Introduzione ad HTTP

Fabio Vitali

WWW

Introduzione

Oggi esaminiamo in breve:

HTTP (HyperText Transfer Protocol)

An application-level protocol for distributed, collaborative, hypermedia information systems

- **La storia**
- **Il funzionamento**
- **Caching e autenticazione**
- **I cookie**

WWW

HTTP

HTTP é un protocollo client-server generico e stateless utilizzato non solo per lo scambio di documenti ipertestuali, ma per una moltitudine di applicazioni, incluso name server e sistemi object-oriented distribuiti.

Caratteristiche di HTTP sono

- la negoziazione del formato di dati, per l'indipendenza del sistema dal formato di rappresentazione dei dati.
- Specifiche di politiche di caching sofisticate a seconda del tipo di connessione
- Specifiche di autenticazione dell'utente di varia sofisticazione.

Storia di HTTP

HTTP è esistito in tre versioni:

- 0.9: un semplicissimo protocollo client-server di sola richiesta di risorse HTML, senza flessibilità né nella direzione, né nel formato delle risorse. Utilizzata nel primo prototipo WWW e nei primi server NCSA.
- 1.0 (RFC 1945): il protocollo diventa generico e definisce la statelessness, e definisce alcuni metodi anche per l'upload di dati. Utilizzato fino al 1998-99
- 1.1 (RFC 2068, 2069 e poi 2616, 2617): la versione attuale di HTTP, specifica meglio i meccanismi di caching, permette multi-homing e connessioni persistenti.
- HTTP-NG doveva essere la naturale evoluzione di HTTP, ma il WG IETF fallì miseramente nel raggiungere l'obiettivo, e l'evoluzione del protocollo si fermò.
- I cookie, originariamente proposti da Netscape, vennero descritti nell'RFC 2109 e poi 2965).
- I lavori di WebDAV estendono di fatto HTTP, ma non generano una nuova versione del protocollo.

Alcune definizioni

Client-server

- In HTTP esistono due ruoli specifici: il *client* attiva la connessione e richiede dei servizi. Il server accetta la connessione, nel caso identifica il richiedente, e risponde alla richiesta. Alla fine chiude la connessione.

