

# Introduzione ad HTTP

---

Fabio Vitali



# Introduzione

Oggi esaminiamo in breve:

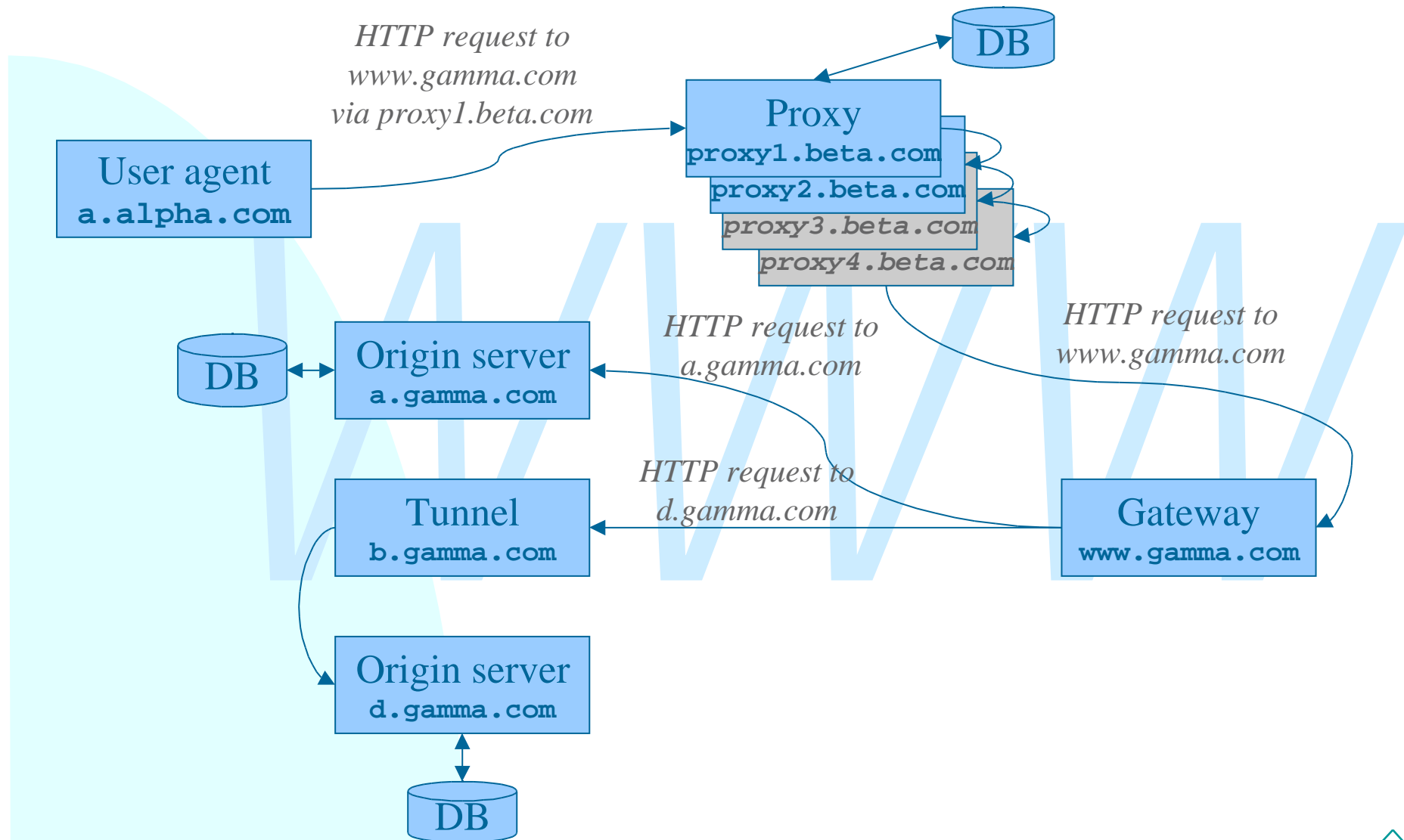
## *HTTP (HyperText Transfer Protocol)*

*Un protocollo stateless per la ricerca, il recupero e la manipolazione*

- HTTP é un protocollo con la leggerezza e la velocità necessari per un sistema informativo ipertestuale distribuito e collaborativo.
- E' un protocollo generico, stateless, object-oriented, che può essere usato anche per scopi diversi dallo scambio di documenti ipertestuali, come name server, per sistemi object-oriented distribuiti, ecc.
- Caratteristica importante in HTTP é la negoziazione del formato di dati utilizzato, per garantire l'indipendenza del sistema dal formato di rappresentazione dei dati.
- HTTP è esistito in tre versioni: 0.9, 1.0 (RFC 1945) e 1.1 (RFC 2616)
- Netscape ha proposto il meccanismo dei cookie nel'RFC 2109.



