alternativi entro un certo lasso di temporale.

Un server HTTP include nella propria risposta un header di tipo **`Alt-Svc:`**

```
Alt-Svc: h3=":50781"
```

Questo indica che HTTP/3 è disponibile via UDP sulla porta `50781`, sullo stesso host utilizzato in prima istanza. Un client può a quel punto tentare ad instaurare una connessione QUIC con la destinazione ricevuta e in caso di successo

[16] https://tools.ietf.org/html/rfc7838

continuerebbe a comunicare con l'origine attraverso il nuovo canale invece della versione HTTP utilizzata precedentemente [4].

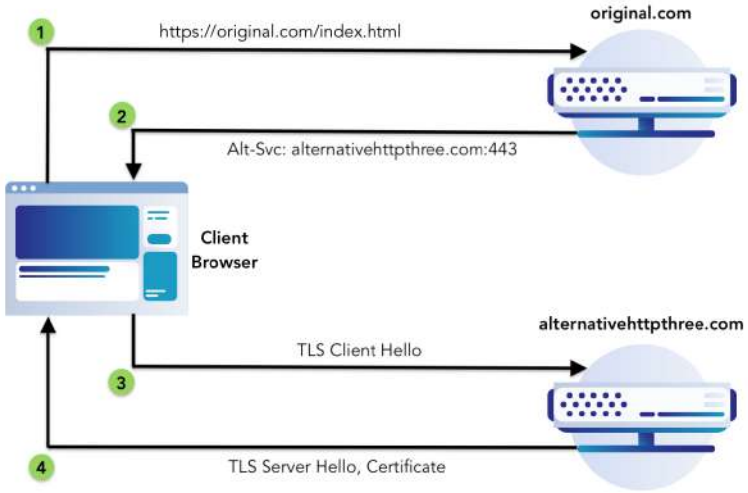

Figura 3.6: Esempio del funzionamento dell'Alternative Service

3.2.3 Gli stream QUIC e HTTP/3

I frame HTTP vengono trasportati sugli stream QUIC. Come già detto, HTTP/3 definisce tre tipi di stream: stream control, stream request e stream push. La Tabella 3.1 fornisce una panoramica sui tipi di frame e su quali stream possono essere utilizzati.

Tabella 3.1: Frame HTTP/3 e panoramica sui tipi di stream

Frame	Control	Request	Push
DATA (0x0)	No	Si	Si
HEADER (0x1)	No	Si	Si
CANCEL_PUSH (0x3)	Si	No	No
SETTINGS (0x4)	Si	No	No
PUSH_PROMISE (0X5)	No	Si	No
GOAWAY (0X7)	Si	No	No
MAX_PUSH_ID (0Xd)	Si	No	No
Reserved	Si	Si	Si

Il frame `SETTINGS` può essere utilizzato solo come primo frame di un stream di controllo. Notare che a differenza dei frame QUIC, i frame HTTP/3 possono estendersi su più pacchetti.

HTTP/2 è stato costretto a concepire il proprio schema di stream e multiplexing basandosi su TCP, mentre HTTP/3 è studiato per lavorare con QUIC e quindi trae massimo vantaggio dall'utilizzo degli stream. Le richieste HTTP effettuate con HTTP/3 usano un set specifico di stream.

Dialogare in HTTP/3 implica l'attivazione di uno stream QUIC e l'invio di una serie di frame all'altro capo della connessione. Al momento esistono una serie abbastanza limitata di frame predefiniti in HTTP/3, i più rilevanti sono senza dubbio:

- `HEADERS`, si occupano di comprimere e spedire gli HTTP header
- `DATA`, tratta l'invio di dati binari
- `GOAWAY`, si occupa della chiusura della connessione [4] [7]

FORMATO DEI FRAME

Tutti i frame hanno il seguente formato:

```
HTTP/3 Frame Format {
  Type (i),
  Length (i),
  Frame Payload (..),
}
```

Il frame deve avere i seguenti campi:

