

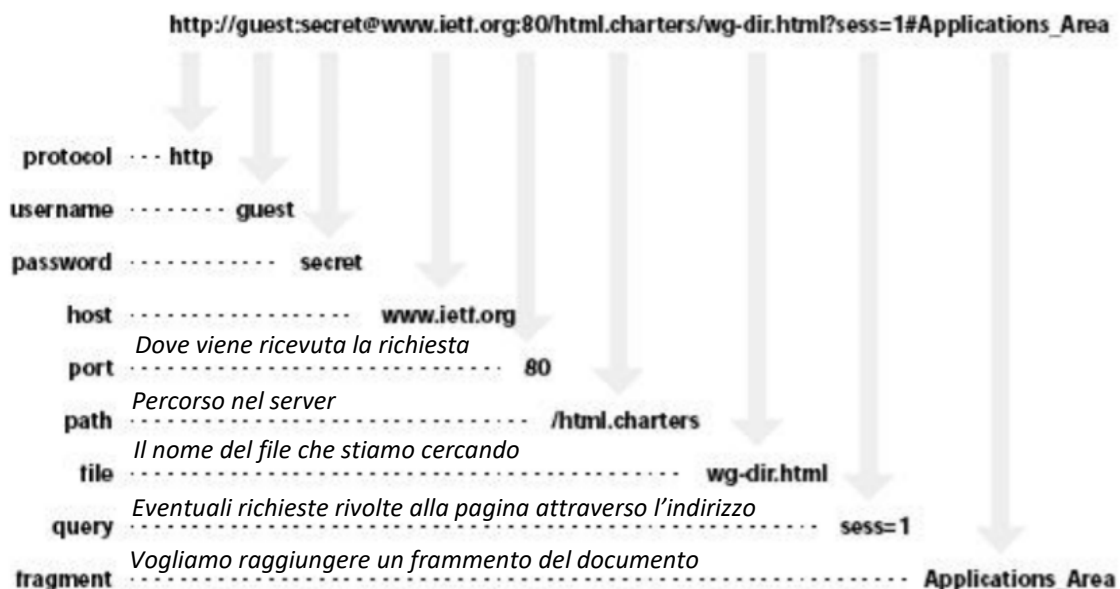
HTTP

- Fino ad ora abbiamo solo accennato qualcosa sul protocollo http. Introduciamolo adesso, tenendo conto che il grosso di questo argomento sarà affrontato a Reti informatiche il prossimo anno.
- **Modello OSI:** un modello a layer (cioè a strati). Abbiamo 7 layer:
 1. *Physical Layer:* descrivo i segnali elettrici e il cablaggio
 2. *Data Link Layer:* regola la trasmissione dei pacchetti da nodo a nodo usando gli indirizzi delle stazioni.
 3. *Network Layer:* instrada i dati a differenti reti locali o reti WAN
 4. *Transport Layer:* assicuro la trasmissione di interi documenti e/o messaggi.
 5. *Session Layer:* avvia, stoppa sessioni e mantiene l'ordine dei dati.
 6. *Presentation Layer:* codifica
 7. *Application Layer:* email, file transfer, client-server
- **Modello TCP/IP:** in parallelo abbiamo questo modello. Presenti i seguenti layer:
 3. Protocollo IP
 4. Protocollo TCP (per quello che ci riguarda solo questo)
 - 5,6,7. Protocollo HTTP (solo questo per quello che ci riguarda).



OSI MODEL		TCP / IP
7	Application Layer Type of communication: E-mail, file transfer, client/server.	FTP, Telnet, HTTP .
6	Presentation Layer Encryption, data conversion: ASCII to EBCDIC, BCD to binary, etc.	SNMP, DNS, OSPF, RIP, Ping, Traceroute
5	Session Layer Starts, stops session. Maintains order.	TCP (delivery ensured) UDP (delivery NOT ensured)
4	Transport Layer Ensures delivery of entire file or message.	IP (ICMP, IGMP, ARP, RARP)
3	Network Layer Routes data to different LANs and WANs based on network address.	
2	Data Link (MAC) Layer Transmits packets from node to node based on station address.	
1	Physical Layer Electrical signals and cabling.	

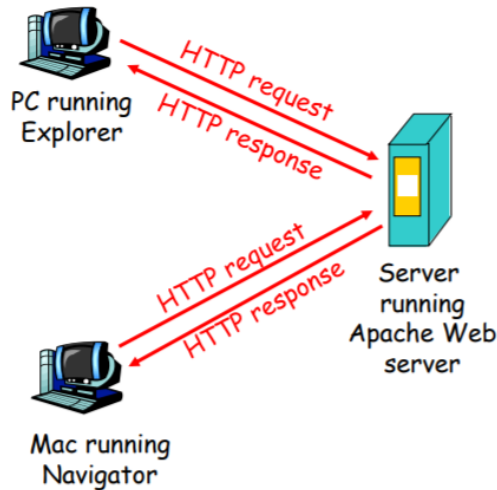
Protocollo tipicamente utilizzato nei servizi web. Ogni pagina web consiste in una serie di oggetti: HTML files, immagini, audio files... Ogni oggetto presente nella pagina web è indirizzabile attraverso un URI. Nella forma più estesa l'URI è strutturato in:



HTTP sta per *Hypertext Transfer protocol*. Questo protocollo è nato per trasmettere ipertesto: la cosa si è evoluta e adesso trasportiamo qualcosa di più di un semplice ipertesto (immagini, audio, video...).

