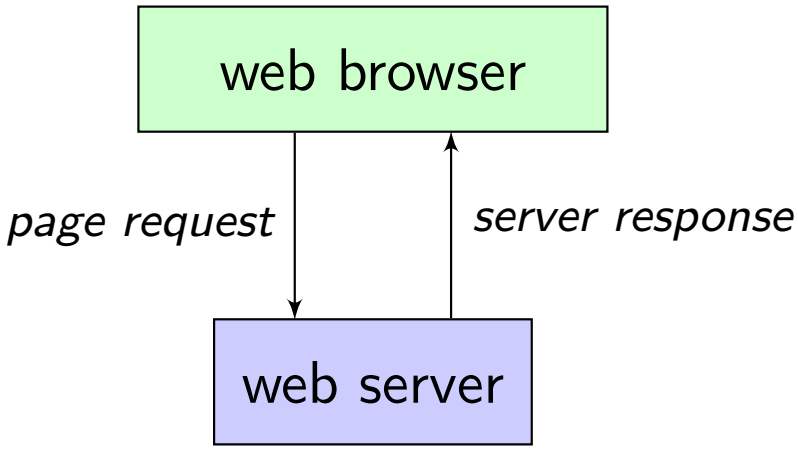
Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali
Anno Accademico 2013/2014

Sicurezza delle applicazioni web: protocollo HTTP

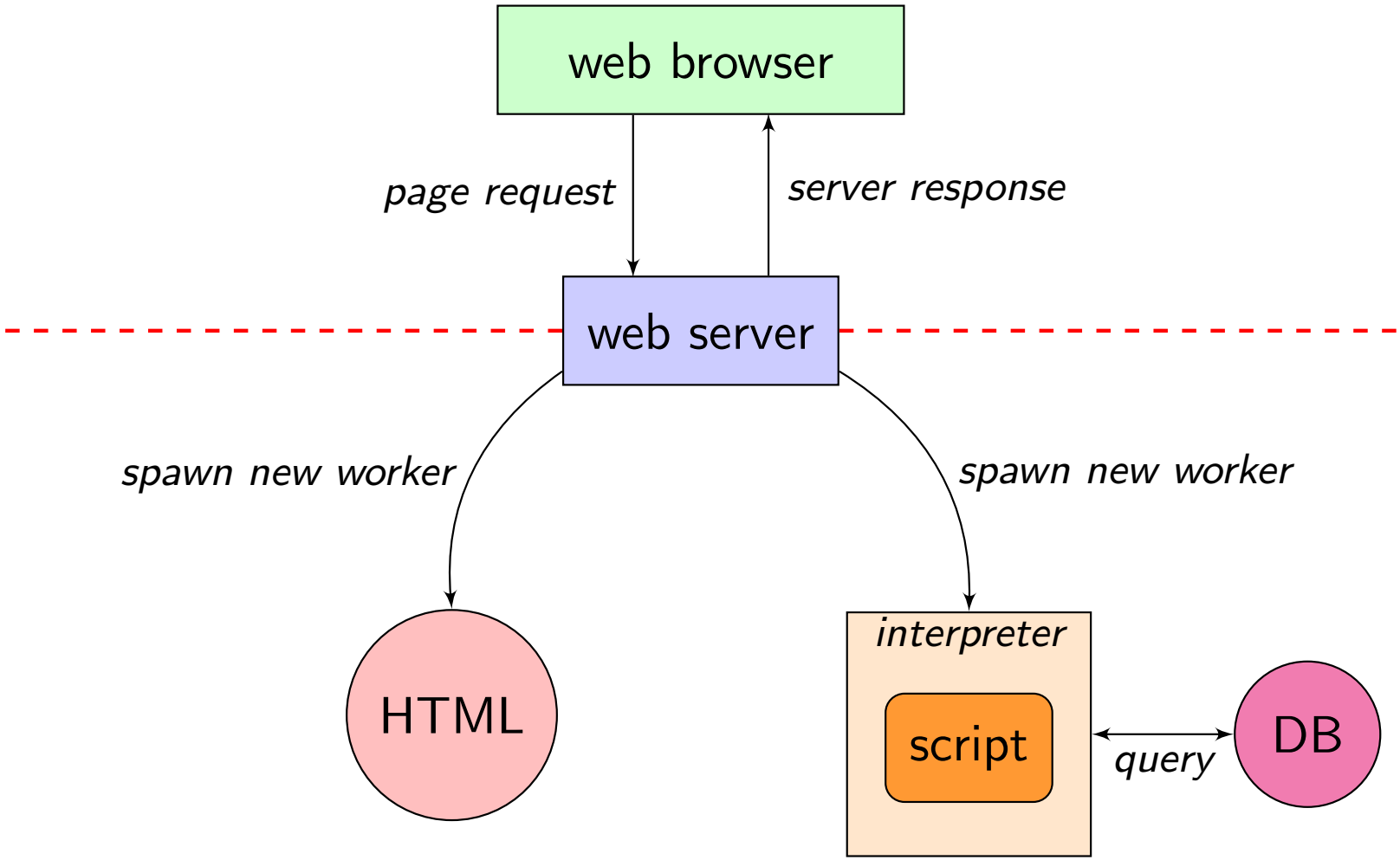
Andrea Lanzi

14 Maggio 2014

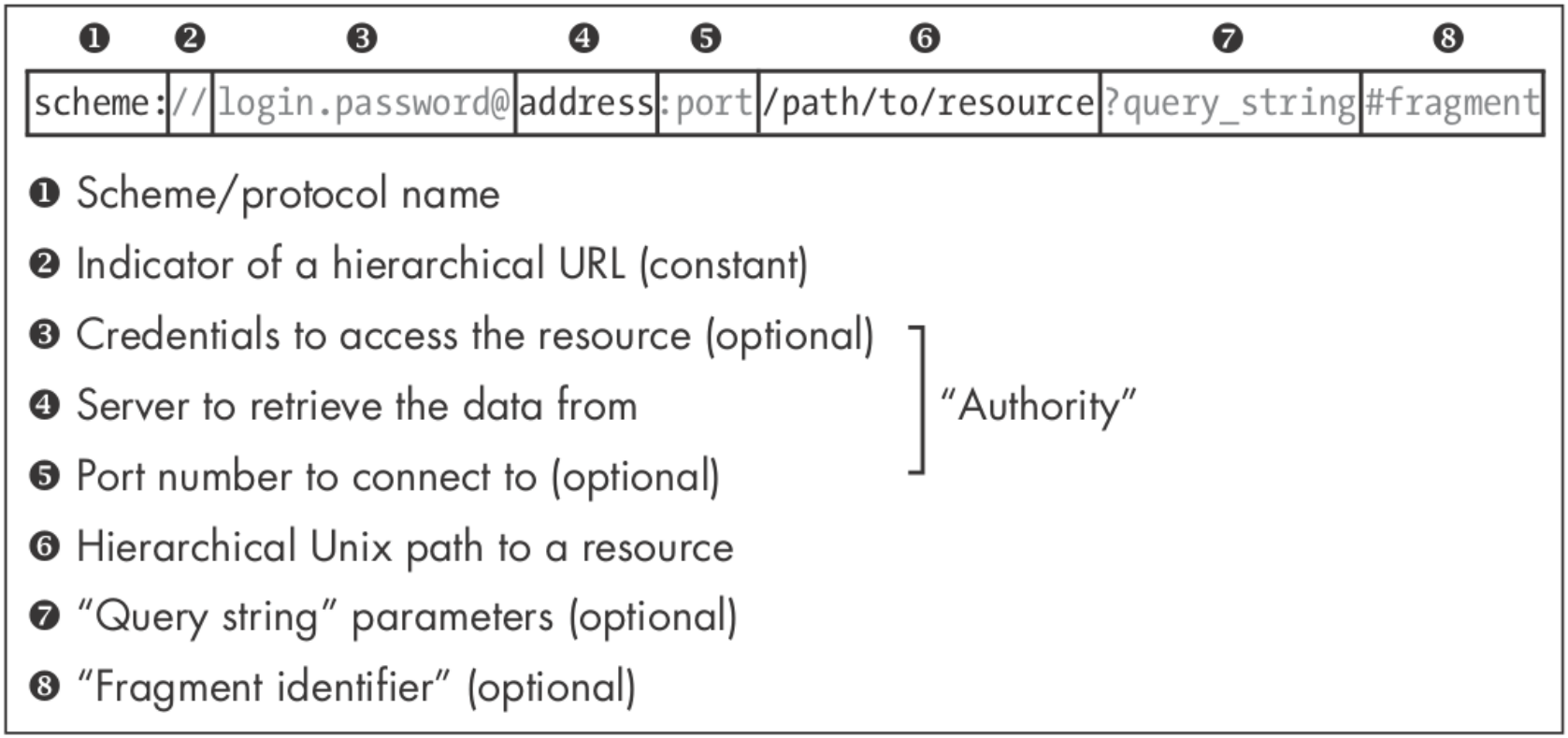
Architettura infrastruttura web



Architettura infrastruttura web



URL: struttura



FONTE: "The Tangled Web", di Michael Zalewksi, ED. No Starch Press, 2011.