# Ruoli delle applicazioni HTTP (1)



# Ruoli delle applicazioni HTTP (2)

HTTP è un protocollo di comunicazione piuttosto semplice, basato sulla comunicazione tra due applicazioni, il browser, che manda richieste di documenti, ed il server, che risponde.

In realtà i ruoli sono un po' più precisi:

- ◆ **Client:** un'applicazione che stabilisce una connessione HTTP, con lo scopo di mandare richieste.
- ◆ **Server:** un'applicazione che accetta connessioni HTTP, e genera risposte.
- ◆ **User agent:** Quel particolare client che inizia una richiesta HTTP (tipicamente un browser, ma può anche essere un bot).
- ◆ **Origin server:** il server che possiede fisicamente la risorsa richiesta (è l'ultimo della catena)



# Ruoli delle applicazioni HTTP (3)

- ◆ **Proxy:** Un'applicazione intermediaria che agisce sia da client che da server. Le richieste sono soddisfatte autonomamente, o passandole ad altri server, con possibile trasformazione, controllo, verifica.
- ◆ **Gateway:** un'applicazione che agisce da intermediario per qualche altro server. A differenza del proxy, il gateway riceve le richieste come fosse l'origin server: il client può non essere al corrente che si tratta del gateway.
- ◆ **Tunnel:** un programma intermediario che agisce da trasmettitore passivo di una richiesta HTTP. Il tunnel non fa parte della comunicazione HTTP, anche se può essere stato attivato da una connessione HTTP.

In più è importante ricordare:

- ◆ **Cache:** memoria locale di un'applicazione e il sistema che controlla i meccanismi della sua gestione ed aggiornamento. Qualunque client o server può utilizzare una cache, ma non un tunnel.



# Tipi di proxy HTTP

In generale un proxy si pone come intermediario tra client e server e decide se e come rispondere al client

## Proxy di cache

- ◆ Richieste multiple agli stessi URL possono essere salvate in una locazione intermedia per una maggiore efficienza nella gestione delle risposte

## Proxy di filtro

- ◆ Esigenze di sicurezza o di controllo degli abusi di una rete possono richiedere l'effettiva esecuzione della richiesta solo in certi casi, e altrimenti la risposta con un generico messaggio di mancata autorizzazione.

## Intermediari

- ◆ Un proxy trasparente esegue tutte le richieste e fornisce tutte le risposte, ma in certi casi può convertire o modificare la risposta.
- ◆ Ad esempio fornire link a vocabolari, togliere i banner, convertire i formati ignoti, ecc.
- ◆ Ad esempio, WBI di IBM (<http://www.almaden.ibm.com/cs/wbi/>)

