



# **Tecnologie Web T**

## **Il protocollo HTTP**

Home Page del corso: <http://lia.disi.unibo.it/Courses/twt2021-info/>  
Versione elettronica: 1.03.HTTP.pdf  
Versione elettronica: 1.03.HTTP-2p.pdf

WWW = URL + HTTP + HTML

- **HTTP** è l'acronimo di **HyperText Transfer Protocol**
- È il **protocollo di livello applicativo** utilizzato per trasferire le risorse Web (pagine o elementi di pagina) da server a client
- Gestisce sia le **richieste** (URL) inviate al server che le **risposte** inviate al client (pagine)
- È un protocollo **stateless**: né il server né il client mantengono, a livello di protocollo, informazioni relative ai messaggi precedentemente scambiati
- Ci sono state diverse versioni particolarmente significative di HTTP: noi esamineremo v1.0, v1.1 e v1.1 con pipelining (a fine corso, anche qualche elemento di HTTP/2 e WebSocket)

# HTTP: Terminologia

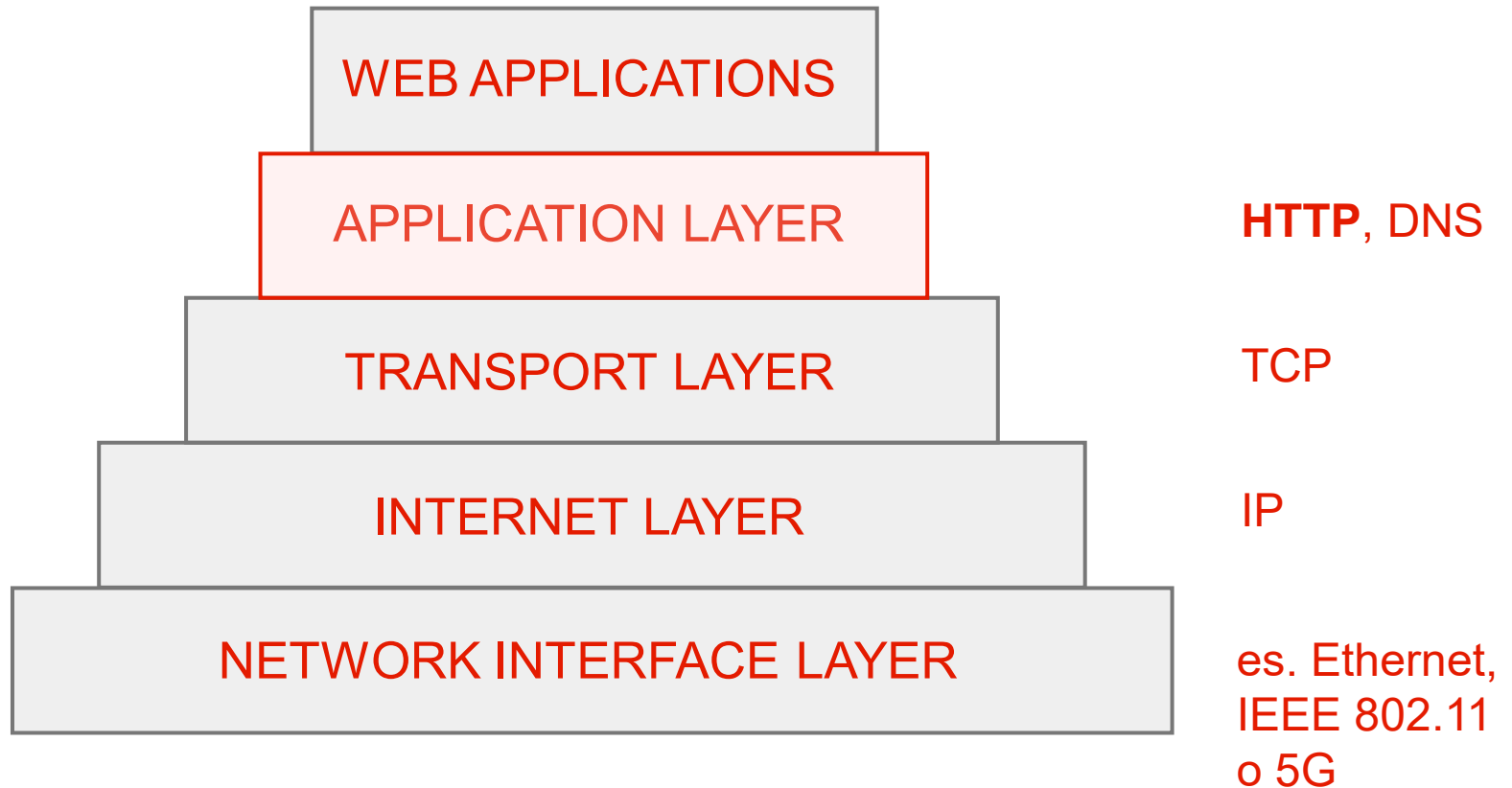
---

- **Client:** programma applicativo che stabilisce una connessione al fine di inviare delle richieste
- **Server:** programma applicativo che accetta connessioni al fine di ricevere richieste ed inviare specifiche risposte con le risorse richieste
- **Connessione:** circuito virtuale stabilito a livello di trasporto tra due applicazioni per fini di comunicazione