Il protocollo HTTP si basa su un modello *client-server*:

- Il client è il browser, colui che compie richieste e riceve risposte (oggetti).
- Il server è colui che replica alle richieste del client inviando oggetti.



Il protocollo è indipendente dal tipo di macchina e browser utilizzati: fissa come devono essere strutturati i pacchetti che viaggiano da lato client a lato server e viceversa (in modo tale che qualunque dispositivo possa comprenderli).

Il **protocollo HTTP** si basa sul **protocollo TCP**:

- Il client inizia una connessione TCP (crea un *socket*, un canale) verso il server.
- Il server accetta questa connessione TCP dal client.
- I *messaggi HTTP* vengono scambiati tra client e server sfruttando questo *socket*.
- Alla fine la connessione TCP viene chiusa.

Ricordiamo che il protocollo HTTP è *stateless*: il server non mantiene alcune informazione sulle richieste passate dei client. Questo significa che se il client perde la connessione non potrà recuperare la risposta del server, quindi dovrà fare una nuova richiesta al server stesso.

La motivazione di questa proprietà è molto semplice: memorizzare per ogni client la storia passata significa gestire una grande quantità di stati che possono perdere consistenza a causa di crash di server e/o client. Gli algoritmi per gestire il mantenimento della consistenza dei dati sul server sono altamente complessi, superflui per quello che dobbiamo fare.

Le connessioni HTTP possono essere:

- **Non persistenti**, viene inviato al più un oggetto sulla connessione TCP;
- **Persistenti**, posso inviare oggetti multipli su una singola connessione TCP.

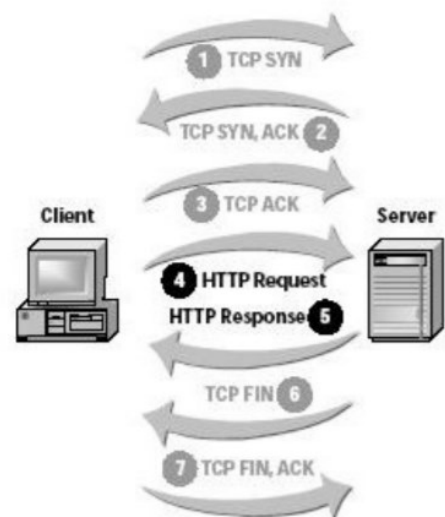
HTTP Nonpersistente

Supponiamo di voler visitare la seguente pagina

www.someSchool.edu/someDepartment/home.index

che presenta ipertesto, ma anche dieci immagini jpeg.

1. Il client HTTP inizia una connessione TCP al server HTTP sulla porta 80.
2. Il server HTTP aspetta una connessione TCP sulla porta 80. La accetta avvertendo il client.
3. Il client HTTP invia un *messaggio di richiesta* attraverso una connessione TCP.
4. Il server HTTP riceve la richiesta, formula la risposta contenente gli oggetti richiesti, invia la risposta attraverso il socket.
5. A quel punto il server HTTP chiude la connessione TCP. Il client invia un messaggio di acknowledgment sulla fine della connessione.
6. Il client HTTP riceve il *messaggio di risposta* il file HTML. Per ogni oggetto (le immagini JPEG riferite nel file HTML) si compiono nuove richieste ricominciando dal primo step.



Round-trip time

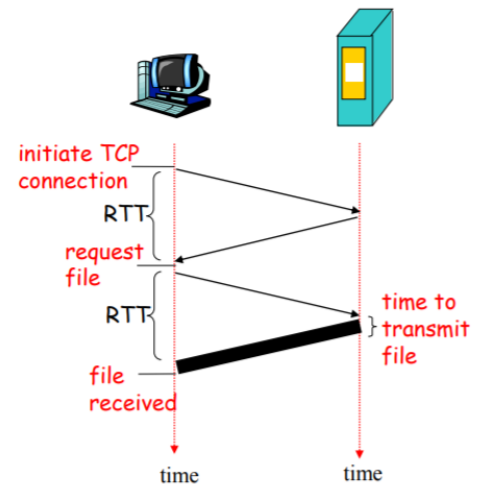
Con Round-trip time intendiamo il tempo necessario a piccolo pacchetto per viaggiare dal client al server e tornare indietro.

