

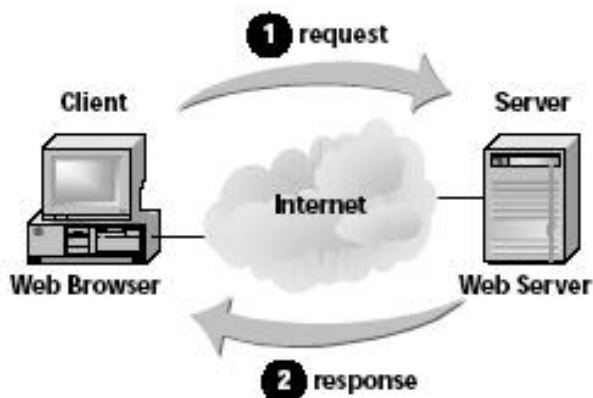
Cos'è l'Http

L'HyperText Transfer Protocol, protocollo dello strato dell'applicazione, costituisce il cuore del web. E' importante distinguere fra **applicazioni della rete** e **protocolli dello strato dell'applicazione**. Un protocollo dello strato dell'applicazione è solo un pezzo (quantunque, un grande pezzo) di un'applicazione della rete. Nel nostro caso, il Web è un'applicazione della rete che permette agli utenti di ottenere a richiesta "documenti" dai server Web. L'applicazione Web è costituita da molti componenti, che comprendono gli standard per i formati dei documenti (cioè, Html), i browser del Web (per esempio Netscape Navigator e Microsoft Internet Explorer), i server del Web (per esempio, i server Apache, Microsoft, e Netscape) e un protocollo dello strato dell'applicazione. Il protocollo dello strato dell'applicazione Web, HTTP (HyperText Transfer Protocol), definisce come i messaggi vengono passati fra browser e server Web. Quindi, HTTP è solo una parte dell'applicazione Web.

Il protocollo dello strato dell'applicazione definisce come i processi delle applicazioni, che funzionano su differenti terminali, si scambiano i messaggi. In particolare il protocollo dello strato dell'applicazione definisce:

- i tipi di messaggi scambiati, per esempio, messaggi di richiesta e messaggi di risposta;
- la sintassi dei vari tipi di messaggio, per esempio i campi del messaggio e come questi campi vengono caratterizzati;
- la semantica dei campi, cioè il significato dell'informazione nei campi;
- le regole per determinare quando e come un processo invia o risponde a messaggi;

Un protocollo dell'applicazione tipicamente ha due "lati", un **lato client** e uno **server**. Il lato client all'interno di un terminale comunica con il lato server di un altro terminale. Nel caso dell'HTTP, un browser web implementa il lato client e un server web ne implementa il lato server. Come nel caso di quasi tutte le applicazioni, *l'host che inizia la sessione è etichettato come client*.



URI (Uniform Resource Identifier)

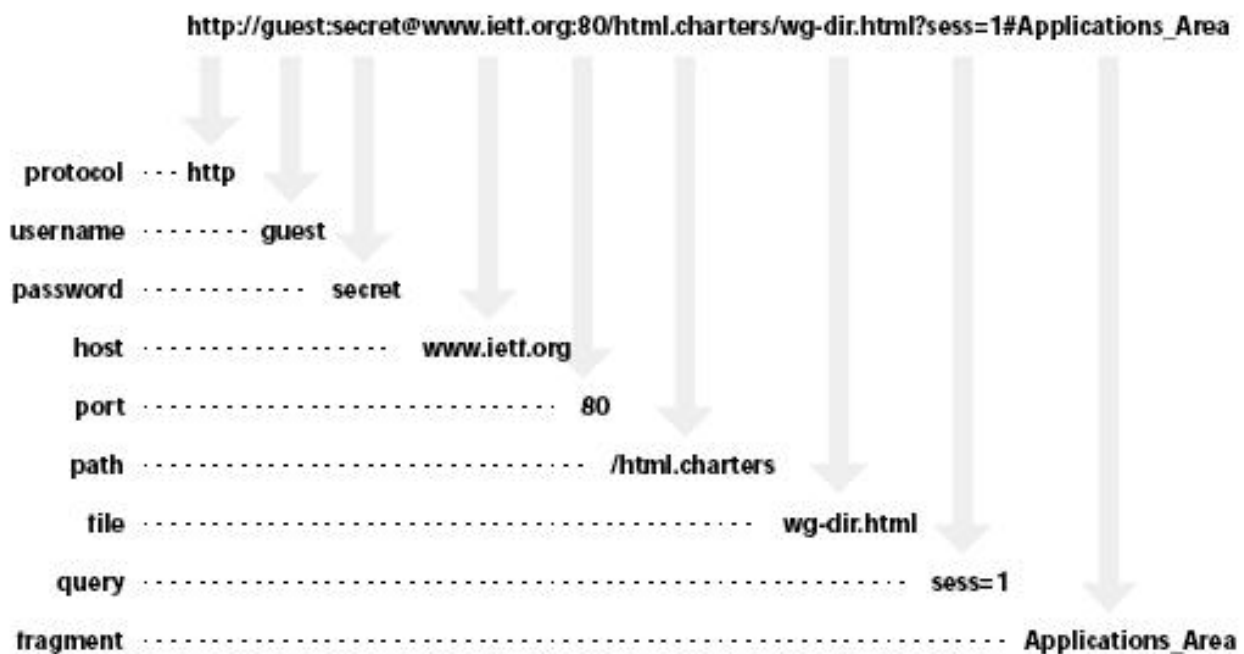
HTTP è usato per trasmettere risorse, non solo files. Una risorsa è identificata da un URI o URL. Il tipo più comune di risorsa è un file, ma può anche essere il risultato di una query generato dinamicamente, l'uscita di uno script CGI, un documento disponibile in diversi linguaggi, o altro ancora.

In base alle informazioni del W3C (World Wide Web Consortium) sull'indirizzamento, un URI è definito come "il set generico di tutti i nomi/indirizzi che sono stringhe corte che si assegnano alle risorse".

Un URL (Uniform Resource Locator) è definito come "un termine informale (più usato nelle specifiche tecniche) associato a popolari protocolli URI: http, ftp, mailto, etc".

HTTP fa riferimento continuamente agli Uniform Resource Identifiers (URIS). Attualmente, non c'è in verità molta distinzione tra i due concetti. Tecnicamente, un URL è solo un tipo di un URI, che quindi riveste un significato più generale. Dopo tutto, un modo per identificare un oggetto è quello di descrivere la sua locazione. In pratica, i due termini sono equivalenti.

La figura in basso mostra un URI con quasi tutti i possibili elementi.



Componenti di un URI

Analizziamo ora i componenti dell'URI, insieme ad una breve descrizione dell'uso di ognuno di essi:

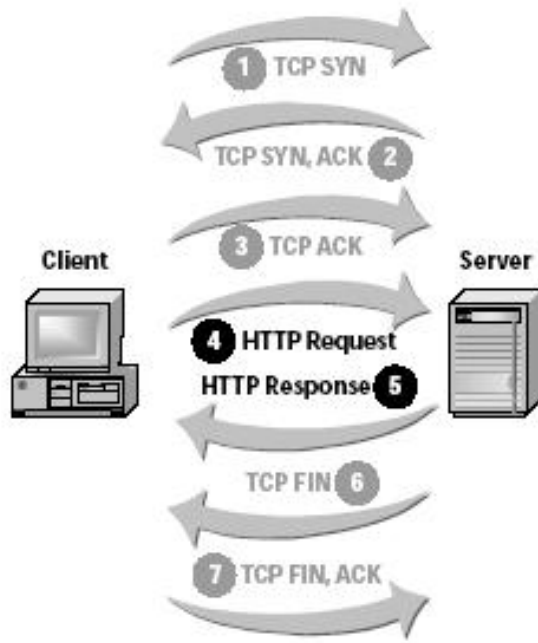
Componente	Uso

protocollo	Identifica il protocollo dello strato dell'applicazione di cui si ha bisogno per accedere alla risorsa, in questo caso HTTP.
username	Se il protocollo supporta il concetto di user name, questo componente ne fornisce uno che ha accesso alla risorsa.
password	La password associata all'user name.
host	Il sistema di comunicazione che ha la risorsa; per HTTP questo è il server Web.
porta	La porta TCP che i protocolli dello strato dell'applicazione dovrebbero usare per accedere alla risorsa; a molti protocolli è stata assegnata una porta TCP specifica, per l'HTTP è 80.
percorso	Il percorso attraverso un'organizzazione gerarchica tramite la quale la risorsa è localizzata, spesso una struttura di directory del file system o qualcosa di equivalente.
file	La risorsa stessa.
query	Informazione aggiuntiva sulla risorsa o sul client.
frammento	Una particolare ubicazione all'interno di una risorsa.

Sono tutti campi opzionali, tranne protocollo, host, percorso e file. Sebbene il protocollo, il percorso e il file possano essere omessi nella maggior parte dei moderni browser Web, essi sono ancora richiesti per un corretto URL. I browser semplicemente prenderanno in considerazione il protocollo HTTP e la directory radice (/) se questi non sono esplicitamente specificati.

Connessioni

Come ogni protocollo dell'applicazione che usa TCP come protocollo di trasporto, HTTP richiede una connessione TCP. Poichè un client HTTP è responsabile dell'inizio della comunicazione, esso è anche l'iniziatore del processo che crea la connessione TCP. Come si nota in figura, questo processo richiede lo scambio di tre segmenti TCP. Dopo tale scambio, il client può mandare la sua richiesta HTTP. Una volta soddisfatta la richiesta con un opportuno messaggio di risposta, è il server a chiudere la connessione tramite un ulteriore scambio di segmenti TCP.



Tipi di connessione

L'HTTP può impiegare sia la connessione non permanente sia quella permanente. L'HTTP/1.0 usa la connessione non permanente, viceversa la connessione permanente è di default per l'HTTP/1.1.

La prima versione di HTTP richiedeva ai client di stabilire una connessione TCP separata per ogni richiesta. Per semplici pagine Web, tale requisito non presentava molti problemi. Ma poichè i siti Web diventarono sempre più complessi e ricchi di grafica risultando così pieni di molti oggetti (dove per oggetto si intende semplicemente un file, come un file HTML, un'immagine JPEG o GIF, un Java Applet, un clip audio e così via), lo stabilire per ogni oggetto e quindi ogni richiesta una connessione TCP separata incominciò ad avere un notevole effetto negativo sulle performances del Web.

La versione 1.1 del protocollo HTTP eliminò il problema delle connessioni multiple TCP con una caratteristica chiamata "persistenza" o "permanenza". Essa abilita un client a continuare ad usare una connessione TCP esistente dopo che la sua iniziale richiesta è stata soddisfatta dal server. Semplicemente, il client fa una nuova richiesta sulla stessa connessione.

La permanenza richiede collaborazione fra client e server: il client deve ovviamente richiedere la connessione di tale tipo e può usarla solo se il server lo permette. Quest'ultimo non deve chiudere la connessione dopo aver mandato la sua risposta all'iniziale richiesta del client.

Connessione non permanente

Compriamo ora alcuni passi seguendo il trasferimento di una pagina Web dal server al client nel caso di una connessione non permanente. Supponiamo che la pagina consista di un file base HTML e di 10 immagini JPEG, e che tutti questi 11 oggetti risiedano sullo stesso server. Assumiamo che l'indirizzo URL per il file base HTML sia

```
www.ing.unipi.it/students/home.htm
```

Ecco ciò che accade:

1. Il client HTTP inizia una connessione TCP al server `www.ing.unipi.it`. La porta numero 80 è usata come numero di default della porta a cui il server HTTP ascolterà i client HTTP che vogliono recuperare i documenti usando l'HTTP.
2. Il client HTTP invia un messaggio di richiesta HTTP al server attraverso il socket associato alla connessione TCP che è stata stabilita in 1. Il messaggio di richiesta include il nome del percorso `/students/home.htm`
3. Il server riceve il messaggio di richiesta attraverso il socket associato alla connessione stabilita al punto 1, trova l'oggetto `/students/home.htm` nel sito in cui è memorizzato (RAM o disco), incapsula l'oggetto all'interno di un messaggio di risposta HTTP e invia il messaggio di risposta al client attraverso il socket.
4. Il server HTTP dice al TCP di concludere la connessione TCP. (Ma il TCP non può chiudere realmente la connessione finché il client non ha ricevuto il messaggio di risposta intatto.)
5. Il client HTTP riceve il messaggio di risposta. La connessione TCP si conclude. Il messaggio indica che l'oggetto incapsulato è un file HTML. Il client estrae il file dal messaggio di risposta, analizza il file HTML e trova i riferimenti ai 10 oggetti JPEG.
6. I primi quattro passi vengono quindi ripetuti per ciascuno degli oggetti JPEG cui si fa riferimento.

Come visto, ciascuna connessione TCP si chiude dopo che il server invia gli oggetti: la connessione non rimane disponibile per altri oggetti. Si noti che ciascuna connessione TCP trasporta esattamente un messaggio di richiesta e un messaggio di risposta. Quindi, in questo esempio, quando un utilizzatore richiede una pagina Web, si generano 11 connessioni TCP.

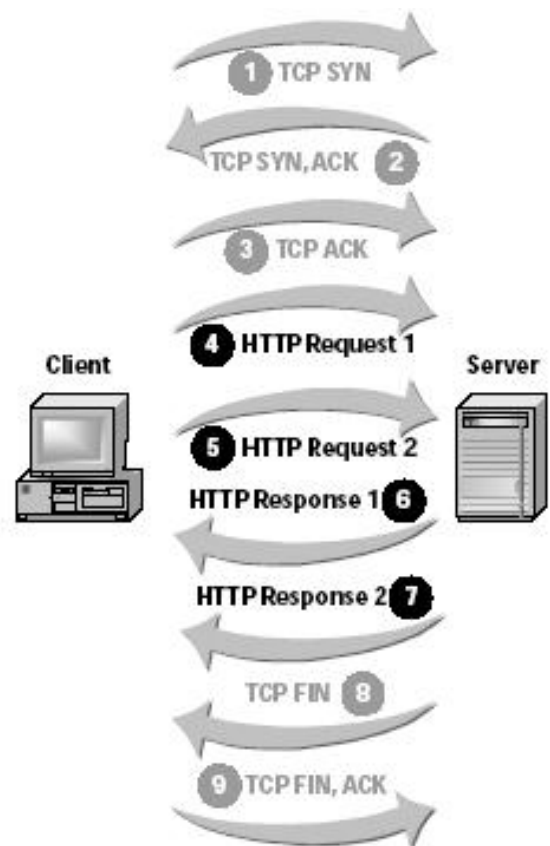
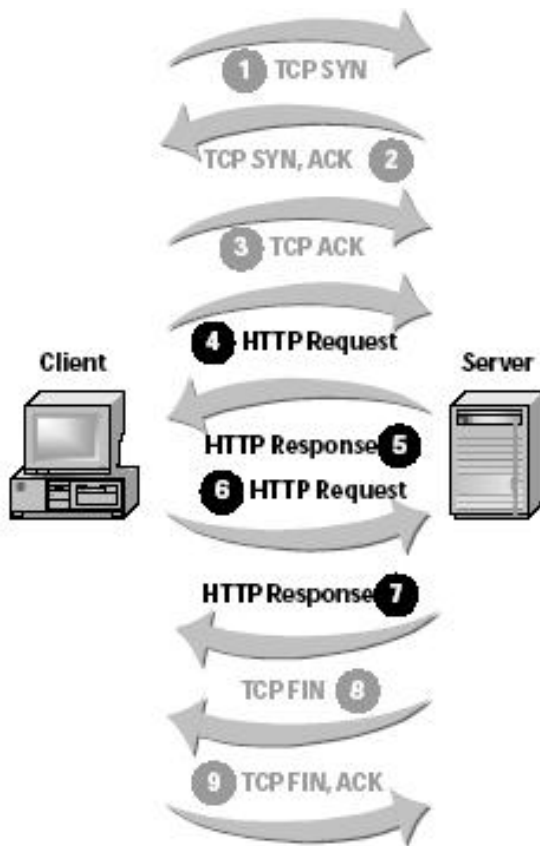
Connessione permanente

Con questo tipo di connessione, il server lascia aperta la connessione TCP dopo aver spedito la risposta. Le successive richieste e risposte fra gli stessi client e server possono essere inviate sull'identica connessione. In particolare, un'intera pagina Web (nell'esempio sopra il file base HTML e 10 immagini JPEG) può essere spedita su una singola connessione TCP permanente; inoltre, pagine Web multiple residenti sullo stesso server possono essere spedite sulla stessa connessione TCP permanente. Tipicamente, il server HTTP chiude una connessione quando non è usata da un certo tempo (intervallo di *timeout*), che spesso è configurabile.

Esistono due versioni della connessione permanente: **incanalata** e **non incanalata**. Mentre per la versione non incanalata il client passa una nuova richiesta solo quando la risposta alla precedente è stata ricevuta, per quella incanalata le richieste possono essere messe in pipeline, ma le risposte devono essere date nello stesso ordine delle richieste in quanto non è specificato un metodo esplicito di associazione.

Il modo di default di HTTP/1.1 usa la connessione permanente con incanalamento. In questo caso, il client passa una nuova richiesta non appena incontra un nuovo riferimento. Quindi il client può fare richieste consecutive (back-to-back) per gli oggetti in riferimento. Quando il server riceve le richieste, può inviare gli oggetti back to back.

Nelle sottostanti figure vediamo i due tipi di connessione permanente, non incanalata a sinistra e incanalata a destra.





Struttura dei messaggi Http

Come detto, HTTP usa il modello client-server: il client HTTP apre una connessione e manda un messaggio di richiesta a un server HTTP che rimanda un messaggio di risposta, contenente solitamente la risorsa richiesta.

Da notare che il server invia gli oggetti richiesti ai client senza immagazzinare alcuna informazione di stato relativa al client. Se uno stesso client chiede lo stesso oggetto due volte nel giro di pochi secondi, il server lo rispedisce come se avesse completamente dimenticato ciò che ha appena fatto. Poichè un server HTTP non conserva le informazioni relative ai client, l'HTTP è detto **protocollo senza stato** (*stateless protocol*).

Il formato dei messaggi di richiesta e di risposta è simile, entrambi consistono in:

- una linea iniziale
- nessuna o più linee di intestazione
- una linea vuota
- un corpo del messaggio opzionale

In generale, il formato di un messaggio HTTP è quindi il seguente:

```
<linea iniziale, differente da richiesta a risposta>
Intestazione1: valore1
Intestazione2: valore2
Intestazione3: valore3

<corpo del messaggio opzionale>
```

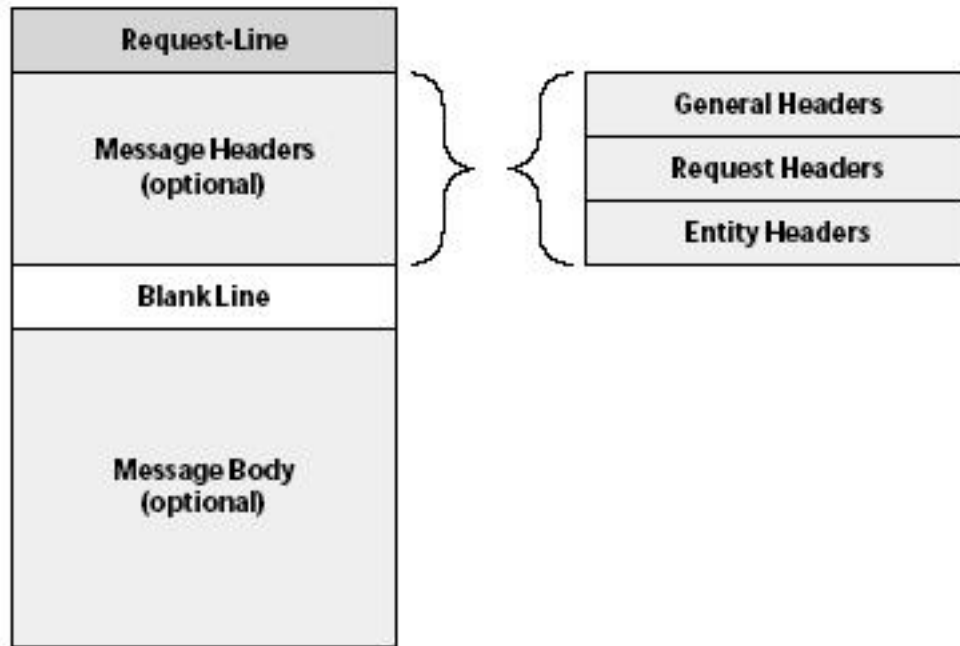
Come si può notare, linee di intestazione e corpo del messaggio sono elementi opzionali all'interno di un messaggio HTTP, di richiesta o di risposta che sia. Solo l'intestazione `Host` è obbligatoria nei messaggi di richiesta di un client a un server entrambi HTTP/1.1 per evitare che il server risponda con un codice di stato `400 Bad Request`.

Richieste HTTP

Una richiesta HTTP, che rappresenta il messaggio mandato da un client HTTP (browser) ad un server HTTP (server Web), comprende tre elementi di base:

- linea di richiesta
- intestazioni HTTP
- corpo del messaggio

La seguente figura mostra la struttura di base delle richieste HTTP.



Ogni richiesta HTTP incomincia sempre con una linea di richiesta.

Linea di richiesta

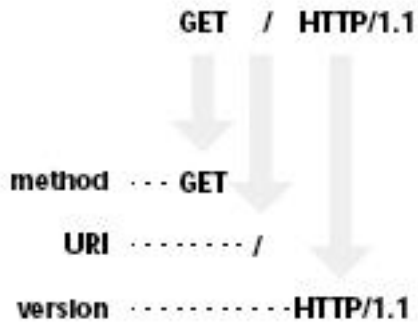
Questa linea di testo è composta da tre campi, separati da spazi:

- metodo
- URI
- versione HTTP

Essa indica quindi il metodo che il client sta richiedendo, la risorsa alla quale applicare il metodo e la versione HTTP che si sta usando.

Un tipica linea di richiesta è:

GET / HTTP/1.1



Nell'esempio:

- il metodo richiesto è GET (il metodo HTTP più comune), ma HTTP definisce un totale di otto metodi differenti, ognuno dei quali viene descritto nella sezione [Metodi](#). I nomi dei metodi sono sempre in maiuscolo.
- l'URI è /, che indica una richiesta per la radice della risorsa. Per richieste che non si applicano a nessuna specifica risorsa, come una richiesta TRACE o in alcuni casi una richiesta OPTIONS, il client può usare un asterisco come URI.
- il browser implementa la versione HTTP/1.1. Questo campo prende sempre la forma "HTTP/x.x", in maiuscolo.

Esempio di richiesta HTTP

Per trovare il file all'URL:

```
http://www.somehost.com/path/file.html
```

dapprima si apre un socket con l'host `www.somehost.com`, porta 80 (si usa la porta di default perchè non ne è specificata una nell'URL). Poi si manda un messaggio di richiesta tipo il seguente al socket stesso:

```
GET /path/file.html HTTP/1.1
Host: www.somehost.com
Connection: close
User-Agent: Mozilla/4.0
Accept-Language: en-us
Accept-Encoding: gzip, deflate
(ritorno carrello extra, line feed)
```

Come detto, la prima linea è quella di richiesta. Tutte le linee successive sono le intestazioni del

messaggio, descritte nella sezione [Intestazioni](#), terminate da una linea vuota (CRLF).

Le intestazioni HTTP includono informazioni di supporto che possono aiutare a spiegare in modo più chiaro la richiesta del client Web. Ci sono tre tipi di intestazioni che possono apparire in una richiesta:

- intestazioni generali
- intestazioni della richiesta
- intestazioni del corpo dell'entità

Nell'esempio non compare il corpo dell'entità (*entity body*) in quanto non è usato con il metodo GET, bensì con il metodo POST.

Risposte HTTP

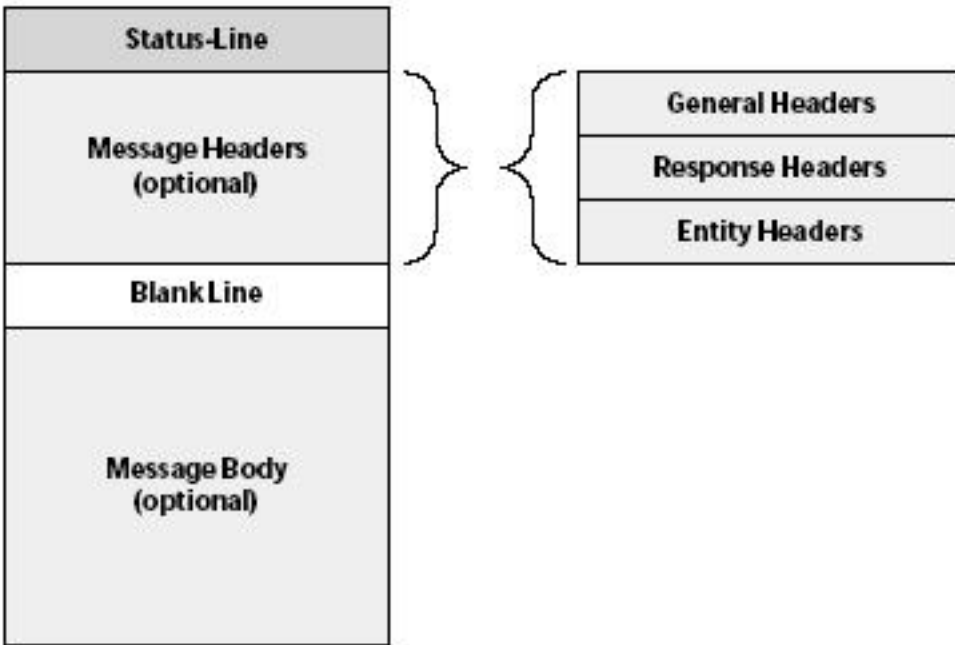
Dopo che un server Web riceve una richiesta HTTP, intraprende le azioni necessarie a fornire la risorsa richiesta. Queste possono includere l'esecuzione di uno script CGI o altro. Fatto ciò, il server Web risponde al client con una risposta HTTP. Questa risposta è organizzata in modo simile ad una richiesta HTTP.

L'unica differenza significativa è che le risposte cominciano con una linea di stato piuttosto che con una linea di richiesta.

Quindi, come nel caso di una richiesta, una risposta HTTP comprende tre elementi di base:

- linea di stato
- intestazioni HTTP
- corpo del messaggio

La seguente figura mostra la struttura di base delle risposte HTTP.



Ogni risposta HTTP incomincia sempre con una linea di stato.

Linea di stato

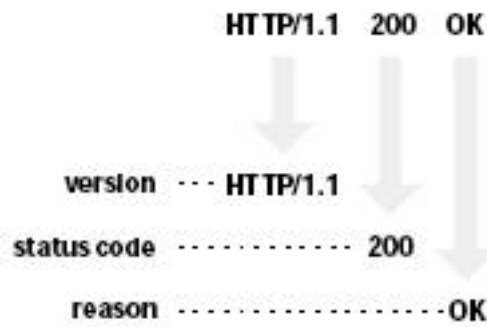
Una linea di stato è composta da tre campi, separati da spazi:

- versione HTTP
- codice di stato
- messaggio di stato

Essa include quindi la più alta versione che il server supporta, un codice di stato che indica il risultato della richiesta e un messaggio di stato corrispondente.