HTTP: HyperText Transfer Protocol

- elemento fondamentale su cui si basa il Web
- protocollo di *livello applicazione* usato per trasferire dati tra *client* ed *web server*
- protocollo *text-based*, stateless
- in uso le versioni 1.0 (RFC 1945[1]) e 1.1 (RFC 2616[2])
- incapsulato all'interno di connessioni TCP, di default su porta 80
- obiettivo originale: trasmissione di documenti HTML
- oggi utilizzato anche per trasportare altri file ed informazioni (e.g, SOAP)

[1] $\approx$ 50 pagine di documentazione
[2] $\approx$ 150 pagine di documentazione

HTTP/0.9

Sviluppato nel 1991 da Tim Berners-Lee (draft di $\approx$ 1.5 pagine)

- CLIENT invia:
  1. `GET` + `path/to/resource`
  2. `address` + `?query_string`
  3. `CRLF` [ASCII: 0x0D 0x0A]
- SERVER risponde:
  1. `HTML` payload

Estremamente limitato, numerose lacune:

- Come specificare la lingua di preferenza dell'utente?
- Come fa il server a comunicare che il file non è presente?
- Come inviare un file che non è in formato HTML?
- Come gestire *virtual servers*?

HTTP/1.0 e HTTP/1.1

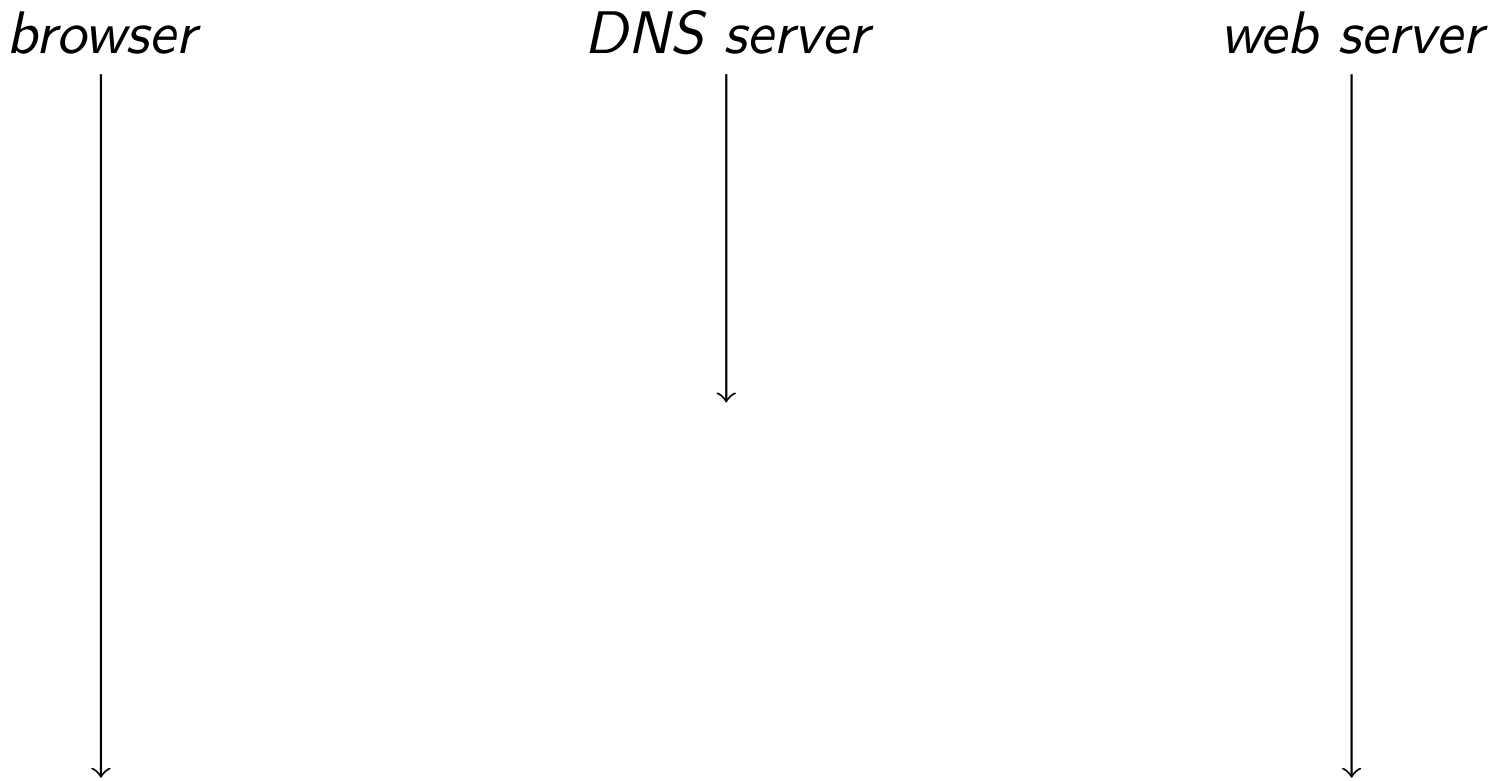
Definiti rispettivamente nel 1996 e nel 1999.

- CLIENT invia:
  1. `scheme` + `path/to/resource?query_string` + Protocol-version
  2. Headers [name: value]
  3. empty line
  4. payload[3]
- SERVER risponde:
  1. Supported-protocol-version + Status-code + Status-message
  2. Headers
  3. content

[3] opzionale, la dimensione di *payload* deve venire indicata nel header *Content-Length*

Richiesta pagina `http://www.example.com/test.html`

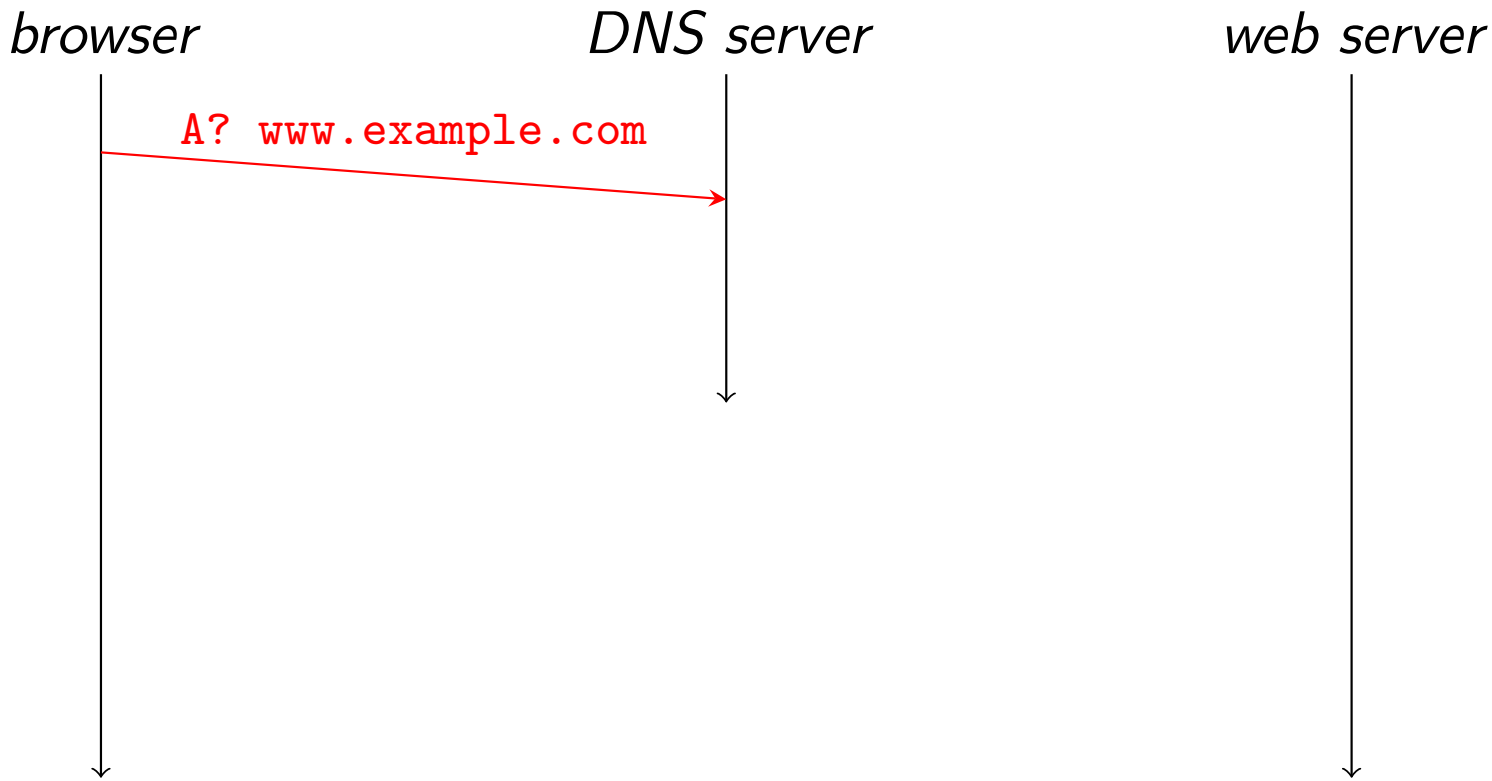
Richiesta pagina http://www.example.com/test.html