Analizziamo il tempo di risposta con una connessione HTTP non persistente:

- Abbiamo un RTT per aprire la connessione TCP
- Un RTT per una richiesta HTTP (per ricevere i primi bytes di una risposta)
- Tempo di trasmissione del file

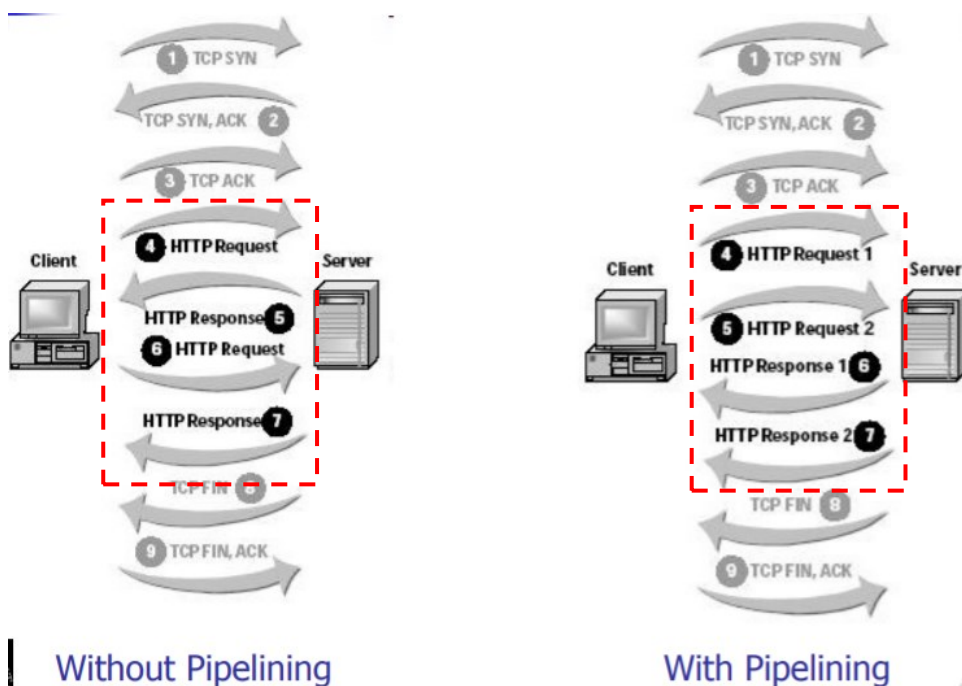
Complessivamente abbiamo come tempo totale

$$\text{Tempo totale} = 2 \cdot \text{RTT} + \text{Tempo di trasmissione}$$



Confronto tra HTTP non persistente ed HTTP persistente

- Nel caso di una connessione persistente il server http lascia la connessione aperta in attesa di ricevere ulteriori richieste dallo stesso client. Evitiamo l'apertura della connessione TCP per l'invio di ogni file da lato server a lato client.
- In HTTP non persistente abbiamo bisogno di 2 RTT per oggetto. Dobbiamo gestire, inoltre, buffer TCP e variabili TCP per ogni connessione TCP. I browser certe volte aprono connessioni TCP in parallelo.
- **Abbiamo due tipi di connessioni http persistenti:** una versione *pipelining* e una *senza pipelining*.
 - o Nella versione *pipelining* il client invia richieste non appena incontra un riferimento a un oggetto. Basta un solo RTT per tutti gli oggetti riferiti (inviando le richieste subito senza attendere risposte dal server per le precedenti).
 - o Nella versione *senza pipelining* il client invia una nuova richiesta solo dopo aver ricevuto risposte per le precedenti. Il client deve attendere un RTT per ogni richiesta. Il server rimane in attesa di eventuali richieste dopo aver inviato una risposta.



Tipi di messaggi di richiesta

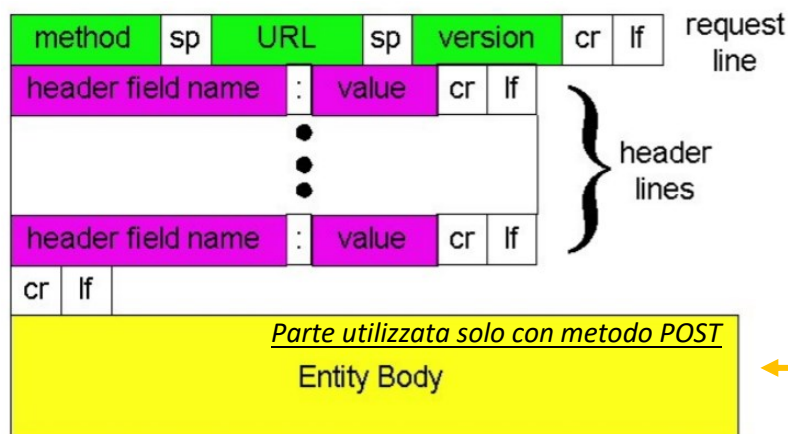
- I messaggi sono di due tipi: *request* e *response*.