Una tipica linea di stato è:

HTTP/1.1 200 OK



Nell'esempio:

- il server implementa la versione HTTP/1.1 (che è nello stesso formato della linea di richiesta, "HTTP/x.x"). Come per il client, ciò non indica che la risposta includa necessariamente opzioni definite da quella versione. Un server HTTP/1.1 che riceve una richiesta da un client HTTP/1.0, ad esempio, può ancora indicare HTTP/1.1 nella sua linea di stato. Quel server comunque deve stare attento a includere solo opzioni HTTP/1.0 nella sua risposta, altrimenti può mandare informazioni che il client non riesce a capire.
- *200* è il codice di stato più comune. Questo valore indica che la richiesta del client è stata soddisfatta.
- *OK* è il messaggio di stato corrispondente al codice di stato 200. Esso aiuta soltanto l'utente a interpretare il valore del codice di stato. I server lo includono per facilitare l'utente, ma i client non pongono attenzione al suo contenuto.

Codici di stato

Il codice di stato è un numero di tre cifre che indica il risultato di una richiesta.

La prima cifra identifica il tipo di risultato e dà un'indicazione di alto livello, le altre forniscono più dettagli.

Codice di Stato	Significato
100-199	<u>Messaggio informativo</u> : il server ha ricevuto la richiesta ma un risultato finale non è ancora disponibile.

200-299	<u>Successo</u> : il server è stato capace di agire sulla richiesta con successo.
300-399	<u>Redirezione</u> : il client deve reinstradare la richiesta a un differente server o risorsa.
400-499	<u>Errore sul lato client</u> : la richiesta conteneva un errore che ha impedito al server di agire su di essa con successo
500-599	<u>Errore sul lato server</u> : il server ha fallito ad agire su una richiesta anche se la richiesta sembra essere valida

In ogni classe, il codice di stato "x00" è lo stato principale per la classe. Se un client riceve un valore di codice di stato che non conosce, può trattarlo nello stesso modo in cui tratterebbe il valore x00 della classe. Per esempio, il codice di stato 237 sarebbe trattato come il 200.

Esempi di stato HTTP

Ecco un elenco dei codici di stato più comuni e degli stati principali di ogni classe:

- 100 `Continue`: il client può chiedere al server di accettare una richiesta prima di mandare l'intero corpo del messaggio. L'intestazione `Expect: 100-Continue` chiede al server di segnalare la sua accettazione inviando un tale codice di stato. Una volta che il client riceve il codice 100 dal server, continua mandando il resto della richiesta. Se il server non risponde entro un ragionevole periodo di tempo, il client deve procedere con la sua richiesta comunque.
- 200 `OK`: la richiesta ha avuto successo e la risorsa richiesta (ad esempio un file o l'uscita di uno script) è inviata nel corpo del messaggio.
- 300 `Multiple Choices`: quando il server non può fornire la risorsa richiesta dà al client una lista di risorse alternative. Queste possono essere fornite nel corpo del messaggio della risposta e una può essere inclusa nell'intestazione `Location`.
- 301 `Moved Permanently`: l'URI di una risorsa è cambiato in maniera permanente, d'ora innanzi i client devono usare l'URI indicato per tutti i futuri riferimenti alla risorsa. Tutti gli altri codici di stato della stessa classe rappresentano condizioni temporanee.
- 302 `Found`: redirezione a un nuovo URL in quanto l'originale è stata spostato; non si tratta di un errore, i browser compatibili cercheranno la nuova pagina.
- 304 `Not Modified`: se una richiesta include una condizione (come un'intestazione `If-`

Match o If-Modified-Since) e questa non è soddisfatta, il server risponde con tale stato. Tipicamente questo permette al client (o al proxy server che inoltra la richiesta) di usare la copia della risorsa in cache senza che il server la mandi nuovamente.

- 400 Bad Request: è il codice di stato standard per gli errori dei client. Questa risposta indica che il server non ha capito la richiesta, forse perchè c'è un errore nella sua strutturazione. Il client non dovrebbe riformulare la stessa richiesta poichè sarebbe nuovamente rifiutata.
- 401 Unauthorized: l'utente ha richiesto un documento ma non ha fornito uno username o una password validi. Il client deve riformulare la richiesta con un'appropriata intestazione Authorization.
- 403 Forbidden: un client che riceve un tale codice di stato ha tentato di accedere a una risorsa a cui non si può avere accesso. A differenza del caso del codice di stato precedente, nessuna intestazione garantirà l'accesso al client. Mandando una simile risposta, il server implicitamente rivela che la risorsa richiesta esiste, altrimenti userebbe il codice di stato 404.
- 404 Not Found: indica che la risorsa richiesta non esiste. Non viene data alcuna informazione sulla condizione di tale mancanza, se è temporanea o permanente.
- 500 Internal Server Error: si è verificato un errore del server inaspettato. La causa più comune è uno script lato server con cattiva sintassi, errori o altro che non può essere eseguito correttamente. Se il server può fornire ulteriori dettagli, lo può fare nel corpo del messaggio della risposta.

Esempio di risposta HTTP

Qui sotto viene fornito un tipico messaggio di risposta HTTP. Questo messaggio di risposta, rimandato attraverso lo stesso socket aperto dal client, può essere la risposta al messaggio di richiesta visto in precedenza.

```
HTTP/1.1 200 OK
Connection: close
Date: Fri, 31 Dec 2004 23:59:59 GMT
Server: Apache/1.3.6 (Unix)
Last-Modified: Mon, 23 Nov 2004 08.23.21 GMT
Content-Type: text/html
Content-Length: 1354
```

```
<html>...
```

A differenza di un messaggio di richiesta dove la prima linea è quella di richiesta, in un messaggio di risposta vi è la linea di stato. Tutte le linee successive sono le intestazioni del messaggio, descritte nella sezione [Intestazioni](#), terminate da una linea vuota (CRLF).

Le intestazioni HTTP includono informazioni di supporto che possono aiutare a spiegare in modo più chiaro la risposta del server Web. Ci sono tre tipi di intestazioni che possono apparire in una risposta:

- intestazioni generali
- intestazioni della risposta
- intestazioni del corpo dell'entità

Il corpo dell'entità è il nucleo del messaggio. In esso c'è la risorsa richiesta e rimandata al client (questo è l'uso più comune del corpo del messaggio), o forse testo che spiega al client se si è verificato un errore per cui l'operazione richiesta non può essere effettuata con successo. In un messaggio di richiesta invece rappresenta la parte dove si trovano i dati che l'utente ha immesso (ad esempio in una form) o file di upload da mandare al server.

Metodi di richiesta

Uno dei più importanti attributi di una richiesta HTTP è il metodo di richiesta. Esso indica "l'intenzione" della richiesta del client. Sebbene molti metodi siano usati raramente nella pratica, sono tutti importanti per scopi diversi.

In HTTP/1.1 sono otto i metodi di richiesta:

- [GET](#)
- [POST](#)
- [PUT](#)
- [DELETE](#)
- [HEAD](#)
- [TRACE](#)
- [OPTIONS](#)
- [CONNECT](#)

La versione 1.0 dell'HTTP include due metodi, `LINK` e `UNLINK`, che non esistono nella versione 1.1. Questi metodi, che non erano tra l'altro largamente supportati da browser o server Web, permettevano a un client HTTP di modificare le informazioni su una risorsa esistente senza cambiare la risorsa stessa.

Mentre HTTP/0.9 supporta solo un metodo di richiesta, il più comune `GET`, HTTP/1.0 specifica tre metodi (`GET`, `HEAD` e `POST`), sebbene altri quattro sono implementati da alcuni server e client. Il supporto per questi altri quattro metodi (`PUT`, `DELETE`, `LINK` e `UNLINK`) è inconsistente e per lo più indefinito, anche se ognuno di essi è brevemente citato nell'appendice D dell'RFC 1945.

Method	HTTP/0.9	HTTP/1.0	HTTP/1.1
CONNECT			●
DELETE		●	●
GET	●	●	●
HEAD		●	●
LINK		●	
POST		●	●
PUT		●	●
OPTIONS			●
TRACE			●
UNLINK		●	

Metodi sicuri e idempotenti

Gli implementatori, consapevoli che il software rappresenta l'utente nelle sue interazioni su Internet, dovrebbero trasmettere all'utente la consapevolezza che alcune azioni intraprese possono risultare dannose.

In particolare, i metodi GET e HEAD non svolgono un'azione diversa dalla ricerca. Questi metodi dovrebbero quindi essere considerati "sicuri". Questo permette agli agenti dell'utente (interfaccia fra l'utente e le applicazioni di rete, nel caso dell'applicazione web l'agente dell'utente è un browser come Netscape Navigator o Microsoft Internet Explorer) di rappresentare gli altri metodi, come POST, PUT e DELETE, in un modo speciale, cosicchè l'utente è reso cosciente del fatto che potrebbe essere possibile un'azione non sicura..

I metodi possono anche avere la proprietà di idempotenza, se gli effetti collaterali di N>0 richieste identiche sono gli stessi di una singola richiesta. I metodi GET, HEAD, PUT e DELETE condividono questa proprietà, non è così per POST.

Comunque è possibile che una sequenza di alcune richieste non sia idempotente anche se lo sono tutti i metodi eseguiti in quella sequenza (una sequenza è idempotente se una singola esecuzione dell'intera sequenza produce sempre un risultato che non è cambiato da una riesecuzione di tutta, o parte, della stessa).

Per esempio, una sequenza non è idempotente se il suo risultato dipende da un valore che più tardi è modificato nella stessa sequenza. Per definizione, una sequenza che non ha mai effetti collaterali è

idempotente (purchè nessuna operazione concomitante sia eseguita sullo stesso set di risorse).

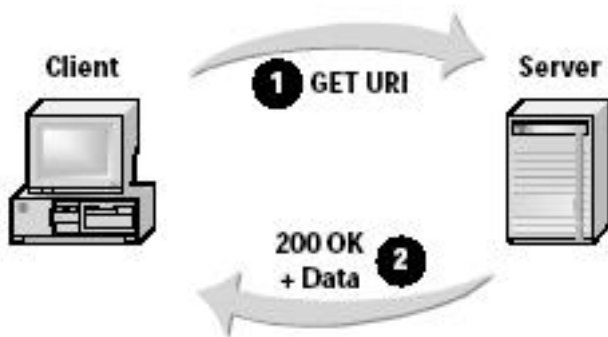
In breve, un metodo HTTP può essere:

- 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.

Metodo GET

La più semplice operazione HTTP è GET. Serve a un client per recuperare un oggetto dal server. Sul Web, un browser richiede una pagina da un server con un GET. Questo è il tipo di richiesta che un browser usa quando l'utente clicca su un link o immette un URI nell'apposito campo del browser.

Come mostrato nella figura in basso, GET è un semplice scambio di due messaggi. Il client inizia mandando un messaggio con tale metodo al server. Il messaggio identifica l'oggetto che il client sta richiedendo tramite il campo URI della linea di richiesta.



Se il server può inviare l'oggetto richiesto, lo fa nella sua risposta. Come mostra la figura, il server indica il successo dell'operazione con l'appropriato codice di stato 200 nella linea di stato relativa alla sua risposta e l'oggetto è ritornato nel corpo del messaggio. Se il server non può spedire l'oggetto (o sceglie di non farlo), inserirà un opportuno codice di stato.

GET è sicuro e idempotente, e può essere:

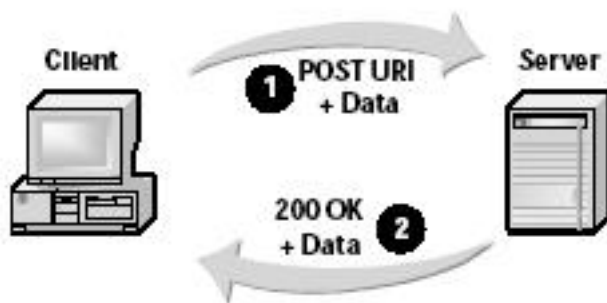
- Assoluto: normalmente, cioè quando la risorsa viene richiesta senza ulteriori specificazioni.
- Condizionale: se il messaggio di richiesta include un'intestazione del tipo If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match o If-Range. Un metodo di richiesta GET condizionale richiede che l'oggetto sia trasferito solo sotto le circostanze descritte dai campi delle intestazioni condizionali.

- Parziale: se il messaggio di richiesta include l'intestazione Range. Un GET parziale richiede che sia trasferita solo parte dell'entità.

Metodo POST

A differenza di GET, che permette a un server di mandare informazioni a un client, l'operazione POST fornisce ai client un modo per trasmettere informazioni ai server. La maggior parte dei browser web usa comunemente tale metodo per inviare forms ai server Web.

Come si vede nella figura, l'operazione POST è semplice quasi come la GET. Il client manda un messaggio POST e include l'informazione che desidera mandare al server.



Come in un messaggio GET, viene specificato anche un URI. In questo caso esso identifica l'oggetto nel server che può processare l'informazione inclusa. Sui server Web, questo URI è frequentemente un programma o uno script. Un server può rimandare l'informazione stessa come parte della risposta. Tipicamente, essa è una nuova pagina da visualizzare, spesso influenzata dai dati inseriti dall'utente; ad esempio, nel caso di una form di ricerca, come avviene in un motore di ricerca, la nuova pagina Web spesso mostra i risultati della richiesta.

POST non è nè sicuro nè idempotente e si differenzia da GET per i seguenti motivi:

- c'è un blocco di dati spediti con la richiesta, nel corpo del messaggio. Di solito ci sono intestazioni extra per descrivere il corpo stesso, come Content-Type e Content-Length.
- l'URI richiesto non è una risorsa da prelevare; è di solito un programma per maneggiare i dati che si stanno spedendo.
- la risposta HTTP è normalmente l'uscita di un programma, non un file statico.

Un 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. In tal caso è presente anche l'intestazione Location.
 - 204 No Content: dati ricevuti e sottomessi alla risorsa specificata. Non è stata data risposta.
-

Differenze tra GET e POST

GET e POST sono due metodi definiti in HTTP i cui compiti sono differenti, ma entrambi sono capaci di mandare sottomissioni di form ai server.

Normalmente GET è usato per ottenere un file o altra risorsa, possibilmente con parametri che specificano più esattamente ciò di cui si ha bisogno. Nel caso di dati di ingresso di una form, GET li include tutti dopo il punto interrogativo nella URL, ad esempio:

```
http://myhost.com/mypath/myscript.cgi?name1=value1&name=value2
```

Con GET, tutti i dati inseriti dall'utente sono impacchettati nella variabile d'ambiente QUERY_STRING e spediti come parte della URL. Ci sono però dei limiti alla dimensione della stessa: la soluzione è usare il metodo POST.

POST si usa per mandare dati che devono essere processati a un server. Quando una form HTML è sottomessa usando tale metodo, i dati immessi sono inseriti alla fine del messaggio di richiesta, nel corpo dell'entità, così che arrivano ad uno script CGI come standard input anzichè nella variabile d'ambiente QUERY_STRING. Ciò non è semplice come l'uso di GET, ma è più versatile. Ad esempio, si possono mandare interi file e la dimensione dei dati, come detto, non è limitata.

Risulta così che POST sia più affidabile di GET, poichè le informazioni di input vengono mandate come dati del pacchetto HTTP anzichè nella URL. Quindi, per le form si può usare uno dei due metodi, ma GET è appropriato se l'ammontare di dati è piccolo e se non ci sono effetti collaterali sul server essendo un metodo idempotente (molte ricerche effettuate su database non hanno effetti collaterali visibili e costituiscono le applicazioni ideali per il metodo GET).

Tutto ciò avviene comunque "dietro le scene", nel senso che per il programmatore CGI entrambi lavorano in modo identico e sono ugualmente facili da usare. Da notare che con POST lo script viene richiamato ogni volta che una form è sottomessa, con GET invece browser e proxy possono cachare le risposte dei server, così due identiche sottomissioni di form possono non essere passate entrambe allo script CGI; è bene quindi non usare GET se si vuole documentare ogni richiesta, o magazzinoare dati o fare altro che intraprenda un'azione per ogni richiesta.

Nota: il metodo GET limita il valore dell'insieme di dati del modulo ai caratteri ASCII. Solo il metodo POST (con enctype="multipart/form-data") è indicato per coprire l'intero insieme di caratteri

[ISO10646].

Codifica URL

I dati di una form HTML sono di solito codificati per essere impacchettati in una sottomissione GET o POST. Ecco i passi da compiere per codificare le coppie nome-valore dei dati della form:

1. Convertire tutti i caratteri "non sicuri" dei nomi e valori a "%xx", dove "xx" è il valore ASCII del carattere, in esadecimale. I caratteri "non sicuri" includono =, &, %, +, caratteri non stampabili e qualsiasi altro carattere si voglia codificare, non c'è pericolo nel codificarne troppi. Per semplicità si possono codificare tutti i caratteri non alfanumerici.
2. Cambiare tutti gli spazi in "+".
3. Mettere insieme i nomi e valori con "=" e "&", come

`nome1=valore1&nome2=valore2&nome3=valore3`

Questa stringa è il corpo del messaggio per sottomissioni POST, o la stringa di query per sottomissioni GET.

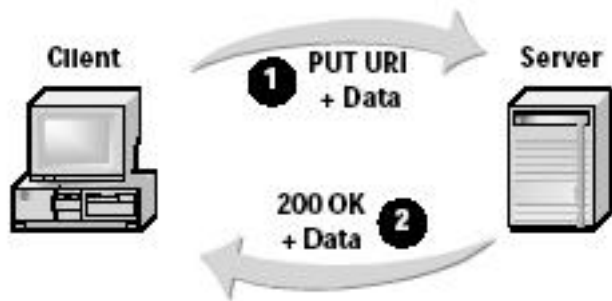
Se ad esempio una form ha un campo chiamato "nome" che è settato con "Marco", e un campo "azienda" settato con "Rossi & Bianchi", i dati della form sarebbero codificati nel modo seguente:

`nome=Marco&azienda=Rossi+%26+Bianchi`

Tipicamente il termine "codifica URL" si riferisce solo al passo 1.

Metodo PUT

Analogamente a POST, l'operazione PUT fornisce ai client un modo per fornire informazioni ai server. Tale metodo è significativamente differente da quello trattato precedentemente, anche se i due sembrano molto simili come la figura in basso mostra. Come per il metodo POST, il client nella sua richiesta manda un metodo, un URI e dati. Il server rimanda un codice di stato e, in modo opzionale, dei dati anch'esso.



La differenza tra POST e PUT sta nell'interpretazione dell'URI da parte di un server. Con il metodo POST, l'URI identifica un oggetto sul server che può processare i dati inclusi nel corpo dell'entità. Con PUT, invece, l'URI indica un oggetto in cui il server dovrebbe immettere i dati. Mentre un URI POST indica generalmente un programma o uno script, l'URI PUT è di solito il percorso o il nome di un file.

Se l'URI richiesto fa riferimento ad una risorsa già esistente, l'entità inclusa nella richiesta viene considerata come una versione modificata di quella residente sul server. Se invece esso non "punta" ad una risorsa esistente, il server crea la risorsa e le assegna l'URI richiesto.

Se è stata creata una nuova risorsa, il server deve informare l'user agent (browser Web) con una risposta 201 Created. Se invece viene modificata una risorsa esistente già, dovrebbero essere mandati i codici di risposta 200 OK oppure 204 No Content ad indicare il successo della richiesta. Se la risorsa non può essere creata o modificata con l'URI richiesto, viene data un'appropriata risposta di errore che evidenzia la natura del problema.

PUT è idempotente ma non sicuro e comunque non offre nessuna garanzia di controllo degli accessi o locking.

Metodo DELETE

Con le operazioni GET e PUT, HTTP diventa un protocollo utile per semplici trasferimenti di file. L'operazione DELETE completa questa funzione dando ai client un modo per rimuovere oggetti dai server.

Come è possibile notare in figura, il client manda un messaggio DELETE insieme all'URI dell'oggetto che il server dovrebbe rimuovere. Il server risponde con un codice di stato e, opzionalmente, con informazioni per il client.



La sola cosa interessante di tale metodo è che una risposta con successo marcata dal codice di stato 200 OK dal server non necessariamente indica che la risorsa è stata eliminata. Essa indica soltanto che il proposito del server è quello di rimuovere il contenuto. Questa eccezione lascia spazio all'intervento umano come una precauzione di sicurezza.

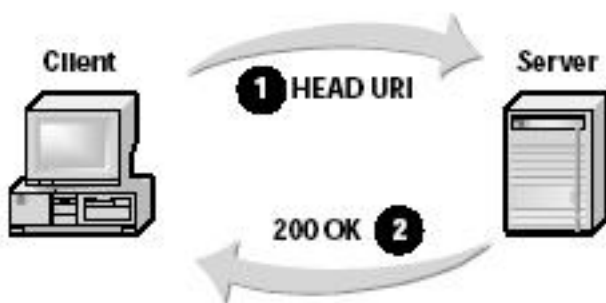
Un server può rispondere positivamente in tre modi:

- 200 OK: se la risposta include un'entità che descrive lo stato.
- 202 Accepted: se l'azione non è stata ancora effettuata.
- 204 No Content: se l'azione è stata effettuata ma la risposta non include un'entità.

Le risposte a questo metodo non possono essere cachate.

Metodo HEAD

L'operazione HEAD è quasi come la GET, tranne che il server non rimanda l'oggetto richiesto. Come mostra la figura, il server rimanda un codice di stato ma non dati (HEAD è l'abbreviativo di "header", poichè il server inserisce solo le intestazioni nel suo messaggio di risposta).



I client possono usare un messaggio HEAD quando vogliono verificare che un oggetto esiste ma attualmente non hanno bisogno di recuperare l'oggetto. I programmi che verificano i link nelle pagine Web, ad esempio, possono usare tale metodo per assicurarsi che un link si riferisce a un oggetto valido senza consumare l'ampiezza di banda della rete e le risorse del server che una ricerca piena richiederebbe.

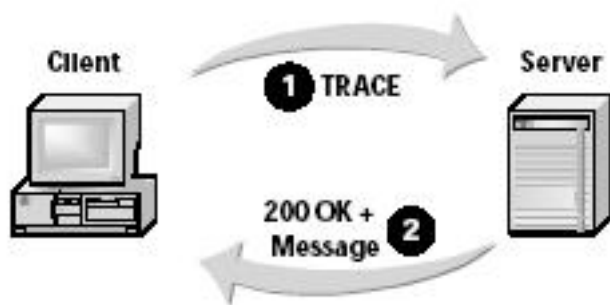
I server con cache possono usare questa operazione; essa dà a loro modo di vedere se un oggetto è cambiato senza al momento ricercare l'oggetto stesso.

HEAD è sicuro e 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.

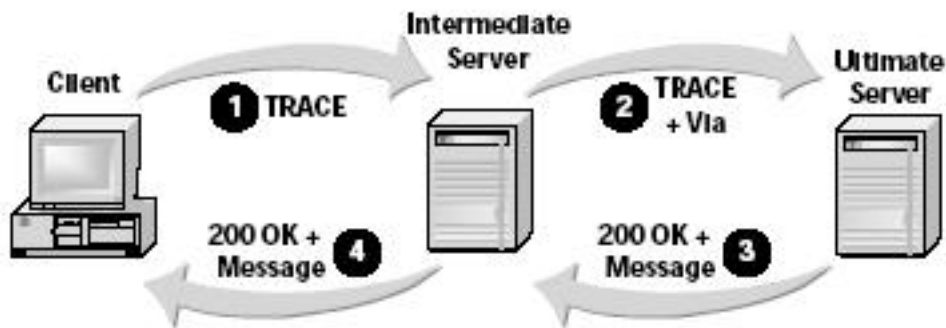
Metodo TRACE

TRACE è un altro metodo di richiesta diagnostico. Esso dà ai client un modo per scoprire il percorso di rete che i messaggi seguono per arrivare fino ai server. Quando un server riceve una richiesta TRACE risponde semplicemente copiando il messaggio TRACE stesso nel corpo dell'entità della sua risposta (l'intestazione Content-Type è difatti del tipo "message/http"). La figura in basso mostra il caso più semplice.



I messaggi TRACE sono più utili quando nella risposta a una richiesta sono coinvolti molti server. Un server intermedio, ad esempio, può accettare richieste da client ed inoltrarle a server supplementari (proxy e cache server sono esempi di server intermedi).

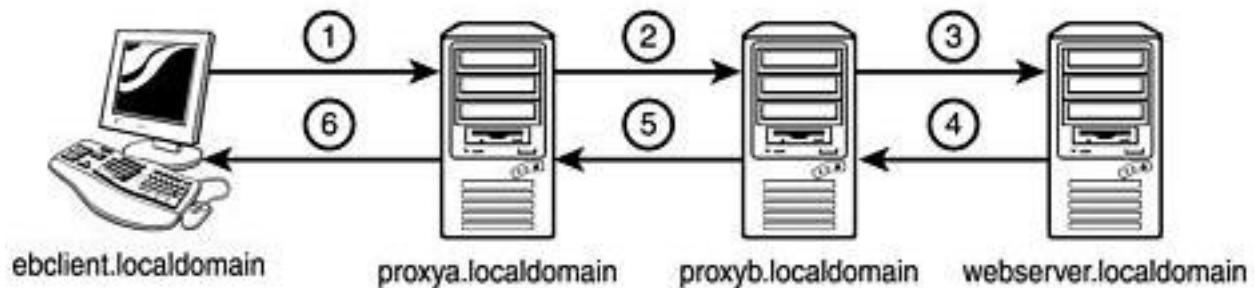
Quando è coinvolto un server intermedio, il metodo TRACE lavora come in figura.



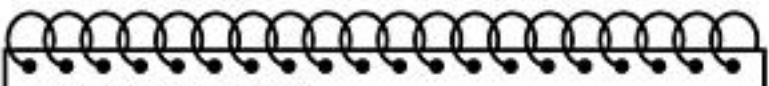
Il server intermedio modifica la richiesta inserendo l'intestazione *Via* nel messaggio. Questa è parte del messaggio che arriva al server di destinazione ed è copiata nel corpo del messaggio della risposta del server. Quando il client riceve la risposta può vedere l'intestazione *Via* nel corpo del messaggio e identificare ogni server intermedio del percorso.

Esempio di transazione TRACE

Per capire meglio l'uso del metodo TRACE, vediamo una semplice transazione HTTP che usa tale metodo di richiesta. Essa inizia con una richiesta TRACE e coinvolge due proxies tra client e server HTTP.

