Esercitazione lezione 20

Durante la prima parte della lezione, vengono mostrati diverse implementazioni di web server.

🏁 00:50

HTTP - Continuo

L'HTTP/1.1 non soddisfa più

Molti utenti abbandonano la visita di un sito se il sito non è sufficientemente reattivo; un problema alla base di questo tempo di caricamento è dato dal numero di oggetti presenti all'interno della pagina. Ricordiamo come il **round trip time** sia una tecnica usata per analizzare le prestazioni di una rete, ed in questo caso stiamo comparando il tempo di caricamento di una pagina con il RTT.

Per non dimenticare: Il round trip time è la quantità di tempo che serve ad un pacchetto di dati per essere inviato a destinazione, più il tempo che esso impiega per "tornare" al mittente.

Anche la dimensione delle pagine diventa sempre più grande, dovuta anche all'aumento costante della grandezza dei singoli oggetti. Un aspetto importante che ha contribuito all'evoluzione del protocollo è il numero di oggetti. Oggi ci sono tanti vincoli nella progettazione di applicazioni, che sono importanti per avere uno score **negli indici di valutazione**, usati per assegnare la qualità ad un sito.
Alcuni di questi vincoli impongono che il numero delle interazioni debba essere minimo.

Sicuramente tutti conosciamo **css e javascript**;

- Un CSS è un foglio di stili, usato per suggerire al browser come interpretare il contenuto trasferito, dal punto di vista grafico. Ad esempio possiamo specificare fonts, colori, ecc.
  Queste informazioni sono recuperate da un file separato; infatti una pagina web moderna non è solo composta da un file HTML, ma anche da altri file, come file .css, o file che contengono codice javascript;
- uno script Javascript ci permette di eseguire del codice in ambiente web; questo codice può infatti essere direttamente inserito all'interno di una pagina web (HTML). Ovviamente possiamo prevedere file separati.

Evoluzione di HTTP

Per molti anni HTTP/1.1 ha dominato le pagine web, essendo utilizzato praticamente ovunque. Nel 2015 viene introdotto la versione 2.0, proprio perchè completamente diverso dalla versione 1. Sebbene abbiamo avuto questo grande cambiamento, la semantica dei messaggi non cambia, infatti continuiamo ad le intestazioni, con la stessa formattazione delle linee, ma cambia il modo in cui i messaggi sono rappresentati.

HTTP/1.1 ha apportato dei miglioramenti rispetto alla versione precedente corrispondenti a tutte le feature che abbiamo già analizzato utilizzando il protocollo stesso:

- Autenticazione per accedere a risorse protette
- Gestione delle connessioni persistenti (permanenti)
- Trasferimento basato su **chunk** , ovvero trasferire il body in diversi pezzi
- Caching
- Concetto di virtual host (visto nelle lezioni precedenti)

HTTP/2, non cambia la semantica, ma va a migliorare le prestazioni

- Introduce il concetto di **stream** ;
- Un altro concetto importante è il concetto di **frame** : i frames sono dei pacchetti applicativi specificati da questo protocollo, che vengono scambiati tra client e server per dar vita a richieste di risposta HTTP.
- Compressione del layer.
- Server push.

Cosa cambia tra HTTP 1.1 e 2?

Con le connessioni permanenti, una volta attivata la connessione, è possibile dialogare; tutto il dialogo relativo che riguarda la gestione di una richiesta con relativa risposta, è realizzato all'interno di quella singola connessione (che non viene chiusa).

- Una richiesta richiede una connessione
- Con le connessioni permanenti la connessione è aperta **solo per la prima pagina** .
- Le risposte preservano l'ordine delle richieste per ogni connessione

Con HTTP/2, invece, la connessione assume un ruolo diverso. Prevediamo infatti degli stream, che operano in modo concorrente, quindi ogni stream può trasferire dati in modo concorrente rispetto agli altri. Prima, per ogni richiesta che prevedeva più interazioni, era necessario la richiesta di una connessione. Con la versione 2 del protocollo un'unica connessione viene usata per recuperare diversi contenuti.

In questa immagine possiamo vedere le due interazioni di tipo richiesta-risposta che sono indipendenti, nel senso che possiamo eseguire prima una e poi l'altra, sono quindi **concorrenti**. Immaginiamo uno stream come un **thread**, quindi all'interno dello stream abbiamo un ordine prefissato, dove avviene prima la richiesta e poi la risposta.

Il messaggio di richiesta è composto da diversi "pezzi", che vengono chiamati **frames**: abbiamo un frame che contiene l'header ed un frame che contiene i dati.