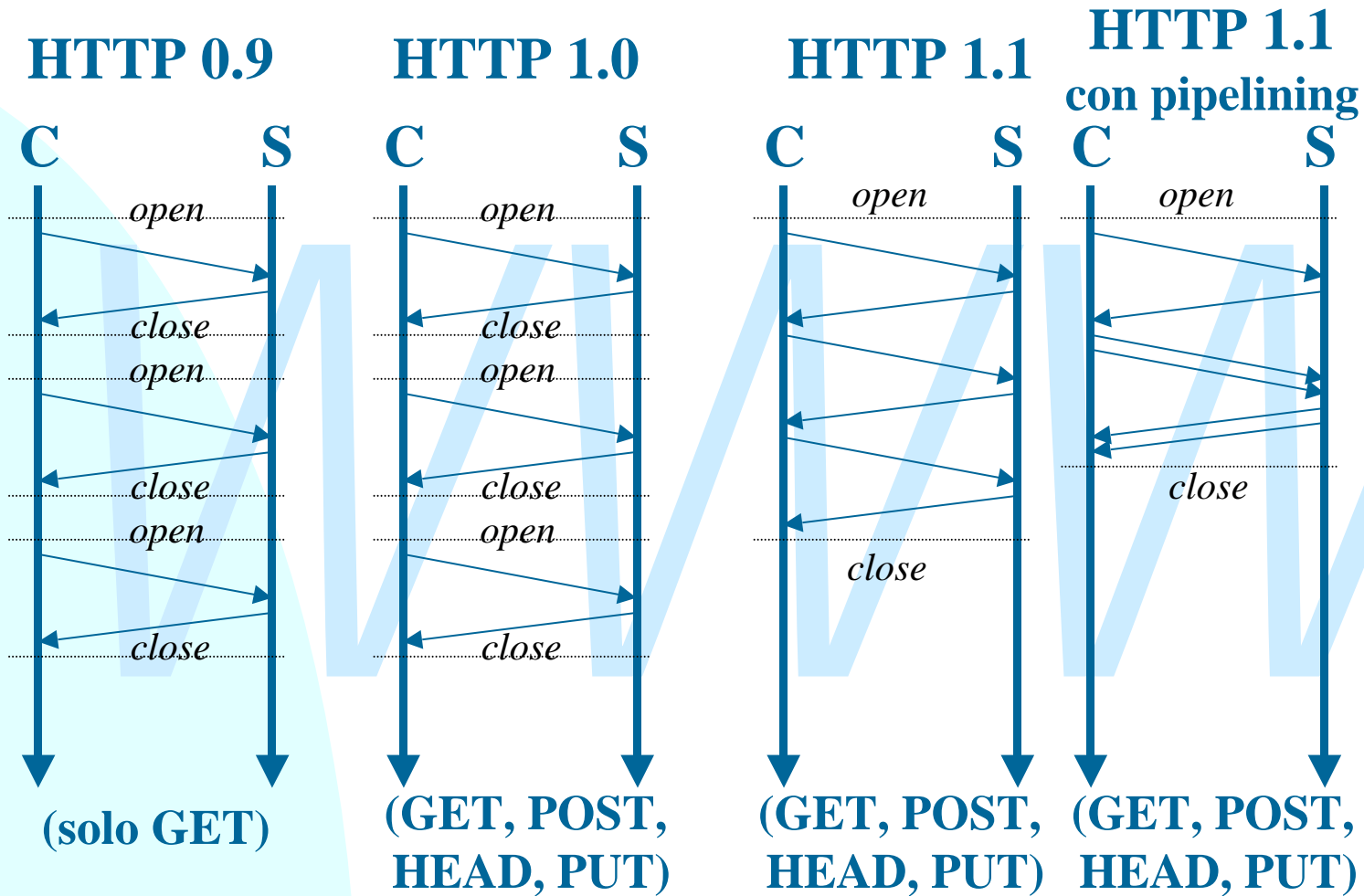


# La connessione HTTP (1)

- La connessione HTTP è composta da una serie di richieste ed una serie corrispondente di risposte.
- La differenza principale tra HTTP 1.0 e 1.1 è stata la possibilità di specificare coppie multiple di richiesta e risposta nella stessa connessione.
- Le richieste possono essere messe in pipeline, ma le risposte debbono essere date nello stesso ordine delle richieste, poiché non è specificato un metodo esplicito di associazione.



# La connessione HTTP (2)





# La richiesta (1)

La richiesta è un messaggio MIME formato da una riga di richiesta e da dati ulteriori facoltativi.

- ◆ La richiesta semplice (HTTP 0.9) è:

```
GET URI CrLf
```

- ◆ La richiesta completa (HTTP 1.0 e segg.) è:

```
Method URI Version CrLf
```

```
[Header]*
```

```
CrLf
```

```
[Body]
```



# La richiesta (2)

- La richiesta semplice è stata introdotta nella versione 0.9 (la prima versione di HTTP) e ne è ancora obbligata l'implementazione.
- La richiesta completa è il meccanismo completo di comunicazione offerto da HTTP 1.0 e 1.1.
- La presenza o meno di `Version` nella linea di richiesta fanno capire al server se si può direttamente creare la risposta o se è necessario attendere altri dati.