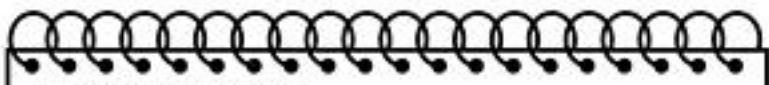


②



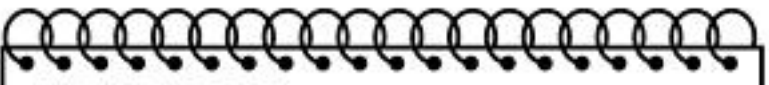
```
TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain
```

③



```
TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain, 1.1proxyb.localdomain
```

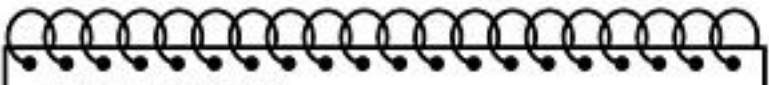
④



```
HTTP/1.1 200 OK
Date: Tue, 21 May 2002 12:34:56 GMT
Server: Apache/1.3.22 (Unix)
Content-Type: message/http

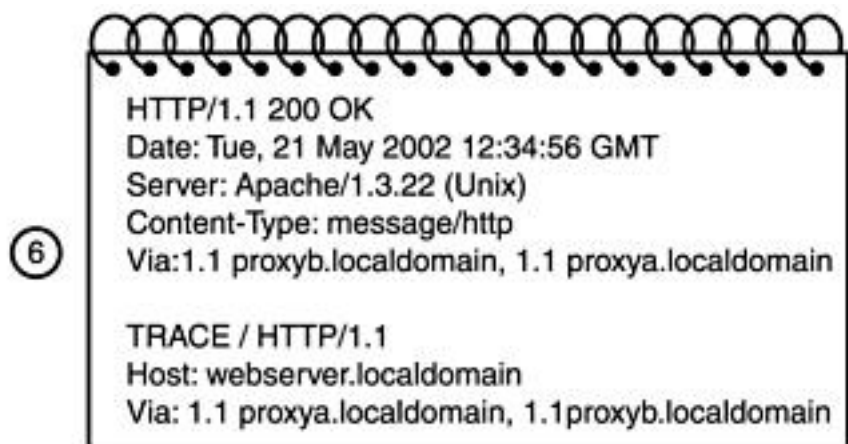
TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain, 1.1proxyb.localdomain
```

⑤



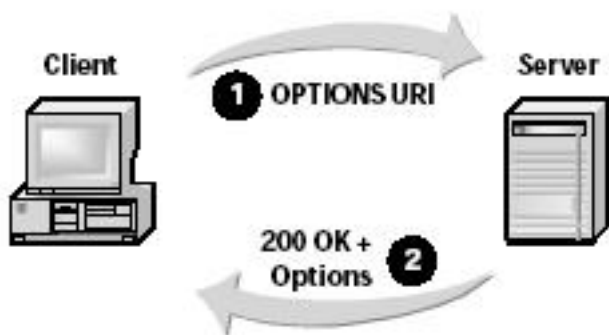
```
HTTP/1.1 200 OK
Date: Tue, 21 May 2002 12:34:56 GMT
Server: Apache/1.3.22 (Unix)
Content-Type: message/http
Via: 1.1 proxyb.localdomain

TRACE / HTTP/1.1
Host: webserver.localdomain
Via: 1.1 proxya.localdomain, 1.1proxyb.localdomain
```



Metodo OPTIONS

I client possono usare un messaggio OPTIONS per scoprire le capacità di un server Web. Lo scambio è rappresentato dalla richiesta e risposta standard, come viene illustrato in figura.



Se il client include un URI, il server risponde con le opzioni attinenti a quell'oggetto, se invece manda un asterisco (*) come URI, il server invia le opzioni generali che applica a tutti gli oggetti che mantiene.

Un client potrebbe usare un messaggio OPTIONS per determinare la versione HTTP che il server supporta o, nel caso di uno specifico URI, per sapere quali metodi di codifica può fornire il server per l'oggetto. Tali informazioni lascerebbero aggiustare al client il suo modo di interagire col server.

Il metodo OPTIONS può anche essere usato con l'intestazione Max-Forwards per testare le capacità di un proxy server intermedio.

Quando un proxy riceve una richiesta OPTIONS su un URI assoluto per il quale la richiesta di inoltra è permessa, deve verificare il campo Max-Forwards. Se il valore di tale campo è zero, il proxy non deve inoltrare il messaggio, deve invece rispondere con le sue stesse opzioni di comunicazione. Se il valore è un intero maggiore di zero, esso deve decrementare il valore del campo quando inoltra la

richiesta.

Metodo CONNECT

Il metodo di richiesta CONNECT è riservato esplicitamente ai server intermedi per creare un tunnel verso il server di destinazione. Solo ad essi è concesso fare uso di tale operazione e non ai client.

Un tunnel è differente da un proxy in quanto non interpreta le richieste HTTP (ad esempio per esaminare l'intestazione `Max-Forwards` o modificare l'intestazione `Via`).

Sia dalla prospettiva di un client che da quella di un server, un tunnel è trasparente. Esso rimane stabilito finchè la connessione TCP rimane aperta. La connessione è chiusa una volta che il server di destinazione chiude la connessione con il client.

L'uso più comune del metodo CONNECT si ha quando un client deve usare un proxy per richiedere una risorsa sicura usando SSL (Secure Sockets Layer) o TLS (Transport Layer Security).

Il client creerà un tunnel per la richiesta attraverso un proxy cosicchè il proxy semplicemente indirizzerà i messaggi HTTP ai e dai server Web senza tentare di esaminarli o interpretarli.

Lo standard non definisce come CONNECT lavora, ma solo che esso è inteso come supporto per il tunneling. Future estensioni dell'HTTP definiranno tale metodo con più dettagli.



Campi dell'intestazione

Come visto nella sezione "Messaggi Http", le richieste e risposte HTTP possono entrambe includere nessuna o più intestazioni.

Possono essere di quattro tipi:

- generali
- di richiesta
- di risposta
- del corpo dell'entità

Le intestazioni generali si applicano alle comunicazioni HTTP in generale, quelle di richiesta e di risposta ai messaggi specifici di richiesta o risposta e quelle del corpo dell'entità all'eventuale corpo del messaggio incluso in essi.

Esse cominciano con un nome seguito dai due punti (:). In alcuni casi è sufficiente solo il nome, ma nella maggior parte dei casi viene inclusa informazione addizionale dopo i due punti. Generalmente un'intestazione termina con ritorno carrello (CR) e line-feed (LF) su una linea, ma se un client ha bisogno di più di una linea o semplicemente per più leggibilità, può farlo incominciandone una con uno o più spazi o con tab (carattere ASCII 8).

Il nome dell'intestazione non è case-sensitive sebbene il valore possa esserlo, e tra i due punti e il campo del valore ci può essere un numero qualsiasi di spazi e tabulazioni.

Così, le due seguenti intestazioni sono del tutto equivalenti:

```
Intestazione1: valore_lungo_1a, valore_lungo_1b
```

```
INTESTAZIONE1:      valore_lungo_1a,  
                   valore_lungo_1b
```

Intestazioni HTTP

HTTP/1.0 definisce sedici intestazioni, sebbene nessuna sia richiesta. HTTP/1.1 ne definisce 51, e una, Host, è richiesta (obbligatoriamente) nei messaggi di richiesta di un client a un server entrambi HTTP/1.1 per evitare che il server risponda con un codice di stato 400 Bad Request.

Ecco l'elenco completo delle intestazioni definite e il tipo relativo:

Header	General	Request	Response	Entity
Accept		•		
Accept-Charset		•		
Accept-Encoding		•		
Accept-Language		•		
Accept-Ranges			•	
Age			•	
Allow				•
Authentication-Info			•	
Authorization		•		
Cache-Control	•			
Connection	•			
Content-Encoding				•
Content-Language				•
Content-Length				•
Content-Location				•
Content-MD5				•
Content-Range				•
Content-Type				•
Cookie		•		
Cookie2		•		
Date	•			
ETag			•	

Header	General	Request	Response	Entity
Expect		•		
Expires				•
From		•		
Host		•		
If-Match		•		
If-Modified-Since		•		
If-None-Match		•		
If-Range		•		
If-Unmodified-Since		•		
Last-Modified				•
Location			•	
Max-Forwards		•		
Meter		•	•	
Pragma	•			
Proxy-Authenticate			•	
Proxy-Authorization		•		
Range		•		
Referer		•		
Retry-After			•	
Server			•	
Set-Cookie2			•	
TE		•		
Trailer	•			
Transfer-Encoding	•			
Upgrade	•			
User-Agent		•		
Vary			•	
Warning	•			
WWW-Authenticate			•	

Accept

L'intestazione `Accept`, che è un'intestazione di richiesta, lascia indicare esplicitamente a un client che tipi di contenuto può accettare nel corpo del messaggio della risposta del server, così come l'ordine di preferenza per ogni tipo di contenuto accettabile.

Ecco un esempio di tale intestazione che il client potrebbe includere in una sua richiesta:

```
Accept: text/plain; q=0.5, text/html,  
       text/x-dvi; q=0.8, text/x-c
```

Come si può notare dall'esempio, l'intestazione `Accept` può includere una lista di tipi di contenuto multipli. Le virgole separano i tipi individuali e ogni tipo può includere un fattore di qualità opzionale. Quest'ultimo è un parametro di un tipo, e un punto e virgola (;) serve a separare i due.

L'esempio precedente indica che il client può accettare ognuno dei seguenti quattro tipi di contenuto:

- `text/plain; q=0.5`
- `text/html`
- `text/x-dvi; q=0.8`
- `text/x-c`

Ogni tipo di contenuto consiste di un tipo e di un sottotipo e una slash (/) che separa i due. Il primo indica il formato generale di un contenuto mentre il secondo quello specifico. Si può usare il carattere asterisco (*) in sostituzione del sottotipo, ad esempio `text/*`, o di entrambi, cioè `*/*`. Il primo esempio indica che il client può accettare qualsiasi contenuto di testo, il secondo che accetta qualsiasi contenuto.

Il fattore di qualità è un numero fra 0 e 1 (le specifiche HTTP pongono un limite di tre cifre dopo il punto decimale). Se non viene specificato nessun fattore di qualità, implicitamente si assume pari a uno. Se un server è capace di mandare tipi di contenuto multipli per una data richiesta allora sceglie quello con il più alto fattore di qualità.

Ecco l'interpretazione dell'esempio: il client preferisce che la risposta contenga un corpo del messaggio di tipo `text/html` o `text/x-c`. Se il server non può soddisfare quella preferenza, il client è disposto ad accettare il tipo `text/x-dvi`, altrimenti un contenuto di tipo `text/plain`.

Accept-Charset

I client possono includere un'intestazione `Accept-Charset` nelle loro richieste per dire ai server

quali **codifiche di caratteri** preferiscono per il corpo del messaggio incluso nelle loro risposte. Questa intestazione condivide una sintassi simile all'intestazione di base `Accept` trattata prima (così come le altre della famiglia `Accept-`).

I client possono includere una lista di differenti codifiche di caratteri e possono indicare una preferenza relativa per ognuno di esse tramite un fattore di qualità. Se assente, il server considera implicito un valore di 1.0.

Il protocollo HTTP tratta la codifica ISO 8859-1 in modo diverso dalle altre. A meno che il client elenchi esplicitamente tale codifica e le assegni un fattore di qualità diverso da uno, è di default tale valore (massimo della preferenza). Da notare che i messaggi HTTP stessi sono codificati con l'ISO 8859-1 nonostante il contenuto.

Ecco un esempio di tale intestazione:

```
Accept-Charset: utf-8, *; q=0.66
```

L'esempio indica che il client che ha inserito questa linea nel suo messaggio di richiesta preferisce una codifica `utf-8`, ma accetta qualsiasi altra (inclusa l'ISO 8859-1) con una preferenza relativa di 0.66.

L'Internet Assigned Numbers Authority (IANA) mantiene la lista delle codifiche definite che sono 235.

Da notare che questa intestazione (e tutte le intestazioni `Accept-`) si applica solo al corpo del messaggio della risposta. Essa non influenza nè la linea di stato nè le altre eventuali intestazioni della risposta, che sono codificate ISO 8859-1.

Accept-Encoding

L'intestazione `Accept-Encoding` dà ai client un altro modo per esprimere le loro preferenze per il corpo del messaggio delle risposte dei server.

In aggiunta al tipo di contenuto (`Accept`) e alla codifica dei caratteri (`Accept-Charset`), questa intestazione lascia specificare ai client quali tipi di **codifiche del contenuto** riescono a decodificare.

Il formato è lo stesso delle altre intestazioni `Accept`, una lista di codifiche accettabili, ognuna con un fattore di qualità opzionale:

```
Accept-Encoding: compress, gzip; q=0.9,
                identity; q=0.8
```

Ecco la spiegazione dell'esempio: il client richiede che la risposta sia codificata con il formato di compressione UNIX, il formato gzip (GNU zip). Se non è disponibile questo e se tutti gli altri falliscono, viene richiesta allora la codifica `identity`.

Come per le codifiche dei caratteri, i possibili valori per questa intestazione sono registrati dall'IANA per assicurare l'interoperabilità.

Accept-Language

L'intestazione `Accept-Language` è l'ultima della serie `Accept-` che dà modo ai client di esprimere le loro preferenze per la risposta (l'intestazione `Accept-Ranges` agisce in maniera piuttosto differente).

Tale intestazione permette ai client di esprimere le loro preferenze per il linguaggio del messaggio inviato.

Per designare linguaggi particolari, i client possono usare una gerarchia multilivello, con ogni livello separato da trattini. Nella sua forma più comune, il primo livello è un'abbreviazione di due lettere ISO 639 del linguaggio e il secondo livello, se presente, è un codice di due lettere ISO 3166 del paese. Ad esempio, il codice `en` rappresenta l'inglese, e il codice `en-us` indica l'inglese degli Stati Uniti.

`Accept-Language` supporta gli stessi fattori di qualità delle altre intestazioni `Accept-`, così un client può esprimere le sue preferenze fra molti linguaggi.

Esempio:

```
Accept-Language: en-uk, en; q=0.8
```

Nell'esempio un client richiede che il linguaggio sia l'inglese del Regno Unito come prima opzione e ogni altro tipo di inglese se non è disponibile la prima scelta.

Da notare che i server non privilegiano automaticamente i livelli più alti di una gerarchia di linguaggio. L'esempio seguente sarebbe soddisfatto solo da una risposta `us`; il server non invierebbe una versione in `uk`, anche se fosse l'unica disponibile:

```
Accept-Language: en-us, *; q=0.0
```

Accept-Ranges

A differenza delle altre intestazioni `Accept-`, l'intestazione `Accept-Ranges` è di risposta, così appare nelle risposte dei server piuttosto che nelle richieste dei client.

Le specifiche HTTP limitano questa intestazione a due forme.

La prima forma, mostrata nell'esempio, lascia indicare a un server che esso può accettare richieste di "range" per la risorsa:

```
Accept-Ranges: bytes
```

Come vedremo in seguito con l'intestazione [Range](#), i client possono fare richieste per un range (cioè per una parte) di byte di una risorsa piuttosto che per l'intero oggetto. Questa caratteristica è particolarmente utile per scaricare grandi file. Se un download fallisce prima di finire, il client può usare una richiesta di range per chiedere solo i byte mancanti, così evita di ricevere l'intero file di nuovo.

La seconda forma viene usata da un server se non è in grado di accettare richieste di range per una risorsa:

```
Accept-Ranges: none
```

Da notare che ai server non è richiesto di includere un'intestazione `Accept-Ranges`, nonostante possano accettare richieste di range. I client sono liberi di inoltrare ai server richieste di tale genere anche se non hanno ricevuto l'`Accept-Ranges`.

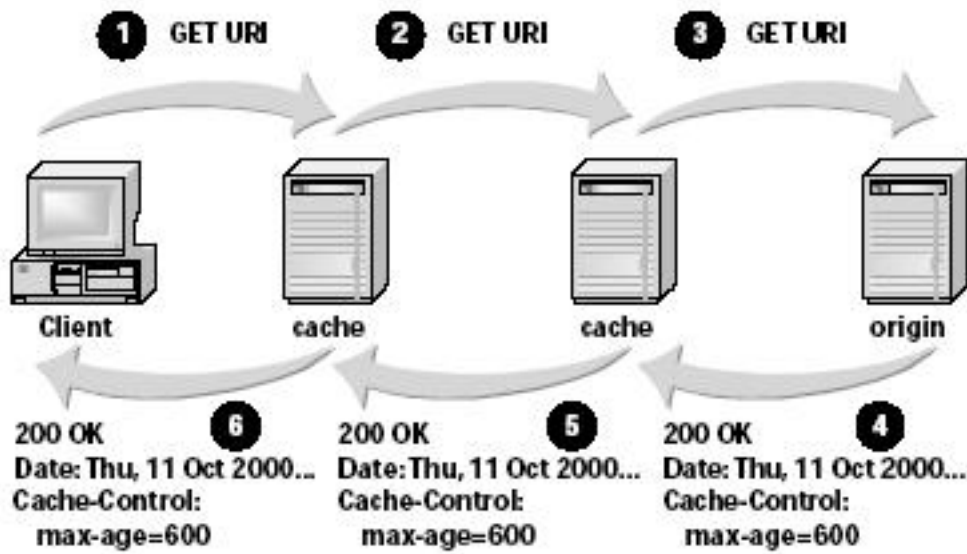
Se un client spedisce una richiesta di range a un server che non può supportarla, il server ritorna semplicemente l'intera risorsa.

Age

`Age` è un'intestazione di risposta che fa una stima, in secondi, dell'età della risorsa associata (che il client ha richiesto) da quando è stata richiesta l'ultima volta al server.

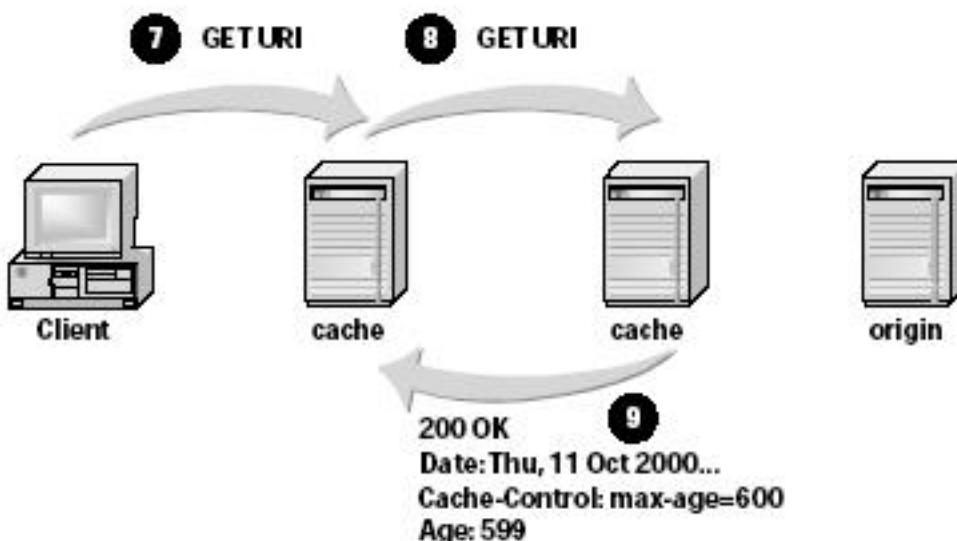
I proxy server usano questo valore per capire se una risorsa in cache è ancora valida o se è diventata obsoleta e deve essere quindi ricaricata dal server. Il valore di tale intestazione è il numero di secondi che il mittente stima che siano passati da quando il server di origine ha generato o convalidato nuovamente la risposta.

Il modo migliore per capire come funziona tale intestazione è con un esempio. Consideriamo lo scenario che comincia con la figura in basso:



La figura mostra la richiesta iniziale per una risorsa, con due proxy server tra il client e il server di origine. Come si può notare, il server include due importanti intestazioni nella sua risposta, Date e Cache-Control; la prima indica l'ora e la data del momento in cui la risposta è stata creata dal server, la seconda specifica l'età massima consentita. Nell'esempio il server indica che la risposta può essere considerata valida (e quindi essere messa in cache) per 600 secondi al massimo.

Lo scenario continua nella figura in basso, considerati trascorsi circa 10 minuti:



Un client fa una richiesta per la stessa risorsa. Il primo proxy server non ha più una copia nella sua cache, così passa la richiesta al secondo proxy server (passo 8 in figura). Questo rimanda l'oggetto con le

intestazioni che la figura include al passo 9. A questo punto il primo proxy server deve prendere un'importante decisione: è ancora valido l'oggetto che il secondo proxy server ha mandato? Per rispondere a questa domanda deve calcolare alcuni valori basati sui parametri della tabella che segue.

- Parametri per calcolare la validità di una cache -

Parametro	Interpretazione
age_value	Il valore dell'intestazione Age della risposta (passo 9); 599 nell'esempio.
date_value	La data assegnata alla risorsa dal server di origine (passo 4); 11 Ottobre 2000 nell'esempio.
now	Tempo corrente sul primo proxy server.
request_time	Tempo in cui il proxy fa la richiesta (passo 8).
response_time	Tempo in cui arriva la risposta (passo 9).

Ecco i passi per il calcolo dell'età:

1. Calcolare l'età apparente come differenza tra response_time e date_value.
2. Stimare l'età della risorsa come la massima fra l'età apparente del passo 1 e l'intestazione Age della risposta.
3. Sommare la differenza tra response_time e request_time all'età stimata del passo 2 (per dare conto dei ritardi del transito sulla rete).
4. Sommare la differenza tra now e response_time (per dare conto dei ritardi all'interno del proxy server).

Da notare che il server basa la sua stima dell'età della risorsa in base a due fonti indipendenti. Guarda esplicitamente l'intestazione Age e calcola il tempo trascorso dall'intestazione originale Date della risorsa (un accurato calcolo del tempo trascorso presume che il proxy e il server abbiano orologi sincronizzati). I passi del calcolo assicurano che il proxy scelga il valore più reale dei due tramite la sua stima, minimizzando così la possibilità di mandare una risorsa obsoleta.

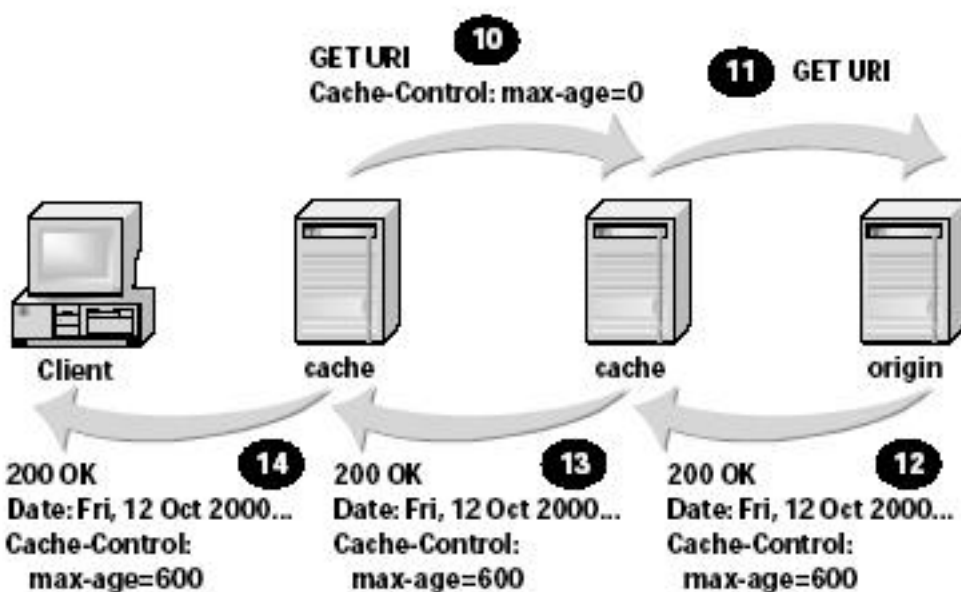
Il proxy usa questo risultato come l'età attuale della risorsa. Se questo valore eccede il max-age del server di origine, allora il proxy non deve usare l'oggetto cachato per la risposta, anzi deve inoltrare nuovamente la richiesta al server di origine.

- Continuazione esempio Age -

Per continuare l'esempio, supponiamo che sia trascorso un secondo tra la richiesta del proxy server al passo 8 e la risposta ricevuta dal secondo proxy al passo 9 (differenza tra response_time e request_time). Supponiamo inoltre che occorra un secondo di ritardo dopo che arrivi la risposta ma prima che il proxy possa processarla (differenza tra now e response_time).

In questo caso, il proxy server calcolerà che l'età dell'oggetto è di 601 secondi. Questo valore eccede il limite di 600 secondi imposto dal server di origine, così il proxy rifiuterà la risposta.

Quindi inoltrerà una nuova richiesta al server, come si può notare nella figura in basso.



Nella figura, anche il primo proxy server aggiunge l'intestazione `Cache-Control` alla richiesta del passo 10; settando la direttiva max-age a 0 in una richiesta, il primo proxy forza il secondo a fare un refresh dell'oggetto in cache.

Le specifiche HTTP limitano il valore possibile per l'intestazione Age a 2^{31} (2.147.483.648) secondi. Ogni qualvolta un valore di età eccede quel limite, viene usato il valore massimo (2^{31}).

Allow

`Allow` identifica quali metodi HTTP sono supportati da una particolare risorsa. L'intestazione elenca semplicemente tali metodi nel suo campo valore.

Il testo seguente, ad esempio, indica che una risorsa supporta i metodi GET, HEAD e PUT.

`Allow: GET, HEAD, PUT`

Questa intestazione è particolarmente utile (in effetti obbligatoria) quando il server deve inviare un codice di stato 405 `Method Not Allowed`.

I client possono anche usarla per mandare una risorsa a un server con il metodo PUT. In questo caso il client raccomanda al server quali metodi sono permessi per la risorsa. Il server, comunque, non è costretto a rispettare la raccomandazione.

Authentication-Info

L'intestazione `Authentication-Info` rappresenta il passo finale nel processo di autenticazione dell'utente composto dallo scambio di tre messaggi tra server e client.

Essa è un'intestazione di risposta che i server possono includere in una risposta che ha avuto successo, cioè in cui è inclusa la risorsa protetta, e dà ai client informazioni aggiuntive sullo scambio di autenticazione. Tra le altre cose, permette a un client di verificare l'identità del server.

Un'intestazione `Authentication-Info` contiene un elenco di parametri delimitati da virgole, ognuno dei quali ha un valore associato.

Ecco un esempio di una risposta HTTP che contiene questa intestazione:

```
HTTP/1.1 200 OK
Authentication-Info: qop="auth-int",
rspauth="5913ebca817739aebd2655bcfb952d52",
                        cnonce="f5e2d7c0b6a7f2e3d2c4b5a4f7e4d8c8b7a",
nc="00000001"
```

I parametri permessi sono i seguenti:

- `cnonce`: è il valore del parametro "cnonce" incluso nell'intestazione `Authorization` della

richiesta del client.

- nc: è il valore del parametro "nc" incluso nell'intestazione `Authorization` della richiesta del client.
 - nextnonce: è il "nonce" che il server richiede e che il client usa per l'autenticazione in richieste future.
 - qop: è il valore del parametro "qop" incluso nell'intestazione `Authorization` della richiesta del client.
 - rspauth: è il digest del server che può essere usato per autenticare l'identità del server.
-

Authorization

I client usano l'intestazione `Authorization` per autenticare se stessi o i loro utenti a un server.

In figura vediamo uno scambio di messaggi fra client e server per l'autenticazione:



Il caso più comune coinvolge il browser che fa una richiesta HTTP di un contenuto che è stato ritenuto riservato e soggetto a username e password. La risposta HTTP sarà composta da un codice di stato `401 Unauthorized` per la prima richiesta, ad indicare che il browser deve fornire una corretta intestazione `Authorization` per avere garantito l'accesso al contenuto.

Per fornire questa informazione, il browser chiederà username e password come mostrato in figura.

A screenshot of a web browser authentication dialog box. It features two input fields: 'Username' and 'Password'. Below the 'Password' field is a checkbox labeled 'Use Password Manager to remember these values.' At the bottom right, there are two buttons: 'OK' with a green arrow icon and 'Cancel' with a red 'X' icon.