- **Messaggio:** è l'unità base di comunicazione HTTP, è definita come una specifica sequenza di byte concettualmente atomica
  - **Request:** messaggio HTTP di richiesta
  - **Response:** messaggio HTTP di risposta
- **Resource:** oggetto di tipo dato univocamente definito
- **URI:** Uniform Resource Identifier – identificatore unico per una risorsa
- **Entity:** rappresentazione di una risorsa, può essere incapsulata in un messaggio, tipicam. di risposta

# HTTP nello stack TCP/IP

---

HTTP si situa a livello **application** nello stack TCP/IP



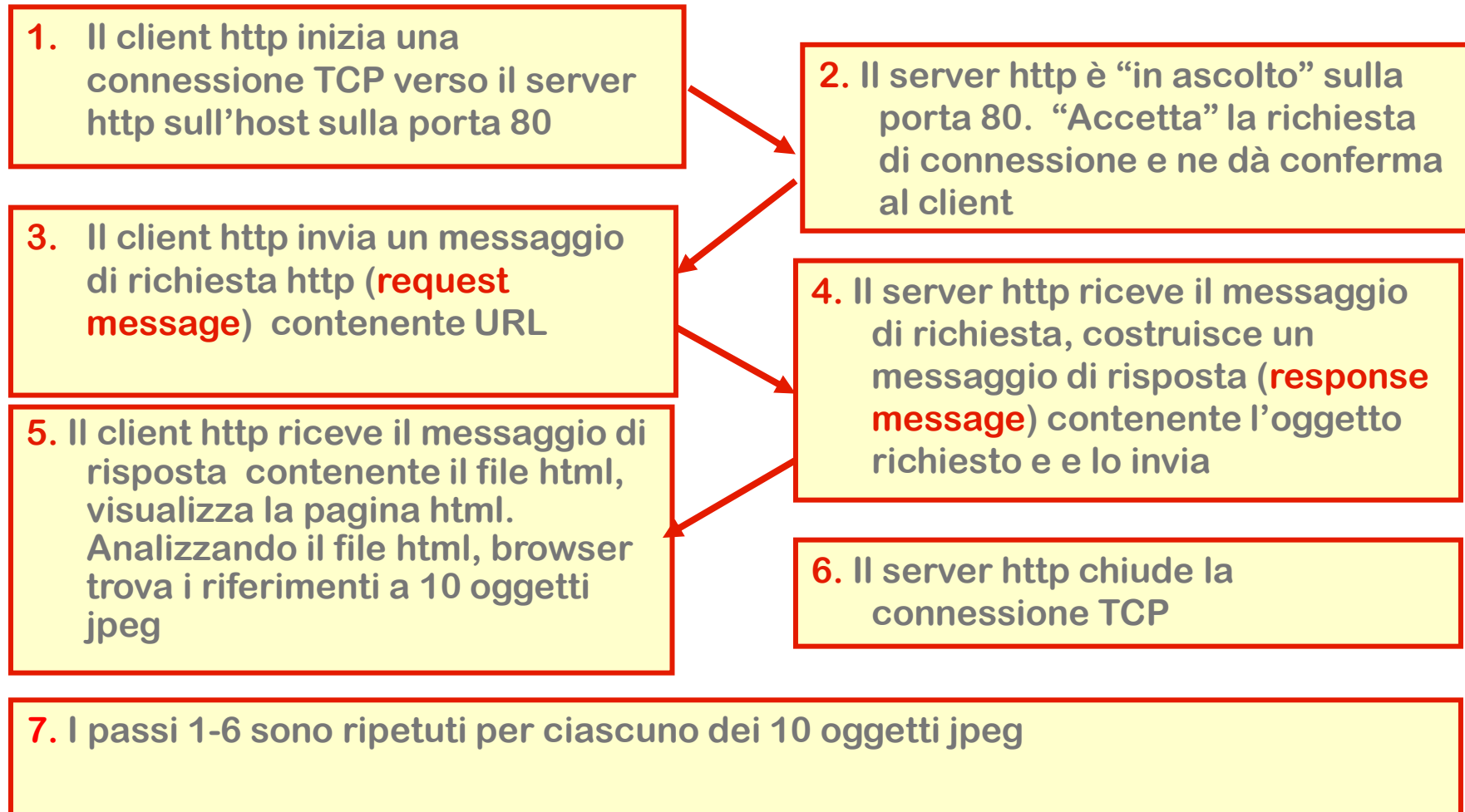
# HTTP

---

- Almeno per v1.0: *protocollo request-response, stateless, one-shot*
- È un protocollo basato su TCP
- Sia le richieste al server che le risposte ai client sono trasmesse usando stream TCP (solamente, no UDP)
- Segue uno schema di questo tipo:
  - **server** rimane in ascolto (server passivo), tipicamente sulla porta 80
  - **client** apre una connessione TCP sulla porta 80 (cliente attivo, da quale porta locale?)
  - **server** accetta la connessione (possibili più connessioni in contemporanea?)
  - **client** manda una richiesta
  - **server** invia la risposta e chiude la connessione