- `Type` — Numero intero a lunghezza variabile che identifica il tipo di frame
- `Length` — Numero intero a lunghezza variabile che descrive la lunghezza di byte del campo `Frame Payload`
- `Frame Payload` — Il carico utile, la cui semantica è determinata dal campo `Type` [7]

DATA FRAME

I frame `DATA` (`Type (i) = 0x0`) trasmettono sequenze arbitrarie di byte di lunghezza variabile associate alla richiesta HTTP o al contenuto della risposta. I frame `DATA` **devono** essere associati a una richiesta o risposta HTTP. Il formato del frame `DATA` è il seguente [7]:

```
DATA Frame {
  Type (i) = 0x0,
  Length (i),
  Data (..),
}
```

HEADERS FRAME

Il frame `HEADERS` (`Type (i) = 0x1`) viene usato per trasportare una sezione di campo HTTP, codificata utilizzando QPACK.

```
HEADERS Frame {
  Type (i) = 0x1,
  Length (i),
  Encoded Field Section (..),
}
```

I frame `HEADERS` possono essere inviati solo su stream request o push [7].

GOAWAY FRAME

Il frame `GOAWAY` (`Type (i) = 0x7`) viene utilizzato per avviare l'arresto di una connessione HTTP/3 da entrambi gli endpoint. `GOAWAY` consente a un endpoint di interrompere l'accettazione di nuove richieste o push e nel mentre completa l'elaborazione di richieste e push ricevuti in precedenza. Ciò consente azioni amministrative, come la manutenzione del server. `GOAWAY` da solo non chiude la connessione.

```
GOAWAY Frame {
  Type (i) = 0x7,
  Length (i),
  Stream ID/Push ID (..),
}
```

Il frame GOAWAY viene sempre inviato nello stream di controllo. Nella direzione da server a client, il frame trasporta uno Stream ID QUIC per uno stream bidirezionale, avviato dal client, codificato come un numero intero di lunghezza variabile.

Nella direzione da client a server, il frame trasporta un Push ID codificato come un numero intero di lunghezza variabile.

Il frame si applica all'intera connessione, non a uno stream specifico [7].

CONFRONTO DEI TIPI DI FRAME TRA HTTP/2 E HTTP/3

- DATA(0X0): Il padding non è definito nei frame DATA di HTTP/3
- HEADERS(0X1): I campi PRIORITY del frame HEADERS non sono definiti in HTTP/3
- PRIORITY(0X2): HTTP/3 non fornisce un mezzo per segnalare la priorità
- RST_STREAM(0X3): Il frame RST_STREAM non esiste in HTTP/3, dal momento che QUIC fornisce una gestione del ciclo di vita degli stream. Lo stesso codice (0X3) viene utilizzato per il CANCEL_PUSH
- SETTINGS(0X4): Il frame SETTINGS viene inviato solo all'inizio della connessione
- PUSH_PROMISE(0X5): Il frame PUSH_PROMISE non si riferisce ad uno stream. Invece lo stream Push fa riferimento al frame PUSH_PROMISE utilizzando un Push ID
- PING(0X6): Il frame PING non esiste in HTTP/3 dato che questo frame è già presente in QUIC
- GOAWAY(0X7): Il frame GOAWAY non contiene un codice di errore e in HTTP/3 trasporta, in direzione client verso il server, un Push ID invece di uno stream ID
- WINDOW_UPDATE(0X8): Il frame WINDOW_UPDATE non esiste in HTTP/3 dato che QUIC fornisce il flow control

- `CONTINUATION(0X9)`: Il frame `CONTINUATION` non esiste in HTTP/3 e per compensare la mancanza sono permessi frame `HEADERS`/`PUSH_PROMISE` più grandi [7]

RICHIESTA HTTP

Il client invia la propria richiesta HTTP su uno stream Request, che è uno stream QUIC bidirezionale iniziato dal client. Un client **deve** inviare solo una singola richiesta su un determinato stream. Un server invia zero o più risposte HTTP temporanee sullo stesso stream Request, seguite da una singola risposta HTTP finale. Un messaggio HTTP (richiesta o risposta) è composto da:

- L'header, inviato come un singolo frame `HEADERS`
- Il contenuto (opzionale), inviato come una serie di frame `DATA`
- Il trailer[17] (opzionale), inviato come un singolo frame `HEADERS`

Uno scambio di richieste/risposte HTTP consuma completamente uno stream QUIC bidirezionale avviato dal client. Dopo aver inviato una richiesta, un client **deve** chiudere lo stream per l'invio. A meno che non si utilizzi il metodo CONNECT, i client **non devono** fare in modo che la chiusura dello stream dipenda dalla ricezione della risposta alla loro richiesta. Dopo aver inviato una risposta finale, il server **deve** chiudere lo stream per l'invio. A questo punto lo stream QUIC è completamente chiuso.

Dato che alcuni messaggi sono di grandi dimensioni o senza limiti, gli endpoint **dovrebbero** iniziare a elaborare i messaggi HTTP parziali una volta che ne hanno ricevuto un numero sufficiente.

Un server può inviare una risposta completa prima che il client invii la richiesta per intero se la risposta non dipende da alcuna parte della richiesta che non è stata inviata e ricevuta. Quando il server non ha bisogno di ricevere il resto della richiesta, **potrebbe** interrompere la lettura dello stream di richiesta, inviare una risposta

[17] I campi inviati/ricevuti dopo che la sezione dell'header è terminata si chiamano Trailer. In breve servono per fornire controlli di integrità dei messaggi, firme digitali, metriche di consegna o informazioni sullo stato post-elaborazione.

completa e chiudere in modo pulito la parte di invio dello stream. I client **non devono** scartare le risposte a seguito della chiusura improvvisa della loro richiesta, sebbene i client possano sempre scartare le risposte a loro discrezione o per altri motivi. Se il server invia una risposta parziale o completa ma non interrompe la lettura della richiesta, i client **dovrebbero** continuare a inviare il corpo della richiesta e chiudere lo stream normalmente [7].

HEADER QPACK

I frame `HEADERS` contengono gli header HTTP compressi con l'algoritmo QPACK. QPACK risulta simile al vecchio HPACK utilizzato in HTTP/2, benchè modificato per consegnare i differenti stream in modo disordinato (out-of-order).

QPACK si avvale di due stream QUIC unidirezionali fra i due endpoint. Tali stream sono utilizzati per trasportare la tabella dinamica di compressione nelle due direzioni opposte [4].

3.2.4 Confronto con HTTP/2

HTTP/3 è stato concepito per QUIC, che è un protocollo di trasporto che gestisce i propri stream in modo autonomo. Invece HTTP/2, che è stato concepito su TCP, gestisce gli stream nello strato HTTP.

Entrambi i protocollo offrono ai client delle funzionalità praticamente identiche:

- Offrono il supporto al server push
- Hanno la compressione degli header (QPACK e HPACK sono simili nel design)
- Offrono il multiplexing su una singola connessione usando gli stream

Le differenze tra i due si trovano nei dettagli e principalmente queste differenze sono dovute all'utilizzo di HTTP/3 su QUIC:

- HTTP/3 funziona meglio con gli "early data" (dati anticipati) grazie all'handshake a 0-RTT, mentre TCP Fast Open e TLS spesso incontrano problemi inviando comunque meno dati
- HTTP/3 ha handshake più veloci grazie a QUIC rispetto a TCP + TLS

- Non esiste una versione non sicura o non criptata di HTTP/3. HTTP/2 può essere implementato e utilizzato senza HTTPS (anche se è raro trovare un caso di questo tipo)
- HTTP/2 può essere negoziato direttamente nell'handshake TLS con l'estensione ALPN (2.3.1 HTTPS Negotiation), mentre HTTP/3 essendo posizionato sopra QUIC, ha bisogno di una risposta header **`Alt-Svc:`** per informare il client del possibile utilizzo di HTTP/3
- HTTP/3 non ha prioritizzazione. L'approccio alla prioritizzazione di HTTP/2 è stato ritenuto fin troppo complicato, o anche un vero e proprio fallimento. Di fatto si sta lavorando ad un nuovo approccio[18] [7]