Una volta che il browser si è autenticato con successo col server in questa maniera, sembrerà all'utente che tutte le richieste successive non richiedano nuovamente l'autenticazione. Data però la natura **senza stato** (*stateless*) del Web, ogni richiesta deve includere l'intestazione `Authorization` altrimenti il server risponderà con 401 `Unauthorized`.

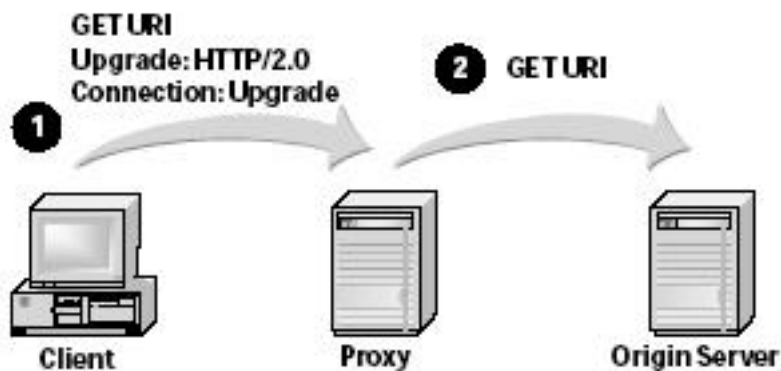
Cache-Control

L'intestazione è trattata nella sezione "[Caching](#)".

Connection

In base alle specifiche HTTP, l'intestazione `Connection` permette al mittente del messaggio (il client nel caso di richieste, il server per le risposte) di indicare ai proxy ogni altra intestazione del messaggio che non deve essere inoltrata ulteriormente.

Consideriamo l'esempio in figura.



Come si può notare, il client include due intestazioni nella sua richiesta: `Upgrade` e `Connection`. Il

proxy server, appena vede la linea `Connection`, rimuove l'intestazione indicata, `Upgrade`, dalla richiesta prima di inoltrarla.

L'intestazione `Connection` quindi identifica altre intestazioni HTTP che devono essere consegnate solo al prossimo nodo della rete.

L'uso principale di tale intestazione è però quella di gestire connessioni permanenti. I due valori possibili sono:

- `Connection: Keep-Alive`
- `Connection: Close`

Poichè una connessione può solamente essere aperta o chiusa, questi due valori, usati in richieste e risposte, permettono la gestione di tutti i possibili scenari di una transazione HTTP.

Come visto nella sezione "Connessioni", una delle distinzioni più importanti tra le versioni 1.0 e 1.1 dell'HTTP è come sono trattate le connessioni. In entrambe le versioni, sono supportate le connessioni permanenti. HTTP/1.0 usa una connessione non permanente di default, così l'uso di `Connection: Keep-Alive` è necessario per sfruttare i vantaggi della permanenza. In HTTP/1.1 sono di default invece le connessioni permanenti, così l'uso di `Connection: Close` serve ad indicare che la connessione deve essere chiusa dopo il completamento della transazione corrente.

Content-Encoding

L'intestazione `Content-Encoding` identifica ogni codifica speciale che è stata compiuta sulla risorsa. Insieme a `Content-Type`, questa intestazione specifica il formato della risorsa.

Se ad esempio un client richiede il file `example.ps.gz`, potrebbe riceverlo in una risposta con le seguenti intestazioni:

```
HTTP/1.1 200 OK
Content-Type: application/postscript
Content-Encoding: gzip
```

Le specifiche HTTP riconoscono quattro differenti codifiche di contenuto:

- `compress`: formato di codifica che usa l'algoritmo di compressione reso popolare dal programma Unix con lo stesso nome.
- `deflate`: formato di codifica "zlib" definito in RFC 1950.

- gzip: formato di codifica gzip (GNU zip).
- identity: assenza di ogni formato di codifica speciale.

Da notare che Content-Encoding è simile ma allo stesso tempo leggermente differente da Transfer-Encoding. Mentre il primo è una caratteristica intrinseca della risorsa, il secondo è applicato esternamente dal server al solo scopo di trasferire la risorsa.

L'uso principale della codifica è quello di comprimere la risorsa per facilitare e accelerare il traffico di rete.

Content-Language

L'intestazione Content-Language identifica il linguaggio naturale della risorsa inclusa.

Il formato è lo stesso dell'intestazione [Accept-Language](#) descritta precedentemente.

Da notare in più che le specifiche HTTP intendono che in questo campo vadano linguaggi umani e non di computer come C o Java.

Content-Length

L'intestazione Content-Length serve a dare la dimensione del corpo del messaggio in byte. Ecco un esempio:

```
HTTP/1.1 200 OK
Date: Tue, 17 Nov 2004 12:34:56 GMT
Content-Type: text/html
Content-Length: 102

<html>
<head>
<title>Content-Length Esempio</title>
</head>
<body>
Content-Length: 102
</body>
</html>
```

In questo esempio, il numero di byte fra il primo carattere del contenuto ("`<`" di `<html>`) e l'ultimo ("`>`" di `</html>`) inclusi è 102.

Tale intestazione è attualmente uno dei possibili differenti modi per il destinatario di determinare la dimensione di un messaggio. Esso può infatti conoscere la lunghezza del messaggio tramite la codifica del trasferimento (Transfer-Encoding) o il formato del tipo di contenuto (Content-Type), inoltre può determinare la fine di un messaggio nel momento in cui viene chiusa la connessione TCP.

Il mittente non deve includere l'intestazione in questione se il messaggio è una risposta che non permette l'inclusione del corpo dell'entità, o se quest'ultimo è codificato usando il formato "chunked", tramite cui un server incomincia a mandare una risposta mentre la sta componendo, tecnica usata per migliorare le prestazioni dei server.

Content-Location

L'intestazione Content-Location si usa nei casi in cui l'URI della risorsa che si invia differisce da quello richiesto per alcuni motivi.

Ad esempio un server può avere una risorsa disponibile in più linguaggi e può decidere di ritornare una traduzione particolare basata sull'intestazione Accept-Language nella richiesta. In casi del genere, l'intestazione Content-Location può identificare l'oggetto tradotto piuttosto che quello richiesto in origine.

Come esempio pratico, consideriamo la seguente richiesta HTTP fatta ad `httpd.apache.org`:

```
HEAD /docs/index.html HTTP/1.1
Host: httpd.apache.org
Accept-Language: it
```

Poichè la richiesta indica che la lingua preferita sia l'italiano, il server invia una versione alternativa della risorsa che è disponibile nel linguaggio richiesto. La risorsa ricevuta, indicata in Content-Location, differisce da quella richiesta:

```
HTTP/1.1 200 OK
Date: Tue, 17 Nov 2004 12:56:42 GMT
Server: Apache (Unix)
Content-Location: index.html.it
Content-Type: text/html
Content-Language: it
```

Nella pratica, questa intestazione è raramente usata. Essa non dovrebbe essere confusa con l'intestazione `Location` che appare invece piuttosto frequentemente nelle transazioni del Web. Mentre `Content-Location` specifica l'URI della risorsa presente nel corpo dell'entità, `Location` identifica un URI alternativo per la risorsa richiesta; l'oggetto stesso non è parte del corpo del messaggio quando appare quest'ultima intestazione.

Content-MD5

L'intestazione `Content-MD5` assicura che un corpo del messaggio raggiunga la sua destinazione senza subire modifiche.

Il valore di questa intestazione è il risultato dell'applicazione dell'algoritmo Message Digest 5 (MD5) al corpo del messaggio (prima di eventuali codifiche). Tale algoritmo assomiglia ad un checksum, ma usa i principi di crittografia affinché il risultato sia relativamente immune da errori non rilevabili.

Ecco un esempio. Ammettiamo che il corpo di un messaggio sia il seguente:

```
<HTML>
  <BODY>
    <P>Hello World!</P>
  </BODY>
</HTML>
```

Applicando l'algoritmo MD5 alla pagina HTML, il risultato sarà il seguente valore binario di 128 bit (16 byte, ognuno rappresentato in notazione esadecimale):

```
B2 B3 59 59 1E 96 1C 6B 0F 46 8F E5 36 BC D9 20
```

Poichè l'algoritmo crea un valore binario, e le intestazioni HTTP devono essere di testo, l'intestazione `Content-MD5` usa l'algoritmo Base-64 per la conversione da binario ad ASCII. Il risultato è il seguente:

```
Content-MD5: srNZWR6WHGsPRo/lNrZZIA==
```

L'intestazione `Content-MD5` fornisce una protezione end-to-end del contenuto così che i destinatari possono rilevare problemi introdotti dalla rete o da proxy server. Per assicurare questo comportamento, le specifiche HTTP proibiscono espressamente ai server intermedi di creare o modificare l'intestazione. Solo il server di origine (per le risposte) o il client (per le richieste) possono creare questa intestazione.

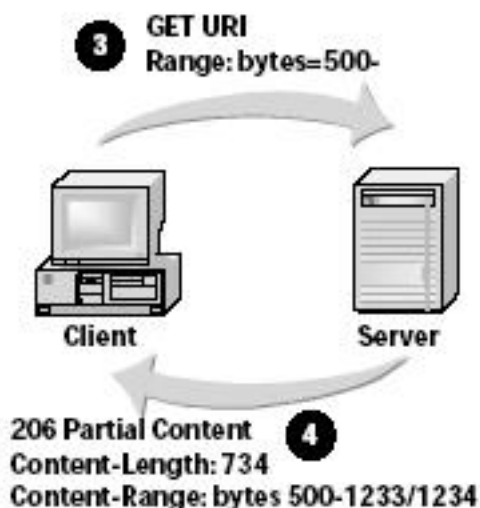
Content-Range

Quando un server include solo parte della risorsa nel suo corpo del messaggio, l'intestazione Content-Range specifica quale parte.

Questa caratteristica è particolarmente utile per riprendere il download di un file dopo che lo stesso era stato abortito. Vediamo come avviene questo processo con l'esempio seguente.



L'esempio comincia con il client che richiede un oggetto. Il server comincia a mandare la risorsa, che consiste di 1234 byte di informazione. Il trasferimento abortisce però dopo che sono stati trasferiti 500 byte. Nella sua risposta originale il server ha indicato comunque, tramite l'intestazione Accept-Ranges, che può accettare richieste di range per l'oggetto. Di conseguenza, quando il client si accorge che il trasferimento è stato abortito, non deve richiedere l'intero oggetto nuovamente. Invece, come mostra la figura seguente, esso include un'intestazione Range nella sua nuova richiesta.



Con la richiesta del passo 3, il client chiede i byte che sono rimasti da trasferire. Il server risponde di conseguenza al passo 4, dove compare l'intestazione `Content-Range`.

La prima parte del valore dell'intestazione identifica l'unità di misura; al momento, HTTP supporta solo byte. La parte successiva elenca il range di byte inclusi. Nell'esempio, la risposta del server comincia con il byte 500 e finisce con il 1233. L'ultima parte dell'intestazione fornisce la dimensione totale dell'oggetto, 1234 byte nell'esempio.

Come indicato nell'esempio, i byte in HTTP vengono numerati a partire da zero; il primo byte di una risorsa è quindi 0.

Content-Type

L'intestazione `Content-Type` identifica il tipo di oggetto che il corpo del messaggio contiene.

In una risposta a un metodo `HEAD`, essa identifica il tipo di oggetto che ci sarebbe nel corpo del messaggio, ammesso che ne fosse presente uno.

I valori possibili per tale intestazione seguono lo stesso formato tipo/sottotipo prima visto con l'intestazione `Accept`. In più, molti tipi definiti di contenuto permettono di aggiungere ulteriori parametri che danno ancora più informazione.

Ad esempio, il frammento di testo successivo indica che la risorsa è un file di testo e usa il set di caratteri ISO 8859-4:

```
Content-Type: text/plain; Charset=ISO-8859-4
```

L'esempio più comune di tale intestazione è il seguente:

```
Content-Type: text/html
```

Molti sviluppatori di programmi CGI hanno una certa familiarità con questo statement in quanto è spesso il primo pezzo dell'uscita di uno script CGI. Poichè uno script potrebbe in teoria ritornare qualsiasi tipo di contenuto, il server delega la responsabilità di indicare il tipo di contenuto allo script stesso.

I vari tipi di contenuto permessi sono mantenuti dall'IANA (Internet Assigned Numbers Authority).

Cookie e Cookie2

Entrambe le intestazioni sono trattate nella sezione "[Cookies](#)".

Date

L'intestazione `Date` indica l'ora e la data del momento in cui un messaggio HTTP è stato creato e spedito.

Così, in una richiesta indica il tempo di sistema sul client (il PC dell'utente, ad esempio) al momento in cui è stata generata. In una risposta, allo stesso modo, indica quando la stessa è stata concepita e inviata.

Da notare che i valori di `Date` si applicano al messaggio e non necessariamente alla risorsa identificata o contenuta nel messaggio stesso. L'intestazione `Last-Modified` fornisce invece il tempo della risorsa, cioè quando è stata creata o modificata per l'ultima volta.

Con la versione HTTP/1.1, il formato richiesto per tale intestazione è il seguente (definito nelle RFC 1123):

```
Date: Tue, 23 Nov 2004 12:34:56 GMT
```

Per rimanere compatibile con le precedenti implementazioni, i sistemi HTTP/1.1 dovrebbero accettare date in altri due formati:

- `Date: Tuesday, 23-Nov-04 12:34:56 GMT`
- `Date: Tue Nov 23 12:34:56 2004`

Le specifiche HTTP richiedono che i server di origine includano un'intestazione `Date` nelle loro risposte a meno che non si verifichi una delle tre condizioni seguenti:

- lo stato della risposta è `100 Continue` o `101 Switching Protocols`.
 - lo stato della risposta indica un errore del server, ad esempio `500 Internal Server Error`, e il server non può generare una data valida.
 - server senza un accurato clock di sistema.
-

ETag

L'intestazione ETag serve a dare ai server un modo più affidabile di identificazione delle risorse, specialmente per migliorare le performances del caching.

Senza l'intestazione ETag, può essere difficile per i sistemi che fanno caching identificare in modo non ambiguo le risorse richieste.

Consideriamo, per esempio, l'URL "http://www.yahoo.com/".

La risorsa ritornata può variare non solo in base al tempo, ma anche alla localizzazione geografica. Utenti nel Regno Unito possono vedere una differente home page rispetto a quella vista da utenti in Francia.

Questo problema può complicare seriamente la vita alle cache del Web. L'intestazione ETag risolve il problema fornendo un modo semplice e non ambiguo per identificare le risorse. I server di origine possono assegnare un ETag (abbreviativo di "entity tag") alle risorse non appena le spediscono.

Un'intestazione di tale genere può contenere arbitrariamente caratteri all'interno di virgolette doppie. Ecco il formato:

```
ETag: "abcd1234"
```

I valori ETag possono essere di due tipi: forte e debole.

Le risorse con lo stesso valore forte di ETag sono identiche, byte per byte. Le risorse con lo stesso valore debole di ETag sono soltanto equivalenti. I valori di quest'ultime cominciano con il prefisso "w/", come nell'esempio:

```
ETag: w/ "abcd1234"
```

Le cache normalmente usano i valori di ETag con le intestazioni [If-Match](#) e [If-None-Match](#).

Expect

Con l'intestazione Expect, un client dice a un server che si aspetta un particolare comportamento da esso.

Le specifiche HTTP definiscono Expect estensibile, ma al momento il solo uso definito per esso è quando un client si aspetta che un server gli mandi uno stato 100 Continue. In questo caso il client

include la seguente intestazione:

```
Expect: 100-continue
```

Se un server riceve un'intestazione `Expect` che non può soddisfare, allora risponde con lo stato `417 Expectation Failed`.

Se il client comunica attraverso una serie di proxy server, ci si aspetta che ognuno di essi risponda all'`Expect`. Inoltre il proxy deve passare l'intestazione al server successivo senza applicare modifiche.

Il supporto per l'intestazione suddetta è sporadico, probabilmente perchè è raramente implementata; alcuni sistemi HTTP/1.1 non la riconoscono nemmeno.

Expires

L'intestazione `Expires` indica l'ora e la data oltre le quali una risorsa può non essere più valida.

Fino alla data indicata nell'intestazione, le cache possono tenere una copia dell'oggetto e inviarla in risposta a richieste susseguenti.

Il formato di tale intestazione è quello di `Date`, ad esempio:

```
Expires: Tue, 23 Nov 2004 10:54:22 GMT
```

Ufficialmente, se un server non vuole che una risorsa venga messa in cache, mette in `Expires` il valore di `Date`. In pratica, comunque, molti server settano semplicemente il valore di `Expires` con una data già passata.

Si può vedere nella pratica anche `Expires: -1`, sebbene vietato dalle specifiche. Il significato che Microsoft Internet Explorer dà a tale linea è che la risorsa deve essere considerata scaduta immediatamente. Questo uso improprio dovrebbe però essere evitato.

Le specifiche HTTP proibiscono che un server metta in `Expires` una data superiore di un anno o più rispetto a quella attuale.

Da notare che la direttiva `max-age` dell'intestazione `Cache-Control` ha la priorità su `Expires`. Poichè HTTP introdusse `Cache-Control` con la versione 1.1 e molte implementazioni precedenti supportavano `Expires`, la combinazione di entrambe le intestazioni lascia specificare ai server differenti tempi di validità per le cache rispettivamente con versione 1.1 e precedenti. Un server

potrebbe agire in questo modo se ci fossero caratteristiche dell'1.1 aggiuntive che permettessero di estendere l'età della risorsa in modo sicuro.

From

L'intestazione From identifica l'utente di una richiesta.

Il valore di tale intestazione, come mostra l'esempio seguente, è un indirizzo email:

```
From: sadimo80@hotmail.com
```

A causa della privacy concernente queste informazioni, questa intestazione può essere disabilitata in quasi tutti i browser moderni.

Le specifiche HTTP dichiarano esplicitamente di non usare questa informazione come forma rudimentale di controllo dell'accesso o di autenticazione di alcun genere. Per la stessa ragione, questa intestazione non dovrebbe essere usata come semplice metodo di identificazione del client.

Host

L'intestazione di richiesta Host è un'aggiunta fatta ad HTTP/1.1 che permette il **multihoming**; un server con un indirizzo IP è *multi-homed* se vi risiedono più domini web, ad esempio "www.host1.com" e "www.host2.com" possono stare sullo stesso server con un singolo indirizzo IP.

Così, ogni richiesta deve specificare a quale nome di host (e possibilmente porta) si riferisce, e fa ciò tramite l'intestazione Host di cui vediamo un esempio in basso:

```
GET /path/file.html HTTP/1.1  
Host: www.host1.com
```

Ai server non è permesso di tollerare le richieste HTTP/1.1 senza tale intestazione. Se un server riceve una richiesta da un client HTTP/1.1 senza essa, deve ritornare una risposta 400 Bad Request, ad esempio:

```
HTTP/1.1 400 Bad Request  
Content-Type: text/html  
Content-Length: 111
```

```
<html><body>  
<h2>No Host: header received</h2>  
HTTP 1.1 requests must include the Host: header.  
</body></html>
```

Il requisito si applica solo a client che usano HTTP/1.1, non a future versioni. Se la richiesta userà una versione successiva all'1.1, il server potrà accettare un URL assoluto invece dell'intestazione Host.

Se la richiesta usa HTTP/1.0, il server può accettarla senza nessuna intestazione Host.

If-Match

L'intestazione di richiesta If-Match permette al browser di fare una richiesta condizionale basata sull'[Etag](#) della risorsa che si sta richiedendo.

In modo specifico, l'intestazione If-Match elenca uno o più entity tag, e il server deve processare la richiesta solo se la risorsa identificata si combina con uno degli entity tag. Il server non deve usare valori deboli di quest'ultimi per la verifica della condizione.

Questa intestazione può essere di aiuto quando dei client stanno modificando risorse immagazzinate su un server. In tali circostanze, If-Match può prevenire conflitti che possono accadere nel momento in cui più utenti modificano la stessa risorsa.

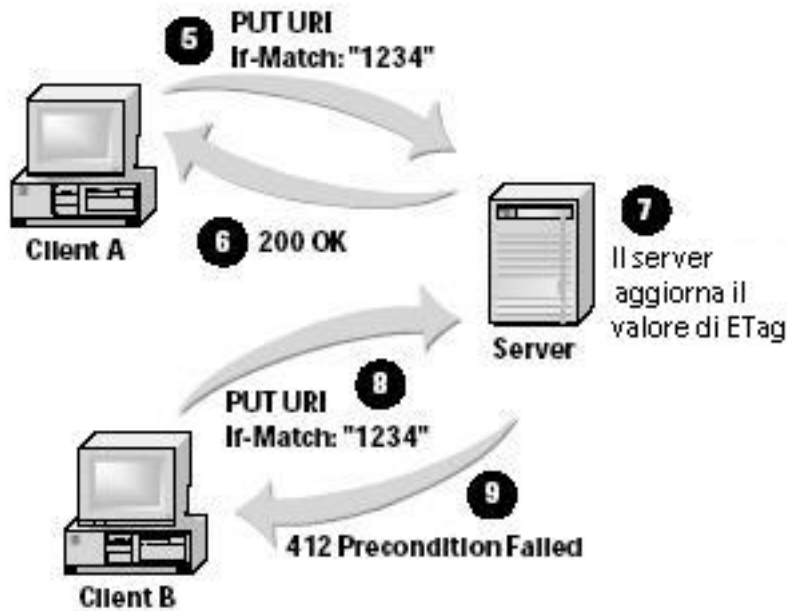
Come esempio, guardiamo lo scenario che comincia con la figura in basso.



Due differenti client richiedono una risorsa. In entrambi i casi il server invia la risorsa con un'intestazione ETag di valore 1234.

- Continuazione esempio If-Match -

L'esempio continua con la seguente figura:



Il primo client finisce di modificare la risorsa e spedisce l'oggetto modificato al server con un metodo PUT. L'intestazione If-Match dice al server di processare la richiesta solo se l'entity tag della risorsa è ancora 1234. La risorsa non è cambiata, il server verifica la condizione e accetta la richiesta.

A questo punto, la risorsa è cambiata. Essa ha preso il nuovo valore datogli dal primo client. A causa di ciò, il server deve assegnare un nuovo entity tag alla risorsa.

Successivamente, il secondo client termina le sue modifiche e tenta di mandare il nuovo oggetto al server. Questo è il passo 8 in figura. In questo caso però il valore 1234 di If-Match non si combina col nuovo entity tag della risorsa, così la condizione non è verificata e il server rifiuta la richiesta con lo stato 412 Precondition Failed.

I client possono anche usare un'intestazione If-Match con un asterisco per l'entity tag, nel seguente modo:

If-Match: *

In questo caso il client chiede al server di eseguire la richiesta solo se la risorsa esiste già, nonostante il suo entity tag corrente. Può usare questa opzione se vuole evitare che la sua richiesta PUT crei una nuova risorsa.

If-Modified-Since

L'intestazione è trattata nella sezione "[Caching](#)".

If-None-Match

L'intestazione If-None-Match ha esattamente l'effetto opposto dell'intestazione If-Match.

Quando un client la include in una sua richiesta, chiede a un server di completarla solo se la risorsa indicata ha un entity tag che differisce da quello presente nell'intestazione.

I server possono considerare valori forti di ETag per tutte le richieste e valori deboli solo con i metodi GET e HEAD.

Per le richieste GET e HEAD, l'intestazione If-None-Match lavora come If-Modified-Since. Se il server scopre che l'entity tag della risorsa coincide con uno elencato in If-None-Match, manda uno stato 304 Not Modified.

Se il client include sia If-None-Match che If-Modified-Since nella sua richiesta, quest'ultima ha la precedenza. Nel caso in cui il server crede che la risorsa sia più recente del tempo presente in If-Modified-Since, la invia interamente nonostante il valore di If-None-Match.

Come per l'intestazione If-Match, anche If-None-Match lascia usare un asterisco a un client per rappresentare un valore di entity tag.

Questo uso, illustrato dall'esempio in basso, serve a chiedere al server di accettare la richiesta solo se al momento la risorsa non esiste. Un client potrebbe usare questo valore per l'intestazione in una richiesta PUT nel caso in cui volesse essere sicuro di non sovrascrivere un oggetto esistente.

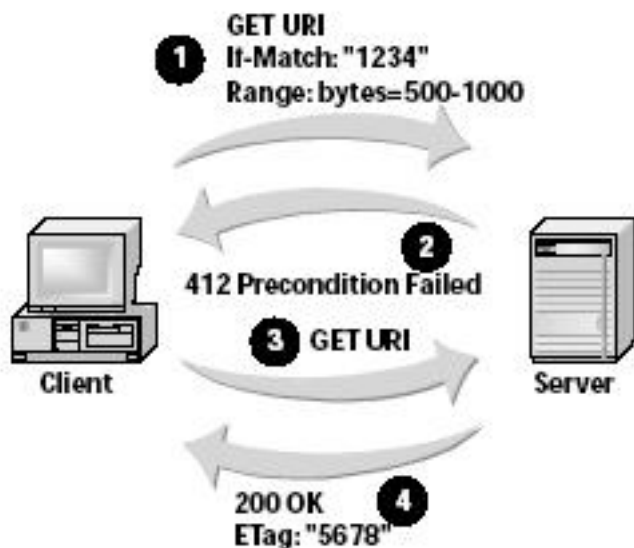
If-None-Match: *

If-Range

L'intestazione If-Range migliora le performances di client e proxy che hanno parte di un oggetto nella loro cache locale.

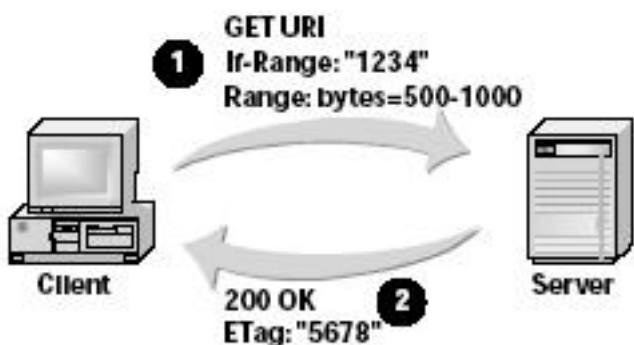
Senza If-Range, il client può richiedere due separati scambi di messaggi al server per ottenere una nuova copia dell'oggetto che è stato modificato.

La seguente figura mostra gli scambi di messaggi quando If-Range non è usata.



Nel passo 1, il client chiede 501 byte della risorsa, ma solo se l'entity tag della risorsa è ancora 1234. Quando il server riconosce che la risorsa è cambiata, risponde con uno stato 412 *Precondition Failed*. Il client deve successivamente riformulare la richiesta, questa volta chiedendo il nuovo oggetto (interamente).

L'intestazione If-Range serve a evitare questo duplice scambio di messaggi, come illustra la figura in basso:



Il client inserisce `If-Range` e `Range` nella sua richiesta. Queste due intestazioni insieme dicono al server di spedire solo il range richiesto se l'entity tag della risorsa è ancora 1234, altrimenti l'intero oggetto. Nell'esempio, l'oggetto è cambiato, così il server manda l'oggetto completo con una risposta `200 OK`.

If-Unmodified-Since

L'intestazione è trattata nella sezione "[Caching](#)".

Last-Modified

L'intestazione `Last-Modified` permette a un server di indicare l'ora e la data del momento della creazione o dell'ultima modifica dell'oggetto richiesto.

I principali benefici dell'uso di tale intestazione sono per i proxy server e i client che fanno cache, in quanto essa permette loro di datare gli oggetti inseriti in cache.