Request message

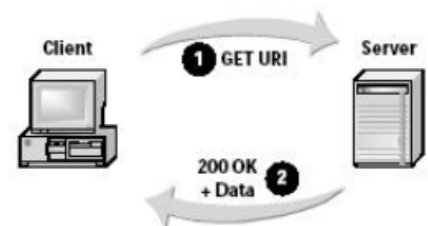
Un messaggio di richiesta è scritto in formato ASCII (in *human-readable format*) e presenta una certa struttura

request line
(HTTP commands) → GET /somedir/page.html HTTP/1.1
header lines
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
Carriage return line feed → (extra carriage return, line feed)
indicates end of message

- La prima riga è la linea di richiesta (con comandi http, il metodo, la risorsa richiesta e la versione del protocollo http adottata)
- Le righe successive consistono nel contenuto della richiesta
 - o **Host:** server a cui rivolgiamo la richiesta
 - o **User-agent:** il browser che sta facendo la richiesta al server.
 - o **Connection:** il browser riferisce al server che non vuole stabilire una connessione persistente (*close* in contrapposizione a *Keep-Alive*)
 - o **Accept-language:** si indica il linguaggio preferito dal client.
- Infine abbiamo il carattere ritorno carrello line feed (che indica la fine del messaggio).
- La specifica del metodo può essere *GET* o *POST* (già conosciamo la differenza, nel metodo POST l'input è caricato nel body della richiesta, mentre nel metodo GET l'input viene posto nell'URL)

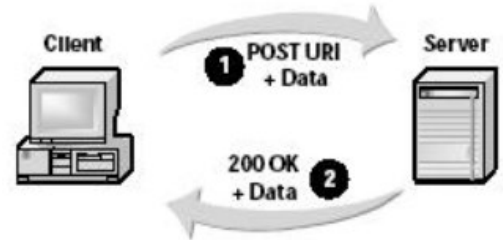


- **HTTP** è passato dalla versione 1.0 alla 1.1. L'evoluzione ha riguardato soprattutto il numero di metodi disponibili:
 - o Nella versione iniziale erano disponibili soltanto i metodi *GET*, *POST*, *HEAD*
 - o Nella versione attuale oltre ai metodi citati abbiamo *PUT*, *DELETE*, *TRACE*.
- **Metodo GET:**
 - o Trascrive le richieste da lato client a lato server utilizzando l'URL
 - o Si possono utilizzare solo caratteri ASCII
 - o Il browser e il proxy possono memorizzare in cache le risposte (attenzione, la cosa può dare problemi quando facciamo prove).



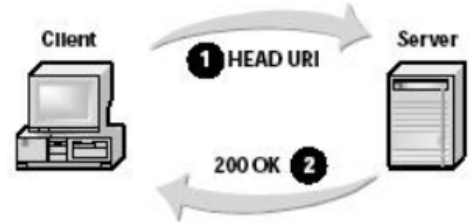
- **Metodo POST:**

- L'informazione è inserita all'interno del corpo del messaggio.
- Viene invocato ad ogni sottomissione di form
- Non si hanno meccanismi di cache.



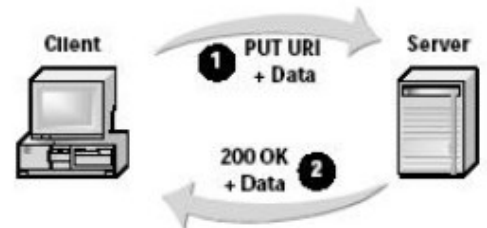
- **Metodo HEAD:**

- Simile al metodo GET, ma....
- Quando il server riceve una richiesta di questo tipo risponde segnalando la ricezione, ma non invia un oggetto in risposta.
- Utilizzato certe volte per fare debugging.



- **Metodo PUT:**

- Permette all'utente di fare upload in un percorso specifico su un server web specifico.
- Se l'URI fa riferimento a un qualcosa di già esistente allora il nuovo caricamento è considerata una versione aggiornata e quanto presente fino a poco prima viene sovrascritto.



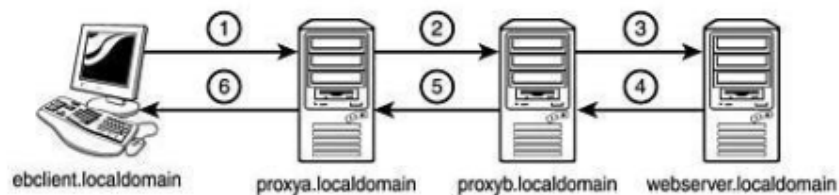
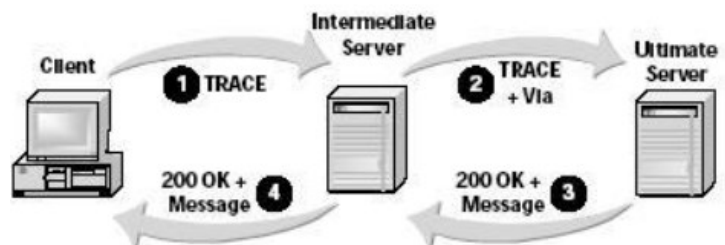
- **Metodo DELETE:**

- Permette all'utente, o a un'applicazione, di cancellare un oggetto da uno specifico server web.