## Esempio HTTP (1.0): request-response, stateless, one-shot

Ipotizziamo di volere richiedere una pagina composta da un file HTML e 10 immagini JPEG:



## Differenze fra HTTP v1.0 e v1.1

---

- La stessa connessione HTTP *può essere utilizzata per una serie di richieste e una serie corrispondente di risposte*
- La differenza principale tra HTTP 1.0 e 1.1 è la possibilità di *specificare coppie multiple di richiesta e risposta nella stessa connessione*
- Le connessioni 1.0 vengono dette **non persistenti** mentre quelle 1.1 vengono definite **persistenti**
- Il server lascia aperta la connessione TCP dopo aver spedito la risposta e può quindi ricevere le richieste successive sulla stessa connessione
  - Nell'esempio precedente l'intera pagina Web (file HTML + 10 immagini ) può essere inviata sulla stessa connessione TCP
- Il server HTTP chiude la connessione quando viene *specificato nell'header del messaggio* (desiderata da parte del cliente) *oppure quando non è usata da un certo tempo (time out)*

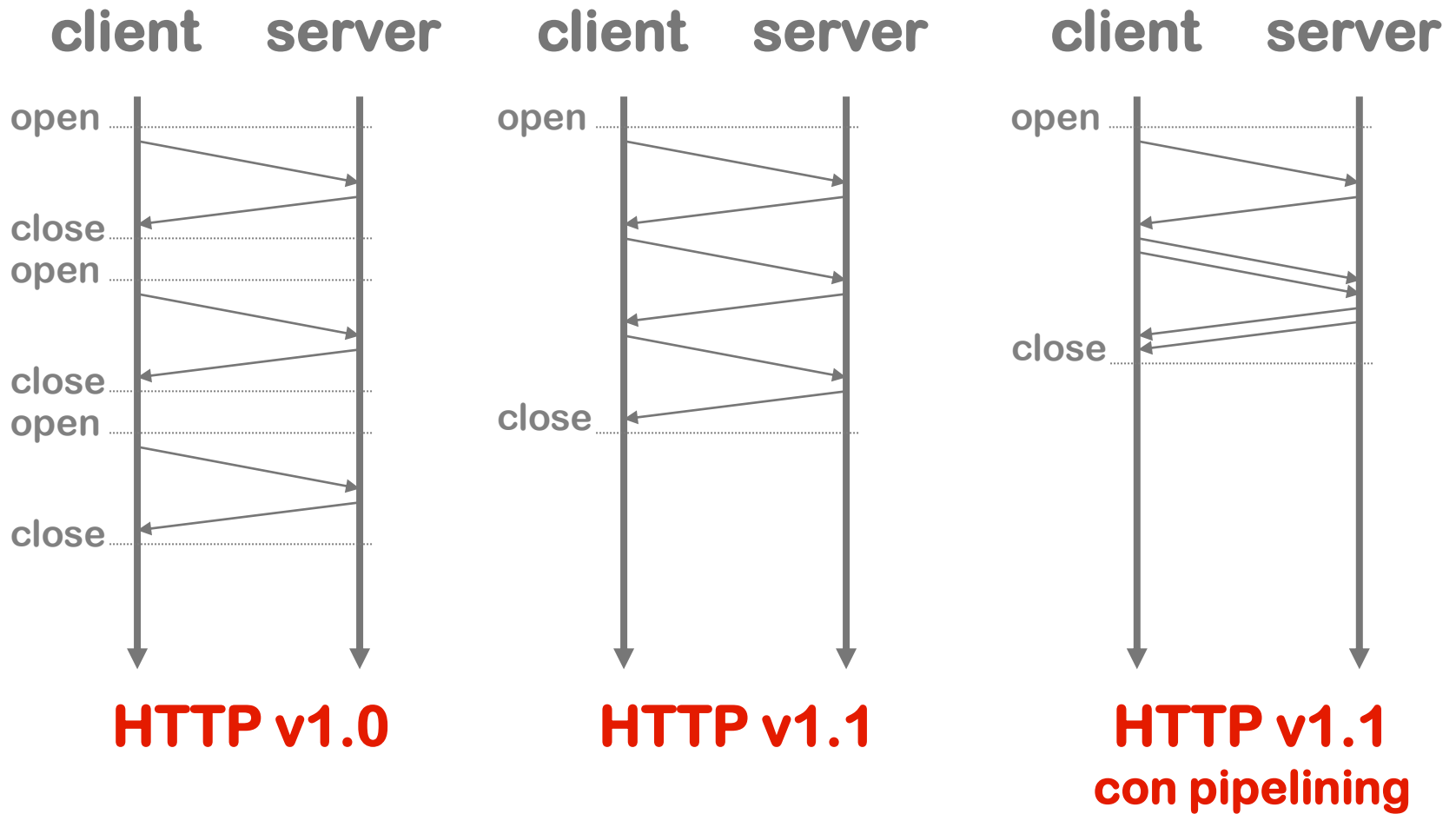
## HTTP v1.1 e pipelining

---

- Per migliorare ulteriormente le prestazioni si può usare la *tecnica del pipelining*
- Il pipelining consiste nell'invio di *molteplici richieste da parte del client prima di terminare la ricezione delle risposte*
- Le risposte debbono però essere date nello *stesso ordine delle richieste*, poiché non è specificato un metodo esplicito di associazione tra richiesta e risposta (si pensi al funzionamento di TCP al sottostante livello di trasporto)



# Confronto fra tipi di connessione



## Ulteriori versioni di HTTP: HTTP/2

---

Dopo ampia discussione e lunga battaglia, processo di standardizzazione sta giungendo a compimento

HTTP/2 sviluppato dal Working Group Hypertext Transfer Protocol (httpbis) di IETF, pubblicato come [RFC 7540](#) a Maggio 2015

**Obiettivo fondamentale di HTTP/2:**

*Miglioramento performance complessiva con full backward compatibility con HTTP 1.1*

- request-response multiplexing
- header compression
- server push

## Ulteriori versioni di HTTP: HTTP/2

---

Anche risposta al successo commerciale di **SPDY** (non sostitutivo di HTTP ma per veicolare HTTP al suo interno: multiplexing, one connection per client), protocollo HTTP-compatibile già supportato in browser Chrome, Opera, Firefox, Internet Explorer 11, ...

*HTTP/2 è basato su SPDY*, protocollo di cosiddetto open networking promosso da Google

Ne parleremo nel dettaglio nella seconda parte del corso...

