1 Lezione del 01-10-25

Continuiamo la discussione del protocollo HTTP.

1.0.1 Tipi di richiesta HTTP

Esistono più tipi di richiesta HTTP:

- **GET:** richiede una risorsa dal server;
- **POST:** invia informazioni al server, ad esempio per trasferire un form.
- **HEAD:** richiede solo l'intestazione o *header* della risorsa, ad esempio per controllare se ha già la versione più recente in cache;
- **PUT:** aggiorna o rimpiazza una risorsa ad un dato URL. Se non esiste, la crea;
- **DELETE:** Rimuove una risorsa a un dato URL;
- **CONNECT:** Stabilisce una connessione col server. Spesso è utilizzato per connessioni SSL (HTTPS);
- **TRACE:** Risponde con la stessa richiesta. Usata per motivi di debug, ad esempio dal programma `traceroute` visto in 2.2.2;
- **OPTIONS:** Descrive le opzioni di comunicazione per la risorsa interessata. Utile per trovare quali metodi HTTP sono supportati dal server.

Più informazioni sulle specifiche del protocollo HTTP per sviluppatori web possono poi essere trovate in `https://raw.githubusercontent.com/seggiani-luca/appunti-web/481107c15776fac537b7882b6e0becfcb88b9886/master/master.pdf`.

1.0.2 Risposte HTTP

Ad una richiesta HTTP su un server web alla porta 80 segue una **risposta**. Questa ha la seguente forma generale. Questa ha un header dalla seguente forma generale:

```
HTTP/1.1 200 OK
Date: Sun, 26 Sep 2010 20:09:20 GMT
Server: Apache/2.0.52 (CentOS)
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT
Content-Length: 2652
Keep-Alive: timeout=10, max=100
Connection: Keep-Alive
Content-Type: text/html
```

Anche queste righe sono separate da `\r\n`, caratteri di ritorno carrello e nuova linea.

La prima riga definisce la **versione** protocollo, e la risposta del server in codice (200) e testo (OK).

Esiste un sistema di codici che comunicano le varie situazioni che si possono creare alla risposta. Ad esempio, alcuni dei codici più tipici sono **200** (*OK*), **301** (*Moved Permanently*), **400** (*Bad Request*), **404** (*Not Found*) e **505** (*Version not supported*). Vediamo come i codici che iniziano da **100** riguardano *messaggi informativi*, **200** condizioni di *successo*, **300** di *redirezione*, **400** errori *client* e **500** errori *server*. Anche questo argomento viene approfondito all'URL riportato nella sezione precedente.

Seguono poi diverse coppie chiave-valore che contengono informazioni sul contenuto ottenuto, il server che l'ha fornito, ecc...

Possiamo rivolgere la nostra attenzione alle linee `Keep-Alive` e `Connection`: queste stabiliscono il tipo di connessione. Dall'esempio della scorsa lezione avevamo visto come una richiesta può esprimere la preferenza per connessioni `keep-alive` (in HTTP persistente). Qui il server ha consentito la connessione `keep-alive`, stabilendo un timeout di 10 secondi e al massimo 100 richieste HTTP accettate.

1.0.3 Meccanismo dei cookie

Avevamo detto che il protocollo HTTP è *stateless*, cioè privo di stato. Un modo per reintrodurre una qualche nozione di stato nel protocollo è adottare il meccanismo dei **cookie**.

Un cookie è una coppia chiave-valore, o semplicemente anche una sola chiave, che il server invia al client assieme ad una pagina per conservare informazione di stato. Il browser che ha intenzione di preservare lo stato potrà a questo punto annettere il cookie ad ogni richiesta successiva, permettendo al server di "ricordare" quale utente sta lanciando la richiesta.

Prevediamo quindi una nuova linea di stato nell'header delle risposte HTTP (`Set-Cookie:`) che dovrà contenere questi cookie, e una linea simile nell'header delle richieste HTTP (`Cookie:`) che vengono dal browser. A questo punto basterà mantenere un file (o volendo un database) sul browser utente che associa i cookie ad un dominio, e un database (qui obbligatoriamente, probabilmente anche massiccio) di *backend* sul Web server che associa un cookie ad ogni utente.