# La richiesta completa

```
Method URI Version CrLf
[Header]*
CrLf
Body
```

dove

- ◆ Method indica l'azione del server richiesta dal client
- ◆ URI è un identificativo di risorsa **locale al server**
- ◆ Version è "HTTP/1.0" o "HTTP/1.1"
- ◆ Header sono linee RFC822, classificabili come header generali, header di entità, ed header di richiesta.
- ◆ Body è un messaggio MIME



# Un esempio di richiesta

```
GET /beta.html HTTP/1.1
Referer: http://www.alpha.com/alpha.html
Connection: Keep-Alive
User-Agent: Mozilla/4.61 (Macintosh; I; PPC)
Host: www.alpha.com:80
Accept: image/gif, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```



# I metodi della richiesta

Un metodo HTTP può essere:

- ◆ **Sicuro:** non genera cambiamenti allo stato interno del server
- ◆ **Idempotente:** l'effetto sul server di più richieste identiche è lo stesso di quello di una sola richiesta.

Noi ci occupiamo dei metodi seguenti:

- ◆ GET
- ◆ HEAD
- ◆ POST
- ◆ PUT



# Il metodo GET

Il più importante (ed unico in v. 0.9) metodo di HTTP è GET, che richiede una risorsa ad un server.

Questo è il metodo più frequente, ed è quello che viene attivato facendo click su un link ipertestuale di un documento HTML, o specificando un URL nell'apposito campo di un browser.

GET è sicuro ed idempotente, e può essere:

- ◆ *assoluto* (normalmente, cioè quando la risorsa viene richiesta senza altre specificazioni),
- ◆ *condizionale* (se la risorsa fa match con un criterio indicato negli header **If-match**, **If-modified-since**, **If-range**, etc.)
- ◆ *parziale* (se la risorsa richiesta è una sottoparte di una risorsa memorizzata).



# Il metodo HEAD

Il metodo HEAD è simile al metodo GET, ma il server deve rispondere soltanto con gli header relativi, senza il corpo.

HEAD è sicuro ed idempotente, e viene usato per verificare:

- ◆ *la validità di un URI*: la risorsa esiste e non è di lunghezza zero,
- ◆ *l'accessibilità di un URI*: la risorsa è accessibile presso il server, e non sono richieste procedure di autenticazione del documento.
- ◆ *la coerenza di cache di un URI*: la risorsa non è stata modificata nel frattempo, non ha cambiato lunghezza, valore hash o data di modifica.



# Il metodo POST

Il metodo POST serve per trasmettere delle informazioni dal client al server, ma senza la creazione di una nuova risorsa.

POST non è sicuro né idempotente, e viene usato per esempio per sottomettere i dati di una form HTML ad un'applicazione CGI sul server.

Il server può rispondere positivamente in tre modi:

- ◆ 200 Ok: dati ricevuti e sottomessi alla risorsa specificata. E' stata data risposta
- ◆ 201 Created: dati ricevuti, la risorsa non esisteva ed è stata creata
- ◆ 204 No content: dati ricevuti e sottomossi alla risorsa specificata. Non è stata data risposta.





# Il metodo PUT

Il metodo PUT serve per trasmettere delle informazioni dal client al server, creando o sostituendo la risorsa specificata.

In generale, l'argomento del metodo PUT è la risorsa che ci si aspetta di ottenere facendo un GET in seguito con lo stesso nome. L'argomento del metodo POST, invece, è una risorsa esistente a cui si aggiunge (es. come input) informazione.

PUT è idempotente ma non sicuro, e comunque non offre nessuna garanzia di controllo degli accessi o locking. Per questo è nato il gruppo di lavoro WebDAV, che ha fornito una semantica sicura e collaborativa per il metodo PUT (tra le altre cose).



# Gli header

Gli header sono righe RFC822 che specificano caratteristiche

- ◆ generali della trasmissione
- ◆ dell'entità trasmessa,
- ◆ della richiesta effettuata
- ◆ della risposta generata



# Header generali

Gli header generali si applicano solo al messaggio trasmesso e si applicano sia ad una richiesta che ad una risposta, ma non necessariamente alla risorsa trasmessa.

- ◆ **Date:** data ed ora della trasmissione
- ◆ **MIME-Version:** la versione MIME usata per la trasmissione (sempre 1.0)
- ◆ **Transfer-Encoding:** il tipo di formato di codifica usato per la trasmissione
- ◆ **Cache-Control:** il tipo di meccanismo di caching richiesto o suggerito per la risorsa
- ◆ **Connection:** il tipo di connessione da usare (tenere attiva, chiudere dopo la risposta, ecc.)
- ◆ **Via:** usato da proxy e gateway.



# Header dell'entità

Gli header dell'entità danno informazioni sul body del messaggio, o, se non vi è body, sulla risorsa specificata.

- ◆ **Content-Type**: il tipo MIME dell'entità acclusa. Questo header è obbligatorio in ogni messaggio che abbia un body.
- ◆ **Content-Length**: la lunghezza in byte del body. Obbligatorio, soprattutto se la connessione è persistente.
- ◆ **Content-Base, Content-Encoding, Content-Language, Content-Location, Content-MD5, Content-Range**: l'URL di base, la codifica, il linguaggio, l'URL della risorsa specifica, il valore di digest MD5 e il range richiesto della risorsa.
- ◆ **Expires**: una data dopo la quale la risorsa è considerata non più valida (e quindi va richiesta o cancellata dalla cache).
- ◆ **Last-Modified**: Obbligatorio se possibile. La data e l'ora dell'ultima modifica. Serve per decidere se la copia posseduta (es. in cache) è ancora valida o no.



# Header della richiesta (1)

Gli header della richiesta sono posti dal client per specificare informazioni sulla richiesta e su se stesso al server.

- ◆ **User-Agent:**

- ✦ una stringa che descrive il client che origina la richiesta; tipo, versione e sistema operativo del client, tipicamente.

- ◆ **Referer:**

- ✦ L'errore di spelling è dovuto a ragioni storiche (si direbbe *Referrer*)
- ✦ l'URL della pagina mostrata all'utente mentre richiede il nuovo URL.
- ✦ Se l'URL è richiesto con altri metodi che non l'attraversamento di un link es. digitando l'URL o selezionandolo dai bookmark, Referer deve essere assente.
- ✦ Referer viene usato per controllo sui percorsi degli utenti, utili nel caso di user profiling (che gusti ha il mio utente? Lo capisco dalla pagina da cui proviene) o pubblicità (il mio utente ha cliccato su un banner. A chi devo pagare i diritti?)



# Header della richiesta (2)

## ◆ Host:

- ◆ Header obbligatorio in HTTP 1.1.
- ◆ Contiene il nome di dominio e la porta a cui viene fatta la connessione.
- ◆ L'URI posto nella riga di richiesta è soltanto la parte locale al server. Manda l'indicazione del nome del server o della porta acceduta.