Client $\leftrightarrow$ server DNS

- il browser interroga un server DNS per ottenere l'indirizzo IP del server web

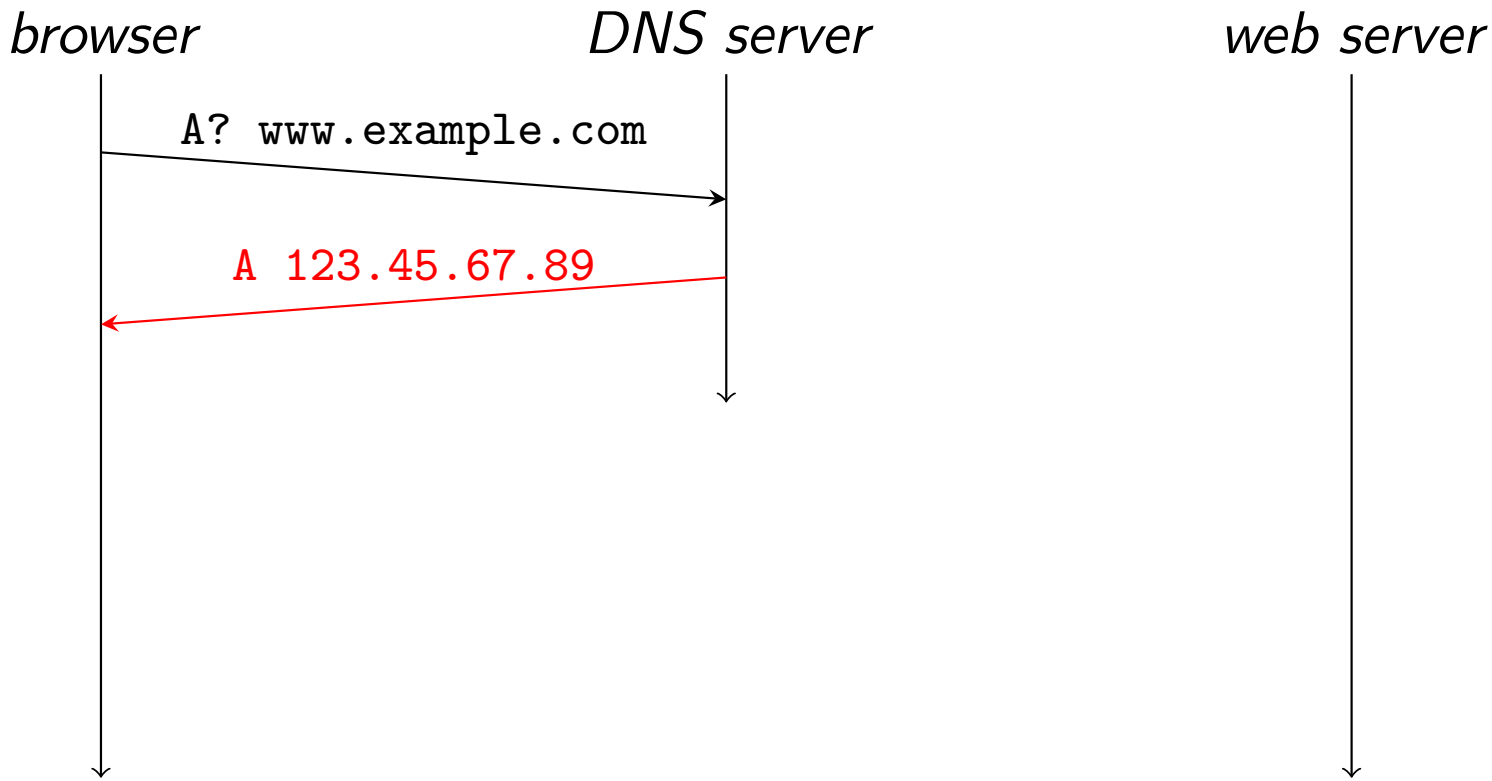
Richiesta pagina http://www.example.com/test.html



Client ↔ server DNS

- il browser interroga un server DNS per ottenere l'indirizzo IP del server web

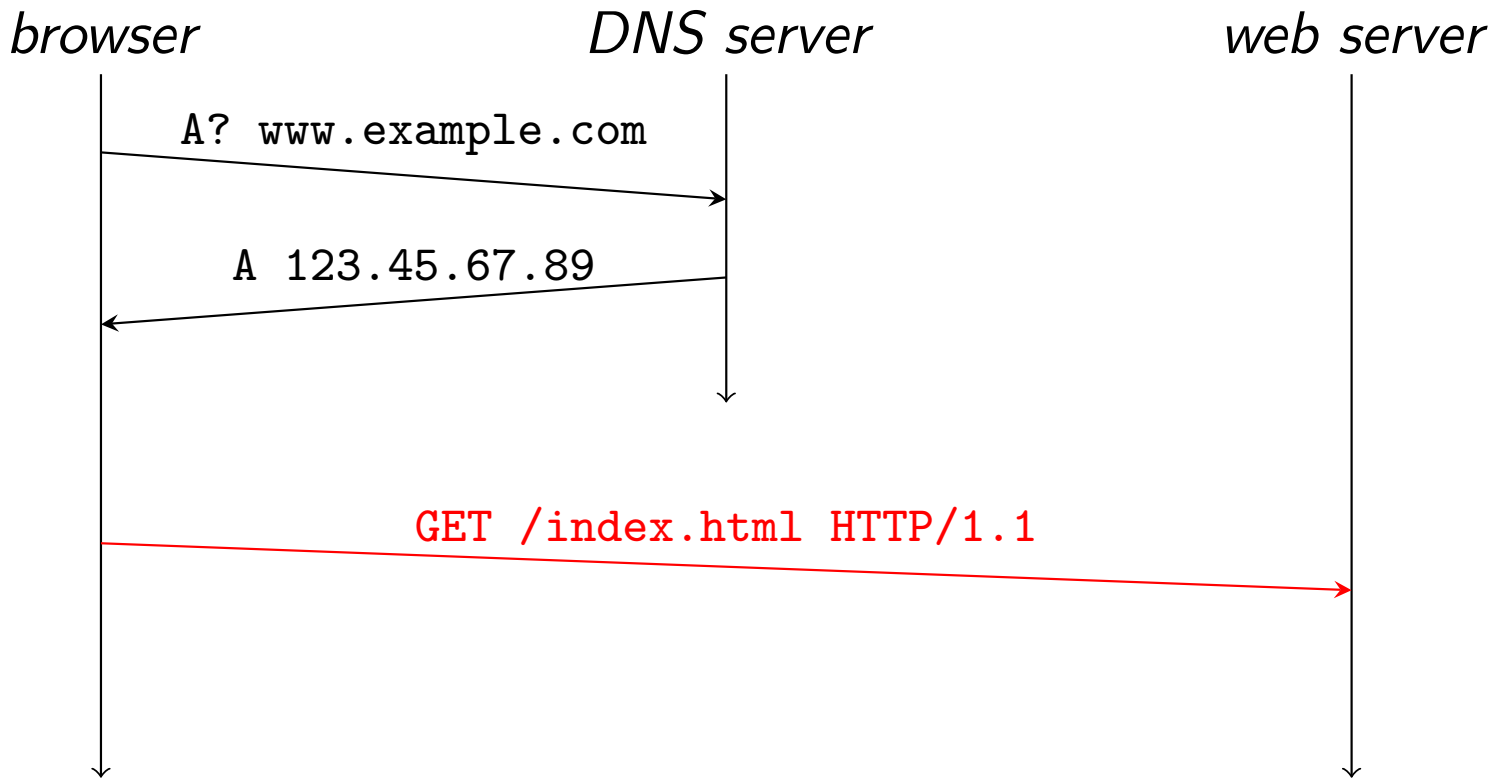
Richiesta pagina http://www.example.com/test.html