Data retro-compatibilità con HTTPv1.1, la diffusione ancora ampia di HTTPv1.1 e soprattutto il suo VALORE FORMATIVO-DIDATTICO, cominciamo con il più semplice modello “tradizionale” del protocollo HTTP e con gli elementi invariante da versione a versione

# Messaggi

---

- Un messaggio HTTP è definito da due strutture:
  - **Message Header**: contiene tutte le informazioni necessarie per identificazione del messaggio (più in generale tutte le intestazioni del messaggio)
  - **Message Body**: contiene i dati trasportati dal messaggio
- Esistono schemi precisi (standard, definiti e non modificabili) per ogni tipo di messaggio relativamente a header e body
- I messaggi di *Response* contengono i dati relativi alle *risorse richieste* (tipicamente una pagina html)
- I dati sono codificati secondo il *formato specificato nell'header*
- Solitamente sono in formato *MIME* (Multipurpose Internet Mail Extensions)

# Header HTTP

---

**Gli header sono costituiti da insiemi di coppie (nome: valore) che specificano caratteristiche del messaggio trasmesso o ricevuto:**

- **Header generali della trasmissione**
  - Data, codifica, versione, tipo di comunicazione, ecc.
- **Header relativi all'entità trasmessa**
  - Content-type, Content-Length, data di scadenza, ecc.
- **Header riguardo la richiesta effettuata**
  - Chi fa la richiesta, a chi viene fatta la richiesta, che tipo di caratteristiche il client è in grado di accettare, quale autorizzazione, ecc.
- **Header della risposta generata**
  - Che server dà la risposta, che tipo di autorizzazione è necessaria, ecc.

# Messaggi HTTP: esempio di richiesta

- Il protocollo utilizza messaggi in formato ASCII (testo leggibile – quali vantaggi e quali limiti?)
- Esempio di messaggio http request:

Request line  
contiene i comandi  
(GET, POST...),  
l'oggetto e la  
versione di  
protocollo

Header lines

Il body è vuoto

```
GET /somedir/page.html HTTP/1.1
Host: www.unibo.it
Connection: close
User-agent: Chrome/37.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
```

Chiudi la connessione  
al termine della  
richiesta

## Un esempio un po' più complesso

---

```
GET /search?q=Introduction+to+XML+and+Web+Technologies HTTP/1.1
Host: www.google.com
User-Agent: Chrome/38.0 (X11; U; Linux i686; en-US; rv:1.7.2)
          Gecko/20040803
Accept: text/xml,application/xml,application/xhtml+xml,
        text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: da,en-us;q=0.8,en;q=0.5,sw;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.google.com/
```

