Azzolini Riccardo

Protocollo HTTP

1 Riassunto: reti e protocolli

Prima di considerare nello specifico il protocollo HTTP, è utile riassumere brevemente i concetti di rete e protocollo.

- Per *rete* si intende un insieme di due o più computer connessi tra loro e in grado di condividere informazioni.
- Un *protocollo* è un insieme di regole (formato dei messaggi, ecc.) che definiscono uno standard di comunicazione tra diversi computer (e diversi programmi in esecuzione su tali computer) attraverso la rete.

  Ciascun protocollo stabilisce le regole per una specifica attività di comunicazione, come ad esempio l'invio di posta elettronica, il trasferimento di file, ecc.

2 HTTP

HTTP (**HyperText Transfer Protocol**) è probabilmente il protocollo di applicazione più utilizzato su Internet, in quanto è il protocollo alla base del **World Wide Web**.

Infatti, il web è formato da numerosi server e client (web browser), che comunicano attraverso TCP/IP a livello di trasporto, e HTTP a livello applicativo. I dati comunicati sono (tipicamente) nel formato *HTML* (*HyperText Markup Language*), anch'esso standard, che permette di rappresentare *ipertesti* contenenti riferimenti (link) ad altre pagine web, le quali possono essere situate su qualunque server.

HTTP è un protocollo **stateless** (senza stato/memoria): esso funziona attraverso un meccanismo di richiesta–risposta, e il server non conserva informazioni relative alle richieste precedenti.

3 URL

Una pagina web è formata da più oggetti (file HTML, immagini, file audio, ecc.), che sono risorse rese disponibili da server potenzialmente diversi. È allora necessario disporre di un metodo per identificare queste risorse. Esso è fornito dagli **URL** (**Uniform Resource Locator**): un URL è una sequenza di caratteri che identifica univocamente l'indirizzo di una risorsa in Internet.

Più in dettaglio, un URL contiene principalmente due informazioni:

- qual è il server a cui rivolgersi;
- qual è la risorsa di interesse all'interno del server.

3.1 Struttura di un URL

Ogni URL si compone di sette parti, 5 delle quali sono opzionali (qui indicate tra parentesi angolari, `<>`):

protocollo`://<`*username*`:`*password*`@>`*nome host*`<:`*porta*`>`
`</`*percorso*`><?`*query string*`><#`*fragment identifier*`>`

- *protocollo*: indica il protocollo applicativo utilizzato.
- *username*`:`*password*`@` (opzionale): è possibile specificare le credenziali di autenticazione (ma è meglio non farlo, perché in questo modo username e password verrebbero trasmessi in chiaro sulla rete, il che è molto pericoloso).
- *nome host*: rappresenta l'indirizzo fisico del server su cui risiede la risorsa.
- *porta* (opzionale): la porta del processo server. Se omessa, viene usata la porta di default del protocollo.
- *percorso* (opzionale): la risorsa richiesta. Se omesso, il server restituisce una risorsa di default (ad esempio la home page del sito web visitato).
- *query string* (opzionale): permette di passare degli argomenti/parametri al programma server, sotto forma di coppie chiave–valore (`?par1=val1&par2=val2&...`).
- *fragment identifier* (opzionale): indica una parte all'interno della risorsa (tipicamente, una specifica sezione di una pagina web).

Un esempio concreto di URL è il seguente:

nome host — fragment identifier

`http://example.com/programmazione-distribuita/#protocolli`

protocollo — percorso

Altri esempi, con protocolli diversi, sono:

- `ftp://ftp.example.org/docs/test.txt`
- `mailto:user@example.com`
- `news:soc.culture.Singapore`
- `telnet://example.net`

4 Fasi di comunicazione HTTP

Quando un client vuole acquisire una risorsa da un server, la comunicazione avviene secondo le seguenti fasi:

1. **Connessione**: il client crea una connessione TCP-IP con il server (usando di default la porta 80, se l'URL non ne specifica una diversa).
2. **Richiesta**: il client invia al server un messaggio di richiesta.
3. **Risposta**: il server invia al client un messaggio di risposta, che può contenere la risorsa richiesta (ad esempio un documento HTML) oppure un messaggio d'errore.
4. **Chiusura della connessione**: subito dopo aver spedito la risposta, il server si sconnette. Comunque, anche il client può interrompere la connessione in ogni momento: in tal caso, il server non registrerà alcuna condizione di errore.