3.2.5 Prestazioni

Secondo uno studio recente che comprende migliaia di siti web, utilizzando HTTP/3 ci sono dei miglioramenti in performance solo in alcuni casi per ora. Sono stati riscontrati vantaggi in scenari con latenza elevata o larghezza di banda scarsa, mentre in caso di elevata perdita di pacchetti, HTTP/3 e HTTP/2 si comportano più o meno allo stesso modo. Inoltre si è riscontrato gran diversità di prestazioni a seconda dell'infrastruttura che ospitava il sito web. In generale i siti web che traggono vantaggio da HTTP/3 sono quelli che caricano oggetti da un set limitato di domini di terze parti, limitando così il numero di connessioni emesse.

Comunque essendo che HTTP/3 è ancora in fase di standardizzazione, alcune aziende leader di Internet che hanno implementato HTTP/3 tuttavia ospitano la maggior parte degli oggetti su server HTTP/2 terzi [9].

[18] https://tools.ietf.org/html/draft-ietf-httpbis-priority-03

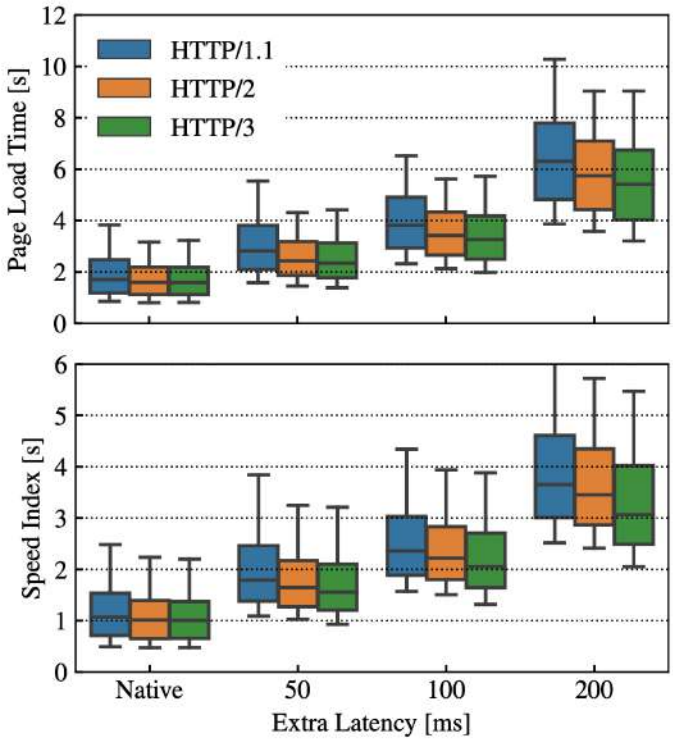

Figura 3.7: onLoad (in alto) e Speed Index (in basso) con diverse latenze

3.2.6 Critiche comuni

UDP NON FUNZIONERÀ MAI

Molte aziende, operatori e organizzatori bloccano o limitano la velocità del traffico UDP al di fuori della porta 53 (utilizzata per il DNS). Blocco dovuto all'abuso di attacchi, in particolare attacchi di amplificazioni in cui un utente malintenzionato può effettuare un enorme quantità di traffico in uscita per colpire vittime innocenti.

QUIC ha un sistema di migrazione incorporato contro gli attacchi di amplificazione richiedendo che il pacchetto iniziale debba essere di almeno 1200 byte e mediante una restrizione nel protocollo che dice che un server **non** deve inviare più di tre volte la dimensione della richiesta in risposta senza aver ricevuto un pacchetto dal client in risposta [4].