Ecco un semplice esempio:

```
Last-Modified: Tue, 30 Nov 2004 09:32:08 GMT
```

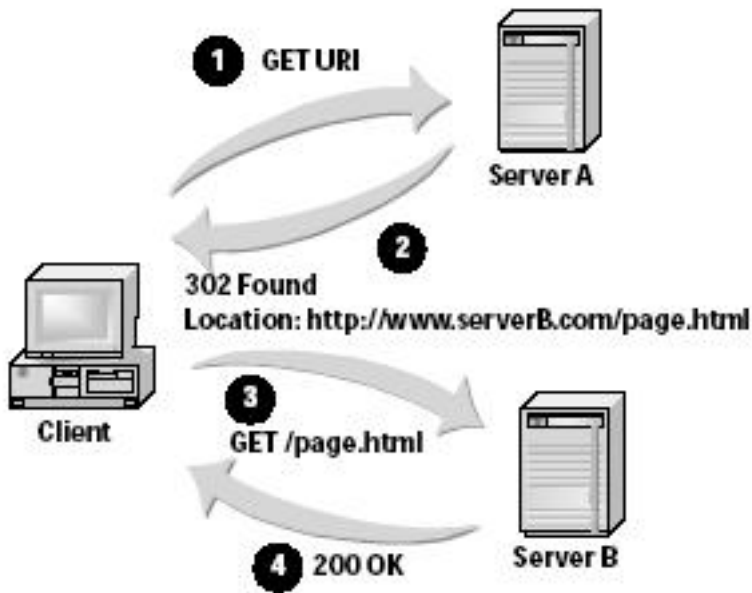
Quando un sistema ha bisogno di ottenere una nuova copia di un oggetto, può usare questa data, insieme all'intestazione `If-Modified-Since`, per evitare che il server mandi nuovamente una copia dell'intero oggetto se questo non è cambiato.

Location

I server usano l'intestazione (di risposta) `Location` per redirigere i client verso un nuovo URI di una risorsa.

L'uso più comune di tale intestazione si ha in risposte con codici di stato `3xx`, ma un server potrebbe usarla anche in una risposta `201 Created`. In questo caso, l'intestazione indicherebbe a un client dove ritrovare la copia della risorsa che esso stesso ha appena mandato al server tramite il metodo `PUT`.

La figura in basso indica l'operazione tipica di tale intestazione.



Il client manda una richiesta standard GET al server A (passo 1). Quest'ultimo non ha la risorsa, ma sa dove può essere trovata. Nella sua risposta inserisce quindi un codice di stato 302 Found e include l'intestazione Location.

Il valore dell'intestazione Location è un URI completo della risorsa. Il client usa questa informazione per fare una nuova richiesta al server indicato, che alla fine spedisce la risorsa (passo 4).

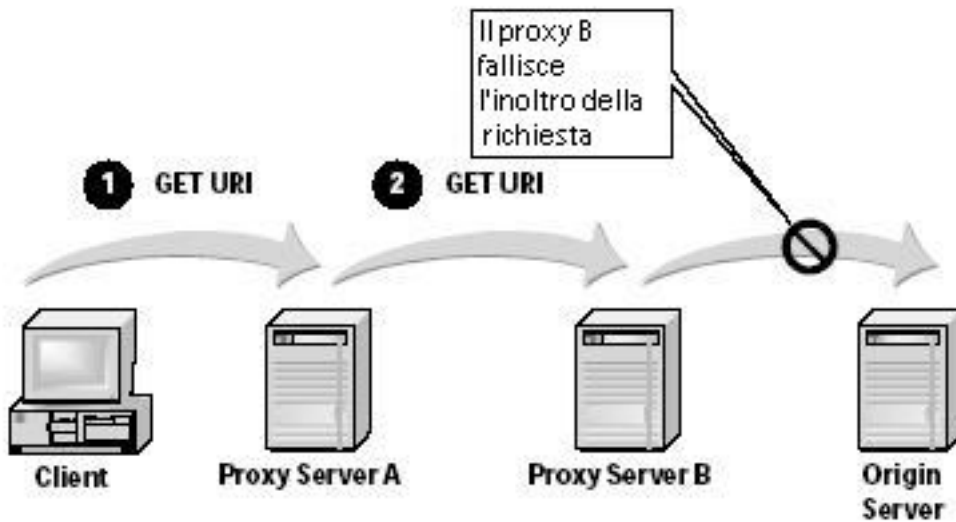
Location è completamente differente dall'intestazione Content-Location nonostante i loro nomi siano simili. Quando un server include Content-Location, dice al client da dove proviene la risorsa; Location, invece, rivela al client dove una risorsa è adesso ubicata.

Max-Forwards

L'intestazione Max-Forwards, insieme ai metodi OPTIONS e TRACE, aiuta i client a risolvere i problemi che impediscono loro di ottenere qualsiasi risposta da un server.

Ci sono due tipi di problemi particolarmente difficili da diagnosticare senza l'intestazione Max-Forwards: intermediari che falliscono e loop della richiesta.

La figura seguente mostra la situazione in cui un intermediario fallisce.



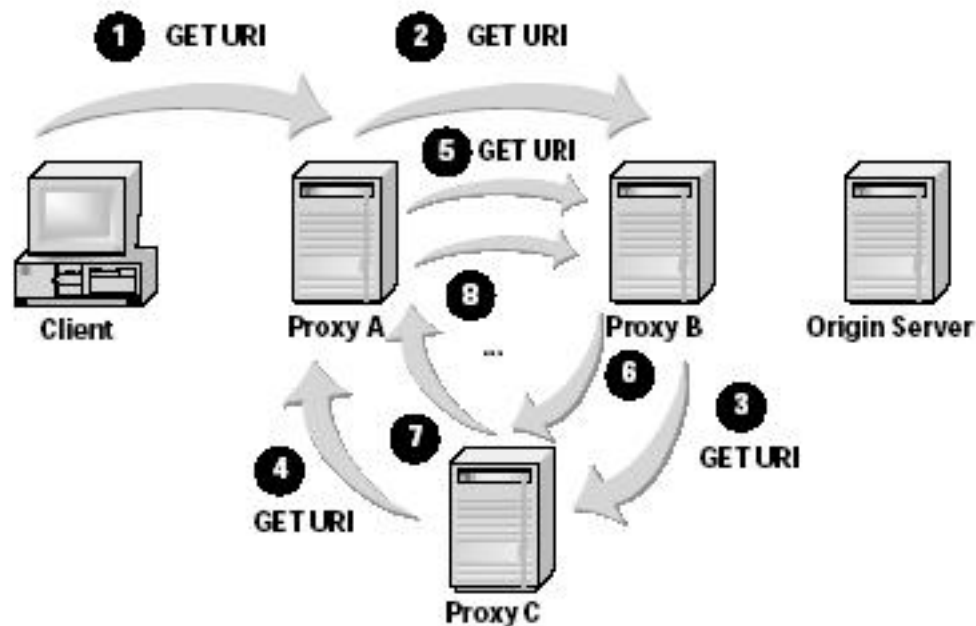
Il proxy server B riceve la richiesta al passo 2, ma fallisce l'inoltro verso il server di origine. La situazione è particolarmente irritante per il client: esso sta comunicando direttamente col proxy A e può probabilmente verificare che tale proxy sta lavorando correttamente; può anche essere in grado di verificare che il server di origine sta funzionando bene (chiamando il supporto tecnico del server, ad esempio). In qualche modo però la richiesta non percorre tutto il percorso verso il server di origine.

- Max-Forwards, loop della richiesta -

Anche il loop della richiesta impedisce a un client di ricevere qualsiasi risposta, ed è perfino più dannoso per la rete nell'insieme.

Quando avviene un loop, le richieste circolano tra i proxy server infinitamente, intasando così la rete e sprestando le risorse del server.

La figura in basso illustra questo problema.



La richiesta del client, invece di raggiungere il server di origine, passa continuamente tra i tre proxy. Questa condizione non necessariamente si verifica per il fallimento di qualche proxy.

Il proxy A, ad esempio, può credere in modo legittimo che il nodo successivo migliore per la richiesta sia il proxy B, che allo stesso modo pensa che la richiesta debba essere inoltrata al proxy C, che può legittimamente inoltrare la richiesta di nuovo al proxy A, creando così il ciclo (Se il proxy A sta inserendo l'intestazione [Via](#) correttamente, dovrebbe essere comunque in grado di scoprire il problema).

- Max-Forwards, soluzione del problema precedente-

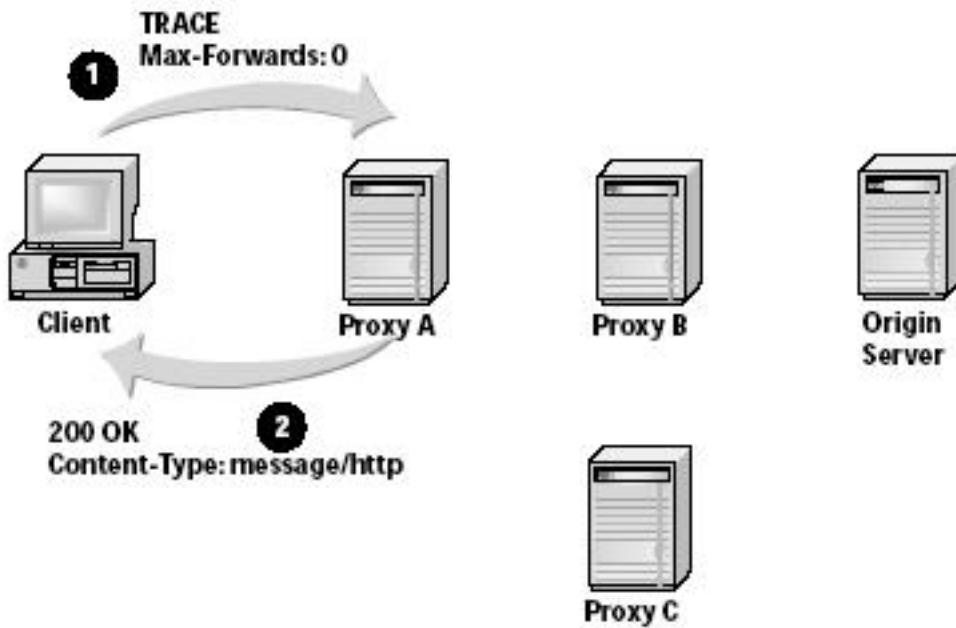
Nei casi presi in considerazione, il client non riceve mai una risposta alla sua richiesta, e finchè la modalità di fallimento persiste, il client non riceverà mai una risposta, anche ripetendo la richiesta.

Quando accade ciò, un client può utilizzare il metodo TRACE insieme alle intestazioni Max-Forwards e Via.

L'intestazione Max-Forwards limita il numero di sistemi intermedi attraverso cui può passare una richiesta.

Il client (o anche un proxy server intermedio) assegna a tale intestazione un valore iniziale, e i proxy seguenti che ricevono la richiesta lo decrementano prima di passarla. Se un server intermedio riceve una richiesta con il valore di Max-Forwards settato a zero, non deve inoltrarla ulteriormente. Al contrario, esso risponde come se fosse il server di origine.

Ecco come un client potrebbe scoprire il loop della richiesta illustrato nella precedente figura.

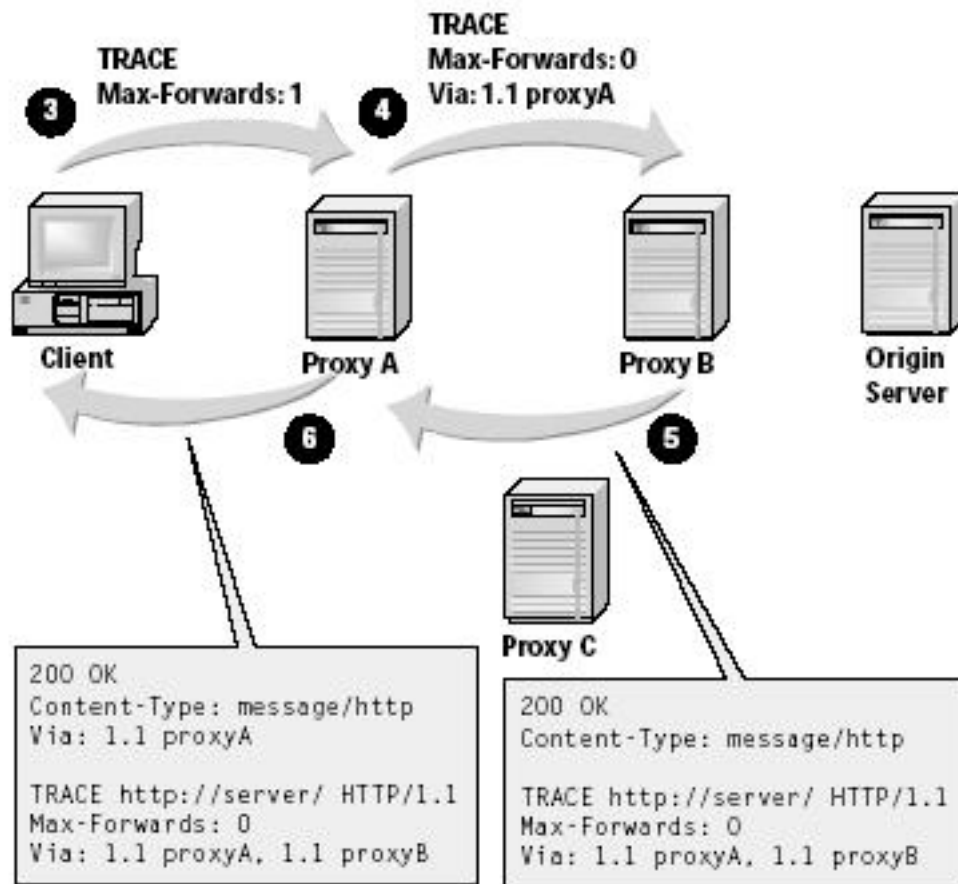


Esso comincia mandando un metodo TRACE con Max-Forwards a zero. Come mostra la figura, il primo proxy server guarda il valore di Max-Forwards e, invece di inoltrare la richiesta, risponde con uno stato 200 OK (immette anche il messaggio originale nel corpo dell'entità della sua risposta).

- Continuazione esempio soluzione -

Quando il client ottiene una risposta dal proxy A, manda un altro TRACE, questa volta con Max-Forwards settato a 1.

La figura seguente mostra cosa avviene questa volta.



Il proxy A accetta la richiesta, decrementa il valore di `Max-Forwards` e lo manda al proxy B. Come indicato in figura, il proxy A inserisce anche la sua identità nella richiesta con l'intestazione `Via` (ogni proxy intermedio inserisce la sua identità in ogni messaggio di richiesta o risposta).

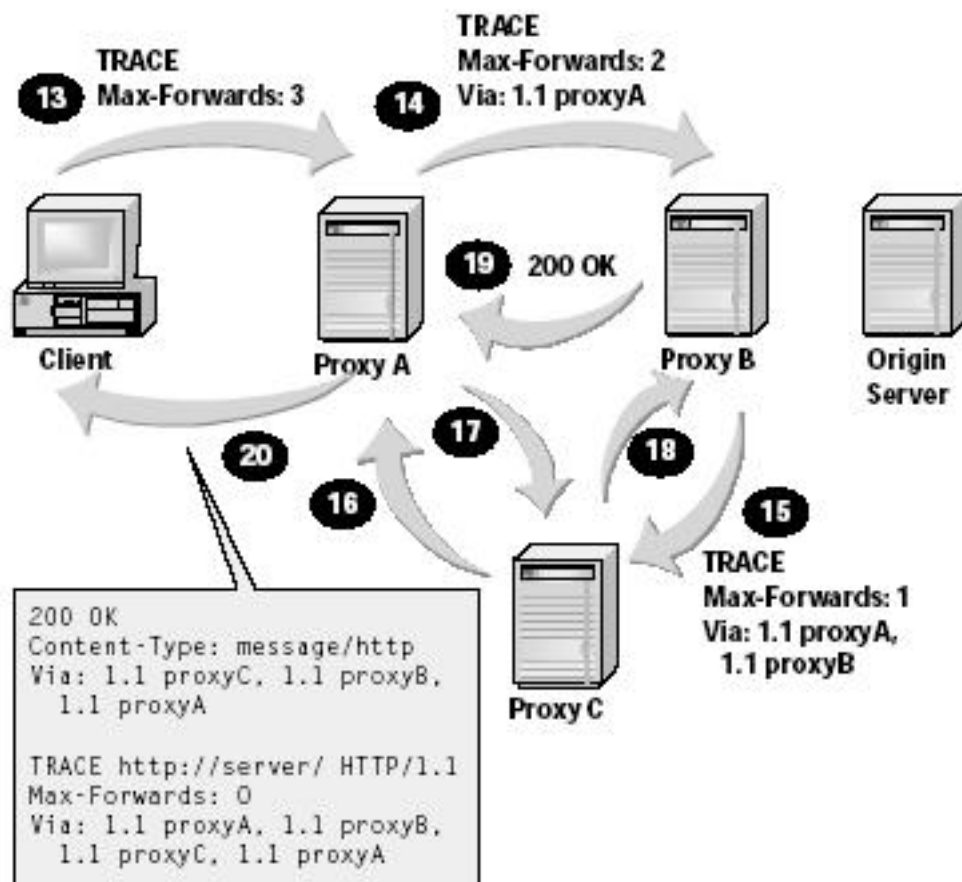
Quando al passo 4 il messaggio raggiunge il proxy B, `Max-Forwards` gli indica di non inoltrare ulteriormente il messaggio. Invece il proxy B ritorna la sua stessa risposta al client.

I passi 5 e 6 mostrano come la risposta del proxy B viaggia indietro fino al client. Quando quest'ultimo riceve la risposta al passo 6, ottiene importanti e nuove informazioni sul problema. Adesso sa, tramite il corpo del messaggio, che il nodo successivo al proxy A è il proxy B.

- Continuazione e finale esempio soluzione -

Il client continua a sondare il percorso in questo modo. Con ogni richiesta incrementa l'iniziale valore di `Max-Forwards` di 1.

Alla fine riceve la risposta del passo 20 in figura.



Questa risposta permette al client di scoprire il loop. Tramite l'intestazione *Via* nel corpo del messaggio, esso può vedere che la richiesta è passata due volte dal proxy A, è così si è generato un loop.

I client possono usare un processo simile per scoprire fallimenti dei server intermedi. Essi cominciano con un valore di *Max-Forwards* pari a zero e lo incrementano ogni volta che ottengono una risposta alla richiesta *TRACE*.

Quando non arrivano risposte, il client viene a conoscenza del nodo dove la richiesta fallisce.

Meter

L'intestazione è trattata nella sezione "[Caching](#)"

Pragma

L'intestazione *Pragma* è un avanzo delle prime versioni dell'HTTP.

Con HTTP/1.1 c'è solo un formato possibile per essa:

`Pragma: no-cache`

Ufficialmente, tramite questa intestazione, i client possono indicare che non vogliono che nessun server intermedio replichi alle loro richieste con risposte prese dalla cache, ma che ogni richiesta sia inoltrata al server di origine.

In pratica, molti server includono la precedente linea nelle loro risposte per dire ai proxy server di non salvare la risposta nella cache. Questo comportamento, anche se non è mai stato standardizzato, è così comune al punto che molti proxy lo adottano.

Un'alternativa più sicura per i server che non vogliono che le loro risposte siano cachate è quella di includere un'intestazione `Expires` con una data nel passato.

In futuro, tutti i server intermedi saranno conformi alla versione 1.1. A quel punto, server e cache potranno usare entrambi l'intestazione `Cache-Control: no-cache`, che è il metodo preferito di controllo della cache in HTTP/1.1.

Proxy-Authenticate

L'intestazione `Proxy-Authenticate` permette ai proxy server di autenticare un client.

Includendo questa intestazione in una risposta, il proxy chiede al client di riformulare la richiesta includendo le sue credenziali di autorizzazione.

I proxy server devono includere `Proxy-Authenticate` in ogni risposta sempre insieme a un codice di stato 407 `Proxy Authentication Required`.

Questa intestazione è simile a `WWW-Authenticate`, tranne per il fatto che è generata da proxy server piuttosto che da server di origine.

Proxy-Authorization

Un client risponde a una domanda di autenticazione formulata da un proxy server includendo un'intestazione `Proxy-Authorization` quando riformula la richiesta.

L'uso (così come la sintassi) di questa intestazione è esattamente uguale a quello dell'intestazione di richiesta [Authorization](#).

Range

L'intestazione Range permette a un client di richiedere parte di una risorsa invece dell'intero oggetto.

I client dovrebbero essere sempre preparati a ricevere l'intero contenuto, in quanto un server che riceve richiesta di un range non valido o che non capisce tale intestazione risponde mandando appunto tutta la risorsa.

Ecco un esempio (da notare che la numerazione dei byte inizia da zero in HTTP/1.1):

```
Range: bytes 500-999, -4
```

Con questa linea, un client chiede i secondi 500 byte (dal byte 500 al byte 999, inclusi) e gli ultimi quattro byte di una risorsa.

Se un server è capace di soddisfare la richiesta del client, ritorna un codice di stato 206 `Partial Content`. Esso include anche l'intestazione `Content-Range` nella sua risposta.

Se il server non può ritornare il range richiesto ma può rispondere con l'intero oggetto, inserisce uno stato 200 `OK`.

Referer

L'intestazione `Referer` compare nelle richieste del client in modo che il server possa identificare dove il client ha ottenuto l'URI della sua richiesta.

Come esempio, consideriamo l'home page dell'IETF (Internet Engineering Task Force), che si trova all'indirizzo <http://www.ietf.org>.

Tale pagina web contiene un link al sito dell'IANA (Internet Assigned Numbers Authority). Il frammento HTML per il link è il seguente:

```
<A href="http://www.iana.org">IANA</A>
```


Se l'utente clicca sul link, il browser manda una richiesta GET all'indirizzo www.iana.org. Poichè il link compare nella pagina www.ietf.org, la richiesta evidenzierà la pagina dell'IETF nell'intestazione `Referer`:

```
GET / HTTP/1.1
Referer: http://www.ietf.org/
Accept-Language: it
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: www.iana.org
Connection: Keep-Alive
```

Consideriamo ora il caso in cui un utente ha una pagina web sul suo server locale (<http://127.0.0.1/>) contenente un link a un URL remoto. Il browser includerà un'intestazione `Referer` quando manderà la richiesta GET al server remoto (quando il link sarà cliccato) nel seguente modo:

```
Referer: http://127.0.0.1/
```

Retry-After

I server usano l'intestazione `Retry-After` per dire a un client quando deve riprovare a riformulare la richiesta.

L'intestazione può specificare una data, cosicchè l'esempio seguente chiede a un client di aspettare fino al primo Gennaio del 2005 prima di riformulare la richiesta:

```
Retry-After: Fri, 31 Dec 2004 23:59:59 GMT
```

Oltre a specificare una data, si può semplicemente indicare un numero di secondi.

L'esempio seguente dice al client di aspettare due minuti (120 secondi) prima di riprovare:

```
Retry-After: 120
```

I server possono usare questa intestazione in risposte con codici di stato 503 `Service Unavailable` o con uno qualsiasi della classe 3xx.

Server

Tramite l'intestazione `Server`, un server identifica il software che usa per implementare l'HTTP.

Tale intestazione è per il server la versione dell'intestazione `User-Agent`.

Gli esempi seguenti mostrano alcuni valori possibili per essa che possono essere trovati sul web attualmente.

`Server: Apache/1.3.0 (Unix) (Red Hat/Linux)`

`Server: IBM-Planetwide/10.45
Domino-Go-Webserver/4.6`

`Server: Microsoft-IIS/5.0`

`Server: NaviServer/2.0 AOLserver/2.3.3`

`Server: Netscape-Enterprise/3.6 SP3`

`Server: Xitami`

Set-Cookie2

L'intestazione è trattata nella sezione "[Cookies](#)".

TE

L'intestazione di richiesta `TE` dice a un server quali codifiche di trasferimento può accettare il client in una risposta, e può anche indicare le preferenze del client.

`TE` è molto simile all'intestazione `Accept-Encoding`, eccetto che si applica alle codifiche del trasferimento piuttosto che a quelle del contenuto.

Il formato di tale intestazione è molto simile a quello dell'intestazione `Accept-Encoding`. Il valore è rappresentato da un elenco di nomi di codifiche di trasferimento, ognuno con un fattore di qualità opzionale, separati da virgole.

Ad esempio, la linea seguente indica che il client può accettare le codifiche "gzip" e "deflate", ma preferisce la prima che ha un fattore di qualità più alto (come per le altre intestazioni, se il client non indica esplicitamente un fattore di qualità per una particolare opzione, il server considera il suo valore pari a 1).

```
TE: gzip, deflate; q=0.9
```

Oltre alle codifiche di trasferimento standard, l'intestazione TE definisce un valore speciale per identificare la codifica di trasferimento "a pezzi" (con questa caratteristica i server possono incominciare a mandare una risposta mentre la stanno componendo). Il valore è semplicemente "trailers", come nell'esempio seguente.

```
TE: trailers
```

Da notare che non c'è bisogno che un client inserisca la codifica del trasferimento "a pezzi" nell'intestazione TE, in quanto tutti i client HTTP/1.1 devono essere preparati ad accettare tale tipo di codifica. L'uso è quindi opzionale; questo valore per l'intestazione permette a un client di dire esplicitamente che capisce quel particolare formato.

Trailer

Client e server possono includere l'intestazione Trailer quando usano la codifica di trasferimento "a pezzi" per il corpo del messaggio ([chunked](#)).

Questa intestazione elenca alcune altre intestazioni HTTP che compaiono dopo il corpo del messaggio, piuttosto che nella loro normale posizione prima del corpo stesso.

Ci sono tre intestazioni che non possono comparire nell'intestazione Trailer:

- Transfer-Encoding
- Content-Length
- Trailer

L'esempio seguente mostra una risposta HTTP che usa questo metodo:

```
HTTP/1.1 200 OK
Content-Type: text/html
Transfer-Encoding: chunked
Trailer: Date
```

```
7f
<html>
<head>
<title>Esempio Transfer-Encoding</title>
</head>
<body>
<p>Attendere il completamento della transazione...</p>
2c
<p>Transazione completa!</p>
</body>
</html>
0
Date: Tue, 30 Nov 2004 23:52:10 GMT
```

Questa allocazione è utile per l'informazione che non può essere determinata accuratamente fino a che la risposta non sia completamente generata. Ad esempio, la risposta precedente può avere un'attesa significativa tra il primo e il secondo pezzo trasmesso.

Transfer-Encoding

L'intestazione Transfer-Encoding identifica il formato di codifica del trasferimento di un corpo del messaggio.

Sebbene le specifiche HTTP/1.1 definiscono questa intestazione in modo generale, le implementazioni correnti la usano quasi esclusivamente per indicare la codifica di trasferimento "a pezzi":

```
Transfer-Encoding: chunked
```

Gli sviluppatori della versione 1.1 dell'HTTP hanno creato tale codifica per migliorare le performance dei server. Con questa caratteristica, i server possono incominciare a mandare una risposta mentre la stanno componendo; senza di essa sono invece costretti a ritardare la risposta finchè l'intero messaggio è completo.

C'è un problema: i server HTTP/1.1 devono indicare la dimensione dei loro messaggi di risposta, cosa che non succedeva nelle versioni precedenti, dove i server mandavano la loro risposta e poi chiudevano la connessione TCP e i client potevano capire di aver ricevuto l'intera risposta quando la connessione chiudeva.