Client ↔ server DNS

- il browser interroga un server DNS per ottenere l'indirizzo IP del server web

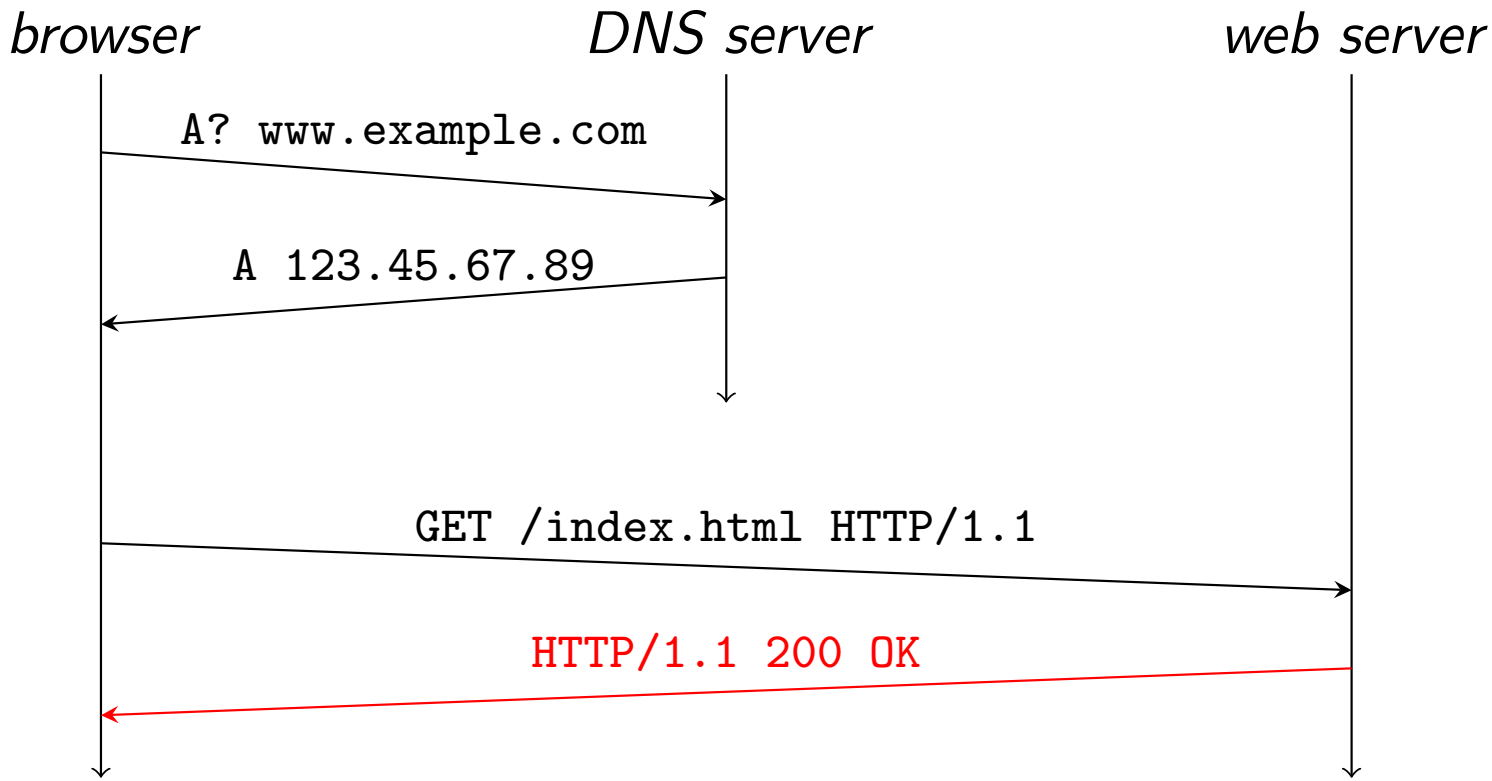
Richiesta pagina http://www.example.com/test.html



Client → server web

- il browser si collega alla porta TCP 80 del server e invia una richiesta HTTP

Richiesta pagina http://www.example.com/test.html



Client ← server web

- il server web processa la richiesta ricevuta e restituisce una riposta (e.g., la pagina HTML)

Richieste HTTP

Struttura

1. **request line** (e.g., GET /index.html HTTP/1.1)
2. **header** (*opzionali*, e.g., User-Agent: Mozilla/5.0 (X11; U; Linux i686)
3. linea vuota
4. corpo del messaggio (*opzionale*)

Note

- `request line` e `header` sono terminati da `CRLF` (*carriage return* + *line feed*: "\r\n")
- la linea vuota è formata da `CRLF`
- spesso le implementazioni sono piuttosto flessibili (e.g., richieste accettate anche con linee terminate dal solo `LF`)
- con `HTTP 1.1` tutti gli `header`, tranne `Host`, sono opzionali

Esempio richiesta HTTP (GET)

```
GET / HTTP/1.1
Host: securitytraps.no-ip.pl
User-Agent: Mozilla/5.0 ...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Proxy-Connection: keep-alive
```

Esempio richiesta HTTP (POST)

```
POST /?d=pre HTTP/1.1
Host: securitytraps.no-ip.pl
User-Agent: Mozilla/5.0 ...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Proxy-Connection: keep-alive
Referer: http://securitytraps.no-ip.pl/?d=pre
Cookie: lang=en; PHPSESSID=iivcvpia24gtarfh1g3irsd471
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

pass=readonly%2C+sorry+%3B%3E
```

Risposte HTTP

Struttura

1. **status-line** (e.g., `HTTP/1.1 200 OK`)
2. **header** (*opzionali*) (e.g., `Server: Apache/2.2.14 (Ubuntu)`)
3. linea vuota
4. corpo del messaggio (*opzionale*)

Esempio

```
HTTP/1.1 200 OK
Date: Mon, 02 May 2011 20:19:15 GMT
Server: Apache/2.2.14 (Ubuntu)
X-Powered-By: PHP/5.3.2-1ubuntu4.8
Expires: Thu, 19 Nov 1981 08:52:00 GMT
...
Vary: Accept-Encoding
Content-Type: text/html
Content-Length: 3259

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
...
```

GET

È il metodo "originario" definito in HTTP/0.9.
In base al RCF, le richieste tramite questo metodo non dovrebbero avere *"significance of taking an action other than retrieval"*; tuttavia esso consente di inviare dati al server per mezzo della `query_string`.
Secondo la specifica del protocollo HTTP di questo metodo i dati inviati al server sono preceduti dall'indirizzo della pagina richiesta e un punto interrogativo.

Esempio:

```
GET /comments.pl?sid=3756217&op=reply&mode=thread HTTP/1.1
Host: tech.slashdot.org
...
```

GET

È il metodo "originario" definito in HTTP/0.9.
In base al RCF, le richieste tramite questo metodo non dovrebbero avere *"significance of taking an action other than retrieval"*; tuttavia esso consente di inviare dati al server per mezzo della `query_string`.
Secondo la specifica del protocollo HTTP di questo metodo i dati inviati al server sono preceduti dall'indirizzo della pagina richiesta e un punto interrogativo.