# I comandi della richiesta - GET

---

## ▪ GET

- Serve per richiedere una risorsa ad un server
- È il metodo più frequente: 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
- È previsto il passaggio di parametri (la parte `<query>` dell'URL)
- La lunghezza massima di un URL è limitata



# I comandi della richiesta - POST

---

## ▪ POST

- Progettato come il messaggio per richiedere una risorsa
- A differenza di GET, i dettagli per identificazione ed elaborazione della risorsa stessa non sono nell'URL, ma sono contenuti nel body messaggio
- *Non ci sono limiti di lunghezza* nei parametri di una richiesta
- POST viene usato per esempio per sottomettere i dati di una form HTML ad un'applicazione CGI sul server (lo vedremo presto...)
- Si ha una trasmissione di informazioni cliente -> servitore che però non porta alla creazione di una risorsa sul server

# I comandi della richiesta - PUT e DELETE

---

## ▪ PUT

- Chiede la memorizzazione sul server di una risorsa all'URL specificato
- Il metodo PUT serve quindi per trasmettere delle informazioni dal client al server
- A differenza del POST però si ha la creazione di una risorsa (o la sua sostituzione se esisteva già)
- L'argomento del metodo PUT è la risorsa che ci si aspetta di ottenere facendo un GET con lo stesso nome in seguito

## ▪ DELETE

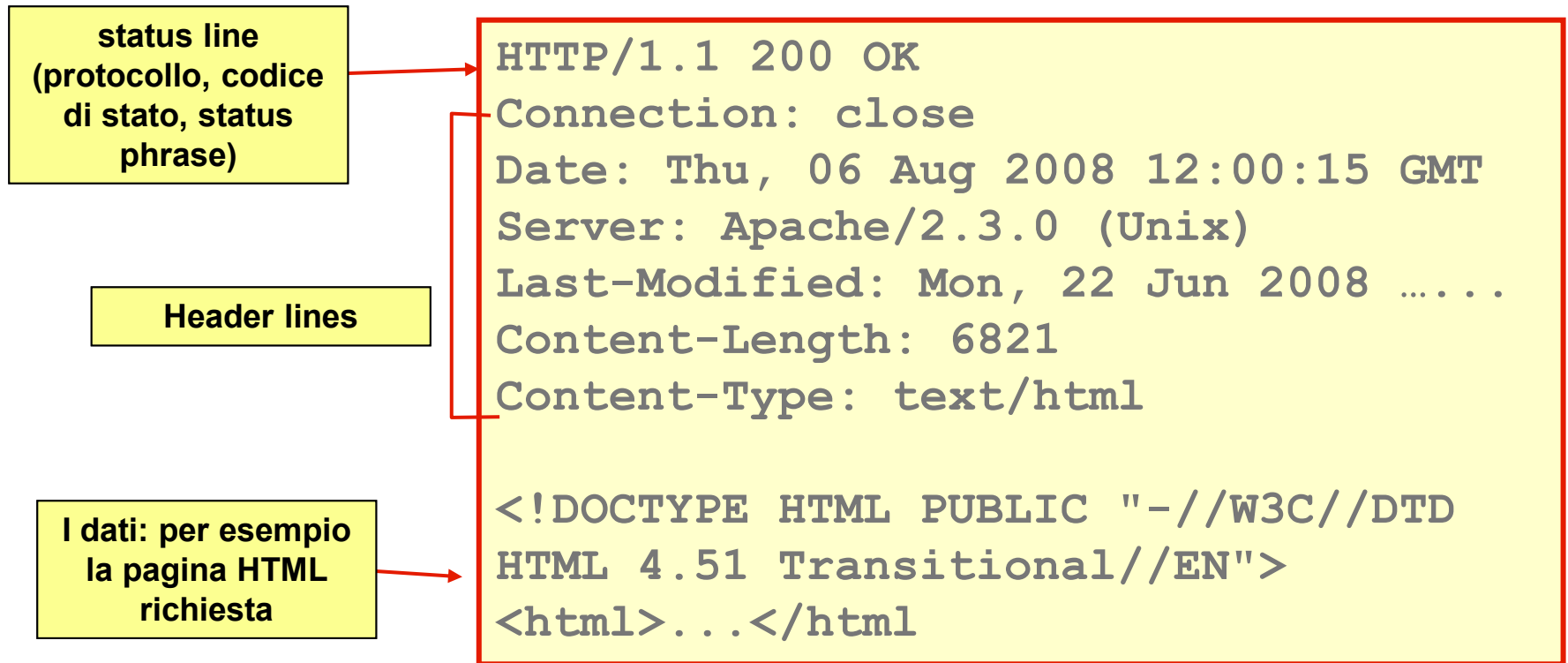
- Richiede la cancellazione della risorsa riferita dall'URL specificato
- **Sono normalmente disabilitati sui server pubblici**