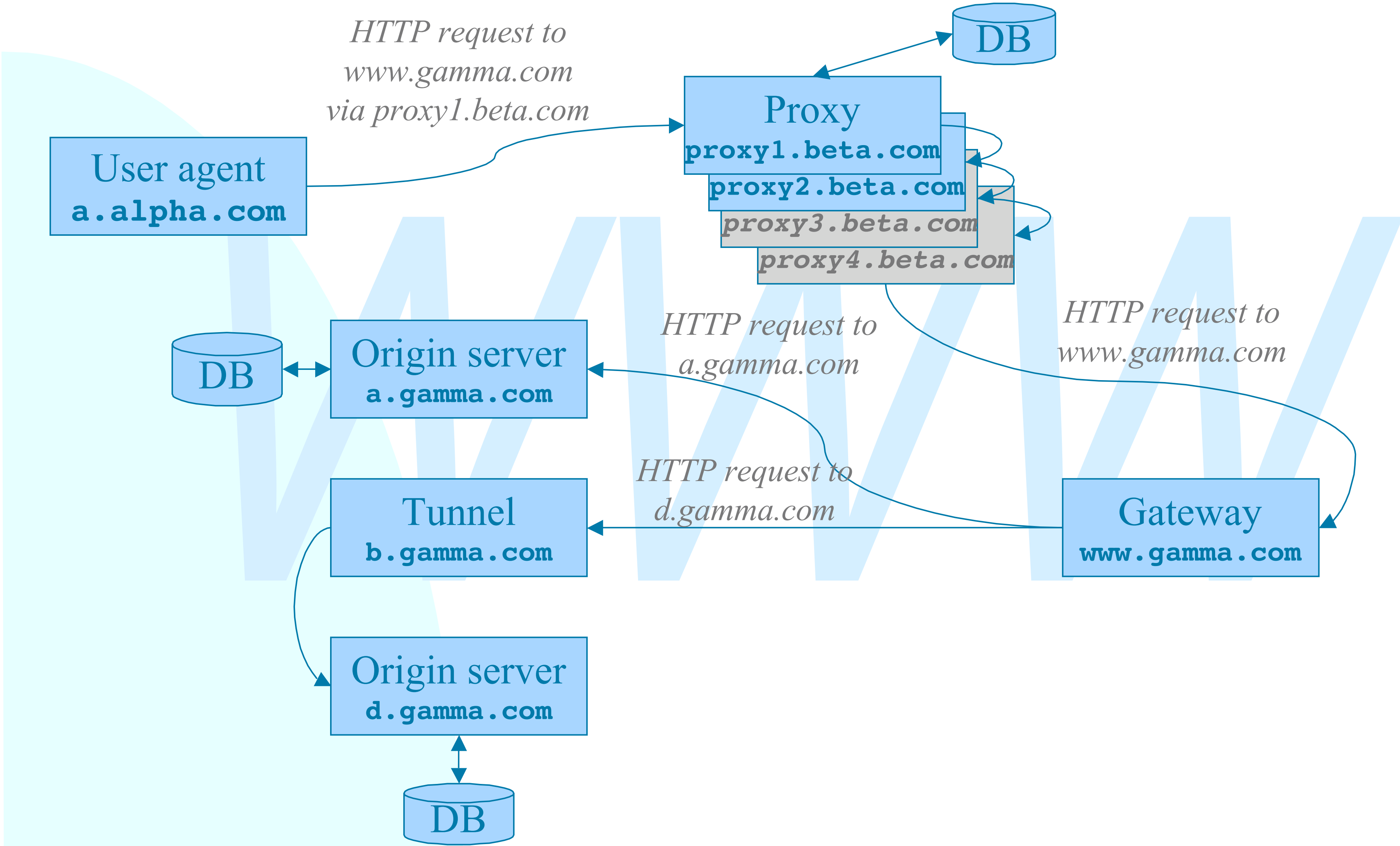
Protocollo generico

- HTTP è indipendente dal formato dati con cui vengono trasmesse le risorse. Può funzionare per documenti HTML come per binari, eseguibili, oggetti distribuiti o altre strutture dati più o meno complicate.

Statelessness

- Il server non è tenuto a mantenere informazioni che persistano tra una connessione e la successiva sulla natura, identità e precedenti richieste di un client. Il client è tenuto a ricreare da zero il contesto necessario al server per rispondere.

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.

A seguire: Ruoli delle applicazioni HTTP (3)

Ruoli delle applicazioni HTTP (3)

- **User agent**: Quel particolare client che inizia una richiesta HTTP (tipicamente un browser, ma può anche essere un bot).
  - Un bot (abbreviazione di robot) è un'applicazione automatica che richiede e scarica pagine HTML e siti web per scopi vari: indicizzazione, catalogazione, verifica di correttezza sintattica, etc. E' uno user agent anche se non vi sono utenti che serve.
- **Origin server**: il server che possiede fisicamente la risorsa richiesta (è l'ultimo della catena)

A seguire: Ruoli delle applicazioni HTTP (4)

Ruoli delle applicazioni HTTP (4)

- **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. I proxy sono trasparenti (non cambiano la risposta) o non trasparenti (possono cambiare la risposta)

Proxy trasparenti

- **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.