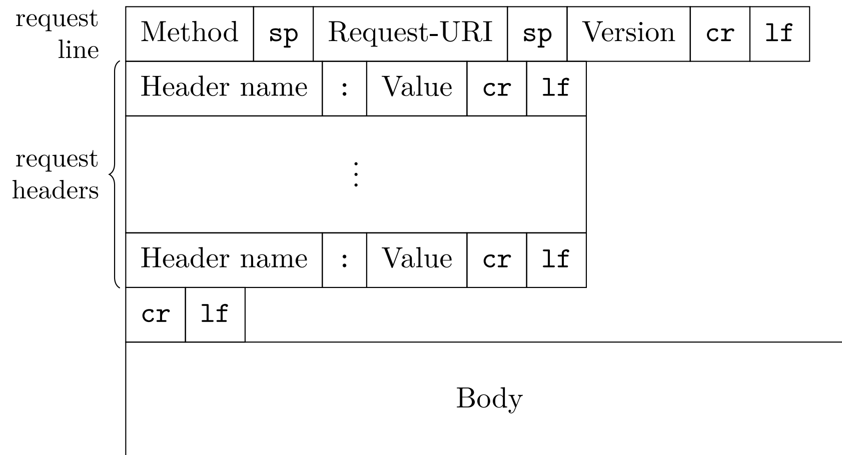
5 Formato dei messaggi

I messaggi di richiesta e risposta HTTP sono messaggi di testo. Ciascuno di essi è composto da un'**intestazione** (**header**) e da un **corpo** (**body**), che è facoltativo; le due parti sono separate da una riga vuota.

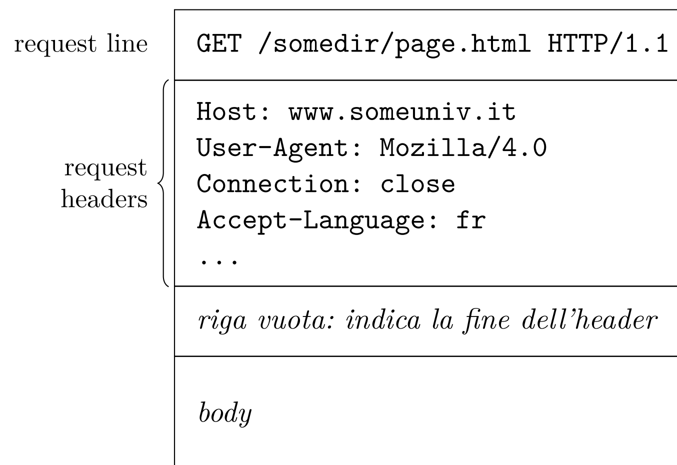
5.1 Messaggio di richiesta

In un messaggio di richiesta (*request message*), l'header è a sua volta suddiviso in due parti:

- la prima riga, chiamata **request line**;
- le righe successive, contenenti i **request headers**.



Un esempio di messaggio di richiesta è il seguente:



5.1.1 Request line

La request line è formata da tre campi, separati da spazi:

- Il **metodo**, che indica quale operazione deve essere eseguita sulla risorsa. Questo campo è case sensitive: il nome del metodo deve essere scritto in maiuscolo.

  I principali metodi sono `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `OPTIONS`, `TRACE` e `CONNECT`. Tra questi, il metodo `GET`, usato per richiedere una pagina, deve essere supportato da qualsiasi server, mentre gli altri sono opzionali: se un server riceve un metodo che non supporta, restituisce un errore.

- L'URL, o `Request-URI`, può avere diversi tipi di contenuti:

```
Request-URI = absolute-path | absolute-URI | authority | "*"
```

  - `absolute-path` è la forma più comune, che specifica il percorso (path) della risorsa sul server. Se è vuoto, assume per default il valore `/`, corrispondente alla root del server.
  - `absolute-URI` è usato quando la richiesta è stata fatta a un proxy.
  - L'asterisco, `*`, indica che la richiesta non deve essere applicata a una particolare risorsa, bensì al server stesso. Esso viene usato con il metodo `OPTIONS`.

- La **versione** del protocollo HTTP usata per la comunicazione, come ad esempio `HTTP/1.1`.