Con la versione 1.1, le connessioni permanenti sono il comportamento di default, e la connessione non

viene chiusa dopo ogni risposta. I client hanno comunque ancora il bisogno di capire se hanno ricevuto tutto il messaggio. L'intestazione `Content-Length` è la soluzione più semplice a questo problema; quando il server la include in una risposta, per il client c'è soltanto da contare i byte per sapere quando ha la risposta completa.

Nonostante sia semplice e facile da usare, `Content-Length` introduce un nuovo problema. Essendo un'intestazione, essa è una delle prime parti di una risposta e in particolare precede il corpo del messaggio. Prima che un server possa calcolare il suo valore, deve sapere la dimensione totale del corpo del messaggio. Questa restrizione impone che prima che un server incominci a mandare la risposta, deve comporre tutto il corpo del messaggio e calcolarne la dimensione.

Se il corpo del messaggio è di grosse dimensioni, e il server genera il messaggio dinamicamente, il ritardo che ne risulta può degradare in modo significativo le performance del server stesso. Un approccio più efficiente consiste nel permettere ai server di incominciare a mandare la risposta non appena iniziano a comporre il corpo del messaggio. Nel momento in cui il server crea nuove parti del messaggio di risposta, le manda immediatamente al client. Questo approccio è esattamente il principio di base del trasferimento "a pezzi".

- Trasferimento "a pezzi" (*chunked*) -

Tramite la codifica di trasferimento "a pezzi", il server divide il corpo del messaggio in uno o più parti (*chunks*).

Nella sua risposta, il server manda queste parti una dopo l'altra. Ogni parte è preceduta da una linea che indica la sua dimensione in esadecimale. L'ultima parte ha una dimensione di zero byte.

Di seguito è possibile notare un messaggio di risposta con tre parti.

```
HTTP/1.1 200 OK
Date: Fri, 4 Dec 2004 22:08:34 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
```

```
1a
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0a
0123456789
0
```

La terza parte ha una dimensione nulla (0 byte), così solo le prime due hanno del contenuto. La

dimensione totale del corpo del messaggio è di 36 byte (il primo pezzo è di 26 byte, 1A₁₆, il secondo di 10, 0A₁₆).

Ecco come potrebbe essere mandato lo stesso corpo del messaggio senza la codifica di trasferimento "a pezzi".

```
HTTP/1.1 200 OK
Date: Fri, 4 Dec 2004 22:08:34 GMT
Content-Type: text/plain
Content-Length: 36

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

Upgrade

L'intestazione Upgrade permette a un client e un server di negoziare un cambio del protocollo di comunicazione.

Il nuovo protocollo può essere una versione più recente dell'HTTP o un protocollo completamente differente come ad esempio il TLS (Transport Layer Security).

Il client propone il protocollo da usare includendo un'intestazione Upgrade nella sua richiesta:

```
GET http://www.banca.it/accett.html?546684684651 HTTP/1.1
Host: www.banca.it
Upgrade: TLS/1.0
Connection: Upgrade
```

Il server può rispondere a questa richiesta con un codice di stato 101 Switching Protocols, e include anch'esso un'intestazione Upgrade:

```
HTTP/1.1 101 Switching Protocols
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
```

Da notare che sia la richiesta che la risposta includono la linea `Connection: Upgrade`. Questa deve sempre apparire quando l'intestazione Upgrade è usata, poichè ogni cambio di protocollo si applica solo all'immediata connessione tra il client e il primo server.

Se un client vuole cambiare il suo tipo di comunicazione con un server, può usare il metodo `CONNECT`

per stabilire una connessione virtuale e poi modificare quel tipo di connessione con l'uso di un altro protocollo di comunicazione

L'esempio mostra anche che la risposta del server elenca una serie di protocolli nell'intestazione Upgrade. Questo perchè il TLS utilizza l'HTTP come mezzo di trasporto, così come l'HTTP si appoggia sul TCP.

User-Agent

L'intestazione User-Agent è per il client la versione dell'intestazione Server.

Tramite essa, un client indica la specifica implementazione HTTP che sta usando.

L'esempio seguente mostra come il browser Netscape Navigator identifica se stesso su un Apple Macintosh.

```
User-Agent: Mozilla/4.x (Macintosh)
```

Da notare l'uso sempre più comune di includere oltre al tipo di browser anche il sistema operativo in tale intestazione.

Vary

Tramite l'intestazione Vary, i server danno ai proxy delle linee guida addizionali per la gestione delle loro cache locali.

Vary elenca altre intestazioni HTTP che, in aggiunta all'URI, determinano quale risorsa deve essere mandata dal proxy server a un client.

Ad esempio, alcuni server possono inviare risorse differenti in base al valore dell'intestazione User-Agent nella richiesta del client (possono avere una pagina ottimizzata per Microsoft Internet Explorer e una differente per Netscape Navigator). In casi del genere, il server dovrebbe includere un'intestazione Vary nella sua risposta.

```
HTTP/1.1 200 OK
Date: Fri, 4 Dec 2004 15:22:58 GMT
Content-Type: text/html
```

Vary: User-Agent

Un proxy server sa così che, a richieste successive, può ritornare una copia in cache di questa risposta, ma solo se queste richieste hanno lo stesso valore di User-Agent della richiesta originale. Un valore differente di quest'ultima forza la cache a consultare nuovamente il server.

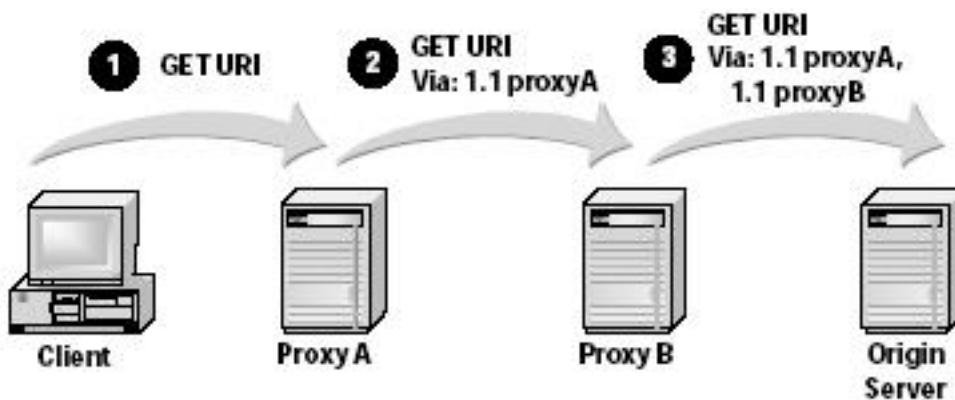
Un asterisco come valore dell'intestazione indica che il contenuto della risposta è influenzato da altri parametri oltre che dalle intestazioni HTTP.

Via

L'intestazione Via traccia il percorso compiuto da un messaggio che viaggia tra proxy server.

Le specifiche HTTP richiedono che ogni server intermedio che maneggia una richiesta o una risposta identifichi se stesso con un'intestazione Via prima di inoltrare il messaggio.

La figura seguente mostra come tale intestazione cresca nel percorso che una richiesta effettua da un client a un server.

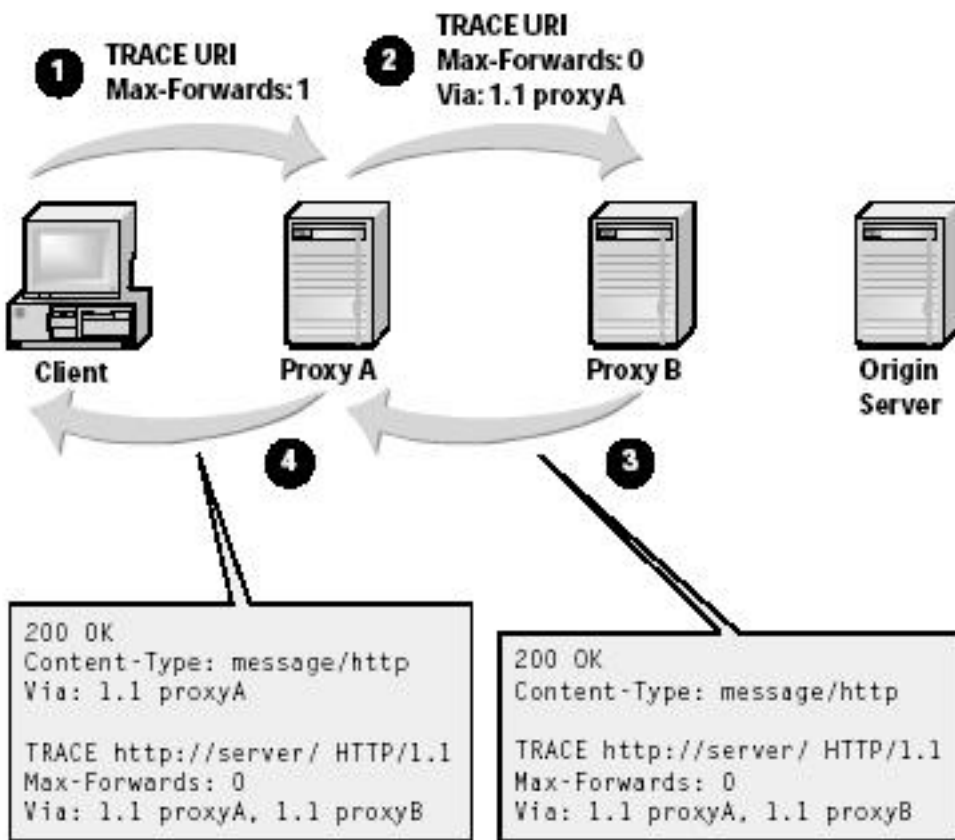


Il primo proxy crea l'intestazione Via e aggiunge la sua identità al valore (sebbene la figura mostra l'identità come ProxyA, il server usa normalmente un nome di dominio completo). L'1.1 che precede il nome del proxy è la versione HTTP usata.

Quando la richiesta passa dal proxyB, quest'ultimo semplicemente inserisce il suo nome all'intestazione Via esistente.

Un'aspetto importante è che i proxy server creano o modificano questa intestazione prima di compiere ogni altra azione sul messaggio. Ad esempio, un proxy può ricevere una richiesta TRACE con Max-Forwards di valore nullo (=0), ad indicare che non può inoltrare la richiesta ulteriormente, come

avviene al proxyB della figura seguente.



Prima che esso risponda alla richiesta TRACE deve comunque inserire la sua identità in Via. Dopo aver fatto ciò, genera la risposta del passo 3, per cui nel corpo del messaggio compare la sua identità in Via.

Warning

L'intestazione Warning dà informazioni aggiuntive su una risposta, solitamente per mettere in allarme l'utente su problemi potenziali della cache.

Il formato è il seguente, la data è opzionale:

```
Warning: 110 proxy.com "Response is stale"
        Fri, 4 Dec 2004 15:22:58 GMT
```

Warning può includere molti avvertimenti individuali, separati da virgole. Il primo campo è un codice di avvertimento e quello successivo identifica il server che lo ha creato. La stringa che segue è una spiegazione dell'avvertimento in linguaggio naturale, adatta agli utenti. Il campo finale, che fornisce il tempo in cui l'avvertimento è stato generato, è opzionale.

HTTP/1.1 definisce i seguenti codici di avvertimento, in modo simile a come avviene per i codici di stato.

Codice	Stringa	Significato
100	Response is stale	Il proxy ha immesso un oggetto obsoleto nella sua risposta (forse perchè il client ha usato la direttiva <code>max-stale</code>).
111	Revalidation failed	Il proxy non può verificare se l'oggetto è ancora valido (forse perchè non può contattare il server).
112	Disconnection operation	Il proxy è stato intenzionalmente disconnesso dalla rete.
113	Heuristic expiration	Il proxy ha supposto che l'oggetto fosse ancora valido, ma invece è vecchio da più di 24 ore.
199	Miscellaneous warning	Avvertimento arbitrario.
214	Transformation applied	Il proxy ha modificato l'oggetto in qualche modo (forse cambiando il formato delle immagini per risparmiare spazio in cache).
299	Miscellaneous persistent warning	Avvertimento arbitrario persistente.

Quando un proxy riceve un'intestazione `Warning` con una data che differisce dall'intestazione `Date` nella risposta, cancella quel particolare avvertimento dall'intestazione. Se questa operazione lascia `Warning` senza avvertimenti, il proxy rimuove anch'essa. Tale comportamento assicura che gli

avvertimenti non siano propagati inappropriatamente attraverso una rete di proxy server.

WWW-Authenticate

L'intestazione `WWW-Authenticate` permette l'autenticazione di un client a un server.

Includendola in una risposta, di solito con un codice di stato `401 Unauthorized`, il server chiede al client di riformulare la richiesta includendo le sue credenziali di autorizzazione.



`WWW-Authenticate` dà così al client le informazioni di cui necessita per identificarsi alla successiva richiesta di accesso alla risorsa.

Caching

Il Caching è uno dei modi più comuni per migliorare le performance dell'HTTP e, specialmente in Internet, è anche uno dei più efficienti.

Come si è visto, le specifiche HTTP riconoscono l'importanza del caching tramite l'appoggio esteso a tale tecnologia all'interno del protocollo stesso.

Lo scopo principale di una cache è di immagazzinare una copia di un oggetto per prevenire la necessità di recuperarla nuovamente in seguito. Ci sono tre tipi principali di caching:

- Caching sul server
- Caching sul client
- Caching tramite proxy server

Abbiamo già visto che l'utente deve connettersi a un server per richiedere documenti o eseguire script. Le risposte a queste richieste possono essere lente, specialmente per server che sono lontani dal client, connessi tramite un link di rete lento o molto intasato. L'intero documento deve essere trasferito sulla rete per ogni richiesta, operazione che richiede spreco della capacità della rete, specialmente per file di grandi dimensioni come ad esempio le immagini, così i costi si accumulano.

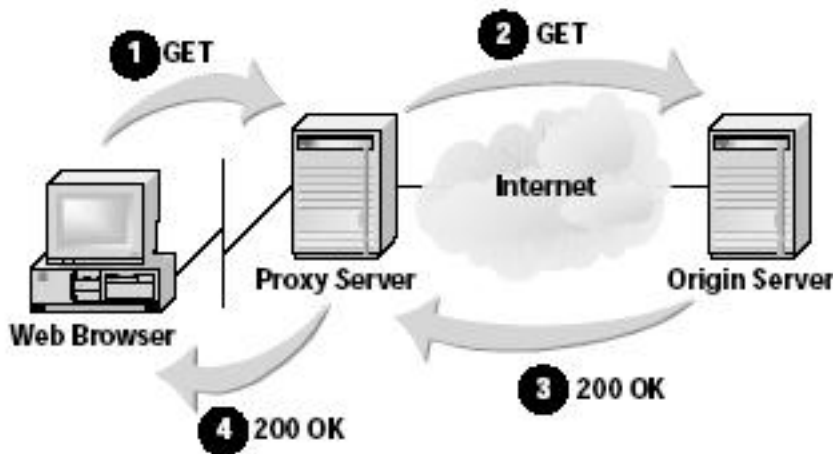
Per le reti di area locale (LAN), il problema può essere acuto. Se ci sono dozzine di utenti sulla rete locale che stanno usando il Web e ognuno di essi richiede gli stessi documenti, questi sono recuperati dallo stesso server ripetutamente. Questo può essere un carico notevole, specialmente sul link di accesso tra la LAN e Internet, che è spesso la parte più lenta e congestionata e inoltre probabilmente costosa da aggiornare.

Per un documento molto richiesto, sarebbe meglio che fosse salvato e riutilizzato, così da evitare il nuovo trasferimento dello stesso dal server di origine in richieste successive, operazione che risulterebbe particolarmente efficiente per grandi organizzazioni come aziende o università. Proprio questo è lo scopo del Caching. Le richieste degli utenti sono soddisfatte il più possibile dalla cache sulla LAN, piuttosto che dal server di origine, e solo poche richieste sono spedite a quest'ultimo.

Proxy server

Un proxy server è un'entità della rete che soddisfa le richieste HTTP da parte di un client. Esso è una cache con un proprio disco di archiviazione e conserva nella sua memoria copie degli oggetti richiesti di recente.

La figura seguente schematizza il suo comportamento.



Nella figura, il client manda la sua richiesta HTTP direttamente al proxy server. Quest'ultimo, comunque, non può (o sceglie di non farlo) rispondere al client immediatamente; invece inoltra la richiesta a un secondo server, il server di origine, così chiamato perchè rappresenta l'origine dell'oggetto richiesto.

Nella maggior parte dei casi il secondo GET ha un URI identico a quello che compare nel primo, è semplicemente mandato a un nuovo server. Il server di origine tratta il secondo GET come se fosse inviato dal client e risponde con l'oggetto richiesto. Il proxy server ha così l'informazione richiesta dal client in precedenza, e gli manda l'oggetto al passo 4.

Ad esempio, supponiamo che il client stia chiedendo l'oggetto `http://www.ing.unipi.it/immagine.gif`. Ecco cosa avviene:

- Il client stabilisce una connessione TCP con il proxy server e invia una richiesta HTTP per l'oggetto al proxy stesso.
- Il proxy controlla se ha una copia dell'oggetto memorizzata localmente. Se c'è, la inoltra all'interno di un messaggio di risposta HTTP al client.
- Se il proxy non ha l'oggetto, apre una connessione TCP al server di origine, cioè a `www.ing.unipi.it`.
- Quando il proxy riceve l'oggetto, ne archivia una copia nella memoria locale e invia la copia, all'interno di un messaggio di risposta HTTP, al client (sulla connessione TCP esistente fra client e proxy).

Da notare che **un proxy server è sia un server che un client allo stesso tempo**. Quando riceve le richieste e invia le risposte a un browser è un server, quando invece invia le richieste e riceve le risposte è un client.

Considerazioni sui proxy server

Un proxy server compie principalmente quattro operazioni:

- riceve richieste dai client
- risponde alle richieste prendendo le risposte dalla sua cache se possibile
- preleva i documenti da altri server se non sono in cache
- gestisce la cache dei documenti

Il proxy è un server Web; il client apre una connessione col proxy e manda le sue richieste. C'è una differenza importante però: la richiesta mandata al proxy deve includere l'intero URL (assoluto), incluso il nome del server da cui proviene il documento originale. Una regolare richiesta non necessita di questa informazione perchè è mandata direttamente al server.

Ad esempio, quando si richiede il documento:

```
http://www.ing.unipi.it/documento.htm
```

la regolare richiesta HTTP, mandata a `www.ing.unipi.it`, sarebbe:

```
GET /documento.htm HTTP/1.0
```

ma la richiesta al proxy sarebbe:

```
GET http://www.ing.unipi.it/documento.htm HTTP/1.0
```

In questo modo il proxy sa da quale server proviene il documento. Quando è configurato per usare un proxy, il browser sa anche che deve mandare l'intera richiesta.

Considerazioni sui proxy server (2)

Un proxy server determina se deve servire una richiesta o passarla al server di origine dopo alcuni passaggi solitamente guidati da regole specificate dall'amministratore del server. Le regole possono specificare, ad esempio, quali documenti devono essere cachati e per quanto tempo.

L'output di ogni script CGI è un documento dinamico che può essere diverso ogni volta che è richiesto, così gli script non possono essere cachati e devono essere richiamati ogni volta che sono richiesti da un client, a differenza dei documenti statici che possono risiedere sulla cache.

Da notare che un proxy considera diversi due documenti con lo stesso nome ma provenienti da due server differenti, anche se per l'utente sono lo stesso documento. Se, ad esempio:

`http://server-1.net/documento.htm`

è nella cache e qualcuno richiede:

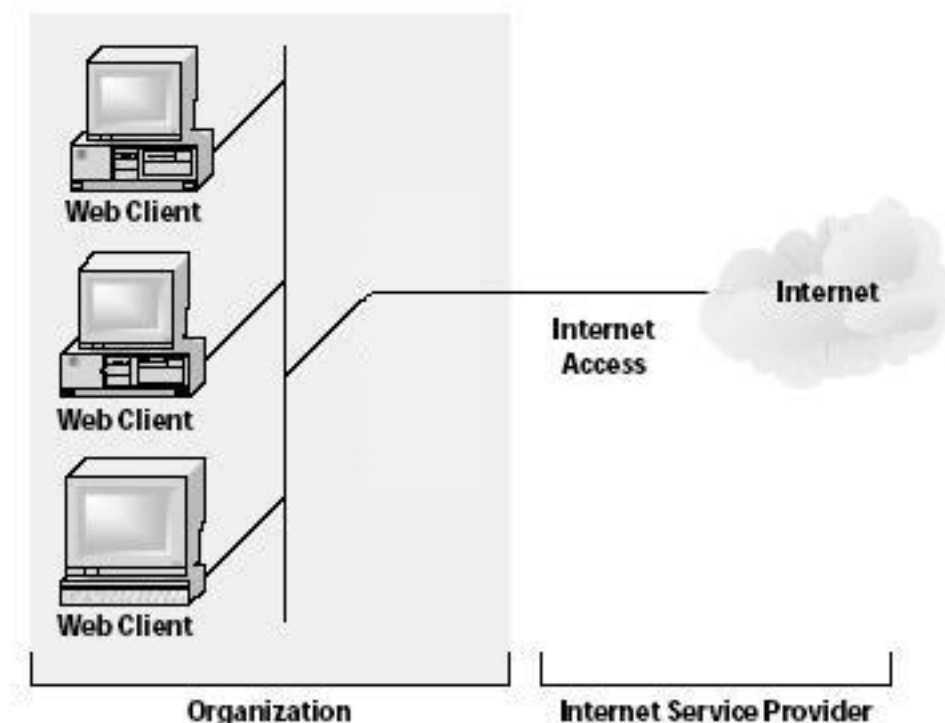
`http://server-2.net/documento.htm`

il documento deve essere recuperato da `server-2.net` e messo nella cache. Così lo stesso documento può essere cachato due volte sotto nomi differenti e, sfortunatamente, è una cosa piuttosto comune.

I documenti possono essere prelevati da molti server e oggetti diversi possono avere lo stesso nome su server differenti, così un proxy deve memorizzare il nome del documento e il server da cui proviene. Inoltre deve tenere conto della data in cui è stato prelevato.

Esempio

Consideriamo la figura seguente:



In essa sono evidenziate due reti: la rete di un'organizzazione, una LAN ad alta velocità, e Internet. Supponiamo che le due siano collegate tramite due router alle due estremità a un link a 1,5 Mbit/s.

Supponiamo che ogni richiesta abbia una dimensione di 100 Kbit e che un client ne formuli in media 15 al secondo verso i server di origine sparsi in Internet. Supponiamo inoltre che il "ritardo Internet", che rappresenta il tempo che intercorre tra una richiesta HTTP (entro un datagram IP) e la corrispondente risposta (tipicamente in molti datagram IP), sia in media di 2 secondi.

Il tempo totale di risposta, cioè il tempo dalla richiesta di un browser per un oggetto alla ricezione dello stesso, è la somma del ritardo della LAN, del ritardo di accesso (sul link fra le due reti), e del ritardo Internet. Cerchiamo di stimare ora questo ritardo.

- Continuazione esempio -

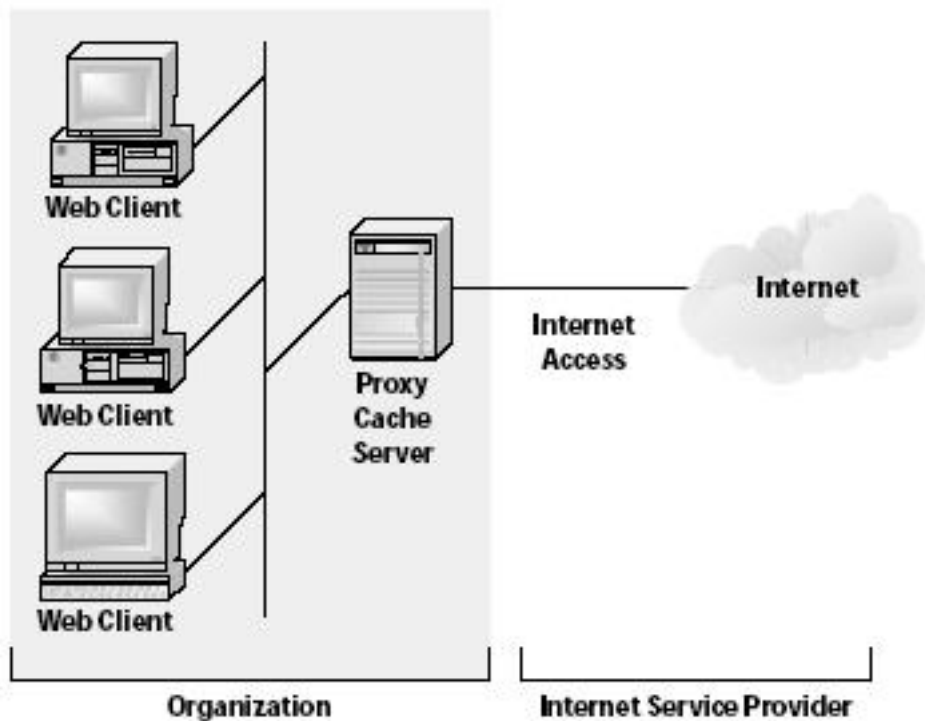
L'intensità del traffico sulla LAN è: $(15 \text{ richieste/s}) * (100 \text{ Kbit/richiesta}) / (10 \text{ Mbit/s}) = 0,15$

L'intensità del traffico sul link di accesso (tra i due router) è: $(15 \text{ richieste/s}) * (100 \text{ Kbit/richiesta}) / (1,5 \text{ Mbit/s}) = 1$

Un'intensità di traffico di 0,15 su una LAN tipicamente introduce un ritardo pari ad alcune decine di millisecondi, ritardo dunque trascurabile. Il ritardo sul link è invece superiore di parecchio: su di esso l'intensità del traffico si avvicina a 1, ovvero vi è un carico del 100%, il canale è saturo. Il tempo di risposta medio per soddisfare le richieste sta raggiungendo le dimensioni di minuti, se non di più, cosa inaccettabile per gli utenti di un'organizzazione: bisogna intervenire.

Una soluzione possibile sarebbe l'aumento della velocità di accesso del link fra le due reti da 1,5 Mbit/s a, per esempio, 10 Mbit/s. In questo caso l'intensità del traffico risulterebbe pari a 0,15 (come per la LAN), e il ritardo trascurabile. Il tempo totale di risposta sarebbe così di 2 secondi, cioè il ritardo Internet. Ma questa soluzione prevede l'aggiornamento del link di accesso da 1,5 a 10 Mbit/s, cosa che può essere molto costosa.

La soluzione alternativa è di installare una cache (proxy server) nella rete dell'istituzione, come mostrato in figura.



- Conclusione esempio -

La velocità delle richieste soddisfatte dalla cache (*hit rate*) in pratica varia di solito tra 0,2 e 0,7.

Supponiamo che per l'organizzazione in figura la cache fornisca un *hit rate* di 0,4.

Essendo i client e la cache connessi attraverso la stessa rete LAN ad alta velocità, il 40% delle richieste saranno soddisfatte pressochè immediatamente, diciamo entro 10 millisecondi, dalla cache.

Il restante 60% delle richieste richiede ancora di essere soddisfatto dai server di origine. Ma con solo il 60% degli oggetti richiesti da passare attraverso il link di accesso, l'intensità del traffico su questo link si riduce da 1 a 0,6.

Tipicamente un'intensità di traffico inferiore allo 0,8 corrisponde a un piccolo ritardo, decine di millisecondi, su un link a 1,5 Mbit/s. Questo ritardo è trascurabile confrontato con il ritardo Internet di 2 secondi.

Detto ciò, il ritardo medio è quindi: $0,4 * (0,01 \text{ secondi}) + 0,6 * (2 \text{ secondi}) = 1,2 \text{ secondi (circa)}$

Allora, questa seconda soluzione fornisce un basso tempo di risposta come la prima e non richiede all'organizzazione l'aggiornamento del suo link a Internet. L'organizzazione deve ovviamente comprare e installare un proxy server, ma questo costo è basso: molte cache usano software di pubblico dominio che

girano su server e PC poco costosi.

Caching su client e server

Quando l'utente clicca su un collegamento, invece di connettersi automaticamente al server indicato nell'URL, sarebbe meglio cercare di vedere se il documento è disponibile localmente e, se così, usare quella copia al posto di quella spedita dal server di origine. L'idea che sta alla base del caching è di "cortocircuitare" una richiesta per un documento sul Web in questo modo ogni qualvolta sia possibile.

Se ci fosse modo di prevedere cosa gli utenti richiedono, sarebbe possibile avere ogni documento disponibile localmente quando richiesto. Ma ciò solitamente non è possibile. Cosa è invece possibile è salvare i documenti dopo che sono stati richiesti, in previsione che saranno richiesti di nuovo. Nella pratica, questa previsione è spesso abbastanza vera da giustificare il caching dei documenti richiesti di recente.

Come detto, i documenti possono essere cachati in vari modi. Il browser stesso può salvare gli oggetti che ha prelevato dai server di origine. Il browser immagazzina documenti e immagini sul PC, e se uno di questi è richiesto una seconda volta, si usa la copia memorizzata piuttosto che inoltrare una nuova richiesta al server.

Il caching sul client è però limitato dalla memoria e dallo spazio disponibile su disco del PC. I documenti cachati inoltre potrebbero non essere salvati dopo che il browser è chiuso, così ogni sessione dovrebbe ricominciare da capo, con nuove richieste ai server. E, considerando che più browser siano collegati ad una stessa LAN, ogni utente che chiede gli stessi oggetti di un altro deve formulare le stesse richieste ai server, così il carico su Internet aumenta. Ci possono essere problemi anche se nella cache sono presenti pagine che possono diventare out of date, cioè obsolete, e quindi inservibili.

Anche i server Web possono agire da cache. Ciò è vero specialmente per server collegati a tanti altri. Un server può così prelevare i documenti e modificare i collegamenti in modo che puntino alla sua cache locale. Questo tipo di caching è utile per quei documenti che sono "puntati" da documenti sul server. I client devono però ancora connettersi ai server Web sparsi su Internet, ciò significa che il carico sullo stesso non si riduce; inoltre in reti locali, come visto nell'esempio, permane il problema del collo di bottiglia (bottleneck) introdotto dal link di accesso a Internet.

La soluzione migliore e con benefici evidenti è quindi quella di usare proxy server. Le richieste vengono così filtrate e reinstradate, con un miglioramento delle performances e l'assunzione di una politica di sicurezza; infatti, poichè un proxy può inoltrare richieste tramite un firewall, è molto importante anche se non compie alcun tipo di caching.

Reinstradamento di richieste al proxy

Come abbiamo visto, tramite l'uso di un proxy server un client manda le sue richieste al proxy invece che al server specificato nell'URL. Ci sono due modi per fare ciò:

- l'URL viene riscritto in ogni richiesta per puntare al proxy
- il browser viene configurato per mandare richieste direttamente al proxy

La prima possibilità è di riscrivere l'URL, inserendo prima l'indirizzo del proxy. In questo caso, una richiesta per l'URL:

`http://www.ing.unipi.it/documento.htm`

cambierebbe in:

`http://cache.local.org/http://www.ing.unipi.it/documento.htm`

Il proxy server all'indirizzo `cache.local.org` esaminerebbe l'URL e invierebbe il documento dalla sua cache o, in caso non fosse presente, lo prelevarebbe dal server di origine. Questo è esattamente cosa noi vogliamo che accada, ma è molto scomodo (se non impossibile) modificare ogni URL in ogni documento. Inoltre, poichè possono esserci molti proxy disponibili, non è sempre chiaro cosa riscrivere nell'URL.

Il modo migliore per reinstradare richieste a un proxy consiste nel configurare il browser a fare ciò automaticamente; è molto più facile configurare un browser a mandare tutte le richieste a un proxy piuttosto che riscrivere ogni volta l'URL. L'idea è quella di configurare manualmente il browser a inviare le richieste a un appropriato proxy e lasciare che entrambi si mettano d'accordo sui dettagli che all'utente non interessano.

La configurazione del browser avviene settando variabili d'ambiente o attraverso menu di preferenze, come vedremo negli esempi seguenti.

Dal punto di vista dell'utente, il browser lavora in modo uguale con o senza caching, le URL non cambiano e quando si clicca su un link avviene la cosa giusta. Dal punto di vista del server, il proxy è come un altro client che ha inoltrato una richiesta. Così, **il proxy è trasparente sia all'utente che al server**.

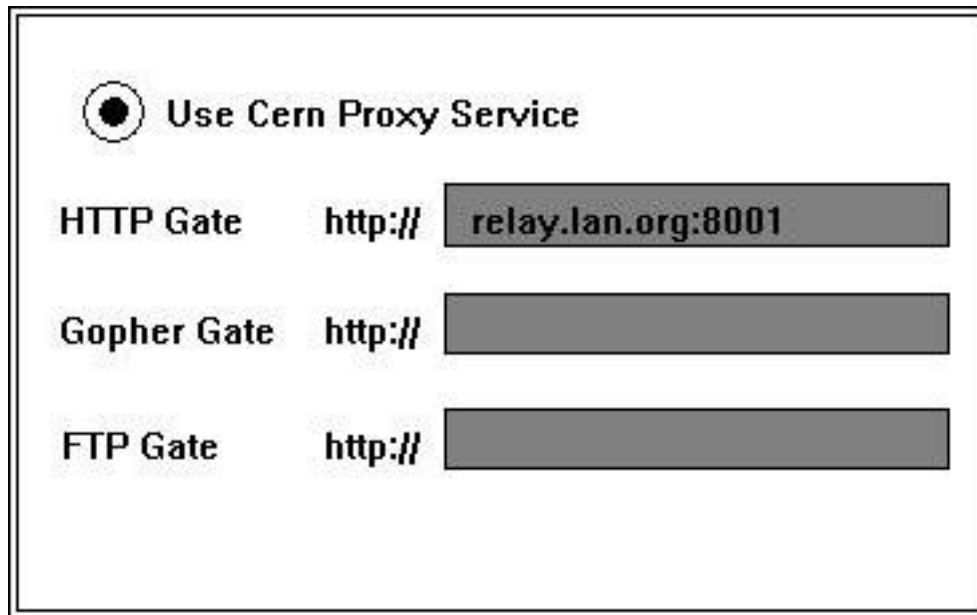
Esempi sulla configurazione di un browser

Vediamo un esempio di configurazione del browser NCSA Mosaic tramite uno script UNIX:

```
#!/bin/csh
#
# Questo script istruisce il browser NCSA Mosaic a usare un proxy
# all'indirizzo "relay.lan.org" alla porta 8001
#

setenv http_proxy http://relay.lan.org:8001
setenv ftp_proxy http://relay.lan.org:8001
setenv gopher_proxy http://relay.lan.org:8001
```

Il secondo esempio di possibile configurazione è raffigurato nella figura seguente, tramite un menu di preferenze:



The image shows a window titled "Use Cern Proxy Service" with a radio button selected. Below the title, there are three rows of proxy settings:

Protocol	Gate	Address
HTTP	Gate	http:// relay.lan.org:8001
Gopher	Gate	http://
FTP	Gate	http://

Entrambi gli esempi fanno la stessa cosa, cioè inoltrano le richieste di un browser verso il proxy server specificato (`relay.lan.org`) che manda le risposte associate. Come mostrato negli esempi, si possono usare differenti proxy per protocolli diversi usati da un browser: HTTP, FTP, Gopher e altri.

Cache Consistency (o Cache Coherence)

Un problema piuttosto evidente che può sorgere quando si fa uso di una cache è che i documenti diventino out of date, cioè obsoleti.

Se e quando il documento originale viene modificato sul server di origine, la copia in cache diventa inconsistente e non deve essere più usata. Il proxy server continuerà ad utilizzare la versione vecchia del

documento finchè non preleverà la nuova versione. Questo è spesso un problema difficile da risolvere, specialmente per un sistema distribuito e decentrato come il Web.

Il problema fondamentale per la cache consiste nel sapere quando è cambiato il documento originale. Due sono le possibilità:

- il server di origine può notificare ai proxy che il documento è cambiato.
- i proxy stessi possono chiedere ai server se la copia di un documento è ancora valida.

La prima opzione è però piuttosto impraticabile. Un server non sa dove è cachato un documento, spesso non è neanche a conoscenza della data della modifica. Ma anche se sapesse queste informazioni, sarebbe impossibile mandare tutti questi aggiornamenti in quanto la quantità di traffico generata risulterebbe enorme sul server stesso e sulla rete.

Si adotta così la seconda opzione; è abbastanza facile per un proxy verificare se la sua copia di un documento è obsoleta. Esso può ottenerne una nuova e metterla in cache se è stata modificata. Ma contattare un server di origine ogni volta che un client richiede un documento al proxy sarebbe costoso per Internet e il server quasi come prelevare il documento ogni volta che viene richiesto, operazione che annullerebbe i vantaggi dell'uso delle cache.

Per questo motivo, i proxy generalmente non verificano la validità di un documento cachato ogni volta che viene richiesto, ma periodicamente. La strategia consiste nell'aggiornare i documenti in cache ogni 24 o 48 ore, e, se questi vengono richiesti dopo che è passato questo periodo di tempo (stabilito da regole settate dall'amministratore del proxy, vedi esempio seguente), sono nuovamente prelevati dal server di origine per vedere se sono cambiati.

Esempio di direttive per un proxy

Come detto, il tempo di validità dei documenti in cache è stabilito da regole settate dall'amministratore del proxy.

Bisogna cercare di stabilire il "tempo di vita" di ogni documento in cache in base alla frequenza con la quale viene modificato nel server di origine. Ovviamente è piuttosto difficile per chiunque fare una previsione su quando un documento verrà cambiato, in quanto esso può essere modificato un secondo dopo che è stato prelevato o nel caso limite non cambiare mai. Così, la stima di un tempo di validità per un documento non è una soluzione priva di errori.

Vediamo un esempio di direttive date a un proxy dal suo amministratore:

#

```
# Abilitazione della cache e memorizzazione dei file cachati in
# '/usr/local/WebCache fino a 200 MB di spazio sul disco
#
Caching On
CacheRoot /usr/local/WebCache
CacheSize 200
#
# Documentazione su "hit" (documenti in cache) e "miss" (documenti da
prelevare)
#
CacheAccessLog /usr/http/logs/hit_log
ProxyAccessLog /usr/http/logs/miss_log
#
# Per non mettere in cache documenti del nostro server locale
#
NoCaching http://www.my.local.net/*
#
# Settaggio dei limiti sul tempo di mantenimento dei documenti in
cache, massimo 2 mesi
CacheClean 2 months
# Per quanto tempo mantenere un documento dall'ultimo uso, http 2
settimane
CacheUnused httpd:* 2 weeks
```

Supporti forniti dall'HTTP per la consistenza di una cache

Il protocollo HTTP stesso fornisce meccanismi che aiutano una cache a rimanere aggiornata.

L'intestazione opzionale `Expires`, trattata nella sezione "Intestazioni", può essere usata da un server di origine per specificare per quanto tempo un documento dovrebbe rimanere in cache.

Purtroppo, molti server non includono questa informazione quando spediscono un documento al proxy. Il motivo per cui questa intestazione non viene sfruttata è abbastanza ovvio: è molto difficile per il software del server sapere quando il documento verrà modificato la prossima volta.

La seconda possibilità consiste nell'uso del GET condizionato. Un messaggio di richiesta HTTP è detto anche messaggio GET condizionato se:

- il messaggio di richiesta usa il metodo GET
- il messaggio di richiesta comprende una linea di intestazione `If-Modified-Since`

In risposta a un GET condizionato, un server invia ancora un messaggio di risposta, ma se il documento non è stato modificato rispetto alla copia che risiede nella cache, non viene allegato l'oggetto richiesto. L'inserimento di quest'ultimo provocherebbe solo uno spreco della larghezza di banda e aumenterebbe il tempo di risposta percepito, soprattutto se l'oggetto ha grandi dimensioni.

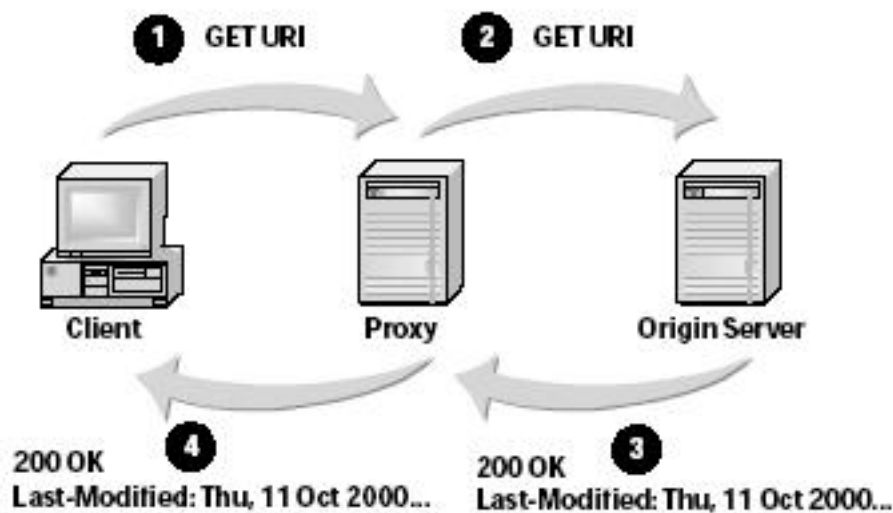
Di seguito vengono descritte le intestazioni If-Modified-Since, che include un esempio di GET condizionato, la duale If-Unmodified-Since, e due intestazioni per il controllo della cache, Cache-Control e Meter.

Nessuno degli approcci per la consistenza di una cache è infallibile, nè in teoria nè in pratica. Ma, per quanto possa essere imperfetto, il caching è molto pratico e solitamente fornisce vantaggi evidenti in grado di rendere irrilevanti i suoi stessi problemi.

If-Modified-Since

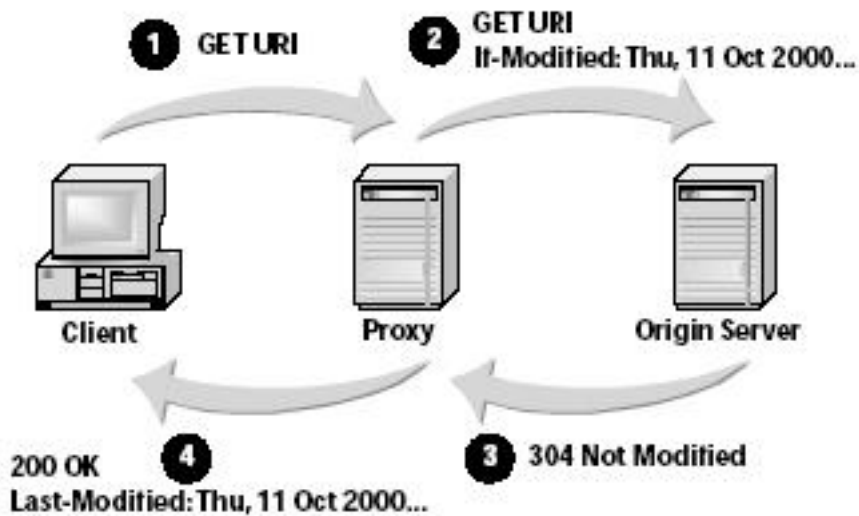
L'intestazione If-Modified-Since permette un uso più efficiente delle cache a client e proxy server. Essa chiede a un server di rispondere a una richiesta solo se la risorsa è cambiata dalla data specificata.

La figura seguente mostra come i sistemi HTTP possono usare questa intestazione.



Viene evidenziata una richiesta standard GET che passa da un proxy server. Un elemento chiave della risposta del server è l'intestazione Last-Modified, tramite cui il server identifica l'ora e la data del momento della creazione o dell'ultima modifica dell'oggetto richiesto.

L'esempio continua nella figura successiva.



- Continuazione esempio If-Modified-Since e considerazioni -

Un pò di tempo dopo, il client fa un'altra richiesta per la stessa risorsa. Il proxy ha una copia della risposta precedente nella sua cache locale, così inserisce l'intestazione `If-Modified-Since` nella richiesta prima di passarla al server di origine. Il valore di questa intestazione è lo stesso di quello di `Last-Modified` che il server aveva inserito nella sua precedente risposta.

Nell'esempio, la risorsa non è cambiata. Piuttosto che ritornare l'intero oggetto, il server di origine risponde con uno stato `304 Not Modified`. Questo stato dice al proxy che la sua copia dell'oggetto in cache è ancora valida, così esso può spedire tale copia al client.

Se l'oggetto è di grosse dimensioni, questo passaggio può avere risparmiato un'ampiezza di banda e un ritardo considerevoli, in quanto l'oggetto non deve viaggiare dal server di origine al proxy una seconda volta.

I client possono usare `If-Modified-Since` non solo per richieste standard, ma anche per quelle parziali con l'intestazione `Range`. In tal caso il valore dell'intestazione (`If-Modified-Since`) si riferisce all'intero oggetto, non solo alla parte richiesta.

I client che usano tale intestazione devono prendere in considerazione due problemi possibili con alcuni server:

- alcuni server fanno una comparazione precisa tra il valore di `If-Modified-Since` e quello di `Last-Modified` della risorsa. Anche se il valore di `If-Modified-Since` è più alto del secondo (cioè successivo nel tempo), questi server mandano l'intero oggetto. I client che vogliono evitare questo comportamento errato devono usare solo il valore esatto dell'intestazione `Last-`

Modified (e ricopiarlo di conseguenza in If-Modified-Since) rilasciato dal server in precedenza.

- l'altro problema è relativo alla sincronizzazione del clock. I client devono essere consapevoli che il clock del server non sempre può essere corretto; esso può essere soggetto a imprecisioni nel tempo e a errori umani. Quindi, anche in questo caso, il modo migliore per prevenire questi errori è quello di usare solo il valore dell'intestazione Last-Modified del server.
-

If-Unmodified-Since

Come ci si può aspettare, l'intestazione If-Unmodified-Since ha esattamente l'effetto opposto dell'intestazione If-Modified-Since.

Se un client la include nella sua richiesta, chiede al server di accettare quest'ultima solo se la risorsa considerata non è cambiata dalla data indicata.

Una risposta di un server con stato 412 `Precondition Failed` indica che la risorsa è stata modificata dalla data specificata nell'intestazione.

Un client, ad esempio, può usare questa intestazione in richieste PUT se vuole essere sicuro che nessun altro abbia modificato una risorsa mentre esso stesso la stava cambiando.

Cache-Control

Cache-Control è l'intestazione principale per alcune differenti direttive che specificano il comportamento che ci si aspetta da un sistema che fa caching, ad esempio da un proxy server.

Queste direttive, alcune delle quali hanno dei parametri associati, sono separate da virgole in una intestazione.. Il seguente frammento di testo specifica ad esempio tre direttive per il controllo della cache.

```
Cache-Control: max-age=3600, no-transform,  
              no-cache="Accept-Ranges"
```

Come avviene per altre intestazioni, le direttive possono essere usate in richieste e risposte. La tabella seguente elenca le direttive possibili per Cache-Control.

Direttive	Parametri	Richiesta	Risposta
max-age	Richiesto	●	●
max-stale	Opzionale	●	
min-fresh	Richiesto	●	
must-revalidate	Nessuno		●
no-cache	Opzionale	●	●
no-store	Nessuno	●	●
no-transform	Nessuno	●	●
only-if-cached	Nessuno	●	
Private	Opzionale		●
proxy-revalidate	Nessuno		●
Public	Nessuno		●
s-maxage	Richiesto		●

Da notare che la direttiva no-cache è opzionale nelle risposte e non ha parametri nelle richieste.

Le direttive sono discusse di seguito.

- Cache-Control: direttive max-age e max-stale -

In questo paragrafo e nei successivi vengono illustrate la sintassi e l'uso delle direttive di Cache-Control, ognuna delle quali è introdotta da un esempio.

Cache-Control: max-age=3600

Gli scopi principali dell'uso della direttiva max-age sono due. Il primo: quando viene usata da un server, indica il tempo massimo, in secondi, che una risorsa deve essere tenuta in cache senza il bisogno che sia aggiornata. Questo uso rende max-age simile all'intestazione Expires.

Se nella stessa risposta sono presenti sia una direttiva max-age che un'intestazione Expires, i proxy server dovrebbero ignorare la seconda, anche se il suo valore è più restrittivo del valore della prima. Questa regola permette ai server di origine di specificare comportamenti per le cache che supportano HTTP/1.0 e la versione successiva 1.1, poichè i proxy con versione 1.0 non capiranno, e di conseguenza ignoreranno, nessuna direttiva max-age.

Il secondo scopo è per l'uso che ne possono fare i client. Quando un client include tale direttiva nella sua

richiesta, indica che è disposto ad accettare un oggetto in cache non più vecchio del valore indicato. Se un proxy ha una copia dell'oggetto più vecchia rispetto al valore che compare nella richiesta, non deve ritornarla anche se la risposta originale del server di origine indica che è ancora valida.

Nel caso estremo, il client può assegnare il valore zero a `max-age`. In questo caso i proxy dovrebbero sempre passare la richiesta al server di origine per aggiornare l'oggetto in cache.

```
Cache-Control: max-stale
```

Tramite questa direttiva, un client indica che è disposto ad accettare una risposta che include un oggetto cachato, anche se l'oggetto è apparentemente diventato out-of-date. Il client può opzionalmente includere un valore per indicare che accetta risposte con oggetti apparentemente obsoleti anche dopo il numero di secondi evidenziato nella direttiva. Ad esempio, la direttiva `max-stale=600` indica che il client è disposto ad accettare risposte ormai vecchie di 10 minuti (600 secondi).

- Cache-Control: direttive min-fresh, must-revalidate e no-cache -

```
Cache-Control: min-fresh=60
```

Quando un client include la direttiva `min-fresh` nella sua richiesta, indica ai proxy di mandare un documento in cache solo se rimarrà ancora valido almeno per il numero di secondi specificato.

Se ad esempio una cache contiene un oggetto che diventerà obsoleto dopo 45 secondi che un client che ha incluso la riga di esempio nelle sue intestazioni lo richiede, il proxy non lo invierà al client. Questo perchè l'esempio precedente richiede che ogni copia locale abbia almeno 60 secondi di vita rimanenti (prima di diventare out-of-date).

```
Cache-Control: must-revalidate
```

Questa direttiva lascia che i server rispondano all'uso che i loro client fanno di `max-stale`. Quando un server include `must-revalidate` nella sua risposta, i proxy devono ignorare la direttiva `max-stale` in tutte le richieste future dei client.

```
Cache-Control: no-cache
```

Tale direttiva può apparire sia in richieste che in risposte. In una richiesta, indica che il client non è disposto ad accettare risposte cachate; ogni proxy intermedio deve passare la richiesta al server di origine.

Da notare che questa richiesta differisce leggermente da una richiesta che include la direttiva `max-age=0`. Nel caso di richieste con `no-cache`, i proxy devono sempre recuperare la risposta dal server di origine. Nel caso di richieste con `max-age=0`, invece, i proxy devono solo validare un'entrata della loro cache locale con il server di origine solo se è out-of-date.

Quando è il server di origine a includere tale direttiva nella sua risposta, indica ai proxy di non usare la risposta per richieste seguenti senza prima averla aggiornata. Questo non proibisce esattamente ai proxy di cachare la risposta (nonostante il nome della direttiva); li forza soltanto ad aggiornare la copia in cache ad ogni richiesta.

- **Cache-Control: direttive no-store, no-transform, only-if-cached e private** -

`Cache-Control: no-store`

Questa direttiva identifica informazioni private, sia in una richiesta (e nella sua risposta successiva) che in una risposta da sola. La direttiva `no-store` indica ai proxy di non memorizzare i messaggi in nessuna memoria locale, in modo particolare se i loro contenuti possono essere trattenuti dopo lo scambio (ad esempio su nastri di backup).

`Cache-Control: no-transform`

La direttiva `no-transform`, che può apparire sia in una richiesta che in una risposta, dice ai proxy di non modificare il formato del corpo del messaggio della risposta. Alcuni proxy potrebbero fare ciò, ad esempio, per salvare spazio sulla cache convertendo un'immagine ad alta risoluzione nella stessa in bassa risoluzione.

`Cache-Control: only-if-cached`

Tramite questa direttiva, un client chiede ai proxy di rispondere successivamente solo se hanno l'oggetto nelle loro cache locali. In particolare, il client dice alla cache di non caricare nuovamente o aggiornare la risposta dal server di origine.

Questo comportamento può essere utile specialmente in ambienti con una scarsa connettività alla rete, dove il client sente che il ritardo nel raggiungere il server di origine è inaccettabile. Se un proxy non può rispondere alla richiesta dalla sua cache locale, invia un codice di stato 504 Gateway Timeout.

`Cache-Control: private`

La direttiva `private` in una risposta indica che la stessa è intesa esclusivo per uno specifico utente. I proxy possono trattenere una copia per rispondere successivamente alle richieste dello stesso utente, ma

non devono mandare quell'oggetto cachato ad altri utenti, anche se questi formulano la stessa richiesta.

- Cache-Control: direttive proxy-revalidate, public e s-maxage -

Cache-Control: proxy-revalidate

La direttiva `proxy-revalidate` indica a tutti i proxy intermedi di non mandare la risposta a richieste successive senza che sia stata nuovamente convalidata. A differenza della direttiva `must-revalidate`, `proxy-revalidate` permette ai client di cachare la risposta e usarla nuovamente senza convalida.

Cache-Control: public

Questa direttiva è l'opposto di `private`. Tramite essa, un server indica esplicitamente che la sua risposta può essere cachata e inviata ad altri utenti, anche se la stessa è ristretta all'utente originale o non può essere cachata del tutto.

Se un client fornisce informazioni per l'autenticazione dell'utente, ad esempio, i proxy dovrebbero normalmente trattare ogni risposta come privata per quell'utente. Ma se il server risponde ad esempio con uno stato 301 Moved Permanently, può anche usare la direttiva `public` per dire ai proxy di non adottare il loro comportamento normale e mettere in cache la risposta.

Cache-Control: s-maxage=1800

La direttiva `s-maxage` agisce in modo simile alla direttiva `max-age` nelle risposte, eccetto che si applica solo a cache che servono più utenti. Per questi proxy, tale direttiva nasconde sia `max-age` che l'intestazione `Expires`. I proxy che rispondono allo stesso utente più volte possono comunque ignorarla.

Meter

Come nel caso di `Cache-Control`, l'intestazione `Meter` supporta alcune differenti opzioni conosciute come direttive. Sia i proxy che i server di origine usano queste direttive per documentare gli accessi a una pagina in cache e limitare il caching di risorse.

Questo processo di "misurazione" (metering) avviene in tre fasi.

- Dapprima, il proxy manifesta la sua volontà di supportare il metering nella richiesta iniziale.
- Poi, il server di origine chiede specifici servizi di metering nella sua risposta.
- Infine, il proxy documenta gli accessi in richieste successive per lo stesso oggetto.

La tabella seguente elenca le direttive di Meter, così come la fase in cui ognuna di esse è usata. Come si può notare, ogni direttiva ha una forma regolare e una abbreviata.

Direttiva	Abbrev.	Si usa in	Uso
count=n/m	c=n/m	Richiesta successiva	Il proxy indica gli accessi.
do-report	d	Risposta	Il server di origine chiede al proxy di fornire rapporti.
dont-report	e	Risposta	Il server di origine dice al proxy di non fornire rapporti.
max-reuses=n	r=n	Risposta	Il server di origine specifica un limite per il numero di accessi non unici a pagine.
max-uses=n	u=n	Risposta	Il server di origine specifica un limite per il numero di accessi unici a pagine.
timeout=n	t=n	Risposta	Il server di origine specifica il tempo massimo fra i rapporti.

<code>will-report-and-limit</code>	<code>w</code>	Richiesta iniziale	Il proxy può supportare il metering.
<code>wont-ask</code>	<code>n</code>	Risposta	Il server di origine indica che non chiederà il metering di nessun oggetto.
<code>wont-limit</code>	<code>y</code>	Richiesta iniziale	Il proxy supporta il metering ma non vuole limitare l'uso.
<code>wont-report</code>	<code>x</code>	Richiesta iniziale	Il proxy supporta il metering ma non vuole fornire rapporti.

- Processo di metering -

Il processo di metering comincia quando una richiesta passa da un proxy. Se quest'ultimo è disposto a supportare il metering, aggiunge un'intestazione `Meter` alla sua richiesta. Nell'intestazione il proxy può identificare il tipo di supporto che sta offrendo tramite l'uso di `will-report-and-limit`, `wont-limit` o `wont-report`.

Senza una specifica direttiva, il comportamento di default è di fornire rapporti e limitare l'uso di risorse. Il proxy deve anche aggiungere l'intestazione `Connection: Meter` alla richiesta, poichè l'intestazione `Meter` deve essere limitata alla connessione attuale. Quindi, se il proxy è contento del comportamento di default, deve includere solo l'intestazione `Connection`, poichè la linea `Connection: Meter` implica la presenza dell'intestazione `Meter`. Ecco un esempio:

```
GET / HTTP/1.1
Via: proxy
Connection: Meter
```

Quando il server risponde a questa richiesta, guida il proxy con un'intestazione `Meter`, che può includere una serie di direttive, nella risposta. Essa può dire al proxy se il server vuole ricevere rapporti (`do-report` o `dont-report`); può specificare il massimo numero di volte che il proxy può spedire

la risposta dalla sua cache (`max-uses` e `max-reuses`), e può indicare un tempo limite prima del quale il proxy deve mandare un nuovo rapporto (`timeout=n`).

Da notare che, a differenza di altre intestazioni, `Meter: timeout=n` specifica minuti, non secondi. Nell'esempio seguente il server di origine chiede al proxy di fornire rapporti almeno ogni ora. La risposta non specifica inoltre nessun limite. Se il server vuole dire al proxy di non mandare più nessuna intestazione `Meter`, può usare la direttiva `wont-ask` nella sua stessa intestazione `Meter`.