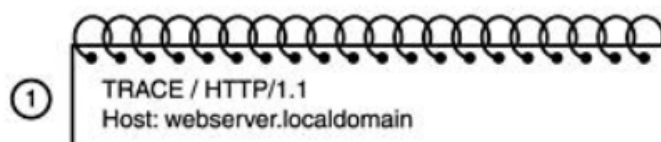


- **Metodo TRACE:**

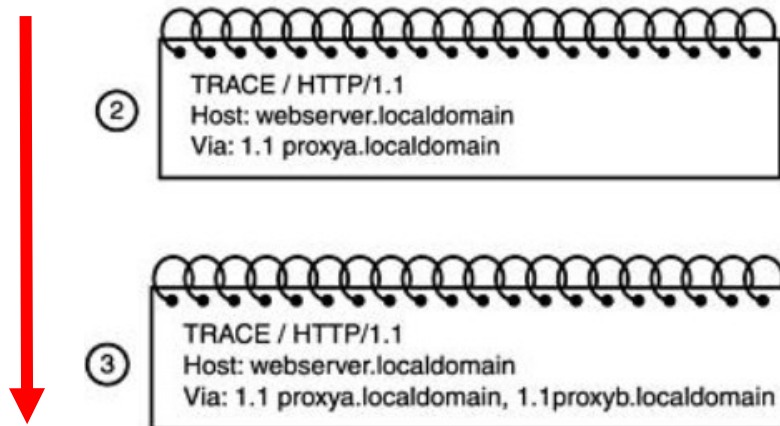
- Metodo usato per invocare un'applicazione remota e verificare quanto ci vuole per ricevere indietro un messaggio di risposta.
- Cerca di tracciare il percorso fatto dalla nostra richiesta per arrivare al server. Stessa cosa per la risposta del server che arriva a noi. Traccio cosa succede nel percorso, quindi anche i server intermedi.
- Utilizzato per fare diagnostica. Interessante per capire le cause dei ritardi.
- **Come funziona questo metodo?**



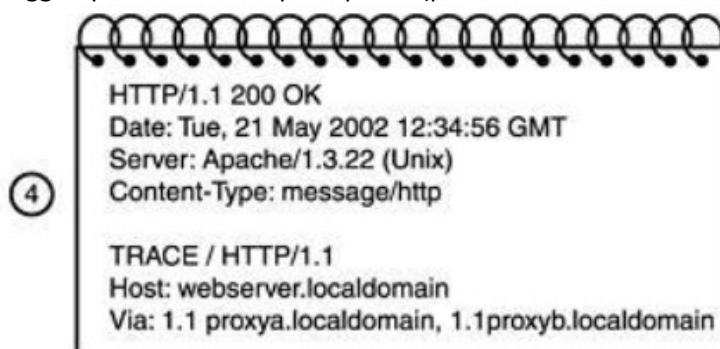
- Si parte dal client che fa la richiesta con metodo TRACE al server (webserver.localdomain)



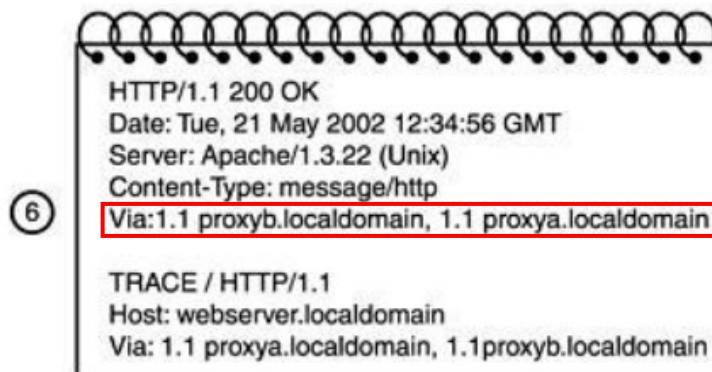
- Ogni volta che passo da un server intermedio questo, vedendo il metodo della richiesta, aggiunge nella linea "Via" il suo nome.



- Il server richiesto, quando viene raggiunto, crea il suo messaggio di risposta. Include nel corpo del messaggio quanto ricevuto poco prima (percorso da client a server).



- La risposta torna indietro, e i server intermedi fanno la stessa cosa di prima.