Proxy non trasparenti

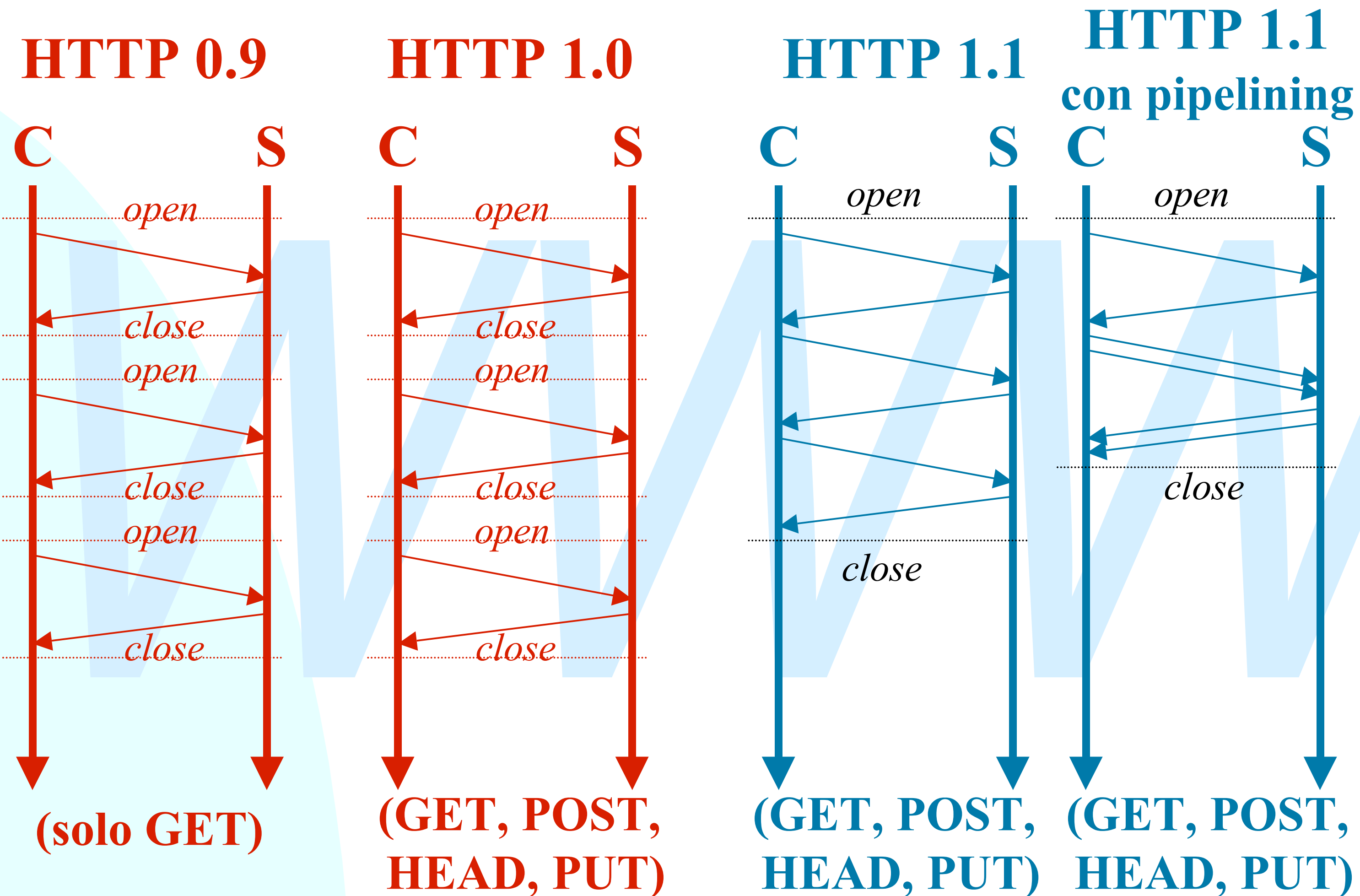
- Un proxy trasparente esegue tutte le richieste e fornisce tutte le risposte, ma in certi casi può convertire o modificare la risposta. Ad esempio fornire link a vocabolari, togliere i banner, convertire i formati ignoti, ecc.
- Ad esempio, WBI di IBM (http://www.almaden.ibm.com/cs/wbi/)

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.

A seguire: La connessione HTTP (2)

La connessione HTTP (2)



Connessioni persistenti e pipelining

Le connessioni persistenti hanno diversi vantaggi:

- Richiedono meno connessioni TCP, con vantaggio per le CPU e per la rete
- Permettono di ridurre l'attesa della visualizzazione
- Permettono di gestire in maniera migliore gli errori

Il pipelining è la trasmissione di più richieste senza attendere l'arrivo della risposta alle richieste precedenti

- Riduce ulteriormente i tempi di latenza, ottimizzando il traffico di rete, soprattutto per richieste che riguardano risorse molto diverse per dimensioni o tempi di elaborazione.
- E' fondamentale che le risposte vengano date nello stessso ordine in cui sono state fatte le richieste (HTTP non fornisce un meccanismo di riordinamento esplicito).

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) ed è ancora obbligatoria 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
```

A seguire: I metodi della richiesta

I metodi della richiesta

Noi ci occupiamo dei metodi seguenti:

- GET
- HEAD
- POST
- PUT

Ma non di

- OPTIONS
- DELETE
- TRACE
- CONNECT

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.

A seguire: Il metodo GET

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).

A seguire: Il metodo HEAD

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.

A seguire: Header dell’entità

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-Encoding, Content-Language, Content-Location, Content-MD5, Content-Range**: 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.

A seguire: Header della richiesta (1)

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.

A seguire: Header della richiesta (3)

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>
```

A seguire: Status code

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).

A seguire: Esempi di status code

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)

A seguire: Header nella risposta

Header nella risposta

La risposta contiene un body MIME che è introdotto da alcuni header e prosegue (se opportuno) con un body.

Gli header sono o generali, o dell'entità (se viene fornito un body) e specifici della risposta.

N.B.: Se viene fornita un'entità in risposta, devono esserci almeno le entità Content-type e Content-length.

- E' solo grazie al content type che lo user agent sa come visualizzare l'oggetto ricevuto.
- E' solo grazie al content length che lo user agent sa che ha ricevuto tutto l'oggetto richiesto.

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).

A seguire: Autenticazione (1)

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)

A seguire: Autenticazione (2)

Autenticazione (2)

Basic authentication

- Introdotto da HTTP 1.0.
- L'header della prima risposta WWW-Authenticate contiene il contesto di sicurezza (*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

- Introdotto da HTTP 1.1, descritto in RFC 2069 e RFC 2617.
- 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.).

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 assunza 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 (RFC 2818)
  - 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 (RFC 2660)
  - 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.

Introduzione ai cookies

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 e poi ancora RFC 2965) 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.

Il termine *cookie* (anche *magic cookie*) in informatica indica un blocco di dati opaco (i.e.: non interpretabile) lasciato in consegna ad un richiedente per poter ristabilire in seguito il suo diritto alla risorsa richiesta (come il tagliando di una lavanderia)

A seguire: Architettura dei cookie (1)

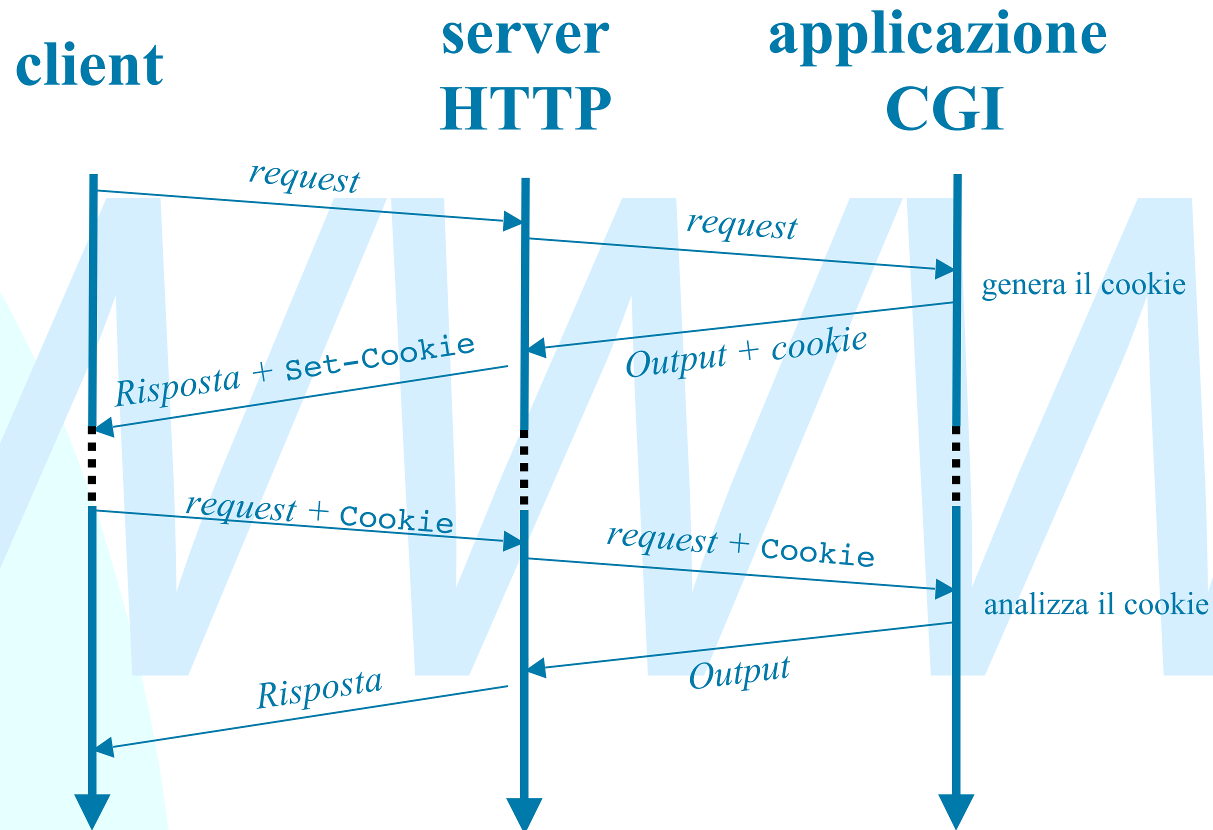
Architettura dei cookie (1)

Alla prima richiesta di uno user-agent, il server fornice la risposta ed un header aggiuntivo, il cookie, con dati arbitrari, e con la specifica di usarlo per ogni successiva richiesta.

Il server associa a questi dati ad informazioni sulla transazione. Ogni volta che lo user-agent accederà a questo sito, rifornirà i dati opachi del cookie che permettono al server di riidentificare il richiedente, e creare così un profilo ottimale.

Di particolare importanza sono la valutazione dei cookie da siti complessi (che comprendono molti domini) e l'uso di cookie di terze parti (ad esempio associati a banner o cose così).

Architettura dei cookies (2)



Architettura dei 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 rispedirlo alla prossima richiesta.
- **Cookie**: header della richiesta. Il client decide se spedirlo sulla base del nome del documento, dell'indirizzo IP del server, e dell'età del cookie.

Architettura dei 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 ilo server usando soltanto un meccanismo sicuro (es. SHTTP) per spedirlo
- **Version**: La versione della specifica a cui il cookie aderisce.

Alternative ai cookie

I cookie permettono al server di riassociare una richiesta a richieste rpecedenti (creare uno stato tra connessioni) attraverso l'uso di un pacchetto di dati opaco. Ci sono altri metodi, ma hanno tutti difetti:

- Posso associare lo stato all'indirizzo IP del richiedente. **Ma:** alcuni computer sono multi-utente, e utenti diversi condividono lo stesso IP; altri computer hanno indirizzi dinamici, e lo stesso IP può essere assegnato a computer diversi
- Posso nascondere informazioni all'interno della pagina HTML (attraverso un form nascosto). Ma: questo significa dover generare dinamicamente TUTTE le pagine, ed essere soggetti a manipolazioni semplici da parte degli utenti. Inoltre sono informazioni che rimangono associate ad una pagina specifica (un back e ho perso il contenuto del mio carrello)
- Posso complicare l'URL della pagina, inserendo dentro le informazioni di stato. Ma: si complica la gestione dei proxy, elle cache, e si rende più facilmente manipolabile la stringa opaca.

I third party cookie

Un uso subdolo (ma alcuni lo giustificano) dei cookie è l'inserimento di cookie nei banner e nelle pubblicità.

Questo permette ad un'agenzia di banner di seguire la navigazione di un utente attrvaerso tutti i siti a cui fornisce banner, e quindi fornire una profilazione più precisa del navigatore, con effetti discutibili sulla sua privacy.

RFC 2965 esplicitamente proibisce questo tipo di comportamento, che è però largamente ignorato dai produttori di browser e dai fornitori di banner.

Si aggiunga che molte versioni di browser hanno bug che permettono a codice Javascript malizioso, nascosto dentro alle pagine dei browser, di *sniffare* i contenuti dei cookie destinati ad altri domini.

Conclusioni

Oggi abbiamo parlato di

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

WWW

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 2965, October 2000
- 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
- D. Kristol, HTTP Cookies: Standards, privacy, and Politics, *ACM Transactions on Internet Technologies*, 1(2), November 2001

Fine Presentazione