Esempio:

```
GET /comments.pl?sid=3756217&op=reply&mode=thread HTTP/1.1
Host: tech.slashdot.org
...
```

Caso 1: passaggio parametri tramite *form*

```
<form action="submit.php" method="get">
   <input type="text" name="var1" />
   <input type="hidden" name="var2" value="b" />
   <input type="submit" value="invia" />
</form>
```

Caso 2: parametri *embedded* nel URL

```
<a href="submit.php?var1=a&var2=b">link</a>
```

Richiesta corrispondente

```
GET /submit.php?var1=a&var2=b HTTP/1.1
Host: www.example.com
...
```

Caso 1: passaggio parametri tramite *form*

```
<form action="submit.php" method="get">
   <input type="text" name="var1" />
   <input type="hidden" name="var2" value="b" />
   <input type="submit" value="invia" />
</form>
```

Caso 2: parametri *embedded* nel URL

```
<a href="submit.php?var1=a&var2=b">link</a>
```

Richiesta corrispondente

```
GET /submit.php?var1=a&var2=b HTTP/1.1
Host: www.example.com
...
```

POST

POST è un metodo per inviare dati usando il protocollo HTTP. Secondo la specifica del protocollo HTTP i dati sono inviati dopo che tutti gli header sono stati inviati dal *client* al *server*. Ciascuna richiesta POST è accompagnata dal header *Content-Length* che specifica la dimensione del payload[a].

[a] generalmente nel formato URL-encoded oppure MIME-encoded

Ex.1: parametri POST

```
<form action="submit.php" method="post">
   <input type="text" name="var1" />
   <input type="text" name="var2" />
   <input type="submit" value="invia" />
</form>
```

```
POST /submit.php HTTP/1.1
Host: localhost
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

var1=a&var2=b
```

Ex.1: parametri POST

```
<form action="submit.php" method="post">
   <input type="text" name="var1" />
   <input type="text" name="var2" />
   <input type="submit" value="invia" />
</form>
```

```
POST /submit.php HTTP/1.1
Host: localhost
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

var1=a&var2=b
```

E se...

provassimo a fare contemporaneamente POST+GET?

Ex.1: parametri POST

```
<form action="submit.php" method="post">
   <input type="text" name="var1" />
   <input type="text" name="var2" />
   <input type="submit" value="invia" />
</form>
```

```
POST /submit.php HTTP/1.1
Host: localhost
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

var1=a&var2=b
```

Ex.2: GET + POST

```
<form action="test.php?var3=c&var4=d"
  method="post">
   <input type="text" name="var1" />
   <input type="text" name="var2" />
   <input type="submit" value="invia" />
</form>
```

```
POST /test.php?var3=c&var4=d HTTP/1.1
Host: localhost
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

var1=a&var2=b
```

HTTP Request Types
Metodi HEAD-OPTIONS-PUT-DELETE

`GET` e `POST` sono i metodi più comunemente utilizzati, ma non sono gli unici previsti dal protocollo `HTTP/1.1`.

HEAD

In risposta alla richiesta `HEAD` in server invia al client *solo* gli Header senza il payload.

OPTIONS

Restituisce tutti i metodi supportatati per uno specifico URL.

PUT-DELETE

Sviluppati per effettuare il *upload* e la rimozione dei file dal server.

TRACE

Restituzione di informazioni su proxy intermedi + funzione "echo".

Server Response Codes

RFC 2616 definisce quasi 50 *status codes* che il server può inviare in risposta al client; tuttavia in pratica solo 1/3 di essi è effettivamente utilizzato.

200-299: Success

"200 OK", "204 No Content" sono i più comuni; indicano che la richiesta è andata a buon fine.

300-399: Redirection

"301 Moved Permanently", "302 Found", "303 See Other", "304 Not Modifed" indicano al browser di ri-provare a inviare al richiesta al URL contenuto nel header *Location* o che la risorsa non ha subito modifiche.

400-499: Client-Side Error

"400 Bad Request", "401 Unauthorized", "403 Forbidden", "404 Not Found" indicano condizioni di errore dovute alla richiesta (sbagliata) del client.

500-599: Server-Side Error

"500 Internal Server Error", "503 Service Unavailable" indicano che il server non può soddisfare la richiesta del client a causa di un errore interno.

Keep-alive Sessions

Originariamente 1 sessione HTTP == 1 connessione TCP.

È un sistema efficiente (e praticabile ancor' oggi)?

Keep-alive Sessions

Originariamente 1 sessione HTTP == 1 connessione TCP.

È un sistema efficiente (e praticabile ancor' oggi)?

Keep-alive Sessions

Originariamente 1 sessione HTTP == 1 connessione TCP.

È un sistema efficiente (e praticabile ancor' oggi)?

PROBLEMA: three-step TCP handshake da completare ogni volta (fork e nuovo processo su sistema UNIX avviato per ogni connessione).
SOLUZIONE: "riciclare" la *connessione TCP corrente*!.

Uso di header *Content-Length* sia per client `request` che per server `response`.
Connessioni `keep-alive` vengono utilizzate di default in HTTP/1.1

Chunked Data Transfers

In sessioni `keep-alive` il server deve conoscere a priori la dimensione del corpo del messaggio inviato in risposta al client. Cosa succede se le informazioni vengono generate dinamicamente (ES. video streaming)?

Il payload viene inviato a pezzi; la risposta del server contiene un apposito header ‘‘`Transfer-encoding: chunked`’’ seguito dalla dimensione di ciascun pezzo.

Esempio di risposta con uso di chunks:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
...
5
Hello
6
world!
0
```

Problema

- *stateless*: ogni richiesta è indipendente dalle precedenti
- le applicazioni web dinamiche richiedono il concetto di *sessione*
- **come fare?**

Cookie

- Dati creati dal server e memorizzati sul client
- Trasmessi tra client e server utilizzando appositi header HTTP
- SERVER usa header
  `Set-Cookie: NAME = VALUE *(; cookie-av)<CRLF>`
- CLIENT usa header
  `Cookie: NAME = VALUE [; path] [; domain]<CRLF>`
- cookie standardizzati in RFC 2109 ("*HTTP State Management Mechanism*")

Problema

- *stateless*: ogni richiesta è indipendente dalle precedenti
- le applicazioni web dinamiche richiedono il concetto di *sessione*
- **come fare?**

Cookie

- Dati creati dal server e memorizzati sul client
- Trasmessi tra client e server utilizzando appositi header HTTP
- SERVER usa header
  `Set-Cookie: NAME = VALUE *(; cookie-av)<CRLF>`
- CLIENT usa header
  `Cookie: NAME = VALUE [; path] [; domain]<CRLF>`
- cookie standardizzati in RFC 2109 ("*HTTP State Management Mechanism*")

Sessioni HTTP

Una *sessione* permette di gestire l'interazione tra client e server web (*stateful*).
L'identificativo viene creato dal server, quindi condiviso con il client.

Caratteristiche

- informazioni e stato devono essere memorizzati
- ogni richiesta HTTP deve contenere un identificativo di sessione
- le sessioni devono avere un timeout

Parametri presenti nei cookie:

- *Expires*: indica la scadenza del cookie (se non specificato tipicamente è la "browser session")
- *Max-age*: durata massima del cookie espressa in secondi[a]
- *Domain*: dominio a cui limitare l'invio del cookie
- *Path*: path alla risorsa a cui limitare l'invio del cookie
- *Secure attribute*: indica che il cookie non deve venire inviato presenza di connessioni *non* cifrate
- *HttpOnly attribute*: disabilita la possibilità di risalire al valore del cookie tramite JS e `document.cookie`

[a] non supportato in Internet Explorer

- il concetto di *sessione* è implementato dall'applicazione web
- le informazioni rigurdanti la sessione *devono* essere passate tra client e server
- la trasmissione può avvenire tramite:
  1. header `HTTP` (e.g., `Cookie`)

```
GET /page.php HTTP/1.1
Host: www.example.com
...
Cookie: sessionid=7456
...
```

  2. URL