Request message

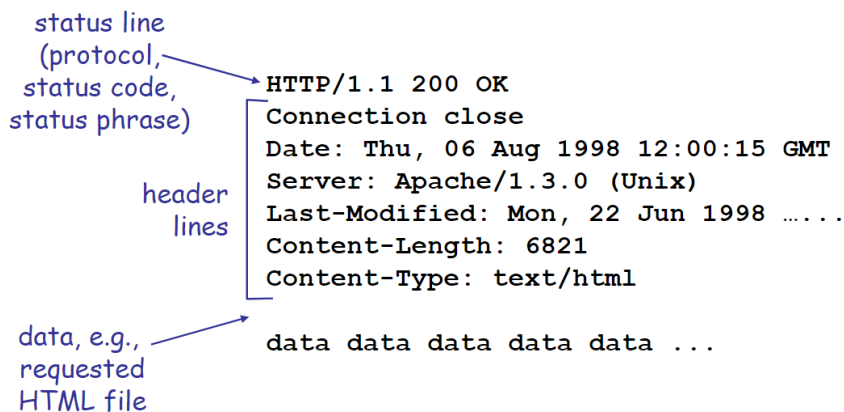
Nel messaggio di risposta abbiamo:

- **linea di stato:** protocollo, codice di stato, frase sintetica che esprime lo stato
- **linee di intestazione** (header)
- **corpo effettivo del messaggio** (sequenza di dati richiesti, inviati al client)

- **Response Status possibili:**

- **200 OK**

La richiesta ha avuto successo, la risorsa è stata recuperata nel server e restituita nel corpo del messaggio



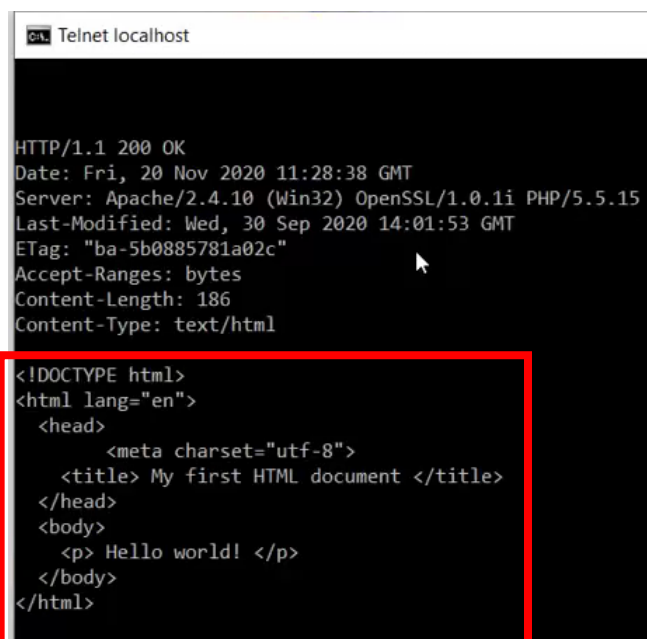
- **301 Moved Permanently**
L'oggetto richiesto è stata spostata in modo permanente. In generale si restituisce l'indirizzo della nuova locazione.
 - **400 Bad Request**
Il server non ha compreso la richiesta
 - **404 Not Found**
Il documento richiesto non è stato trovato sul server
 - **505 HTTP Version Not Supported**
Versione HTTP non supportata.
- **Linee di intestazione:**
- *Date*: indica l'ora e la data di quando la risposta è stata creata e inviata dal server.
 - *Content-type*: indica il tipo dell'oggetto contenuto (per esempio codice HTML)
 - *Content-Length*: specifica la grandezza del codice restituito
 - *Last-Modified*: va a indicare l'ultima data di modifica della risorsa restituita
 - *Server*: indica il server che sta rispondendo.

Testare il protocollo http

- Possiamo sbizzarrirci e testare il protocollo HTTP utilizzando telnet.
- Aprite il prompt dei comandi e stabilite una connessione sul vostro *localhost* (assicuratevi che il server sia acceso)
telnet localhost 80
Apro una connessione TCP sulla porta 80 (la porta usata di default dal protocollo http)
- A questo punto avremo una schermata a parte dove possiamo incollare i nostri messaggi di richiesta.
Vediamo degli esempi sul localhost di Marcelloni:

```
GET /html.html HTTP/1.1
Host: localhost
```

```
GET /paginainesistente.html HTTP/1.1
Host: localhost
```



```

HTTP/1.1 200 OK
Date: Fri, 20 Nov 2020 11:28:38 GMT
Server: Apache/2.4.10 (Win32) OpenSSL/1.0.1i PHP/5.5.15
Last-Modified: Wed, 30 Sep 2020 14:01:53 GMT
ETag: "ba-5b0885781a02c"
Accept-Ranges: bytes
Content-Length: 186
Content-Type: text/html

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title> My first HTML document </title>
  </head>
  <body>
    <p> Hello world! </p>
  </body>
</html>

```



```

HTTP/1.1 404 Not Found
Date: Fri, 20 Nov 2020 11:31:14 GMT
Server: Apache/2.4.10 (Win32) OpenSSL/1.0.1i PHP/5.5.15
Vary: accept-language,accept-charset
Accept-Ranges: bytes
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
Content-Language: en

cb
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="
ge
en" xml:lang="
15
en">
  <head>
    <title>
39
Object not found!</title>
    <link rev="made" href="mailto:
117
postmaster@localhost" />
    <style type="text/css"><!--/*--><![CDATA[*><!--*/
body { color: #000000; background-color: #FFFFFF; }