QUIC PRENDE TROPPA CPU

Per ora è una affermazione giusta se paragonata a TCP e TLS. Ma bisogna anche rendersi conto che TCP ha avuto molto più tempo per maturare e ottenere assistenza hardware oltre che software.

Con alte probabilità ci saranno gli stessi tipi di miglioramenti come ci sono stati per TCP. Già sono stati pubblicati studi[19] che trattano miglioramenti di QUIC da questo punto di vista fino a portarlo, per ora, computazionalmente efficiente come TCP [4].

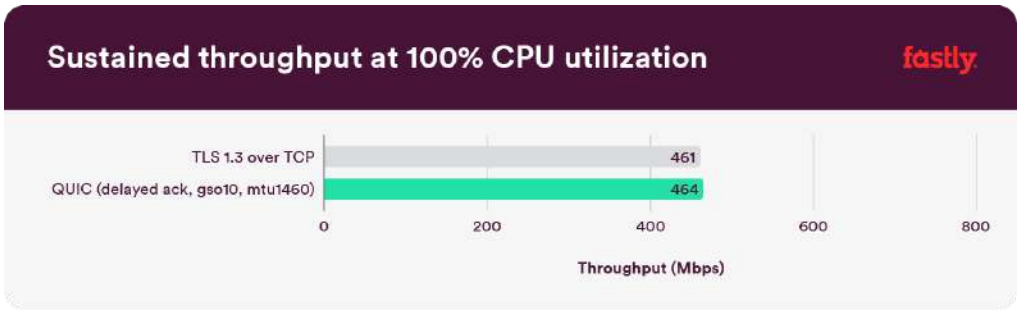

Figura 3.8: Velocità di trasmissione sostenuta al 100% di utilizzo della CPU

3.3 Come implementare HTTP/3 su un proprio sito web

In questa sezione si entrerà nel pratico vedendo passo per passo come io stesso ho implementato HTTP/3 sul mio portfolio nel modo più semplice, veloce ed intuitivo che si possa fare.

Per prima cosa bisogna scegliere una piattaforma che ospiterà il nostro sito. Nel mio caso ho scelto Vercel[20]. Su Vercel è possibile caricare un progetto esistente dal proprio GitHub (ad ogni push su GitHub, Vercel ricaricherà il progetto facendo rimanere il proprio sito sempre aggiornato). Ed ecco che con pochi click, il nostro sito è già online, visitabile tramite il dominio fornitoci da Vercel. Se adesso lo analizziamo tramite gli strumenti per sviluppatori del browser (Figura 3.8), vedremo che tutti i dati che ci sono arrivati sono arrivati con HTTP/2 (o con HTTP/1.1). Infatti la Edge

[19] https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency
[20] https://vercel.com/

Network di Vercel non supporta, per ora, HTTP/3. Quindi per poter utilizzare HTTP/3 ho utilizzato uno dei CDN (Content Delivery Network) più utilizzati al mondo[21], Cloudflare.