Con il meccanismo dei cookie il protocollo stateless dell'HTTP diventa effettivamente *stateful*, cioè capace di preservare lo stato fra più richieste (assunto che il browser sia disposto a reinviare ogni volta la stringa di cookie).

1.1 Cache Web

Per ridurre il tempo di accesso agli Web server (che possono trovarsi anche molto lontano dal browser) e ridurre il traffico sugli stessi, si può usare un sistema di *caching*, cioè redirezionare il browser verso una *cache Web* (o **server proxy**).

Un proxy duplica alcuni dei contenuti presenti sul server originale: se possiede la risorsa richiesta dall'utente, la restituisce, altrimenti contatta il server originale, se la procura e la fornisce all'utente. A questo punto mantiene la risorsa per richieste future (chiaramente gestendo la sua memoria, che è finita).

Chiaramente, i *miss* dei proxy sono più lenti di un accesso diretto al server originale, ma di contro la soluzione diventa viabile e effettivamente utile quando si riesce a raggiungere una certa quota di *hit*.

I proxy permettono a content provider meno distribuiti di distribuire contenuti più facilmente. Solitamente sono installati dagli ISP (sia privati che istituzionali), con l'effetto collaterale (positivo) della riduzione del traffico sia sui server originali che sulla linea di accesso a Internet dell'ISP stesso.

Una nota interessante è che il proxy si comporta contemporaneamente sia come **client** che come **server**: dal browser è visto come *server*, mentre dal server originale è visto come *client*.

1.1.1 GET condizionale

Potremmo chiederci come fa li server proxy ad assicurarsi di mandarci sempre la copia più aggiornata della risorsa richiesta.

La soluzione è, lato server originale, non inviare la risorsa se è gia presente in cache: il server proxy può usare la linea di richiesta `If-modified-since:` per specificare l'ultima versione che ha a disposizione, e il server può rispondere con una nuova copia (se esiste), o alternativamente con il codice **304** (*Not Modified*).

Questo mantiene chiaramente un piccolo traffico in circolazione sul link, che però è comunque più piccolo del traffico richiesto per inviare risorse complete, e quindi non si traduce in carichi significativi per la rete.

1.2 Protocollo HTTP/2

L'obiettivo degli aggiornamenti del protocollo HTTP è principalmente quello di ridurre il ritardo in richieste HTTP multi oggetto.

HTTP/1.1 ha introdotto richieste GET multiple eseguite in *pipeline* su una sola connessione TCP. Il server risponde *in-order* usando l'algoritmo **FCFS** (scheduling First Come First Served). Secondo questo algoritmo, gli oggetti più piccoli devono aspettare la trasmissione dietro gli oggetti più grandi (**HOL blocking**, *Head Of Line blocking*).

Inoltre, il meccanismo di ritrasmissione dei segmenti TCP persi può rallentare ulteriormente le trasmissioni.

Potremmo pensare di risolvere i problemi di HTTP/1.1 usando altri algoritmi di scheduling:

- Magari inviando prima gli oggetti più piccoli. Questo però ritarderebbe indefinitamente la trasmissione di oggetti più grandi, in quanto probabilmente ci sarà quasi sempre un'oggetto più piccolo;
- Usando un approccio **RR** (*Round Robin*), dove gli oggetti più grandi vengono divisi in *segmenti* più piccoli, e il server alterna fra i diversi client trasmettendo tali segmenti.

HTTP/2 ha aumentato la flessibilità lato server nell'inviare oggetti al client. In questo caso l'ordine di trasmissione degli oggetti è stabilito da codici di priorità inviati dai client, e da algoritmi più sofisticati lato server (come il RR visto prima). Inoltre, il server può inoltrare (*push*) ai client oggetti che non hanno richiesto.

In HTTP/2 restano comunque i problemi di stallo nel caso di ritrasmissioni di segmenti TCP persi: i browser sono invitati a mantenere più connessioni TCP parallele per ridurre gli stalli e aumentare il throughput complessivo.

Inoltre, non c'è sicurezza sopra il protocollo TCP.

HTTP/3 ha introdotto meccanismi di sicurezza, e controllo di errori per oggetto e congestioni su UDP. Approfondiremo questo aspetto quando parleremo del livello transport.