## I comandi della richiesta – HEAD, OPTIONS e TRACE

---

- **HEAD:** è simile al metodo GET, ma il server deve rispondere soltanto con gli header relativi, senza body
  - Viene usato per verificare un URL
    - Validità: la risorsa esiste e non è di lunghezza zero
    - Accessibilità: non è richiesta autenticazione
- **OPTIONS:** serve per richiedere informazioni sulle opzioni disponibili per la comunicazione
- **TRACE:** è usato per invocare il loop-back remoto a livello applicativo del messaggio di richiesta
  - Consente al client di vedere che cosa è stato ricevuto dal server: viene usato nella diagnostica e nel testing dei servizi Web

# Il formato della risposta



- **HTTP 1.0:** server chiude la connessione al termine della richiesta
- **HTTP 1.1:** server mantiene aperta la connessione oppure la chiude se si mette la clausola: `Connection: close`

# I codici di stato

---

- 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
- Ci sono 5 classi:
  - **1xx: Informational.** Una risposta temporanea alla richiesta, durante il suo svolgimento (sconsigliata a partire da HTTP 1.0)
  - **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 codici di stato

---

- **100 Continue** (se il client non ha ancora mandato il body, deprecated da HTTPv1.0)
- **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)

# Representational State Transfer (REST)

---

Solo un cenno, diciamo un puntatore in avanti...

Probabilmente avete già sentito nominare REST API in altri ambiti...

Ricordatevi quanto detto su http e sulla possibilità di usare il protocollo per muovere semplici messaggi di GET, PUT, POST, DELETE, ...

Se a queste azioni, che hanno una determinata semantica sul reperimento di documenti Web statici, ne venisse associata un'altra per invocare funzionalità locali al server?

## HTTP server: quali modelli di esecuzione?

---

Vi ricordate vero la distinzione fra **server sequenziale**, **server concorrente** e **server parallelo**?

- **Soluzione concorrente:**

- Processi pesanti o thread?
- Quale cardinalità del pool di processi/thread?
- Quando effettuare generazione processi/thread?
- Code e gestione code (FIFO, con priorità, ...)
- Qualità di servizio: come ottenerla?
- Tradeoff fra costo del pool di processi/thread e latenza, ecc.

Una curiosità: in Unix numero massimo di processi generabili è stato aumentato dopo il successo di utilizzo dei primi HTTP server



## HTTP server: quali modelli di esecuzione?

---

E per server http ad altissima scalabilità, ad es.  
<http://www.google.com> ?

- Può essere sufficiente avere un pool di processi/thread su un unico server ad altissime prestazioni?
- Come andrebbe gestita opportunamente una soluzione con n server disponibili in parallelo?
- Quali problemi (sulla base di quello che già conoscete su reti di calcolatori e uso di socket TCP)?

Idee per soluzioni possibili...

## I cookie

---

- Parallelamente alle sequenze request/response, il protocollo prevede una struttura dati che si muove come un token, dal client al server e viceversa: i **cookie**
- I cookie possono essere generati sia dal client che dal server
- Dopo la loro creazione vengono sempre passati ad ogni trasmissione di request e response
- Hanno come scopo quello di fornire un supporto per il *mantenimento di stato* in un protocollo come HTTP che è essenzialmente **stateless**

# Struttura dei cookie

---

I cookie sono una collezione di stringhe:

- **Key:** identifica univocamente un cookie all'interno di un dominio:path
- **Value:** valore associato al cookie (è una stringa di max 255 caratteri)
- **Path:** posizione nell'albero di un sito al quale è associato (di default /)
- **Domain:** dominio dove è stato generato
- **Max-age:** (opzionale) numero di secondi di vita (permette la *scadenza di una sessione*, ne parleremo a breve...)
- **Secure:** (opzionale) non molto usato. Questi cookie vengono trasferiti se e soltanto se il protocollo è sicuro (https)
- **Version:** identifica la versione del protocollo di gestione dei cookie

# Autenticazione

---

- Esistono situazioni in cui si vuole restringere l'accesso alle risorse ai soli utenti abilitati
- Tecniche comunemente utilizzate
  - Filtro su set di indirizzi IP
  - Form per la richiesta di username e password
  - HTTP Basic
  - HTTP Digest