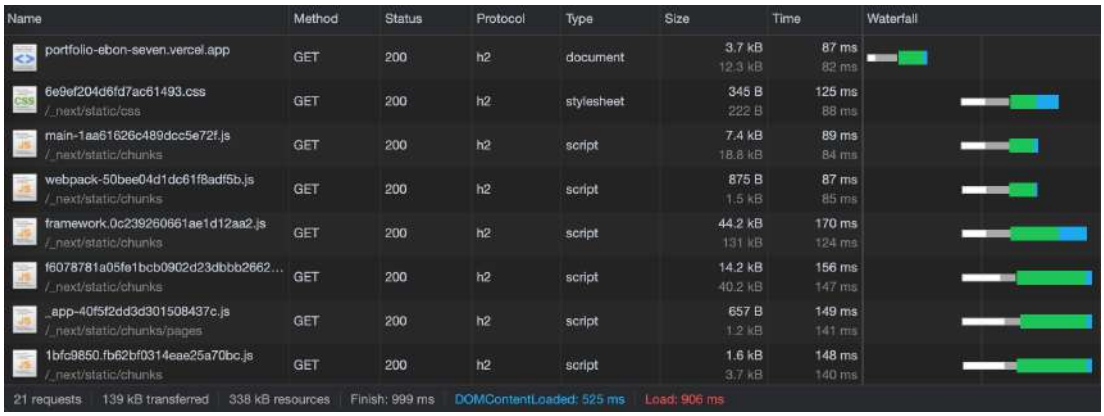

Figura 3.9: Portfolio ispezionato con gli strumenti per sviluppatori (senza Cloudflare)

Un CDN si riferisce a un gruppo di server distribuiti geograficamente che collaborano per fornire una rapida trasmissione di contenuti in Internet. In breve, parte del sito web (come file javascript e immagini) viene inviato nei vari data center del CDN in giro per il mondo in modo da avere questi file più vicini al fruitore e quindi il tutto si traduce in maggior velocità di caricamento del sito web.

Quindi per poter utilizzare Cloudflare bisognerà semplicemente registrarsi e aggiungere il proprio dominio (per poter utilizzare Cloudflare bisogna avere un proprio dominio, quello datoci da Vercel non può essere utilizzato). Seguendo le istruzioni dateci da Cloudflare ed impostando delle preferenze come ad esempio come vogliamo proteggere il nostro sito, si arriverà a dover cambiare i nameserver del nostro dominio. Una volta cambiati, entro un massimo di un paio d'ore sarà possibile accedere alla dashboard del nostro sito web in Cloudflare. Arrivati a questo punto il nostro sito utilizza Cloudflare come CDN!

A questo punto l'ultimo passaggio è quello di attivare HTTP/3 dalle impostazioni. Non è attivo di default dato che HTTP/3, come già detto, è ancora in fase di bozza e non è ancora stato standardizzato. Per attivarlo bisogna semplicemente premere sul toggle button nella tab Network come si può vedere nella Figura 3.10.

[21] https://blog.intricately.com/2020-state-of-the-cdn-industry-trends-market-share-customer-size

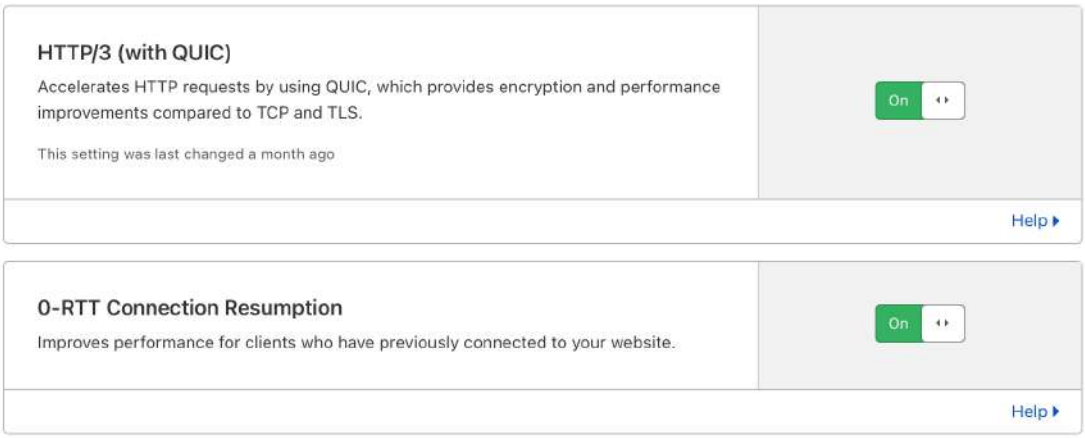

Figura 3.10: Attivazione HTTP/3 e 0-RTT in Cloudflare

Nella Figura 3.10 si può vedere che sempre nella tab Network è possibile attivare il 0-RTT di cui ho parlato in 3.1.1 Connessione.