```
GET /beta.html HTTP/1.0
```

```
...
```

```
Host: www.alpha.com:80
```

- ◆ Se un server contiene più siti Web per scopi diversi, Host permette al server di distinguere il sito a cui la richiesta fa riferimento.
- ◆ Permette l'implementazione di virtual hosting senza manipolazioni del routing e multi-addressing IP.

## ◆ From:

- ◆ l'indirizzo di e-mail del richiedente. Si richiede che l'utente dia la sua approvazione prima di inserire questo header nella richiesta.



# Header della richiesta (3)

- ◆ **Range:**
  - ◆ il range della richiesta. Poco usato.
- ◆ **Accept, Accept-Charset, Accept-Encoding, Accept-Language:**
  - ◆ Implementazione della negoziazione del formato, per quel che riguarda tipo MIME, codice caratteri, codifica MIME, linguaggio umano.
  - ◆ Il client specifica cosa è in grado di accettare, e il server propone il match migliore.
- ◆ **If-Modified-Since, If-Unmodified-Since:**
  - ◆ richieste condizionali (per esempio, per aggiornare una cache) che vanno portate a termine solo se la condizione è vera.
  - ◆ Se la pre-condizione è valida, viene ritornato un 304 (not modified), altrimenti si procede come per un GET normale.
- ◆ **Authorization, Proxy-Authorization:**
  - ◆ una stringa di autorizzazione per l'accesso alla risorsa richiesta. Ne parliamo oltre.



# La risposta

```
Version status-code reason-phrase CrLf  
[Header]*  
CrLf  
Body
```

```
GET /index.html HTTP/1.1  
Host: www.cs.unibo.it:80  
  
HTTP/1.1 200 OK  
Date: Fri, 26 Nov 1999 11:46:53 GMT  
Server: Apache/1.3.3 (Unix)  
Last-Modified: Mon, 12 Jul 1999 12:55:37 GMT  
Accept-Ranges: bytes  
Content-Length: 3357  
Content-Type: text/html  
  