Il messaggio di risposta a sua volta contiene un frame che contiene gli headers e **più frames contenenti i dati**. Tipicamente i frame che vengono scambiati contengono **headers e dati**. Oltre ad avere i frames headers e dati (sempre nell'immagine sopra) abbiamo anche un frame **priority**, proprio perchè i vari frames possono essere gestiti con diverse priorità, proprio come accade con i thread.

Importante: notiamo dall'immagine che seppure gli stream sono diversi, la connessione è unica! Inoltre gli stream avvengono in parallelo.

Domanda: perchè nell'ambito di uno stream prevediamo l'utilizzo di frames e spezziamo quindi in diverse parti un messaggio?
Se operassimo con dei messaggi di grandi dimensione uno stream potrebbe **monopolizzare** l'uso della connessione. L'uso dello stream serve proprio ad evitare questo problema, proprio perchè la singola risorsa (la connessione) può essere usata da più stream.
Siccome la connessione è utile, tra client e server vengono comunque scambiati dei frames; nel frame ci saranno delle informazioni di controllo che ci permettono di capire di quale stream si tratta.

I blocchi (frames) vengono lo stesso trasferiti in sequenza (perchè il canale è FIFO), ma i vari frames appartengono a **diversi** stream, in modo che si possa emulare una sorta di "esecuzione in parallelo".

Per capire a quale stream ogni frame appartiene, viene usato un **header**:

ogni frame viene quindi preceduto da un header che ci dice:

- Lunghezza del frame
- Tipo del frame (HEADER - DATI - PRIORITY)
- Flag che dipendono dal tipo
- Bit non utilizzato
- Identificatore di stream
- Carico dello stream (dati?)

Il carico (payload) del frame dipende dal tipo del frame:

Precisazione sull'intestazione: le linee che avevamo in HTTP/1.1 non sono più presenti in HTTP/2, proprio perchè non è più **basato sul testo**, ma è un protocollo **binario**.

Quindi

I messaggi sono scomposti in frammenti più piccoli chiamati frames per consentire la gestione concorrente degli stream. I contenuti che vengono scambiati tra client e server appartenenti a canali logici diversi gestiti dalla stessa connessione.

Nello stream 1 vediamo un messaggio di richiesta HTTP costituito dal solo frame **HEADERS**; il messaggio di risposta HTTP prevede invece un header ed i dat, come tipicamente accade nei messaggi di tipo GET (sprovvisto di body). I due "pezzi" in celeste sono due **frames diversi**, che messi insieme danno vito al *concetto logico* di messaggio. Capiamo quindi che un singolo messaggio può essere spezzettato in più frames.

Mentre si trasferisce sullo stream 1 questo contenuto, altri frames vengono trasferiti all'interno di **altri stream**. E' l'identificatore di stream che ci permette di identificare correttamente i vari frames ed associarli allo stream corretto.

Server Push

Nel caso in cui un browser voglia recuperare una pagina HTLM contenente delle risorse, ad esempio immagini o files, recupera prima la pagina involucro e solo dopo effettua il parsing della pagina; per ogni oggetto incapsulato nella pagina, viene effettuata una richiesta.

E' necessario che il parsing della pagina venga fatta dal browser? Il file index.html potrebbe analizzarlo, in modo da capire se questo file contiene altri oggetti incapsulati. Se spostiamo il parsing **sul server**, una volta che il server capisce cosa è contenuto all'interno del file, invia al client una **"promessa di spedizione del contenuto embedded"** del file, prima di inviare il file richiesto (index.html).

Cosa sono queste promesse? Con queste promesse inviate dal server al client, il server dice "questo file contiene delle risorse embedded, tu mi chiederai queste risorse inevitabilmente; io posso fornirtele senza che tu me le chieda". Essenzialmente il server sta dichiarando quali risposte è in grado di soddisfare senza che il client produca la relativa richiesta.

Anticipando cosa può trasferire, il client non farà richiesta! Questa promessa blocca il browser (qualora accettasse di ricevere questo contenuto dal server) e possiamo avere diverse possibilità:

1. Se il contenuto non è in cache il browser accetta, ed il server spedisca senza che il client faccia la richiesta, il contenuto che gli aveva promesso, inviando tutti i file previsti all'interno di index.html .

Il vantaggio di questo schema è sicuramente di tipo prestazionale, come mai? Sicuramente si evita l'overhead di ogni interazione, proprio perchè non dobbiamo fare una richiesta, aspettare che arrivi e che il server risponda (evitiamo il round trip time), proprio perchè il client ottiene **tutti i files come se fossero associati alla prima richiesta effettuata**.

Fine lezione 20