```

Codice HTML restituito all'utente

Linee di intestazione

- Nella versione 1.0 di HTTP si potevano usare 16 linee di intestazioni diverse. Nessuna di queste era obbligatoria.
- Nella versione attuale di HTTP sono definite 51 linee di intestazione diverse. La linea Host è l'unica obbligatoria. Si ottiene errore di cattiva richiesta se si omette questa linea.
- **Come sceglie le linee il client?** Le linee nei messaggi di richiesta sono confezionate direttamente dal browser, dipendono dalla versione del browser, dalla sua configurazione, e dal caching.
- **Come sceglie le linee il server?** Il server sceglie il numero di righe e imposta il messaggio. Il messaggio dipende dal tipo di server, dalla sua versione e dalla sua configurazione.

Autorizzazioni

- Il server http è *stateless*: non si memorizza lo stato delle richieste (maggiore efficienza e possibilità di gestire connessioni simultanee, non dobbiamo creare strutture dati particolari e dedicarci al mantenimento della consistenza – ricordare quanto detto indietro).
- In alcuni casi un sito ha bisogno di identificare l'utente: il protocollo HTTP include la possibilità di chiedere dati di accesso (attraverso delle linee di intestazione speciali).

- **Scenario:**

- o Il client richiede un oggetto al server, il server richiede un autorizzazione.
- o Il client invia un messaggio di richiesta senza particolari linee
- o Il server, non avendo ricevuto nulla di particolare, non restituisce niente all'utente. Segnala lo stato *401 Authorization Required* e include l'header *WWW-Authenticate* per segnalare la necessità di autenticazione.
- o L'utente indica username e password.
- o Il client re-invia il messaggio di richiesta includendo una linea di intestazione *Authorization*.
- o Quando il primo oggetto è stato ottenuto il client continua a spedire username e password nelle richieste successive (*caching*).



- **Attenzione:** il meccanismo è molto carino, ma estremamente debole. Un qualunque utente maligno che sniffa la rete (cioè guarda cosa passa nella rete) è in grado di leggere in chiaro i dati di accesso. Nel protocollo HTTPS la situazione è decisamente migliore (le informazioni vengono crittografate).

Cookies

- **Abbiamo quattro componenti per gestire questo meccanismo:**

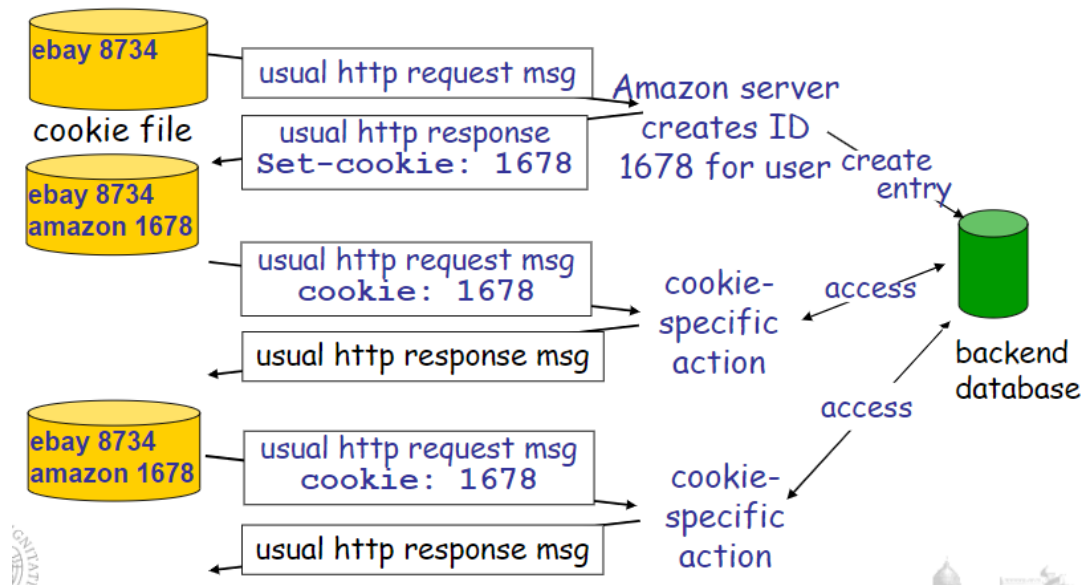
- o La linea di intestazione cookie nel messaggio di risposta del server
- o La linea di intestazione cookie nel messaggio di richiesta
- o Il file di cookie, mantenuto nell'host dell'utente e gestito dal browser
- o Database di back-end nel sito (memorizzazione di cookie lato server)