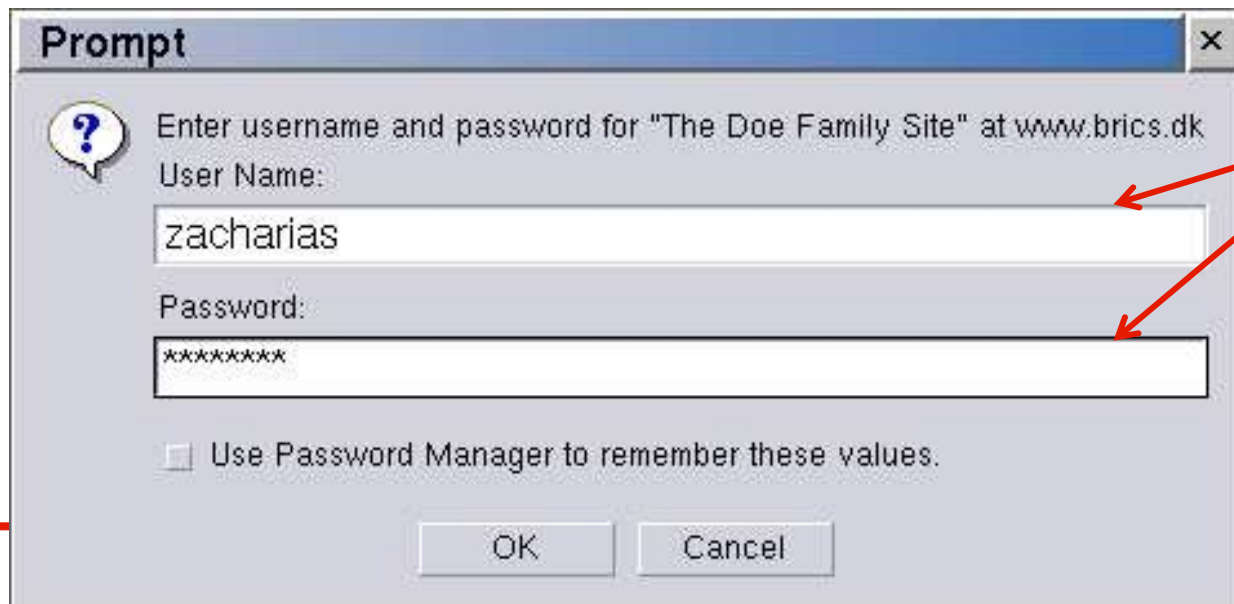
## Riconoscimento dell'indirizzo IP

---

- **Basare l'autenticazione sull'indirizzo IP del client è una soluzione che presenta vari svantaggi:**
  - Non funziona se l'indirizzo non è pubblico (vedi esempio dei NAT)
  - Non funziona se l'indirizzo IP è assegnato dinamicamente (es. DHCP)
  - Esistono tecniche che consentono di presentarsi con un IP fasullo (spoofing)
- **L'autenticazione HTTP Digest è caduta in disuso negli ultimi anni** (invio di password dopo hash)
- **Normalmente si usano**
  - **Form**
  - **HTTP Basic**

# Autenticazione HTTP Basic

- **Challenge (da parte del server):**  
HTTP/1.1 401 Authorization Required  
WWW-Authenticate: Basic realm="Secure Area"
- **Response (da parte del cliente):**  
GET /private/index.html HTTP/1.1  
Host: localhost  
Authorization: Basic QWxhZGRpbjpvYVUyIHNlc2FtZQ==



Prompt

Enter username and password for "The Doe Family Site" at www.brics.dk

User Name:  
zacharias

Password:  
\*\*\*\*\*

Use Password Manager to remember these values.