Adesso il nostro sito web "parlerà" HTTP/3! Infatti utilizzando come prima gli strumenti per sviluppatori del browser (Figura 3.11), adesso si può vedere come HTTP/3 è in uso (il 29 dopo h3 fa riferimento alla versione della bozza di HTTP/3 utilizzata).

Figura 3.11: Portfolio ispezionato con gli strumenti per sviluppatori (con Cloudflare)

Nel caso in cui il vostro sito ancora non "parla" HTTP/3 probabilmente dovrete semplicemente aggiornare la pagina o chiudere e riaprire il browser. Questo è dovuto al funzionamento dell'Alternative Service. Il client ha bisogno di "capire" che il server può "parlare" anche HTTP/3 e quindi cambiare da HTTP/2 ad HTTP/3.

Conclusioni

In questa tesi sono state analizzate le differenze tra ciò che oggi si utilizza nei livelli trasporto, sessione e applicazione, e ciò che verrà utilizzato in futuro. Perchè si, non è un'ipotesi l'utilizzo di HTTP/3 nel prossimo decennio ma una certezza.

Sebbene sia QUIC che HTTP/3 sono ancora in fase di bozza, bisogna rendersi conto che ad esempio Google già da un paio d'anni utilizza QUIC. Qualsiasi connessione a un server di Google, se il browser lo permette, viene stabilita tramite QUIC. Tutto sotto Google utilizza QUIC: YouTube, Blogger, Hangout. Per citare un altro esempio, Uber utilizza QUIC per le sue applicazione mobile. Anche per HTTP/3 vale lo stesso, ed infatti ad oggi HTTP/3 viene utilizzato da circa l'11.3% dei siti web e questa percentuale sale ogni giorno[22]. La maggior parte dei colossi di Internet ha implementato HTTP/3, anche se si trova ancora in fase di bozza!

Come già discusso, ad oggi non ci sono particolari benefici nel passaggio ad HTTP/3 e quindi anche all'utilizzo di QUIC. Ma queste nuove tecnologie sono così promettenti che non si può non rendersi conto come il web ne beneficerà grazie alle loro implementazioni una volta che verranno standardizzati (con gran probabilità prima della fine del 2021).

Ad oggi HTTP ha una storia di circa 30 anni, e dopo tutto questo tempo si sta provando un nuovo approccio al "classico" HTTP sopra TCP + TLS. Il futuro di HTTP si prospetta di sicuro un futuro più veloce.

[22] https://w3techs.com/technologies/details/ce-http3

Bibliografia

[1] Evolution of HTTP, 2020.
https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP

[2] Barry Pollard. *HTTP/2 in Action.* Manning Publications, 2019.
https://www.manning.com/books/http2-in-action?query=http.2

[3] *Intent to Remove: HTTP/2 and gQUIC server push,* 2020.
https://groups.google.com/a/chromium.org/g/blink-dev/c/K3rYLvmQUBY/m/vOWBKZGoAQAJ?pli=1

[4] *HTTP/3 explained*. https://http3-explained.haxx.se/

[5] Alessandro Ghedini. *Even faster connection establishment with QUIC 0-RTT resumption,* 2019.
https://blog.cloudflare.com/even-faster-connection-establishment-with-quic-0-rtt-resumption/

[6] Craig Andrews. *The Sad Story of TCP Fast Open,* 2019.
https://squeeze.isobar.com/2019/04/11/the-sad-story-of-tcp-fast-open/

[7] *Hypertext Transfer Protocol Version 3 (HTTP/3),* 2021.
https://tools.ietf.org/id/draft-ietf-quic-http-34.html

[8] Jana Iyengar. *Modernizing the internet with HTTP/3 and QUIC,* 2020.
https://www.fastly.com/blog/modernizing-the-internet-with-http3-and-quic

[9] Martino Trevisan, Danilo Giordano, Idilio Drago, Ali Safari Khatouni. *Measuring HTTP/3: Adoption and Performance,* 2021.
https://arxiv.org/abs/2102.12358