<HTML> ... </HTML>
```





# Status code

Lo status code è un numero di tre cifre, di cui la prima indica la classe della risposta, e le altre due la risposta specifica.

Esistono le seguenti classi:

- ◆ **1xx: Informational.** Una risposta temporanea alla richiesta, durante il suo svolgimento.
- ◆ **2xx: Successful.** Il server ha ricevuto, capito e accettato la richiesta.
- ◆ **3xx: Redirection.** Il server ha ricevuto e capito la richiesta, ma sono necessarie altre azioni da parte del client per portare a termine la richiesta.
- ◆ **4xx: Client error.** La richiesta del client non può essere soddisfatta per un errore da parte del client (errore sintattico o richiesta non autorizzata).
- ◆ **5xx: Server error.** La richiesta può anche essere corretta, ma il server non è in grado di soddisfare la richiesta per un problema interno (suo o di applicazioni CGI).



# Esempi di status code

- 100 Continue** (se il client non ha ancora mandato il body)
- 200 Ok** (GET con successo)
- 201 Created** (PUT con successo)
- 301 Moved permanently** (URL non valida, il server conosce la nuova posizione)
- 400 Bad request** (errore sintattico nella richiesta)
- 401 Unauthorized** (manca l'autorizzazione)
- 403 Forbidden** (richiesta non autorizzabile)
- 404 Not found** (URL errato)
- 500 Internal server error** (tipicamente un CGI mal fatto)
- 501 Not implemented** (metodo non conosciuto dal server)



# Header della risposta

Gli header della risposta sono posti dal server per specificare informazioni sulla risposta e su se stesso al client

- ◆ **Server:**

- ◆ una stringa che descrive il server: tipo, sistema operativo e versione.

- ◆ **WWW-Authenticate:**

- ◆ l'header di WWW-Authenticate include una challenge (codice di partenza) con cui il meccanismo di autenticazione deve fare match in caso di una risposta 401, (unauthorized). Il client genererà con questo valore un valore di autorizzazione posto nell'header Authorization della prossima richiesta.

- ◆ **Accept-ranges:**

- ◆ specifica che tipo di range può accettare (valori previsti: byte e none).



# Autenticazione (1)

Quando si vuole accedere ad una risorsa su cui esistono restrizioni di accesso, il server richiede l'autenticazione dell'utente.

Al GET viene fornita la risposta 401 (unauthorized), più un header WWW-Authenticate che specifica i criteri con cui autenticarsi (metodo e parametri da usare).

HTTP ha due metodi di autenticazione:

- ◆ Basic authentication (introdotto in HTTP 1.0)
- ◆ Digest access authentication (introdotto in HTTP 1.1)



# Autenticazione (2)

## Basic authentication

- ◆ Introdotto da HTTP 1.0.
- ◆ L'header della prima risposta WWW-Authenticate contiene il realm dell'autenticazione.
- ◆ Il client richiede le informazioni di autorizzazione all'utente e
- ◆ Il client crea una nuova richiesta GET e fornisce le informazioni di autorizzazione codificate in Base64.
- ◆ Il browser continua a mandare lo stesso header per tutte le pagine dello stesso realm.
- ◆ **Problema: La password passa dunque in chiaro sulla rete.**



# Autenticazione (3)

## Digest access authentication

- ◆ Introdotta da HTTP 1.1, descritto in RFC 2069.
- ◆ Non manda la password in chiaro, ma una fingerprint della password, ovvero la password crittografata con il metodo MD5 (RFC 1321).
- ◆ Per evitare l'abuso della password, anche se crittografata, insieme alla fingerprint vengono codificate anche informazioni come lo username, il realm, l'URI richiesto, una time stamp, ecc.).
- ◆ Ovviamente non risolve il problema di fare arrivare al server la password la prima volta!



# Caching (1)

Può essere client-side, server-side o intermedia (su un proxy).

La cache server-side riduce i tempi di computazione di una risposta, ma non ha effetti sul carico di rete.

Le altre riducono il carico di rete.

HTTP 1.0 si basava su tre header:

- ◆ **Expires:** il server specifica la data di scadenza di una risorsa
- ◆ **If-Modified-Since:** il client richiede la risorsa solo se modificata dopo il giorno X. Richiede una gestione del tempo comune tra client e server
- ◆ **Pragma: no-cache:** Fornita dal server, istruisce il client di non fare cache della risorsa in ogni caso.

HTTP 1.1 introduce due tipi di cache control:

- ◆ Server-specified expiration
- ◆ Heuristic expiration



# Caching (2)

## Server-specified expiration

- ◆ Il server stabilisce una data di scadenza della risorsa, con l'header Expires o con la direttiva max-age nell'header Cache-Control
- ◆ Se la data di scadenza è già passata, la richiesta deve essere rivalidata. Se la richiesta accetta anche risposte scadute, o se l'origin server non può essere raggiunto, la cache può rispondere con la risorsa scaduta ma con il codice 110 (Response is stale)
- ◆ Se Cache-Control specifica la direttiva must-revalidate, la risposta scaduta non può mai essere rispedita. In questo caso la cache deve riprendere la risorsa dall'origin server. Se questo non risponde, la cache manderà un codice 504 (Gateway time-out)
- ◆ Se Cache-Control specifica la direttiva no-cache, la richiesta deve essere fatta sempre all'origin server.





# Caching (3)

## Heuristic expiration

- ◆ Poiché molte pagine non conterranno valori espliciti di scadenza, la cache stabilisce valori euristici di durata delle risorse, dopo le quali assume che sia scaduta.
- ◆ Queste assunzioni possono a volte essere ottimistiche, e risultare in risposte scorrette. Se non valida con sicurezza una risposta assunta fresca, allora deve fornire un codice 113 (heuristic expiration) alla risposta.



# Caching (4)

## Validazione della risorsa in cache

- ◆ Anche dopo la scadenza, nella maggior parte dei casi, una risorsa sarà ancora non modificata, e quindi la risorsa in cache valida.
- ◆ Un modo semplice per fare validazione è usare HEAD: il client fa la richiesta, e verifica la data di ultima modifica. Ma questo richiede una richiesta in più sempre.
- ◆ Un modo più corretto è fare una richiesta condizionale: se la risorsa è stata modificata, viene fornita la nuova risorsa normalmente, altrimenti viene fornita la risposta 304 (not modified) senza body della risposta. Questo riduce il numero di richieste