- **Esempio:**

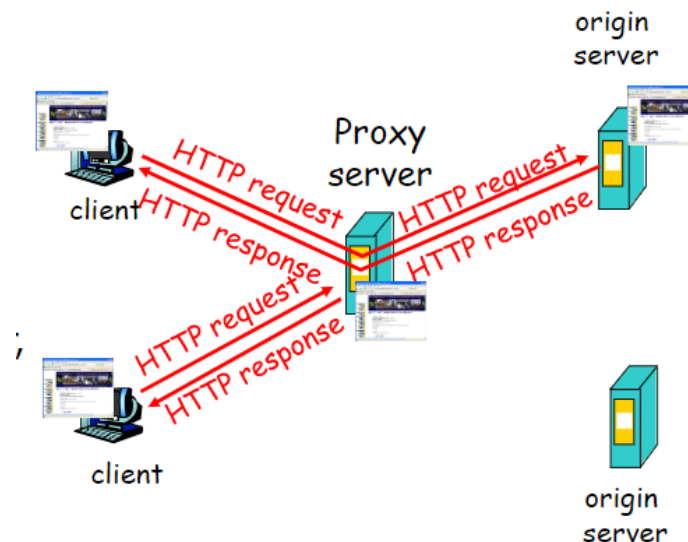
- o Susan accede ad internet usando il PC.
- o Visita per la prima volta un sito di e-commerce.
- o Quando effettua la prima richiesta http il sito crea un identificatore univoco e lo inserisce nel database di *backend*. Successivamente restituisce l'ID al client con la linea di intestazione *Set-cookie* in modo tale che il client sappia come viene identificato.
- o Ogni volta che il client esegue una richiesta verso il server indica l'identificativo attraverso la linea di intestazione *cookie* il server, leggendo l'identificativo, capisce come comportarsi.

client

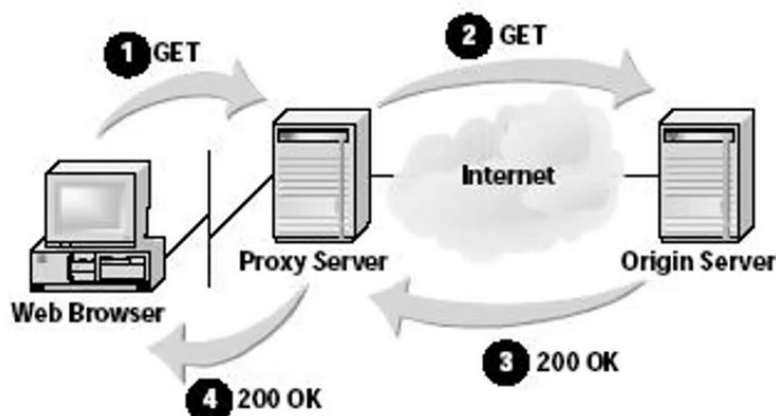
server



Web caching



- L'obiettivo principale di questo meccanismo è soddisfare la richieste del client in modo rapido senza per forza coinvolgere il server di origine.
- Le richieste viaggiano attraverso server intermedi prima di arrivare al server di origine. Le risorse richieste, restituite dai server con messaggi di risposta, vengono salvate nei server intermedi.
- Se lo stesso client, o un altro client, fanno richiesta di questa risorsa, il server intermedio può restituire immediatamente l'oggetto richiesto senza coinvolgere il server di origine.
- **Vantaggi:** risposte restituite più velocemente, minore traffico nel link di accesso dell'istituzione.



- Per capire meglio il meccanismo vedere gli esempi sulle diapositive del docente. Abbiamo una situazione in cui gli oggetti richiesti hanno, in media, una dimensione di un milione di bit. La media di richieste al secondo verso il server è di 15, il ritardo dovuto al client-server-client è di due secondi.
 - o Se non utilizziamo il sistema di web caching il ritardo è intollerabile. Per rendere chiara l'idea immaginiamoci il passaggio dei dati da un tubo molto stretto.
 - o Se invece utilizziamo il web-caching avremo il 40% di richieste soddisfatte in modo immediato (si parla di millisecondi) e il 60% soddisfatte dal server.
 - o Soluzione alternativa può essere l'espansione della bandwidth. La soluzione è valida, ma estremamente costosa.
- **Problemi:**
 - o La web cache può risiedere in un client o il server intermedio. Il problema tipico (visibile quando progettiamo un sito e modifichiamo gli stylesheet o i js frequentemente) è il non aggiornamento dei contenuti (riceviamo in modo veloce un contenuto grazie al web caching, ma non riceviamo la versione aggiornata).
 - o La soluzione a questo problema è il GET condizionale: la richiesta si basa sul metodo GET e include nell'header la linea *If-Modified-Since*. Indicando una data permettiamo al server intermedio di capire come comportarsi (restituire la risorsa da lui posseduta o rimandare la richiesta al server di origine).
 - o Il server intermedio, se restituisce una risorsa da lui posseduta, segnala lo stato *304 Not Modified*. Negli altri casi si ha come stato *200 OK*.