```
http://www.example.com/page.php?sessionid=7456
```

  3. payload `HTTP`

```
<INPUT TYPE="hidden" NAME="sessionid" VALUE="7456">
```

- il concetto di *sessione* è implementato dall'applicazione web
- le informazioni rigurdanti la sessione *devono* essere passate tra client e server
- la trasmissione può avvenire tramite:
  1. header `HTTP` (e.g., `Cookie`)

```
GET /page.php HTTP/1.1
Host: www.example.com
...
Cookie: sessionid=7456
...
```

  2. URL

```
http://www.example.com/page.php?sessionid=7456
```

  3. payload `HTTP`

```
<INPUT TYPE="hidden" NAME="sessionid" VALUE="7456">
```

- il concetto di *sessione* è implementato dall'applicazione web
- le informazioni rigurdanti la sessione *devono* essere passate tra client e server
- la trasmissione può avvenire tramite:
  1. header HTTP (e.g., `Cookie`)

```
GET /page.php HTTP/1.1
Host: www.example.com
...
Cookie: sessionid=7456
...
```

  2. URL

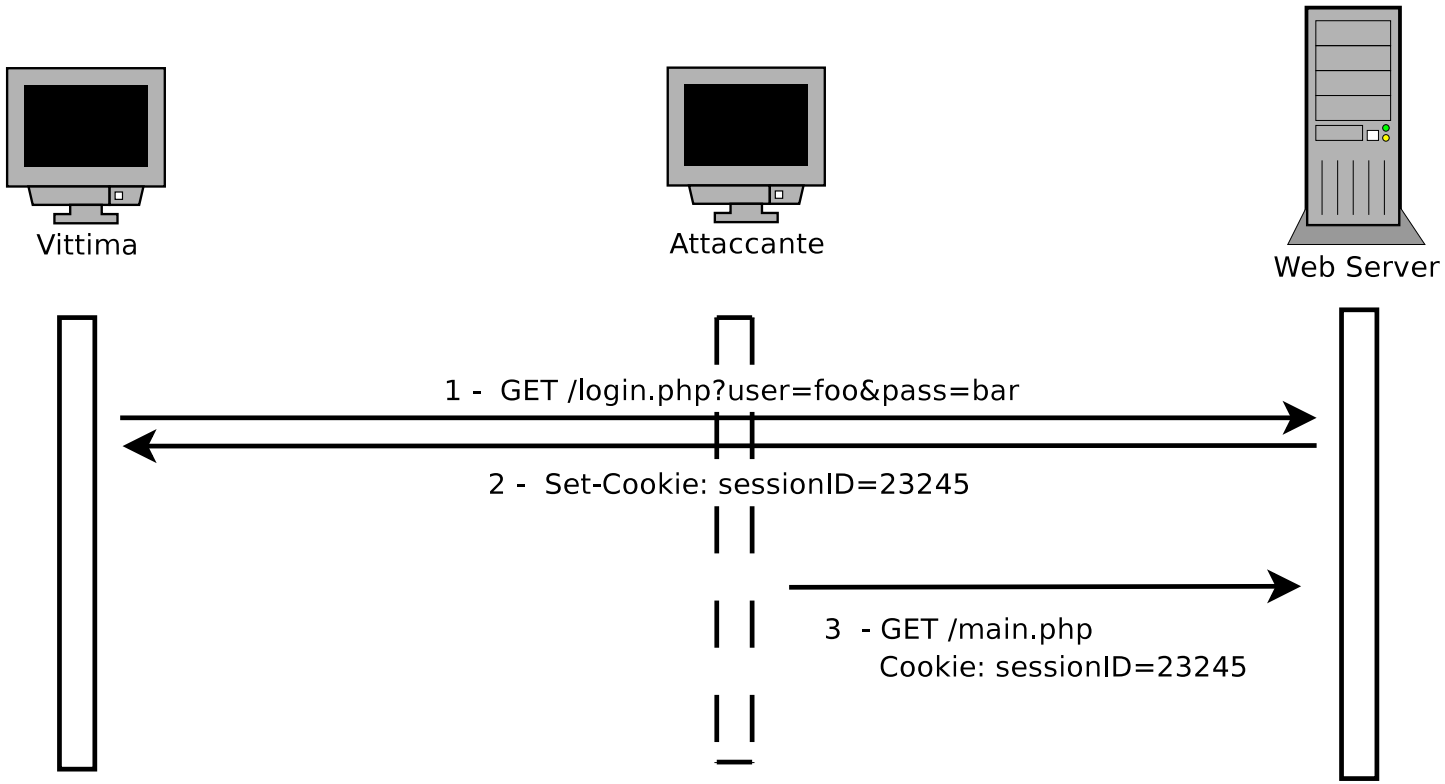
```
http://www.example.com/page.php?sessionid=7456
```

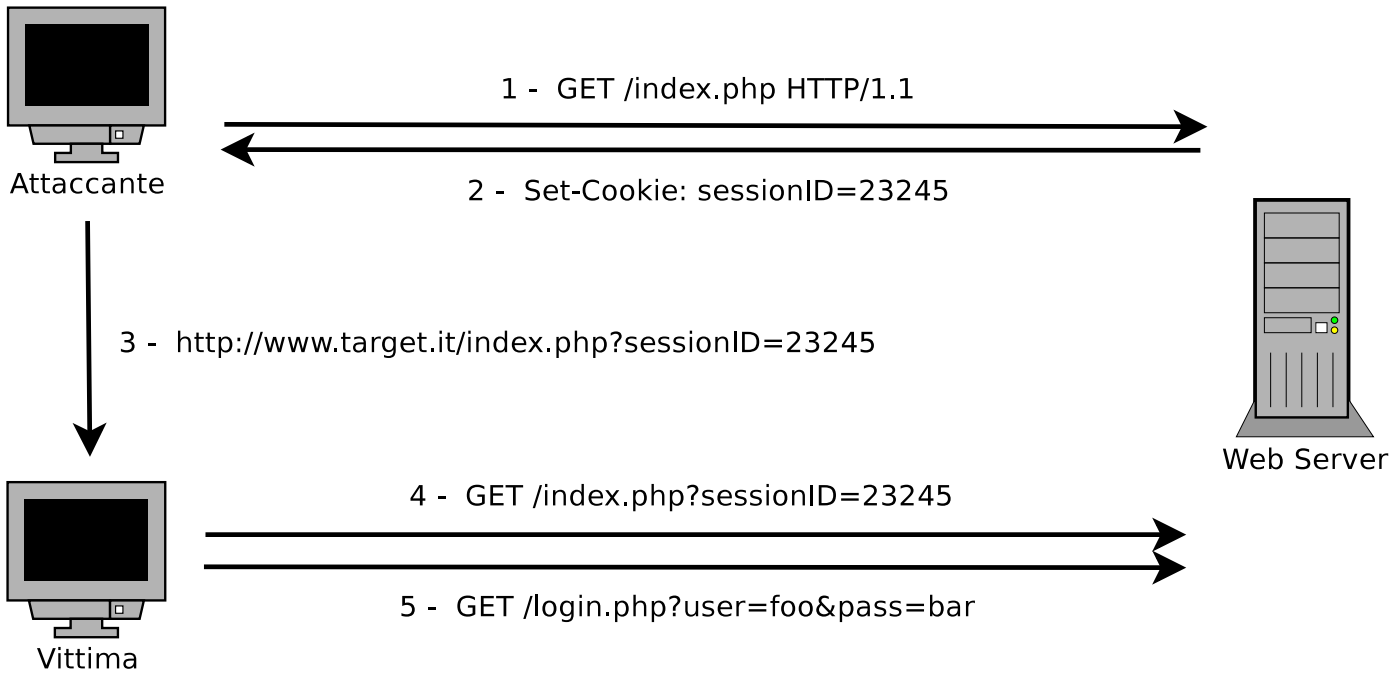
  3. payload HTTP