```
HTTP/1.1 200 OK
Date: Sun, 08 Gen 2005 19:59:17 GMT
Meter: do-report, timeout=60
Connection: Meter
```

Quando il proxy vede la risposta del server e mette in cache il corpo del messaggio, inizia a contare il numero di volte che ritorna l'oggetto dalla sua cache. Esso dovrebbe contare sia il numero di accessi unici alla pagina (richiesti da nuovi utenti) sia il numero di accessi non unici alla stessa (più richieste dello stesso utente).

I proxy considerano ogni risposta in cui ritornano l'oggetto (in altre parole, con un codice di stato 200 OK) come un accesso unico alla pagina e ogni risposta che semplicemente conferma la copia precedentemente memorizzata (codice di stato 304 `Not Modified`) come un accesso non unico alla pagina. Ogni qualvolta uno di questi conteggi raggiunge il massimo valore specificato dal server di origine, il proxy prende una nuova copia dal server prima di mandarla al client.

Il proxy server continua a ricevere richieste per l'oggetto cachato, ma deve anche determinare quando mandare un rapporto di uso al server di origine. Il proxy manda questo rapporto in questi casi:

- quando deve inoltrare un GET condizionato al server.
- quando scade il tempo limite dato dal server.
- quando rimuove l'oggetto dalla sua cache.

Il rapporto consiste in un'intestazione `Meter` con una direttiva `count`. I due valori in `count` sono rispettivamente il numero di usi e riusi. L'esempio seguente riporta 934 usi e 201 riusi.

```
GET / HTTP/1.1
Via: proxy
Meter: count=934/201
Connection: Meter
```

I costi del caching

Nell'esaminare le performances di una cache, ci sono tre casi da considerare:

- il documento è disponibile nella cache (hit)
- il documento non è in cache e il proxy deve quindi prelevare dal server di origine (miss)
- il documento fornito dalla cache è out-of-date (stale hit)

Il successo di una cache dipende dai costi relativi di hit e miss, dal numero degli stessi e dal numero dei documenti vecchi in cache. Il costo di un hit è molto minore del costo di prelievo di un documento. Per un proxy server il costo ha tre importanti componenti:

- latenza della risposta
- network traffic
- carico nel server

La latenza è il tempo che intercorre fra una richiesta inviata da un client e la risposta correlata. La latenza per ritrovare uno stesso documento può variare, perfino tra lo stesso client e server. Il secondo importante costo è costituito dal traffico sulla rete, specialmente nelle parti più lente e congestionate. Ogni link di rete ha una capacità limitata e, quando viene sovraccaricato, il servizio è più lento o disgregato.

Il terzo costo è il carico sul server; è improbabile che un singolo utente possa sovraccaricare un server Web, così il problema è nuovamente costituito dal numero totale di persone che usa un server. Un server sovraccaricato risponderà più lentamente alle richieste e potrebbe pure non rispondere a tutte.

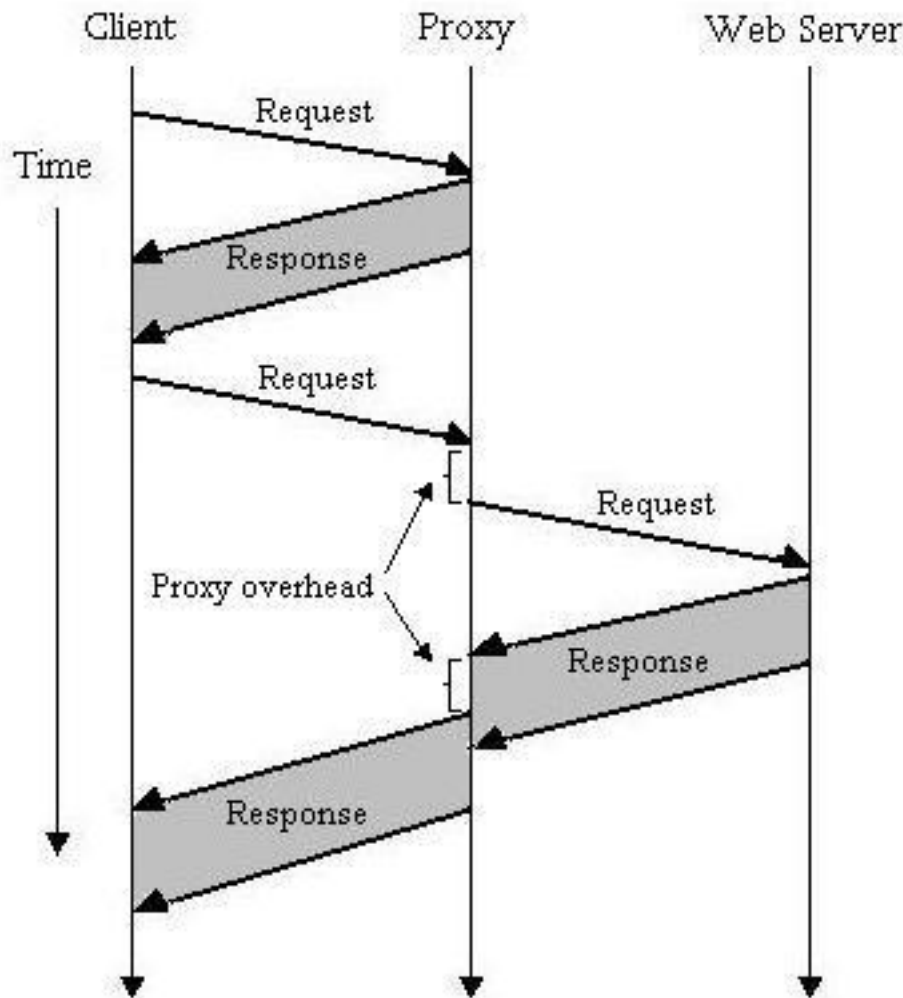
Un proxy server deve decrementare e ridistribuire questi costi. Per esempio, quando una richiesta è servita dalla cache invece che da un server distante, diminuisce il traffico di rete tra la rete locale e il server, e certamente pure il carico nel server.

Per ottenere un miglioramento delle performances, un documento deve essere copiato nella cache alla prima richiesta effettuata da un client, così la prima richiesta ha un costo almeno pari a quello che si ha quando non si fa uso di un proxy server. Ogni eventuale guadagno si ottiene solo in richieste successive, che sono servite dal proxy senza ricorrere al server di origine. Più lo stesso documento è richiesto, maggiori sono i vantaggi di averlo in cache.

Benefici dell'uso del proxy sulla latenza

L'accesso ai dati di un proxy server è quasi sempre più rapido di quello a un server di origine. La differenza varia, ma è all'incirca più veloce dalle 2 alle 10 volte.

L'inconveniente è che se si fa uso di un proxy e il documento richiesto non è in cache (miss), ci vuole più tempo che per prelevare lo stesso dal server di origine direttamente. La figura seguente mostra il ritardo (overhead) aggiunto da un proxy:



Questo ritardo varia, ma il tempo di accesso potrebbe risultare doppio rispetto a quando non si usa un proxy. E poichè i documenti dinamici non sono cachabili, sono sempre serviti più lentamente da un proxy.

Esempio per illustrare gli effetti di una cache sulla latenza

Supponiamo che:

- Tempo per prelevare un documento dal server Web: 1.0 secondo
- Tempo per il prelievo dal proxy (hit): 0.1 secondi
- Tempo per prelevare dal server usando un proxy (miss): 2.0 secondi

Con queste ipotesi, il documento in cache deve essere richiesto tre volte per realizzare un guadagno globale nella latenza media:

Accesso	Senza Proxy	Con Proxy
1	1 sec	2 sec
2	1 sec	0.1 sec
3	1 sec	0.1 sec
4	1 sec	0.1 sec
Totale	4 sec	2.3 sec
Media	1 sec	0.55 sec

Il primo accesso paga la penalità; il secondo e il terzo sono più veloci, così la latenza è "riallocata" e ridotta nel globale. Anche se la latenza con proxy è maggiore di quella senza, con appena quattro accessi siamo riusciti a ridurla di quasi il 50%.



Cookies e mantenimento di stato

Il protocollo HTTP opera normalmente come se ogni richiesta di un client fosse indipendente da tutte le altre.

Il server risponde a ogni richiesta inviando i file richiesti senza memorizzare alcuna informazione di stato relativa ai client.

Se un particolare client chiede due volte lo stesso oggetto entro pochi secondi, il server non risponde dicendo che l'oggetto è appena stato inviato al client, invece lo rispedisce, come se avesse completamente dimenticato ciò che ha appena fatto.

Poichè un server HTTP non conserva le informazioni relative ai client, l'HTTP è detto **protocollo senza stato** (*stateless protocol*).

Il mantenimento dello stato richiede risorse del server (memoria e altro), per cui vengono di solito preferite le operazioni senza stato. In alcune applicazioni però il server ha bisogno di sapere e memorizzare alcune informazioni di stato relative a ognuno dei suoi client.

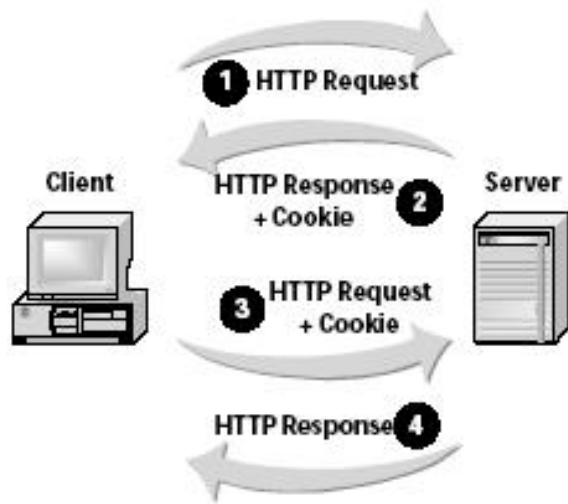
Gli utenti che fanno il "login" su un sito web, ad esempio, non devono ripetere la stessa operazione ogni volta che vedono una pagina differente di quel sito. Così un server può evitare questo inconveniente tenendo traccia dello stato del client. La prima volta che il client richiede una pagina del sito, il server indica all'utente di fare il "login". Appena l'utente continua a navigare tra le pagine dello stesso sito, il server ricorda il precedente login effettuato con successo e non chiede login aggiuntivi.

Il mantenimento di stato richiede una capacità critica: i server devono essere in grado di associare una richiesta HTTP ad un'altra. Il server deve essere capace di dire, ad esempio, che l'utente che sta richiedendo una nuova pagina è realmente lo stesso che ha già fatto il login, non un utente diverso che non è stato autorizzato.

Cookies

Il meccanismo definito dall'HTTP per il mantenimento dello stato è costituito dai *cookies*.

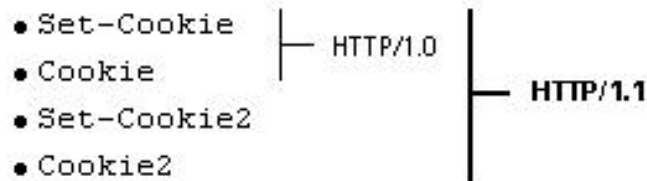
Un server crea i cookies quando vuole tenere traccia dello stato di un client e li manda al client nella sua risposta. Una volta che il client riceve un cookie, può includerlo in richieste successive allo stesso server, come indica la figura in basso.



Il client può continuare a includere il cookie nelle sue richieste finchè non si verifichi una delle due condizioni seguenti:

- il cookie scade.
- il server dice al client di non usarlo più.

Sebbene i cookies siano spesso descritti come entità, sono più facili da capire a livello funzionale se vengono considerati come un'estensione del protocollo HTTP. Essi possono essere definiti come l'aggiunta di due intestazioni in HTTP/1.0 e quattro in HTTP/1.1:



Set-Cookie2 (estensione di Set-Cookie)

L'intestazione `Set-Cookie2` è una forma leggermente aggiornata dell'intestazione `Set-Cookie` della versione 1.0 dell'HTTP.

Entrambe le intestazioni sono usate dai server per iniziare il processo di gestione di stato con un client.

Tramite l'inclusione dell'intestazione `Set-Cookie2` nella sua risposta, un server fornisce un cookie al client, e implicitamente gli chiede di rispedirlo in richieste successive al server stesso.

L'intestazione comincia dando un nome e un valore al cookie, poi può fornire alcuni o tutti gli attributi elencati nella tabella in basso.

Un esempio con tutti i possibili attributi può essere il seguente.

```
Set-Cookie2: NAME="VALUE";  
    Comment="Shopping Cart";  
    CommentURL="http://merchant.com/cookies.html";  
    Discard; Max-Age="300"; Path="/shopping";  
    Port="80"; Secure; Version="1"
```

Di seguito viene fornita l'interpretazione dei vari attributi.

Cookie

Se un client desidera supportare il processo di gestione di stato HTTP, fornisce nelle richieste successive a un server ogni cookie che ha ricevuto da quest'ultimo.

Tali cookies sono inseriti in un'intestazione `Cookie`. L'esempio seguente mostra solo un singolo cookie, ma un client può ricevere più cookies da un server, così li può combinare tutti in una sola intestazione o usare intestazioni separate.

```
Cookie: $Version="1"; NAME="VALUE";  
    $Path="/shopping"; $Domain="www.shop.com";  
    $Port="80"
```

Ogni cookie incomincia con l'identificazione della versione del mantenimento di stato HTTP che il client sta usando; la versione corrente è la 1, come nell'esempio.

La versione è sempre seguita dal nome del cookie e il suo valore. Quest'ultimi sono assegnati dal server nella sua intestazione `Set-Cookie` o `Set-Cookie2`, ma bisogna notare che il server non può dare un nome al cookie che sia `$Version`, altrimenti sarebbe impossibile riconoscere il cookie in un'intestazione. Le specifiche HTTP infatti proibiscono che i nomi dei cookies comincino con il simbolo del dollaro (\$).

I campi aggiuntivi che seguono il nome del cookie sono opzionali.

Nell'esempio compaiono `$Path`, `$Domain` e `$Port` che rappresentano alcuni degli attributi possibili per un cookie, elencati di seguito.

Cookie2

Nonostante la somiglianza dei nomi, la relazione tra le intestazioni `Cookie2` e `Cookie` non è come quella esistente tra `Set-Cookie2` e `Set-Cookie`.

Mentre `Set-Cookie2` è una versione leggermente modificata di `Set-Cookie`, `Cookie2` e `Cookie` sono

intestazioni differenti con usi completamente diversi.

L'intestazione `Cookie2` indica soltanto quale versione di gestione di stato è supportata dal client. La versione corrente è la 1, così l'intestazione assumerà il formato seguente.

```
Cookie2: 1
```

Un client deve includere questa intestazione ogni qualvolta manda l'intestazione `Cookie`. In questo modo il server capisce che può usare sia `Set-Cookie2` che `Set-Cookie` nelle risposte successive.

I client che non supportano pienamente `Set-Cookie2` omettono l'intestazione `Cookie2`, ma possono includere l'intestazione `Cookie`; i server così non manderanno risposte con `Set-Cookie2` a questi client.

Attributi dei cookies

Come detto, i cookies consistono in una serie di attributi. Il server sceglie i valori per gli attributi richiesti e, se lo desidera, anche per quelli opzionali.

La tabella seguente elenca e spiega l'uso dei possibili attributi di un cookie.

Attributo	Stato	Note
NAME	Richiesto	Un nome arbitrario per il cookie, assegnato dal server.
Comment	Opzionale	Un commento che il server può aggiungere al cookie; il commento può essere usato per spiegare come i server usano il cookie, possibilmente rassicurando gli utenti che hanno preoccupazioni in riguardo.
CommentURL	Opzionale	Un URL che il server può fornire con un cookie; l'URL può spiegare come il server usa il cookie.

Discard	Opzionale	Istruisce il client a scartare il cookie non appena l'utente finisce; in pratica, dice al browser web di non memorizzare il cookie nel disco dell'utente.
Domain	Opzionale	Il dominio (dato dal Domain Name System) per cui è valido il cookie; un server può non specificare un dominio oltre a quello a cui appartiene egli stesso, ma può specificarne uno più generale di un singolo server.
Max-Age	Opzionale	Il tempo di vita di un cookie, in secondi.
Path	Opzionale	Gli URL sul server a cui si applica il cookie.
Port	Opzionale	Un elenco di porte TCP a cui si applica il cookie.
Secure	Opzionale	Istruisce il client a rimandare il cookie in richieste successive solo se queste sono sicure; da notare, comunque, che l'HTTP non specifica che cosa vuole dire "sicuro" in questo contesto.
Version	Richiesto	La versione di mantenimento di stato adottata dal cookie; la versione corrente è la 1.

Accettazione di cookies

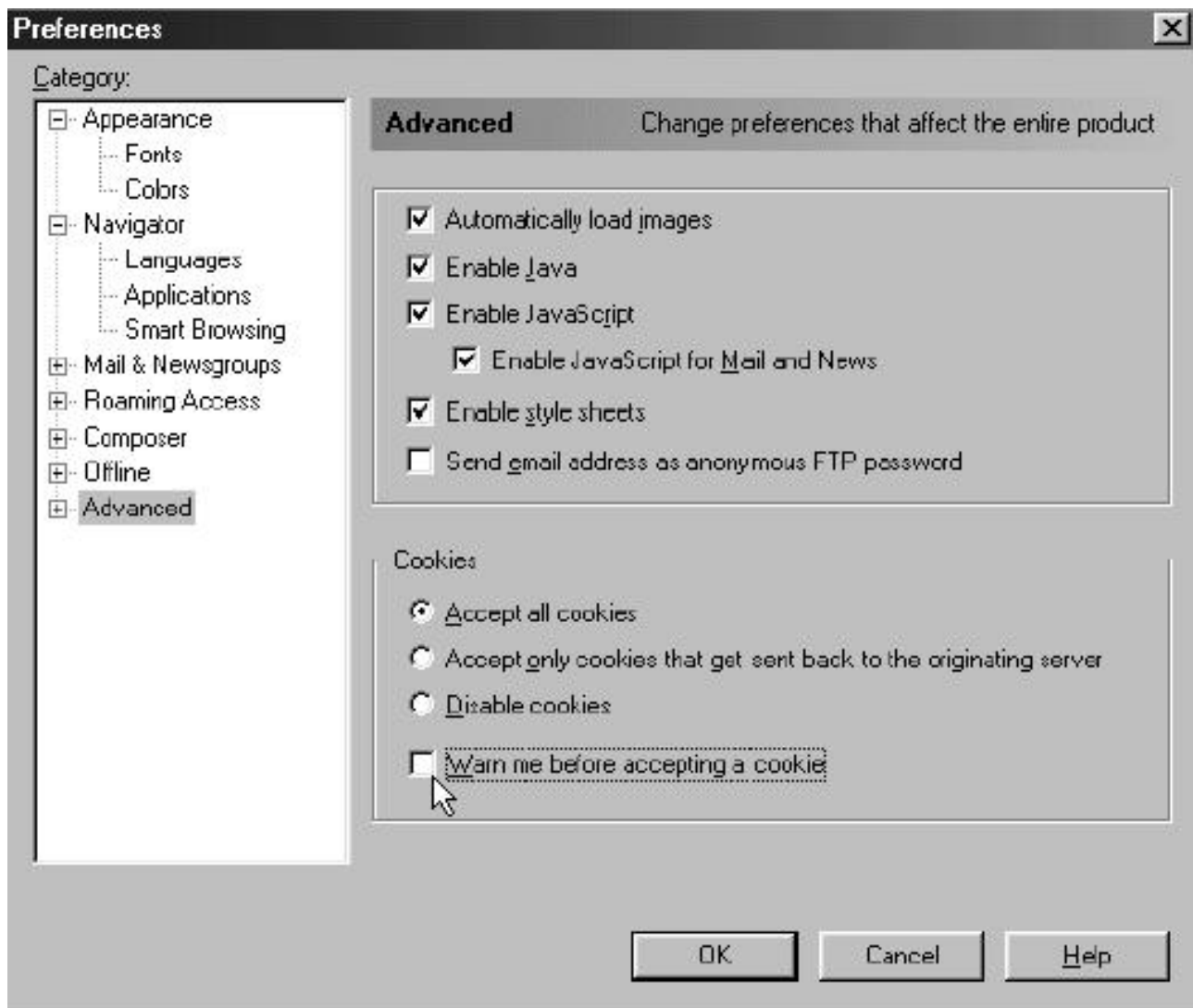
Quando un client riceve un cookie, salva i suoi attributi. In più, se il server ne ha omessi alcuni opzionali, il client inserisce i valori di default.

La tabella seguente elenca i valori di default che il client applica agli attributi mancanti.

Attributo	Valore di default
Discard	Vedi valore di default di Max-Age.
Domain	Il nome di dominio del server che ha creato il cookie.
Max-Age	Si tiene il cookie solo finchè la sessione corrente dell'utente è attiva (quindi non si memorizza il cookie sull'hard disk).
Path	L'URL per cui il cookie è stato ritornato, fino a, ma non incluso, il file specificato da quell'URL.
Port	Il cookie si applica a ogni porta. (da notare che se tale attributo è presente nel cookie ma non ha un valore, il client mette come valore la porta della sua richiesta originale).
Secure	Il cookie può essere rimandato con richieste non sicure.

Rifiuto di cookies da parte dell'utente

Un client non è mai costretto ad accettare cookies. Gli utenti, ad esempio, possono configurare i loro browser web per accettarli o meno, come mostra la figura seguente.



Un server HTTP, quindi, non può essere sicuro che un cookie sia accettato, anche se è configurato in modo appropriato.

Rifiuto di cookies da parte del client

Anche se un utente è disposto ad accettare cookies, le specifiche HTTP richiedono che il client li rifiuti in certe circostanze. I cookies rifiutati sono semplicemente ignorati dal client e quindi non sono mai inclusi in richieste successive.

Le condizioni sotto le quali un client deve rifiutare un cookie di un server sono le seguenti (da notare che il client considera queste condizioni dopo che ha applicato ogni valore di default per gli attributi opzionali):

- Il valore dell'attributo `Path` non è un prefisso dell'URL che appare nella richiesta del client.
- Il valore dell'attributo `Domain` non contiene nessun punto al suo interno (non solo all'inizio), a meno che il valore sia `".local"`.

- Il server che ha ritornato il cookie non appartiene al dominio specificato dall'attributo `Domain`.
- La parte dell'host dell'attributo `Domain`, se presente, contiene un punto al suo interno.
- La porta della richiesta del client non è inclusa nell'attributo `Port` (a meno che quest'ultimo sia assente).

Quando un client accetta un cookie, il nuovo cookie sovrascrive ogni cookie precedentemente accettato che ha gli stessi attributi `NAME`, `Domain` e `Path`.

Cookies rimandati dal client

Non appena un client accetta un cookie e inserisce gli appropriati valori di default, analizza quando tale cookie deve essere rimandato a un server in successive richieste `HTTP`.

Le regole sotto le quali un client include un cookie in una richiesta sono elencate di seguito (da notare che in una richiesta possono essere inclusi più cookies se le condizioni seguenti sono verificate):

- Il nome del dominio della nuova richiesta deve appartenere al dominio specificato dall'attributo `Domain` del cookie.
- La porta della nuova richiesta deve essere inclusa nell'elenco di porte dell'attributo `Port` del cookie, a meno che questo attributo era assente nel cookie (ad indicare tutte le porte).
- Il path della nuova richiesta deve combaciare con l'attributo `Path` del cookie, o rappresentarne una parte.
- Il cookie non deve essere scaduto (visibile dal suo attributo `Max-Age`)

Quando un client rimanda un cookie a un server, include gli attributi `Domain`, `Path` e `Port` se erano presenti nel cookie originale. Viceversa, non li include se erano assenti da quest'ultimo.

Testi di riferimento

- Stephen Thomas, "HTTP Essentials Protocols for Secure Scaleable Web Sites", Wiley.
- Chris Shiflett, "HTTP Developer's Handbook", Sams Publishing.
- Kurose-Ross, "Internet e Reti di Calcolatori", McGraw-Hill.