# Modelli di sicurezza (1)

Ci sono due modi per fornire un trasporto sicuro (cioè non intercettabile da orecchie maliziose durante la trasmissione):

- ◆ Usare un'infrastruttura di trasporto sicura
  - ✦ Il protocollo non cambia, ma ogni pacchetto trasmesso nello scambio di informazioni viene gestito in maniera sicura dal protocollo di trasporto
- ◆ Usare un protocollo sicuro a livello applicazione
  - ✦ Si usa un protocollo anche diverso, che si occupa di gestire la trasmissione delle informazioni.



# Modelli di sicurezza (2)

## ◆ HTTPS

- ◆ Introdotto da Netscape, trasmette i dati in HTTP semplice su un protocollo di trasporto (SSL) che crittografa tutti i pacchetti.
- ◆ Il server ascolta su una porta diversa (per default la porta 443), e si usa uno schema di URI diverso (introdotto da `https://` )

## ◆ S-HTTP

- ◆ Poco diffuso, incapsula richieste e risposte HTTP in un messaggio crittografato secondo o un formato MIME apposito (MIME Object Security Services, MOSS), o un formato terzo (Cryptographic Message Syntax, CMS).
- ◆ E' più efficiente ma più complesso.



# I cookies

HTTP è stateless: non esiste nessuna struttura ulteriore alla connessione, e il server non è tenuto a mantenere informazioni su connessioni precedenti.

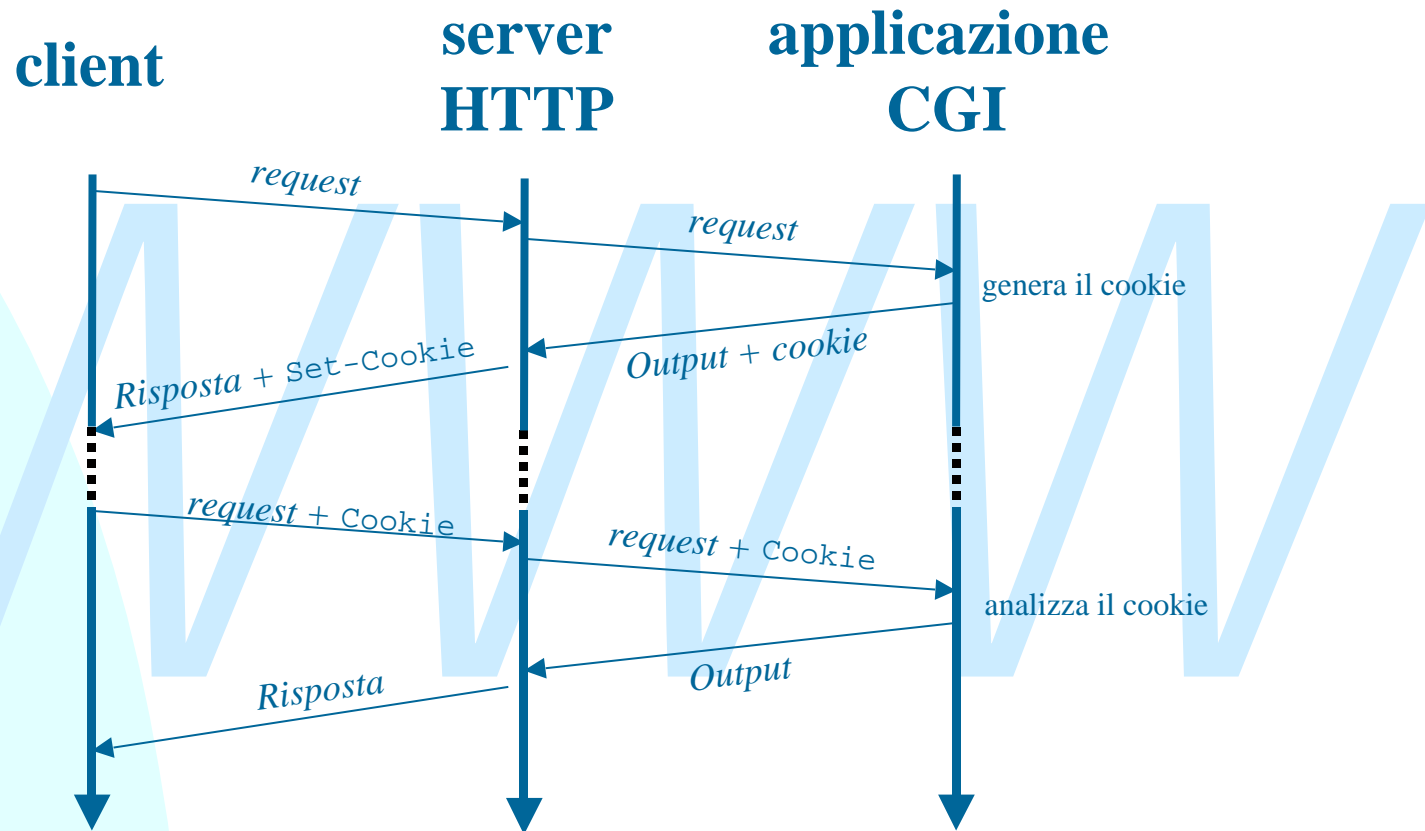
Un cookie (non in HTTP, è un'estensione di Netscape, proposta nell'RFC 2109) è una breve informazione scambiata tra il server ed il client.

Il client mantiene lo stato di precedenti connessioni, e lo manda al server di pertinenza ogni volta che richiede un documento.

Questioni di sicurezza permettono di distinguere tra cookies spediti solo al server di appartenenza e cookies spediti a qualunque computer del dominio.



# I cookies (2)



# I cookies (3)

I cookies dunque usano due header, uno per la risposta, ed uno per le richieste successive:

- ◆ **Set-Cookie**: header della risposta, il client può memorizzarlo e rispedito alla prossima richiesta.
- ◆ **Cookie**: header della richiesta. Il client decide se spedito sulla base del nome del documento, dell'indirizzo IP del server, e dell'età del cookie.



# I cookies (4)

I cookies contengono le seguenti informazioni:

- ◆ **Comment:** stringa leggibile di descrizione del cookie.
- ◆ **Domain:** il dominio per cui il cookie è valido
- ◆ **Max-Age:** La durata in secondi del cookie.
- ◆ **Path:** l'URI per il quale il cookie è valido
- ◆ **Secure:** la richiesta che il client contatti il server usando soltanto un meccanismo sicuro (es. SHTTP) per spedirlo
- ◆ **Version:** La versione della specifica a cui il cookie aderisce.





# Conclusioni

Oggi abbiamo parlato di

- ◆ Protocollo HTTP
- ◆ Meccanismo di gestione dello stato in HTTP (cookie)



# Riferimenti

## *Wilde's WWW, capitolo 3*

Altri testi:

- T. Berners-Lee, R. Fielding, H. Frystyk, *Hypertext Transfer Protocol -- HTTP/1.0*, RFC 1945, May 1996
- D. Kristol, L. Montulli, *HTTP State Management Mechanism*, RFC 2109, February 1997
- R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, RFC 2616, June 1999