```
<INPUT TYPE="hidden" NAME="sessionid" VALUE="7456">
```

Sessioni: sicurezza

- sono un elemento **critico** (e.g., usate per autenticazione)
- rischio: *bypass* del sistema ti autenticazione!
- attacchi possibili:
  - intercettazione $\rightarrow$ `SSL/TLS`
  - predizione $\rightarrow$ *strong pseudonumber*
  - brute force $\rightarrow$ lunghezza `id`
  - session fixation $\rightarrow$ controllo `IP`, `Referer`; rigenerazione `id`; ...
- devono essere valide per un periodo di tempo limitato!


Vittima
Attaccante
Web Server
1 - GET /login.php?user=foo&pass=bar
2 - Set-Cookie: sessionID=23245
3 - GET /main.php
Cookie: sessionID=23245


Attaccante
1 - GET /index.php HTTP/1.1
2 - Set-Cookie: sessionID=23245
Web Server
3 - http://www.target.it/index.php?sessionID=23245
Vittima
4 - GET /index.php?sessionID=23245
5 - GET /login.php?user=foo&pass=bar

Analisi di traffico HTTP

- il payload HTTP è incapsulato all'interno del segmento TCP (default: porta 80)
- comunicazione "in chiaro"
- osservazione del traffico HTTP per analisi *black-box* dell'applicazione
- analisi tramite strumenti di *sniffing* (e.g., `ngrep`, `tcpdump`, `wireshark`, ...)
- questi strumenti **non** consentono l'analisi di traffico TLS/SSL

Manipolazione di traffico HTTP

Traffico HTTP

- Browser tradizionali (e.g., Firefox, Internet Explorer, Chrome, ...)
- `netcat`
- `curl`, `wget`
- ...

Traffico HTTPS

due alternative:

1. estensioni browser (e.g., `Firefox` → `Tamper Data`)