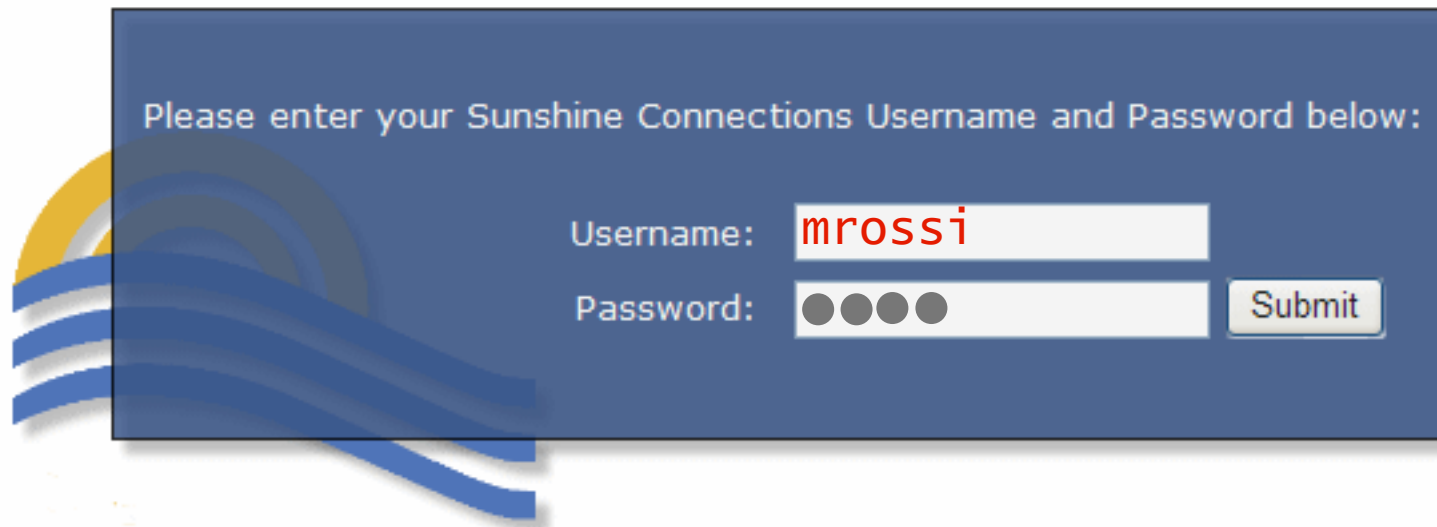
OK Cancel

Testo in chiaro  
(username:password)  
codificato in Base64

# Autenticazione Form

---

- Normalmente si usa il metodo POST
- Analoghe considerazioni a quelle fatte per HTTP Basic



Please enter your Sunshine Connections Username and Password below:

Username:

Password:

# Sicurezza

---

- **Proprietà desiderabili**

- **Confidenzialità**

- **Integrità**

- **Autenticità**

- **Non Ripudio**

**SSL/TLS**

## Sicurezza del canale di trasporto:

- **SSL: Secure Sockets Layer**

- **TLS: Transport Layer Security**

- Sostituisce SSL, originariamente sviluppato da Netscape

- È alla base di HTTPS



# SSL/TLS

---

- Viene posto un livello che si occupa della gestione di confidenzialità, autenticità ed integrità della comunicazione fra HTTP e TCP
- **Accediamo tramite**  
`https://...`
- Basato su crittografia a chiave pubblica
  - private key + public key
  - certificato (in genere usato per autenticare il server)

**Per quelli che proseguiranno con LM, vedrete ampi dettagli in Sicurezza dell'Informazione M...**

# Architetture più distribuite e articolate per il Web

---

Senza cambiare nulla del semplicissimo HTTP...

- **Proxy**: Programma applicativo in grado di agire sia come Client che come Server al fine di effettuare richieste per conto di altri Clienti. Le Request vengono processate internamente oppure vengono ridirezionate al Server. Un proxy deve interpretare e, se necessario, *riscrivere le Request* prima di inoltrarle
- **Gateway**: Server che agisce da intermediario per altri Server. Al contrario dei proxy, il gateway riceve le request come se fosse il server originale e Client non è in grado di identificare che Response proviene da un gateway. Detto anche *reverse proxy* o *server-side proxy*
- **Tunnel**: Programma applicativo che agisce come “blind relay” tra due connessioni. Una volta attivo (in gergo “salito”) non partecipa alla comunicazione HTTP

## Ad esempio per caching distribuito

---

- Idea di base: memorizzare copie temporanee di documenti Web (es. pagine HTML, immagini) al fine di ridurre l'uso della banda ed il carico sul server
- Una Web cache memorizza i documenti che la attraversano. L'obiettivo è usare i documenti in cache per le successive richieste qualora alcune condizioni siano verificate
- Tipi di Web cache
  - User Agent Cache
  - Proxy Cache

# User Agent Cache

---

- Lo user agent (tipicamente il browser) mantiene una cache delle pagine visitate dall'utente
- L'uso delle user agent cache era molto importante in passato quando gli utenti non avevano accesso a connessioni di rete a banda larga
- Questo modello di caching è comunque ora molto rilevante per i dispositivi mobili al fine di consentire agli utenti di lavorare con connettività intermittente ma anche per ridurre latenze dovute a caricamento di elementi statici (icone, sfondi, ...)  
*Anche strumenti addizionali, es. Google Gears, si basano su questo concetto*

# Proxy Cache

---

- **Forward Proxy Cache**

- Servono per ridurre le necessità di banda
- Es. rete locale aziendale, Università, ecc.
- Il proxy intercetta il traffico e mette in cache le pagine
- Successive richieste non provocano lo scaricamento di ulteriori copie delle pagine al server

- **Reverse (o server-side) Proxy Cache**

- Gateway cache
- Operano per conto del server e consentono di ridurre il carico computazionale delle macchine
- I client non sono in grado di capire se le pagine arrivano dal server o dal gateway
- Internet Caching Protocol per il coordinamento fra diverse cache. Base per content delivery network (sapete farmi qualche esempio di content delivery network – CDN - che conoscete?)

# HTTP e Cache

---

HTTP definisce vari meccanismi che possono avere *effetti collaterali positivi* per la gestione «lazy» dell'aggiornamento cache

- **Freshness**: controllata lato server da *Expires response header* e lato cliente da *direttiva Cache-Control: max-age*
- **Validation**: può essere usato per controllare se un elemento in cache è ancora corretto, ad es. nel caso in cui sia in cache da molto tempo (ad es. tramite richieste HEAD)
- **Invalidation**: è normalmente un effetto collaterale di altre request che hanno attraversato la cache. Se per esempio viene mandata una POST, una PUT o una DELETE a un URL il contenuto della cache deve essere e viene automaticamente invalidato