Alcuni esempi di request line sono:

```
GET /test.html HTTP/1.1
HEAD /query.html HTTP/1.0
POST /index.html HTTP/1.1
OPTIONS * HTTP/1.1
```

5.1.2 Request headers

I request headers sono delle coppie nome–valore. Il nome e il valore sono separati dai due punti, e possono essere specificati più valori, separati da virgole:

```
Request-header-name: value-1, value-2, ...
```

Alcuni esempi di header sono:

```
Host: www.xyz.com
Connection: keep-alive
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us, fr, cn
```

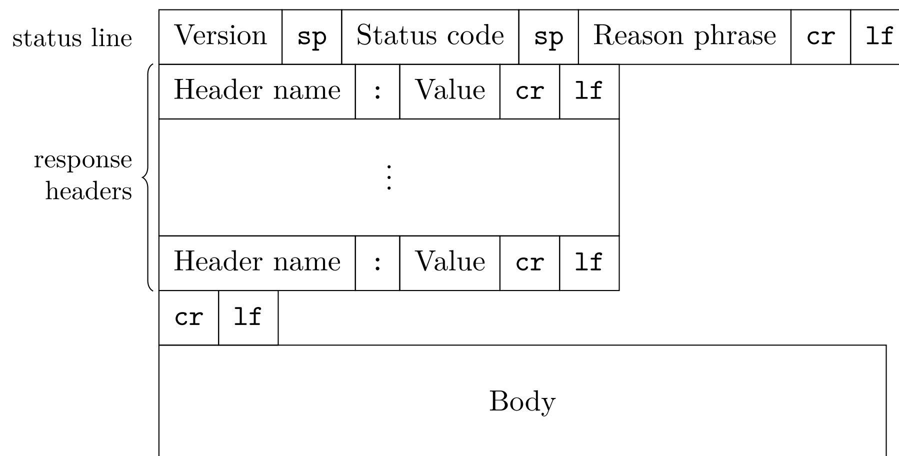
- `Host` indica l'host al quale è rivolta la richiesta;
- `Connection` se chiudere la connessione dopo l'invio della risposta (`close`), oppure tenerla aperta per richieste successive (`keep-alive`);
- `Accept` limita i tipi di oggetti che possono essere inviati nella risposta (permettendo al client di indicare quali formati è in grado di gestire).
- Analogamente, `Accept-Language` indica le lingue in cui può essere scritto il documento contenuto nella risposta.

5.2 Messaggio di risposta

Il formato del messaggio di risposta (*response message*) HTTP è simile a quello del messaggio di richiesta. In particolare, anche qui l'header è diviso in due parti:

- la prima riga, chiamata **status line**;
- le righe successive, contenenti i **response headers**.

Il corpo (che, come già detto, è separato dall'intestazione da una riga vuota) contiene i dati effettivamente richiesti dal client (tipicamente, i documenti ipertestuali in formato HTML).



5.2.1 Status line

La status line contiene tre campi, separati da spazi:

- la versione di HTTP
- lo **status code**, un codice a tre cifre che indica l'esito della richiesta;
- una breve descrizione del significato di tale codice, destinata all'uso umano.

In base alla prima cifra dello status code, si definiscono 5 classi di risposta:

- `1xx`: *informazione* – la richiesta è stata ricevuta, ma è ancora in corso la sua elaborazione;
- `2xx`: *successo* – la richiesta è stata ricevuta, capita e accettata;
- `3xx`: *ridirezione* – il client deve compiere ulteriori azioni per completare la richiesta (ad esempio, deve fare la richiesta a un URL diverso);
- `4xx`: *client error* – errore da parte del client, cioè nella richiesta;

- `5xx`: *server error* – errore da parte del server, che per qualche motivo non è stato capace di eseguire la richiesta.

6 Metodi di richiesta

Come detto in precedenza, il protocollo HTTP definisce un insieme di metodi di richiesta. I principali sono:

- `GET`: serve per ottenere una risorsa web dal server (e perciò è il metodo più utilizzato).
- `HEAD`: analogo a `GET`, ma indica al server di inviare solo l'intestazione della risposta. Ciò è utile, ad esempio, perché l'intestazione contiene la data dell'ultima modifica dei dati, che può così essere confrontata con la data di una copia cache locale: se la cache locale non è aggornata, sarà necessario eseguire una richiesta `GET` per scaricare la versione più recente.
- `POST` e `PUT`: utilizzati per inviare dati al server, tipicamente chiedendo che vengano memorizzati.
- `DELETE`: chiede al server di cancellare dei dati.
- `TRACE`: chiede al server di restituire il messaggio ricevuto, corredato di informazioni diagnostiche.
- `OPTIONS`: chiede al server di restituire l'elenco dei metodi di richiesta che supporta.
- `CONNECT`: usato per dire a un proxy di effettuare una connessione a un altro host.

Oltre a questi, esistono vari altri metodi di estensione.

7 Esempio di client HTTP in Java

Nel seguente esempio di codice Java si usano i socket per implementare un semplice client HTTP, che esegue una richiesta e stampa la risposta ottenuta:

```java
import java.io.*;
import java.net.*;

public class HttpClient {
    public static void main(String[] args)
            throws IOException, URISyntaxException {
        String urlStr = "http://artelab.dista.uninsubria.it/people.html";
        URI uri = new URI(urlStr);
        String host = uri.getHost();
```

```
        String path = uri.getRawPath();

        try (
            Socket socket = new Socket(host, 80);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            PrintWriter out = new PrintWriter(socket.getOutputStream())
        ) {
            out.print(
                "GET " + path + " HTTP/1.1\r\n" +
                "Host: " + host + "\r\n" +
                "Connection: close\r\n\r\n"
            );
            out.flush();

            System.out.println("*** MESSAGE from " + host + ":");
            String line;
            while ((line = in.readLine()) != null) {
                System.out.println(line);
            }

            System.out.println("*** Closing...");
        }
    }
}
```

1. Si costruisce l'URL della risorsa da richiedere, poi si estraggono da esso il nome dell'host e il percorso (i due elementi che serviranno per comporre la richiesta).
2. Usando un socket, si stabilisce la connessione con il server alla porta 80 (quella di default di HTTP). Come sempre, sul socket connesso si costruiscono gli stream di ingresso e uscita.
3. Tramite lo stream di uscita, si manda al server una richiesta `GET`. Per assicurarsi che essa venga effettivamente inviata, e non rimanga invece nel buffer dello stream, si usa una chiamata `out.flush()`.
4. Il server invia la risposta: che il client legge (usando il proprio stream di input) e visualizza a schermo.
5. Si chiude il socket, terminando così l'interazione con il server.

L'esecuzione di questo programma produrrà un output simile al seguente:

```
*** MESSAGE from artelab.dista.uninsubria.it:
```

```
HTTP/1.1 200 OK
Date: Tue, 02 Jun 2020 10:30:29 GMT
Server: Apache
Last-Modified: Fri, 21 Jun 2019 07:51:52 GMT
ETag: "1fc49-102b-58bd0bc4a5600"
Accept-Ranges: bytes
Content-Length: 4139
Vary: Accept-Encoding
Connection: close
Content-Type: text/html

<!DOCTYPE html>
<html lang="en">
...
</html>
*** Closing...
```

7.1 Esempi di errori

Il client appena implementato può essere usato per mostrare alcuni dei possibili errori che il server può restituire.

- Se, nella richiesta, il metodo non è scritto correttamente (ad esempio, `get` invece di `GET`), il server restituisce un errore `501 Method Not Implemented`, e indica i metodi consentiti nell'header `Allow`:

  ```
  *** MESSAGE from artelab.dista.uninsubria.it:
  HTTP/1.1 501 Method Not Implemented
  Date: Tue, 02 Jun 2020 10:49:05 GMT
  Server: Apache
  Allow: GET,HEAD,POST,OPTIONS
  Vary: Accept-Encoding
  Content-Length: 216
  Connection: close
  Content-Type: text/html; charset=iso-8859-1

  <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
  <html><head>
  <title>501 Method Not Implemented</title>
  </head><body>
  <h1>Method Not Implemented</h1>
  <p>get to /people.html not supported.<br />
  </p>
  </body></html>
  ```

```
*** Closing...
```

- Se il path indicato nella richiesta (ad esempio `/pathsbagliato`) non corrisponde ad alcuna risorsa sul server, viene restituito un errore `404 Not Found`:

```
*** MESSAGE from artelab.dista.uninsubria.it:
HTTP/1.1 404 Not Found
Date: Tue, 02 Jun 2020 10:55:15 GMT
Server: Apache
Vary: Accept-Encoding
Content-Length: 211
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /pathsbagliato was not found on this
server.</p>
</body></html>
*** Closing...
```

- Se la richiesta non è formattata correttamente, il server risponde con un errore `400 Bad Request`:

```
*** MESSAGE from artelab.dista.uninsubria.it:
HTTP/1.1 400 Bad Request
Date: Tue, 02 Jun 2020 11:00:16 GMT
Server: Apache
Vary: Accept-Encoding
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not
understand.<br />
</p>
</body></html>
*** Closing...
```