2. proxy HTTP

Proxy HTTP

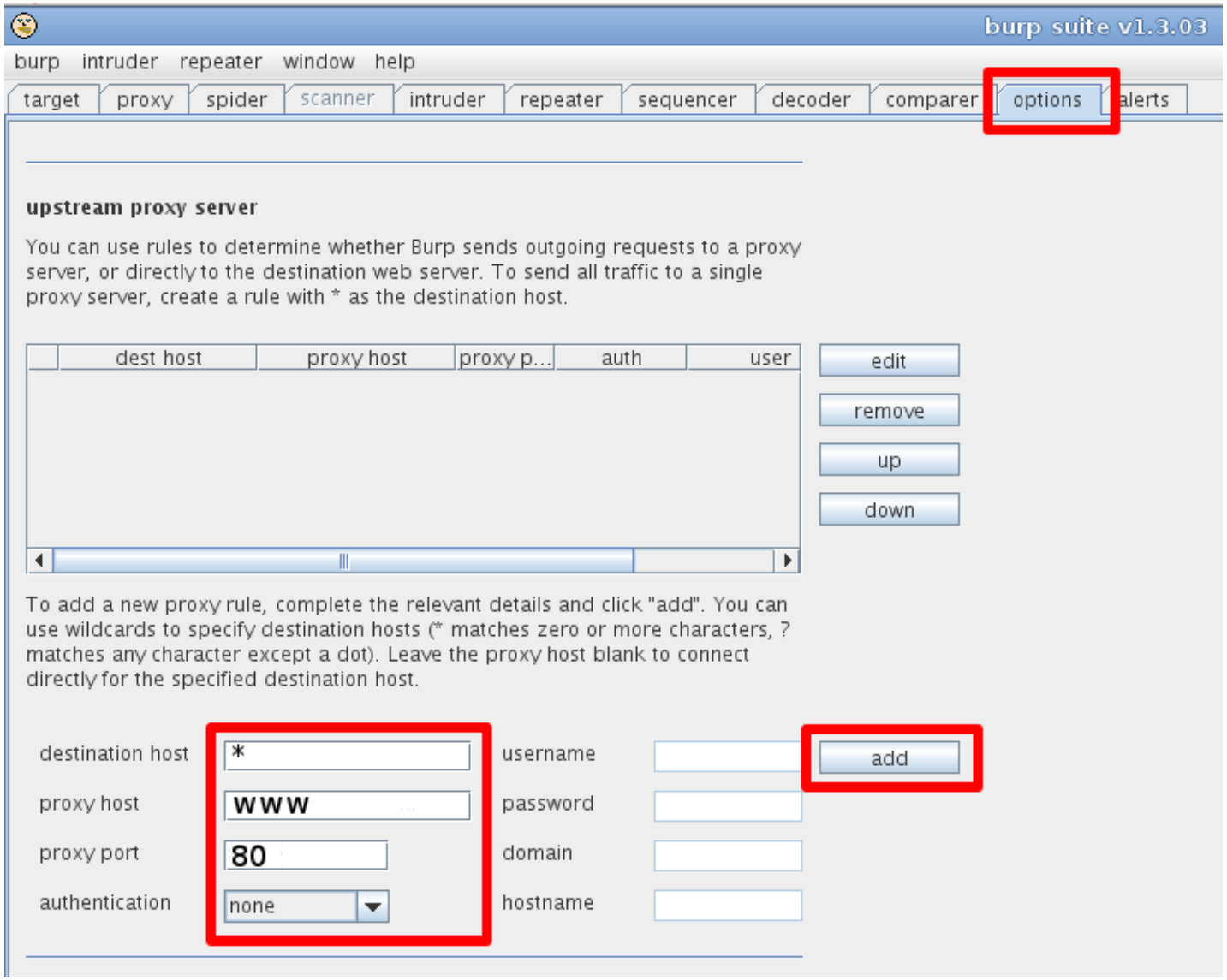
Un `http proxy` funziona come *man-in-the-middle* tra il browser e l'applicazione target.

- modifica del traffico `HTTP`/`HTTPS`
- indipendenti dall'applicazione
- intercettando traffico `HTTPS`, il browser notificherà l'errore nella verifica del certificato `SSL`

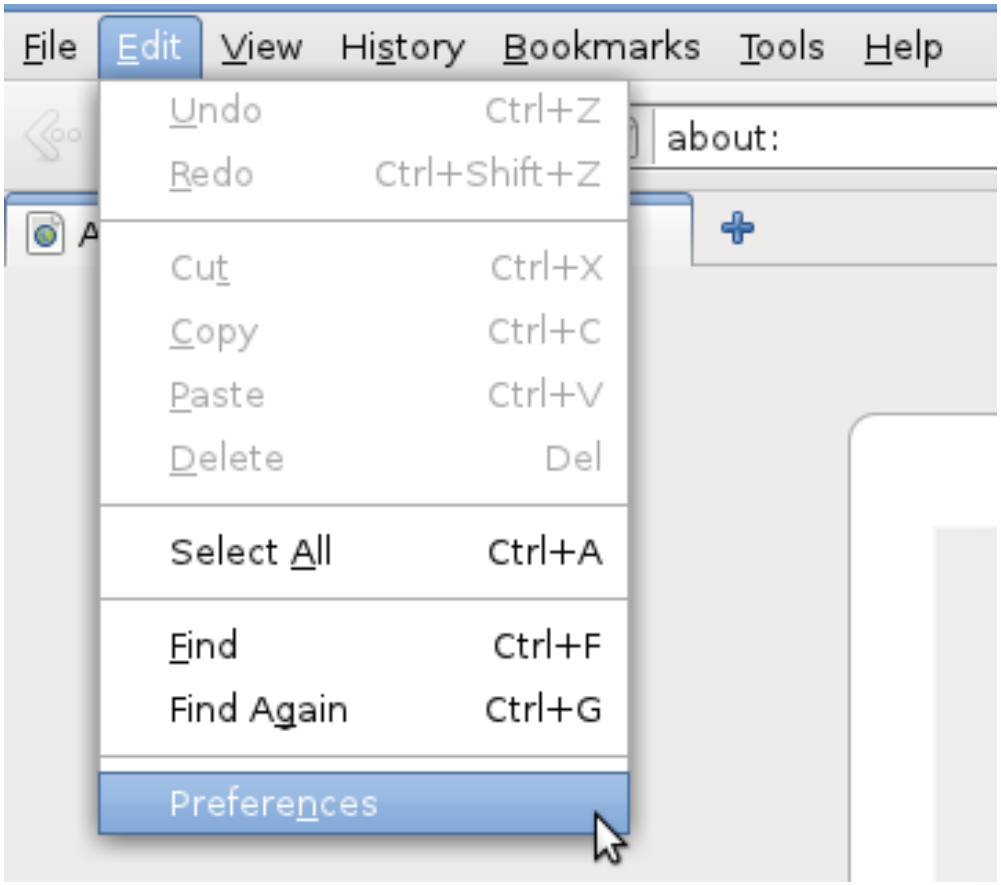
Alcuni proxy HTTP

- `WebScarab` - `http://www.owasp.org/`
- `proxpy` - `http://code.google.com/p/proxpy/`
- `Burp` - `http://portswigger.net/burp/`
- `Paros` - `http://www.parosproxy.org/`

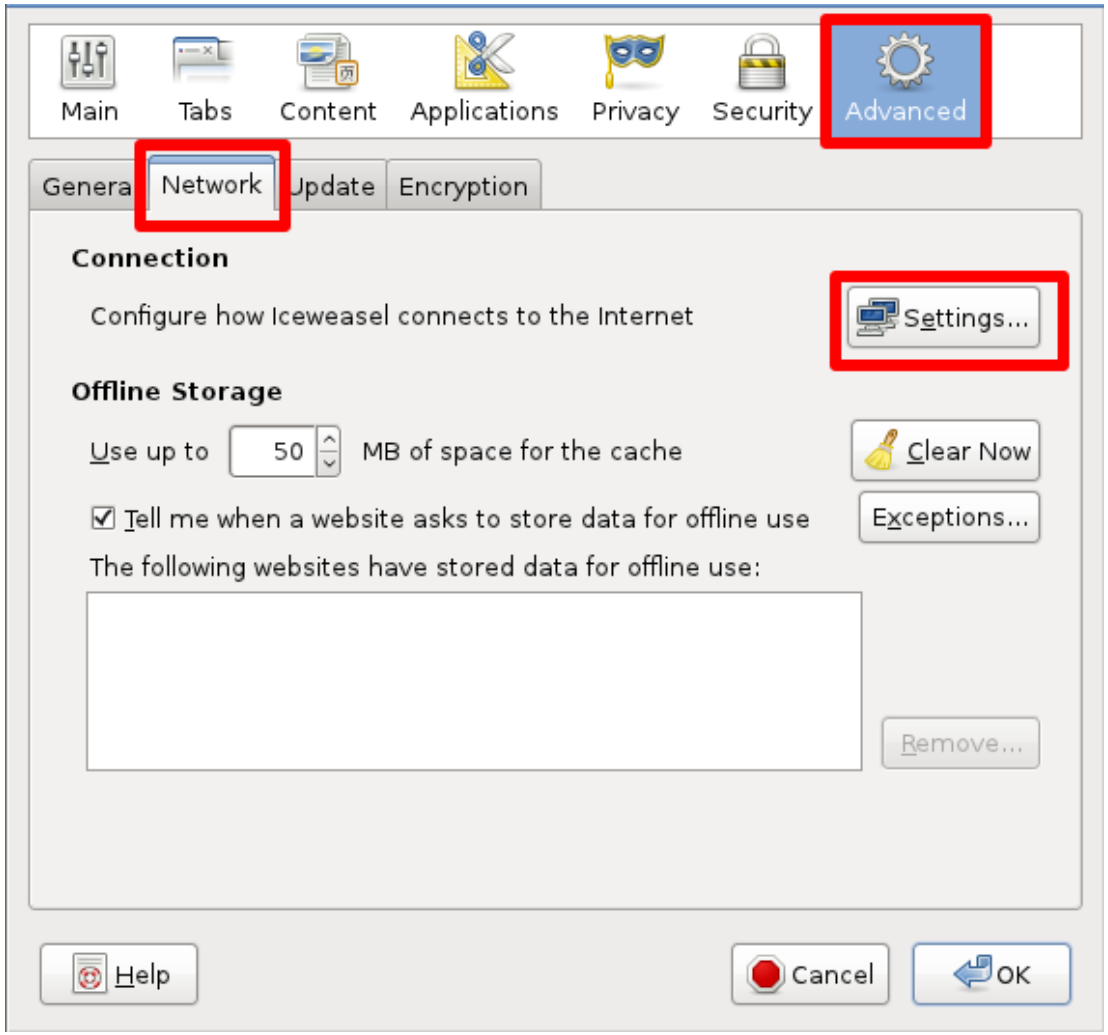
Configurazione Proxy Http
Firefox/IceWeasel



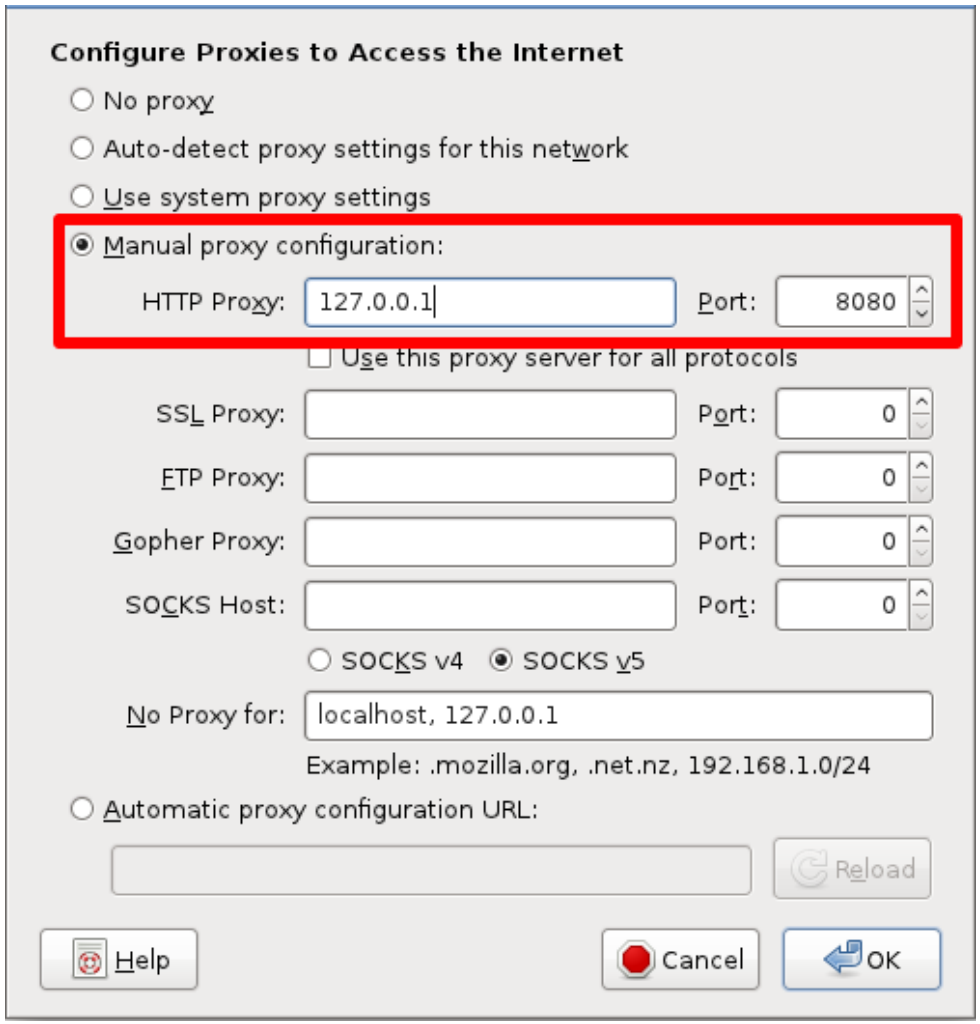
Configurazione Proxy Http
Firefox/IceWeasel



Burp - ProxyHttp
Firefox/IceWeasel



Burp - ProxyHttp
Firefox/IceWeasel



Esercizi

Tutti gli esercizi al seguente url:

`http://gamebox.laser.di.unimi.it/aa1314_sec1_web2/`

HINT

Gli esercizi sono in ordine crescente di difficoltà...
Partite dai primi!

Link e materiale utile:

- *"Hypertext Transfer Protocol – HTTP/1.0", RFC 1954,* `http://www.ietf.org/rfc/rfc1945.txt`
- *"Hypertext Transfer Protocol – HTTP/1.1", RFC 2616,* `http://www.ietf.org/rfc/rfc2616.txt`
- *"HTTP State Management Mechanism", RFC 2109,* `http://www.ietf.org/rfc/rfc2109.txt`
- *"The Tangled Web", Michael Zalewski, ED. No Starch Press, 2011*
- *"Cross-Site Tracing (XST)", Jeremiah Grossma, 2003,,* `http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf`