1 WWW - introduzione

1.2 Componenti semantici del Web

URI (*Uniform Resource Identifier*), HTML, HTTP.

2 Web Clients

Tre tipi di client: browser, spider, agent software.
Le funzioni del *browser* sono legate al web. Un browser è implementato come un web-client. Una sessione è una serie di richieste dell'utente con possibili risposte.
Gli step nel processo di un browser sono:

1. richiesta URL a DNS;
2. connessione TCP con il server
3. richiesta HTTP
4. risposta HTTP
5. connessioni parallele

Tramite il caching è possibile evitare alcuni di questi passaggi, e HTTP rappresenta il protocollo di default (a meno che non siano specificati altri).
Il proxy è presente sul percorso tra client e server.

Browser Caching

Ci sono due tipi di cache: memoria di processo o file system. Il browser è in grado di controllare se ciò che è in cache è aggiornato oppure no.
Un tipo di verifica di cache è detto *strong cache consistency* (si controlla ad ogni richiesta); invece il *weak cache consistency* usa delle euristiche di controllo.
Un utente può decidere di bypassare l'utilizzo della cache facendo richieste direttamente al server, che risponde con il last_modification_time.
Il messaggio di request di HTTP è l'unità di comunicazione tra client e server che consiste di un header obbligatorio e di un body opzionale. Nell'header di trovano info del tipo: info user_agent (browser), formato di codifica e credenziali utente.
Il browser è personalizzabile sia visualmente che semanticamente. La personalizzazione semantica aggiunge header nella richiesta (5 tipi):

1. rigurardanti la connessione (scelta dei proxy)
2. legate al contenute o alla scelta della risorsa (es: linguaggio accettato)
3. legate al caching
4. gestione delle risposte
5. cookies.

Un plug-in è codice che si invoca per gestire un formato di file in un browser (o una risorsa). I cookies sono piccoli file di testo che, se utilizzati, storicizzano info sul computer utente e sono inoltrate dall'origin server per simulare sessioni di lavoro.
I web server sono in grado di tracciare gli utenti di una o più sessioni. Le informazioni nei cookie possono essere per singolo utente (carrello) o comuni a gruppi di utente. Il client effettua una richiesta e il server gli manda il cookie. Alla prossima richiesta il client invia il cookie e il server lo riconosce.
Le **operazioni dell'utente sui cookie** sono:

1. decidere se accettare i cookie
2. limitarli in numero e dimensione
3. specificare i siti di provenienza
4. specificare la sessione per la quale si accettano i cookie
5. richiedere che i cookie siano originati dallo stesso server che mostra la pagina.

Sicurezza per i cookies: possono essere intercettabili e trasmettono info riguardante l'utente (in chiaro). La maggior parte degli utenti non sa neanche cosa siano.

Spider

Uno *spider* (detto anche client automatico) è un programma usato per ottenere certe risorse da un gran numero di siti ed è usato nei motori di ricerca. Per la ricerca testuale nel web si usano indici detti *inverted index* e le *stop words* che sono le parole escluse dagli spider ritenuti di poca importanza.
Lo spider attacca una pagina iniziale di un sito e poi esamina tutti i suoi link facendo una visita in ampiezza. Inoltre gli spider possono decidere di effettuare le ricerche in un'unica visita o in più passate in modo da non sovraccaricare il sito soggetto ad esame; solitamente uno spider agisce su di un sito con una frequenza di uno o due accessi al minuto. Per migliorare le proprie prestazioni gli spider analizzano le pagine in base alla frequenza con cui vengono aggiornate in modo da poter agire in modo settoriale (parti della pagina).
Non c'è un metodo efficace per controllare le richieste da parte degli spider, ma si adottano due tecniche:

1. utilizzo di un file di record d'accesso per gli spider da parte dell'amministratore
2. utilizzo di tag speciali per indicare le risorse da NON includere nell'inverted index.

Intelligent Agent e special purpose browser

Si parla di agenti in grado di confrontare i risultati di una ricerca e presentarli agli utenti in base al loro profilo.
Gli agenti intelligenti sono creati per specifici compiti dell'utente, i browser a scopi speciali sono creati per permettere l'interazione tra utenti e imporre e permettere l'unione di browsing.
Gli agenti intelligenti si dividono in:
Meta motori di ricerca (ricerca su più motori) e poi presentata all'utente. È meno potente di un browser perché ancora non implementa le funzioni base.
Gli *auction agent* sono specifici per ricerche in aste elettroniche.
Gli special purpose si dividono in: *co-browser* che aiutano l'utente aumentando il numero di richieste; *browser collaborativi* che sono come i co-browser ma permettono il lavoro collaborativi ad un gruppo. Ci sono poi gli *offline-browser*.

3 Web Proxy

Per ridurre la ridondanza di comunicazione è necessario l'utilizzo di un nodo intermediario che fa uso di una cache.
Ci sono:

1. firewall
2. gateway
3. tunnel

Una prima classificazione ad alto livello è quella tra i *caching-intermediary* e i *trasparent*, che applicano una trasformazione al messaggio e si dividono in due tipi (trasparent e non-trasparent).
I *trasparent* applicano modifiche molto superficiali (come aggiunta di info di identificazione del proxy o del server), in *non-trasparent* fanno operazioni come rendere anonima la richiesta e oscurare le info del proxy (o del server) al client, o conversione dei dati (da un formato all'altro), traduzione di messaggi…

3.3 Applicazione dei proxy

Funzione di sharing per le risorse web in cache, in modo che i client che accedono alla stessa risorsa vengono forniti di ciò che chiedono.

Caching responses

Funzione opzionale per il proxy e non tutti i client traggono beneficio dal caching (per esempio uno spider).

Oscuramento dei client

Quando il server vuol capire da chi è arrivata la richiesta, allora trova il proxy invece del client. Ci sono due parti potenziali di info presenti in una richiesta, che possono ridurre l'anonimia dell'utente: user-agent e stato (cookie o id-sessione).

Trasformazione di richieste e risposte

Il proxy può modificare req o res (o entrambi) e il client può informare il proxy delle preferenze (come la compressione delle risposte se la connessione è bassa) e può cambiare anche non in base alle capacità del client (aggiungendo info proprie, come la compressione dei dati…).

Gateway per sistemi non HTTP

Le richieste HTTP sono tradotte in altri tipi di protocollo. In alcuni casi può agire come tunnel (comunicazioni in SSL).

Filtering per req e res

Gioca il ruolo di *gatekeeper* (per esempio blocca il traffico verso un sito).

3.4 Ruoli del proxy legati ad HTTP

Di seguito indichiamo i passi del proxy per la comunicazione e lo scambio di req e res:

1. Client ←richiesta→ URL tramite DNS *lookup* per la ricerca del proxy
2. Client contatta → Proxy tramite connessione TCP (il proxy fa la *lookup* per il server)
3. Proxy contatta → Server tramite connessione TCP
4. Client inoltra a → Proxy inoltra a → Server request
5. Server inoltra a→ Proxy inoltra a → Client response

Gestione delle richieste e delle risposte HTTP: un proxy si deve adattare a diversi requisiti:

1. identificazione di se stessi
2. cambio di versione di protocollo (e.g. client 1.0 e server 1.1)
3. aggiunta obbligatoria di info sulle risorse (da quanto tempo è in cache)
4. non deve cambiare il significato semantico di req e res
5. deve mantenere una connessione con il server durante la risposta del server

Se il proxy ha funzionalità di web server ci sono dei meccanismi per fare richieste direttamente al server senza passare per il web proxy. In più può funzionare come web client.
I proxy possono trovarsi in successione l'un l'altro in modo lineare, oppure in modo gerarchico per avere cache molto grandi.

Akamai e risoluzione del bottleneck

Il problema del collo di bottiglia è affrontato in quattro punti della rete:

1. ***Problema del primo miglio***

Riguarda la velocità con cui gli utenti possono accedere al sito, ossia la capacità di connessione del sito web è limitata (si riferisce al miglio nei pressi del server). Non si risolve solo aumentando la larghezza di banda, ma aumentando anche la capacità di rete interna all'ISP.

2. ***Congestione ai peering-point***

I *peering point* sono i punti di'interconnessione tra reti indipendenti. I motivi sono di tipo economico poiché i costi per mettere su rete i peering point sono alti e le grandi reti non mettono i peering point tra di esse (o ce ne sono pochi).
Uno dei tipi più comuni di peering si ha quando piccole reti comprano connettività da grandi reti. Il problema sorge quando la creazione dei circuiti necessari prende molto tempo e le aziende piuttosto che utilizzare i cavi e la banda disponibile preferiscono installare nuove fibre per aumentare i profitti. Il tempo di installazione è così alto che alla fine la richiesta è cresciuta talmente tanto da diventare un collo di bottiglia. In pratica la richiesta è talmente alta che c'è più perdita di pacchetti. Perciò questo è un problema di larga scala inerente alla struttura di Internet.

3. ***Backbone***

Sono le strutture principali della rete. Il problema è che la capacità delle backbone dovrebbe crescere dinamicamente alla richiesta di utilizzo. Un altro problema sono i router che sono limitati dalla tecnologia corrente.
A causa dell'enorme crescita di domanda questo diventa un problema serio.

4. ***Problema dell'ultimo miglio***

Riguarda la connessione dell'utente finale. Non è un vero e proprio problema, poiché se l'utente avesse tutta la banda disponibile allora non ci sarebbe banda sufficiente per tutti.

Delivery content from the network edge

L'*edge delivery* fornisce una soluzione al modello centralizzato di Internet, perché è un modello scalabile nella distribuzione di info e servizi agli utenti finali.
Nell'edge delivery il contenuto di ogni sito web è disponibile su più server, posizionati all'estremità di Internet. In poche parole un browser dovrebbe essere capace di trovare tutti i contenuti richiesti su un server nella sua home network.
L'edge delivery risolve il problema del primo miglio rendendo ogni contenuto web disponibile su più fonti. La soluzione correntemente utilizzata è l'*outsourcing* che sposta semplicemente il problema del primo miglio al centro dati del fornitore di hosting.
Ovviamente anche l'edge delivery ha un problema: se si ha uno schema con 50-100 locazioni sparse per le 10 reti più grandi, non si risolve il problema del peering, ma si attenua quello delle backbone, NON si risolve quello dell'ultimo miglio (si elimina il WWW – World Wide Wait).

Edge delivery: difficoltà e costi l'implementazione

1. I contenuti devono essere sviluppati sugli edge server (non difficile)
2. richiede un grande sviluppo di server edge su migliaia di reti
3. devono essere posizionati in punti strategici a livello geografico
4. ci devono essere algoritmi complessi per la gestione di contenuti che viaggiano attraverso migliaia di reti
5. Il contenuto deve essere sempre consistente, aggiornato e sincronizzato
6. è cruciale la tolleranza ai guasti

7. si deve essere capaci di instradare e reinstradare velocemente il contenuto real-time in risposta alla congestione della rete
8. ci deve essere un modo per localizzare gli utenti per scegliere il server migliore per l'instradamento della richiesta
9. gestire il carico assicurando che gli oggetti risiedano nel minor numero di locazioni con max performance di download
10. monitoring delle performance sugli edge server
11. deve essere capace di supportare migliaia di richieste al giorno e capace di fatturarle attraverso più reti
12. deve garantire la fornitura affidabile dei servizi

Edge delivery – soluzione di Akamai

Akamai ha sviluppato un edge delivery che incrementa il traffico dell'800% implementando un meccanismo chiamato *freeflow* che si basa sull'implementazione degli edge server e sofisticati servizi di monitoring di rete, e inoltre include algoritmi che mappano direttamente ogni richiesta utente sul server edge ottimale assicurando una validità del 100%.
Freeflow permette di focalizzarsi su problemi come marketing, prodotti e sviluppo di contenuti invece che sul difficile ostacolo della consegna dei contenuti tra le reti.

Content Delivery Network – CDN

I CDN forniscono una consegna affidabile e veloce dei contenuti agli utenti web, streaming e trasmissione in Internet.
Akamai fornisce una soluzione e ottimizza le performance dei siti web. CDN risolve i problemi dei siti web relativamente ai tempi di elaborazione dei server e dei ritardi di Internet. Offre il contenuto richiesto da un server più vicino all'utente.
Akamai fornisce un'architettura distribuita implementando affidabilità, scalabilità, ecc…
CDN si occupa solo del rilascio di contenuti statici, mentre Akamai si rivolge a tutte le distribuzioni di siti web (dinamici, streaming). La *edge suite* è stata creata per lavorare il meno possibile con i CMS e gli application server che generano molto contenuto dinamico.
La tipica architettura di una struttura commerciale ospitata da diverse locazioni, prevede tre livelli fisici L'edge suite deve offrire:

1. *generazione contenuti*: deve coordinare e comunicare le info generate che devono essere presentate agli utenti finali
2. *integrazione*: deve fornire connettività su HTTP
3. *consegna dei contenuti*: largamente distribuito si trova sugli edge della rete ed è basato sulla consegna e memorizzazione di contenuti attraverso gli edge di Internet raggiungibili tramite pochi hop
4. *consegna dinamica di contenuti agli estremi*.

Quando un sito web utilizza i server edge di Akamai, tutte le sue richieste iniziali sono inviate al server edge che controlla la validità delle pagine nella cache prima di rispondere al client.
Se la pagina è dinamica viene memorizzata per intero sulla rete akamai, sempre se la pagina non è specifica per l'utente. La maggior parte dei siti personalizzano la propria pagina web sulle specifiche dell'utente, per cui queste pagine non sono memorizzabili in cache.
In ogni caso però risulta più veloce l'utilizzo di Akamai rispetto alla richiesta diretta al server d'origine, per via delle connessioni che ci sono tra client e server.
Akamai inoltre utilizza procedure di compressione per ridurre il traffico dati sulla rete tra edge server e origin server.
Inoltre è possibile utilizzare dei tag speciali per definire solo le parti più dinamiche del sito, questo è possibile grazie ad un accordo tra i leader del settore.
L'edge suite include l'*Edge Side Includes* che è un linguaggio di markup per definire i frammenti più dinamici delle risorse web.

L'ESI deve avere le capacità di

1. includere ed astrarre file per comprendere pagine web
2. variabili d'ambiente
3. inclusione condizionate (basate su costrutti booleani)
4. specifica di pagine alternative e comportamenti in risposta ad eventi d'errore

Invece di mandare pagine complete si invia un template della pagina specificando la parte memorizzabile in cache in modo da:

- velocizzare il caricamento delle pagine
- ridurre il traffico sull'application web server.

FREE-FLOW

Identifichiamo con *ARL*: Akamai Resource Locator l'indirizzo che identifica univocamente una risorsa sui server di Akamai. Ha un formato che consta di:

1. *Serial Number*: identifica il gruppo di appartenenza di oggetto (o risorsa)
2. *Akamai Domain*: assicura che la risorsa sia su un server Akamai senza doverla ricercare sull'home site
3. *type code*: definisce come deve essere interpretata la risorsa dai server Akamai
4. *content provider code*: identifica il customer che ha sottoscritto il contratto con Akamai
5. *object data*: contiene l'expiration time della risorsa ed è utilizzato per assicurare la freschezza della stessa
6. *URL*: rappresenta l'URL assoluta dell'oggetto per ottenere la risors la prima volta (origin server).

Free-flow utilizza un'architettura DNS propria (*FreeFlow DNS*), in cui tutte le richieste di ARLs sono dirette alla rete AKAMAI tramite il *Akamai Domain* presente in ogni ARL. Il sistema di free-flow DNS garantisce una consegna veloce del contenuto richiesto risolvendo ogni indirizzo del tipo *.g.akamai nell'indirizzo IP del server AKAMAI che prima può soddisfare la richiesta, questo lo può fare grazie all'opportunità di conoscere il traffico sulla rete e di conoscere l'indirizzo sorgente della query DNS.

Free-Flow DNS è implementato su una gerarchia a due livelli:

1. 50 server di alto livello .akamai.net (HLDNS high-level-DNS)
2. 2000 di basso livello .g.akamai.net (LLDNS low-level-DNS)

Ogni HLDNS è responsabile di inoltrare la richiesta al LLDNS che è più vicino al client che ha fatto la richiesta. I LLDNS devono risolvere la richiesta e quindi fornire l'IP finale del server su cui risiede la risorsa e che può gestirla nel modo più rapido.

I DNS impiegati nell'architettura freeflow effettuano un monitoring continuo della rete per avere un controllo sulle condizioni della stessa, in modo tale da poter dare la risposta in diversi secondi. Come funziona la gerarchia:

- Client → DNS locale → richiesta di *.g.akamai.net/… (se la risorsa già è presente in cache risponde direttamente, altrimenti esegue query iterativamente finché non viene restituito l'ID di un server Akamai)
- ? Se si deve ricercare l'indirizzo del server Akamai viene fatta una richiesta dal
    - local DNS → *.net root server*, chiedendo di parlare con akamai.
    - .net root server → local DNS (e gli restituisce la lista dei server Akamai)
    - quando viene restituita la lista dei server Akamai, il local DNS ne sceglie uno e lo contatta per ottenere un LLDNS di freeflow DNS (viene scelto quello ottimale).
- local DNS → Client (viene restituito l'IP di un server Akamai – o dalla cache o da "?" che corrisponde a quello ottimale).

Ovviamente in freeflow DNS viene abilitato l'uso della cache e il TTL delle risposte è settato in modo tale che si sia sicuri che quando il TTL è scaduto è già avvenuto un nuovo check dei file in cache. Infatti visto che gli HLDNS vengono aggiornati ogni 7-10 minuti, le loro risposte hanno un TTL di 20 minuti, mentre gli LLDNS vengono aggiornati ogni 2-10 secondi, quindi il loro TTL è settato a 20 secondi.

4 Web Server

Web site → collezione di pagine web, oggetti statici
Web server → serve a soddisfare le richieste dei client web

4.1.2 Web server

Un web server è un programma che dovrebbe essere in grado di gestire richieste HTTP per particolari risorse (ad esempio il trasferimento di una pagina web potrebbe includere più server, script e DB).
Un web server gira su una piattaforma che consiste di computer che hanno accesso alla rete.

4.2 Gestione delle richieste client

Un WS (da ora in poi per web server) per gestire una richiesta deve effettuare i seguenti passi:

1. leggere e tradurre i messaggi di req HTTP
2. convertire l'URL in file
3. determinare se la richiesta è autorizzata
4. costruire ed inviare la response.

È possibile utilzzare un file di log per tener traccia di req e res. Una volta che il client riceve il messaggio di risposta, parsa il file HTML. Quando c'è in atto una politica di caching in browser può inviare una richiesta per validare la copia in cache e dopo aver confrontato il last_modification_time della copia in cache e della data restituita dal server è in grado di decidere la freschezza della risorsa.

4.2.2 Controllo dell'accesso

Un WS dovrebbe limitare l'accesso alle risorse e per fare ciò utilizza due meccanismi:

1. *autenticazione*: è un sistema che si basa sulla richiesta di username e passwor all'utente che vuole accedere alla risorsa. Dopo il primo accesso, le info di autenticazione viaggiano nei messaggi http di req e res
2. *autorizzazione*: il sistema utilizza le policy per controllare l'accesso alle risorse web. Queste politiche sono specificate nella configurazione del web server. Di solito una policy è una lista d'accesso. Le policy di autorizzazione basata su nome e indirizzo del client è rischiosa, perché se non si è autorizzati e si passa per un proxy autorizzato, allora si diventa autorizzati! La gestione delle policy introduce carico sul server, quindi va usata solo se necessario. Un tipo conveniente di autorizzazione è il modo gerarchico, attraverso la gestione delle cartelle.

4.2.3 Risposte generate automaticamente

Oltre a fornire contenuti statici, il web permette di generare dinamicamente delle risorse, ciò è fatto tramite:

- *server-side includes*, che servono ad istruire il WS su come customizzare una risorsa statica, basandosi su direttive scritte in linguaggio simile all'HTML
- *server-script*, programmi separati che generano la risorsa richiesta.

Server side includes

È un processo con il quale si possono aggiungere "macro" o "direttive" che permettono di inserire delle informazioni al tempo della richiesta. Questo tipo di informazioni generate nelle macro sono visualizzate direttamente nel body. Più generalmente le macro possono istruire il server ad invocare un programma ed includere l'output nel documento.
PHP è una piattaforma di scripting integrata in HTML.

Server script

Piuttosto che integrare info in HTML, un programma separato può generare la risorsa, e in questo caso, l'URL HTTP corrisponde ad un programma piuttosto che ad un documento.
Il server può integrare tramite script:

- invocando processi separati: lo script è un processo separato invocato dal server, atto a fornire la risorsa richiesta (CGI)
- lo script può essere un modulo software separato che gira come parte del web server. Chiamando un modulo all'interno del server si evita l'overhead per la creazione di processi separati
- può essere un processo persistente che gestisce più richieste per un lungo periodo ti tempo

4.2.4 Creazione ed utilizzo di cookie

I web sites usano cookie per tracciare e memorizzare informazioni attraverso più trasferimenti HTTP. I cookie non sono gestiti dal WS ma dagli script.
Di solito una richiesta HTTP non fornisce info sufficienti per l'identificazione di un client, perché gli utenti possono utilizzare la stessa macchina oppure l'IP può cambiare, invece il browser può includere un cookie unico in ogni richiesta HTTP, per il browser è visto come una stringa di caratteri, mentre per lo script del server sono info di controllo. Ovviamente i cookie non garantiscono alcun grado di controllo, in quanto un utente esperto potrebbe contraffarli, a meno che non contengano informazioni cifrate.

4.3 Condividere informazioni attraverso richieste

Un WS può memorizzare delle info riguardanti delle richieste HTTP e i loro header.

4.3.1 Condivisione di risposte HTTP attraverso le richieste

Un sito potrebbe memorizzare le risorse richieste frequentemente nella memoria principale (caching). Il caching server-side è diverso dal caching browsing o caching proxy, perché migliora le performance locali del server riducendo i tempi d'accesso al file system.
Per mantenere la consistenza del file confronta il last_modification_time.
Oltre a fare caching di pagine statiche, il server può prevedere caching per risposte generate dinamicamente; ad esempio se tanti utenti cercano la stressa stringa su google il risultato è lo stesso allora ci conserviamo il risultato in RAM.

4.3.1 Condivisione di metadati

Oltre al caching delle risorse web si può effettuare il caching delle info generate attraverso la gestione delle richieste HTTP:

- traduzione dell'URL in filename
- informazioni di controllo riguardanti la risorsa, tra cui File description, dimensione, data modifica (riduce overhead nell'header)
- HTTP response header (si cacha una porzione dell'header anziché interamente)

Inoltre si possono cacheare informazioni per richieste differenti:

- Data e ora corrente (usa la stessa ora finché il server non è libero)
- Nome del client (l'hostname è cacheato per non ripetere il processo di get_host_by_addr)

4.4 Architettura dei server

Tipicamente un server deve gestire più utenti in contemporanea e tutte le richieste devono condividere le sue risorse. Esistono varie tecniche per condividere le richieste competitive.

4.4.1 *Event driven*

Esiste un singolo processo che si alterna tra le richieste in modo sequenziale. Il processo dovrebbe accettare le richieste, generare e trasmettere risposta, prima di processare la prossima richiesta.

Piuttosto che utilizzare questo sistema, il processo elabora una piccola quantità di lavoro a beneficio di ogni richiesta. Tale approccio è più appropriato quando ogni richiesta produce una limitata quantità di lavoro. Una richiesta web può essere suddivisa in un numero più piccolo di step. Alcuni di questi step introducono un ritardo che non può essere gestito dal web server. Per evitare l'attesa del completamento di questi task che introducono overhead un WS event-driver esegue chiamate al sistema non bloccanti, che permettono al processo di continuare la sua esecuzione mentre aspettano la risposta del sistema. Con questo metodo il server può avere uno o più eventi in coda relativi a più richieste. La gestione di un evento può essere vincolata a chiamate di sistema, che possono a loro volta chiamare più eventi.
Un server event-driven può facilmente assicurare che un operazione di scrittura sia completata prima che ne inizi un'altra. Girando sul server come un singolo processo si facilita la condivisione dei dati tra richieste diverse.
Un buon server ED si basa su un buon supporto per le chiamate non bloccanti da parte del sistema. La maggior parte dei WS non hanno architettura ED.

4.4.2 *Process driven*

Il server può dedicare un processo ad ogni richiesta. Il modello PD dipende dal OS che deve alternarsi tra le varie richieste. I vari processi girano tutti insieme nel sistema, in modo che ognuno abbia l'impressione che tutti girino in contemporanea. Tipicamente esiste un processo master che gestisce le nuove connessioni con l'utenza tramite creazione o *fork* di processi. Una volta che il processo ha trasmesso la risposta allora termina e vengono rilasciate le risorse ad esso assegnate evitando il *memory leak*.
Per ridurre l'overhead si può far uso di pool di processi. Quest'ultimo approccio è sensibile agli errori di programmazione e quindi il sistema operativo può decidere di uccidere un processo dopo che ha gestito un certo numero di richieste, o applicare tecniche di gestione e limitazione della memoria.
Limiti: lo switch tra processi introduce overhead, mentre nell'event driven è gestito tutto in un singolo processo. La maggior parte delle richieste web non richiede molta computazione, quindi si crea molto switching. Si crea overhead addizionale quando si devono condividere le risorse dati.

4.4.3 Architetture Ibride

L'idea è quella di gestire ogni processo del sistema process-driven come un server event-driven, collezionando la serie di richieste. In tal modo si abbassa l'overhead per lo switch di processi e per la condivisione dei dati per richieste soddisfatte dallo stesso processo. Come l'event driven ha bisogno della gestione accurata del ritardo per l'esecuzione della singola richiesta. Per cui si riduce il numero di richieste associate ad ogni processo.
Si introduce overhead per il passaggio tra diversi processi o condivisione dati tra di essi.
Un altro approccio è quello in cui si introducono più thread di controllo, di cui ognuno condivide lo spazio degli indirizzi del processo che lo contiene, ma un WS multithread può anche associare ad ogni richiesta un thread (un processo ha tanti thread). Non si necessita di coordinare esplicitamente lo switch tra i thread che in ogni caso introduce meno overhead rispetto a quello tra processi.
I thread devono impiegare un sistema di sincronizzazione dell'accesso ai dati che è difficile da implementare per i programmatori.
Un terzo approccio è quello in cui l'event driven gestisce le piccole richieste (ed è in genere settato di default) e c'è un processo ED che gestisce gli stati iniziali della richiesta. Nel caso in cui noti che la richiesta richiede molta computazione allora viene associata ad un helper, basato su PD.

4.5 Server Hosting

Un singolo PC può ospitare più siti web e un sito web può essere replicato su più macchine. Una web host company ha uno o più computer sparsi nel mondo che ospitano più siti web. L'hosting

provider offre particolari vantaggi, in fatti le host-company devono gestire i dettagli tecnici, ossia la manutenzione del server:

1. gestione del server e dei dettagli tecnici
2. l'azienda deve fornire un'infrastruttura di elaborazione di rete per supportare i picchi di carico
3. l'hosting company deve ammortizzare i costi d'elaborazione e di rete ospitando più siti
4. il proprietario del contenuto non rischia di esporre i propri dati.

Combinando sulla stessa macchina siti che sfruttano risorse differenti si ha il best effort per la macchina, ma questo richiede un modo appropriato per indirizzare il client sul sito appropriato. Questo hosting non è accettabile, perché si vorrebbero hostname separati, e quindi virtual server.
I *Virtual Server* sono più server che girano sulla stessa macchina ed ognuno di essi ha un suo albero di documenti ed opera come se fosse l'unico web server sulla macchina. Quando il client richiede la risorsa ospitata da un'azienda con multiple-hosting deve risolvere il suo IP, cioè ogni virtual server dovrebbe avere un IP diverso. Questo è un approccio efficace ma non efficiente. I problemi sono introdotti da limitazioni sui processi o sul numero di IP. Dall'altro lato, per rendere accessibile un sito che ha molte richieste, si può effettuare il mirroring del sito, ma ciò comporta alcuni svantaggi, come la selezione di un particolare server (tediosa per l'utente), quindi deve essere trasparente. Avere nomi differenti per ogni replica consuma più hostname e si hanno più URL che si riferiscono allo stesso contenuto. Un approccio alternativo è quello in cui un nome può essere associato ad un IP che corrisponde ad un surrogato (proxy) che sceglie la replica da utilizzare per le richieste dell'utente.
Le sfide introdotte sono:

1. una sequenza di richieste da parte dello stesso utente dovrebbe essere gestita dalla stessa replica
2. il contenuto deve essere consistente su tutte le repliche
3. le politiche d'accesso alle varie repliche devono essere consistenti.

4.6 Apache Web Server

Apache 1.3.3 ha un architettura PD (differentemente dalla 2 che è multithread). File e memoria sono allocati in pool rilasciati automaticamente alla terminazione. Piuttosto che creare un processo per ogni connessione, preforka i figli alla partenza (proprietà configurabile tramite max_client). Individua il numero massimo di utenti che si possono gestire simultaneamente (configurabile in compile time).
I parametri min_spare_servers e max_spare_servers determina il num min e max di processi idle e ogni pochi secondi viene fatto il controllo.
Creando nuovi processi prima dell'avvio delle richieste, Apache assicura che non ci siano ritardi nella gestione dei processi in entrata.
Max_request_per_child identifica il numero di richieste http che un child può gestire. La configurazione di Apache è molto flessibile e può essere fatta in base al numero di richieste (se un server ma molti client con banda piccola, può metterci anche tanto tempo visto che i client devono aspettare).
Esistono altri parametri per la gestione delle connessioni per ogni processo figlio. Il padre ascolta le richieste dei client che vogliono stabilire nuove connessioni TCP. Se tutti i figli sono occupati il sistema mantiene una coda con le connessioni in attesa (configurabile). Il server ha un buffer di invio per memorizzare i dati di uscita per ogni connessione stabilita. Le dimensioni del buffer sono configurabili e servono per gestire il throughput tramite il *send_buffer_size*. Il numero di richieste http per singola connessione TCP è configurabile tramite *max_keep_alive_request* e il tempo in cui può restare idle è settato da *keep_alive_timeout*. Quando è raggiunto uno dei due limiti, il processo chiude la connessione.

Pool di risorse

Un web server consiste di moduli SW che consumano risorse del sistema come allocazione di memoria, accesso al FS, fork di processi… ed è difficile assicurare un utilizzo efficiente delle risorse. Apache WS utilizza l'astrazione tramite pool. Una struttura dati che traccia un gruppo di risorse del OS che sono create e distrutte assieme. L'astrazione del pool isola allocazione e deallocazione delle risorse in piccole parti del SW del server. Ciò protegge il server da errori di programmazione come il memory leak.
Altri moduli del server interagiscono con i pool attraverso API che includono funzioni di creazione distruzione e inizializzazione dei pool, funzioni di allocazione di nuove risorse per i pool e la gestione della costruzione degli header.

Processamento richieste HTTP

Apache processa richieste HTTP attraverso:

1. traduzione dell'URL in filename: dipende dalla configurazione del server e in Apache è permessa la specifica degli alias, che permettono di astrarre la locazione del file system dai link visibili sulla barra degli indirizzi, e possono essere utili anche per fare mapping multiplo. Apache ha un modulo per correggere gli errori di ortografia più comuni. Queste capacità aggiuntive introducono overhead ed in più una riposta basata sul matching parziale può esporre inavvertitamente un file. Esiste un modulo per riscrivere l'URL richiesto.
2. determinare se la richiesta è autorizzata. Le politiche di controllo d'accesso dipendono dalla configurazione del server. Una singola politica può essere applicata a più risorse e i file di configurazione consistono di direttive che si basano su: URL, filename, directory e virtual server. Le direttive possono essere specificate in un unico file, una per linea, dall'amministratore del sito web. Un unico file di configurazione ha i suoi svantaggi: riavvio del server dopo ogni modifica e crare tante politiche per tanti siti è oneroso per l'amministratore. Per ovviare a questi problemi si permette all'amministratore del sito di modificare i permessi d'accesso alle proprie cartelle e sottocartelle tramite dei file *.htaccess*. Dopo che l'URL è stato risolto come nome del file, il server inizia un processo di *directory-walk* per determinare le politiche d'accesso al file richiesto. Viene prima esplorato il file generale comune a tutti i siti e poi gli htaccess associati. È possibile aggiungere più direttive d'accesso ad ogni passo dell'algoritmo e possibile limitare la sovrascrittura delle policy.
3. generare e trasmettere la risposta. Come parte del parsing di una richiesta http il server crea e popola una struttura detta *request_record* che è utilizzata da diversi moduli software. Esso contiene informazioni che si basano sulla richiesta (URL, versione protocollo HTTP, codifica, contenuto). Il record ha un puntatore al pool che è stato allocato per gestire la richiesta, che viene ripulito quando la richiesta è stata completata. Un'altra porzione del request record è utilizzata appena il server genera la risposta. Apache ha una collezione di handler che forniscono un'azione sul file richiesto. Il gestore è assegnato in base al tipo del file, e quello di default invia il file al client inserendo l'header aggiornato. La configurazione del server determina quali metadati sono inseriti nell'header della risposta. Il WS associa particolari attributi della risorsa all'estensione dei file. Il server può avere un file di configurazione che definisce il tipo MIME per ogni estensione del file. L'interpretazione della risposta dal lato client non dipende solo dal formato del file ma anche della codifica utilizzata (informazioni presenti nell'header).

6 HTTP 1.0

Proprietà del protocollo HTTP:

1. *Meccanismo di naming per URI fidati*: permette alle risorse di risiedere in una qualsiasi parte di Internet. Una risorsa può avere la stessa URI per sempre, anche se i contenuti interni cambiano. Un'URI è interpretata come una stringa formattata. Un URI contiene URN e URL e inizia con uno schema ed è seguita con una stringa che rappresenta la risorsa ottenibile tramite lo schema. Lo schema è http. Un URI relativo non inizia con uno schema. Visto che la stessa risorsa può essere acceduta tramite diversi protocolli, lo schema è separato dalla sintassi interna del protocollo in modo tale che il meccanismo di naming del web sia in grado di funzionare su tutti i sistemi.
2. *Scambio richiesta/risposta*: HTTP specifica la sintassi e la semantica con la quale i componenti web devono interagire. Ogni messaggio HTTP è strutturato in gruppi di 8 unità in una sintassi specifica. HTTP specifica un insieme di metodi di richiesta estendibili per l'accesso come richiesta, creazione, modifica, cancellazione della risorsa.
   Un messaggio di risposta è formato da:
   - *Codice risposta*
   - *Header* opzionale (non tutti i campi nell'header sono di natura informativa. Alcuni influenzano la generazione della risposta e vengono detti "modificatori di richiesta")
   - *Body*
3. *Protocollo stateless*: ciò implica l'impossibilità di conservare uno stato tra diverse req/res HTTP. Se HTTP non fosse stato stateless, si sarebbe persa la proprietà di scalabilità.
4. *Metadati delle risorse*: sono legati alla risorsa ma non ne fanno parte. Spesso vengono inclusi negli header e le loro funzionalità sono:
   - Svelare informazioni riguardanti l'encoding
   - Permettere di fare il check tra la dimensione specificata e quella di arrivo
   - Danno direttive per il caching

Influenza dei protocolli

– MIME in HTTP

MIME può essere utilizzato per rappresentare un set di caratteri diverso da ASCII. MIME definisce un insieme di oggetti multimediali creando un modo per definire nuovi dati ed ha 2 funzionalità:

1. Classificazione dei formati di dati (*mimetype* con estensione)
2. definizione di un formato per messaggi *multipart* (più entità in un unico body)

Cose di MIME non inglobate in HTTP:

1. meccanismo di Markup simile a SGML (da cui deriva html)
2. metodo di indirizzamento a documenti esterni (http usa url)

Differenze tra MIME e HTTP:

- MIME è utilizzato per scambio mail, HTTP ha alte performance per scambio dati binari.
- L'uso e l'interpretazione dell'header è differente ed è usato differentemente
- In html non ci sono limitazioni al numero di linee di codice in header o body, mentre in MIME il limite è fissato
- In MIME un entità è qualcosa all'interno della mail e ha solo due tipi di dati: *messaggio* e *entity*.
- MIME non è in grado di conoscere il mittente.
- HTML non riesce ad utilizzare la stessa nozione di entity di MIME perché è andato in contro a molte difficoltà.

HTTP vs (ARCHIE, GOOPHER e WAIS)

HTTP è stato influenzato molto dai protocolli già esistenti

- GOOPHER era stateless ma non utilizzava menù, file, testo come hyperlink a differenza di HTML. HTTP è più dinamico ed estendibile.

HTTP e altri protocolli di livello applicativo.

http ha preso il meccanismo di risposta da SMTP, in modo da classificare le risposte in pochi gruppi (function group).

Elementi del linguaggio - termini di HTTP

1. *messaggio*: una sequenza di byte inviate tramite protocollo di trasporto
    a. *request*:
        i. *request line*: metodo, versione protocollo e risorsa richiesta
        ii. *general header*: contiene la data ed un'azione da intraprendere (pragma)
        iii. *request headers*: specificano il client che ha fatto la richiesta da un user agent
        iv. *entity headers*: lunghezza del contenuto e permessi
        v. *CRLF*
        vi. *entity body*: stringa opzionale
    b. *response*:
        i. *status line*: versione protocollo e codice di risposta
        ii. *general header*: contiene la data ed un'azione da intraprendere (pragma)
        iii. *response header*: tipo di server
        iv. *entity header*
        v. *CRLF*
        vi. *entity body*: sempre opzionale
2. *entità*: rappresentazione di una risorsa in un messaggio di req/res strutturato tramite:
    a. *entity headers*: metadati dell'entità richiesta
    b. *entity body*: presente nel response messagge; è il contenuto della risposta senza il response header
3. *risorsa*: una risorsa è un oggetto/servizio sulla rete identificabile tramite URI
4. *user_agent*: il client che inizia la richiesta, che può essere browser, spider, ecc. La distinzione tra user_agent e client generico è importante, poiché l'user_agent è direttamente connesso all'utente. L'user_agent fa parte dei request_header e contiene info su browser e OS.

Metodi di HTTP

I metodi di *request* sono: GET, HEAD, POST, ma i client e server implementano anche altri metodi come PUT, DELETE, LINK, UNLINK.
I metodi sono inclusi nella request_line del request header e sono usati dal server per generare la risposta. Tali metodi non sono applicati dagli intermediari come i proxy e possono godere di due proprietà: *safety* se esamina la risorsa senza alterarla, *idempotenza* se l'esecuzione multipla della stessa richiesta genera sempre la stessa risposta.

1. GET: è applicato sulla risorsa specificata nell'URI e la risposta generata è il valore della risosra. GET è sicuro ed idempotente, può includere richieste da parte dell'utente. Una richiesta GET che include una request_modifier è in grado di compiere diverse azioni. Non ha il request body e se c'è viene ignorato.
2. HEAD: è stato introdotto per ottenere i metadati di una risorsa e non restituisce alcun response body. I metadati possono anche essere memorizzati in cache.
3. POST: è usato principalmente per aggiornare risorse esistenti o per fornire input ad un processo di gestione dati. Può alterare la risorsa e quindi non è sicuro. Il side effect del POST non è sempre lo stesso per ogni invocazione, quindi non è neanche idempotente.

Per modificare le risorse esistenti l'utente di deve autenticare e deve avere permessi.
POST può essere utilizzato per inserire il body come input di un programma identificato dal request-URI che genera output all'interno dell'entity-body della risposta. È importante che il metodo post abbia il content length nel request-header in modo che il server possa essere sicuro che la richiesta ricevuta sia completa. La differenza principale tra GET e POST è che mentre GET codifica l'input nella request URI, POST lo codifica nel body dell'entity. Per tale motivo è preferibile utilizzare il metodo POST anche perché spesso proxy intermediari e server limitano la lunghezza della request-URI.

4. PUT: è simile a POST, visto che può alterare il formato di una risorsa. Se l'URI non esiste viene creata, altrimenti modificata. È utilizzato per cambiare l'identificatore della risorsa. PUT non è sicuro ma è idempotente.
5. DELETE: cancella una risorsa identificata da una request-URI ma, prima di eseguirla, il server controlla che possa essere applicata. Esistono due tipi di risposta per DELETE:
    a. La richiesta sarà processata più tardi
    b. La richiesta è stata completata

   DELETE è safe e idempotente.
6. LINK e UNLINK: LINK permette la creazione di LINK tra request-URI e altre risorse, mentre UNLINK toglie il link. Sono stati eliminati in HTTP 1.1.

Header di HTTP 1.0

Gli header giocano un ruolo fondamentale per la gestione e alterazione delle risorse nei messaggi di request o response. Possono inoltre essere utilizzati per fornire metadati su risorse come lunghezza, encoding, ecc.
Ogni protocollo a livello più basso ha un header e a differenza di IP e TCP, HTTP ha headers di lunghezza variabile, in modo da permettere una più flessibile rappresentazione dei formati.
C'è la possibilità di aggiungere sempre nuovi header. Le implementazioni esistenti possono cooperare oppure decidere di ignorare gli header che non conoscono. La maggior parte degli header sono opzionali e alcuni di essi possono essere ignorati e diversi tipi di header possono riferirsi ad intermediari, o al client o al server.

Formato e sintassi degli header

Sintassi generica dell'header:

<nome>:<valore*1*>,<valore*2*>,…,<valore*n*>

Gerarchia degli header

1. *general*: usati in req e res
2. *request*: solo nella request e includono info esprimendo preferenze sulla natura della risposta o vincoli sulla gestione della richiesta
3. *response*: solo nella response per aggiungere info sulla risposta o per info richieste dall'utente
4. *entity*: in req e res info come il last_modification_time.

GENERAL HEADERS

1. `data`: tre tipi di rappresentazione della data attuale e tutti riconoscibili da client e server anche perché alcuni senders potrebbero essere applicazioni non HTTP
2. `pragma`: consente di dare direttive da inviare al ricevente del messaggio. È un campo opzionale e la sola direttiva definita è "pragma: no chache" che informa i proxy di non utilizzare la copia in cache per le risposte.

REQUEST HEADERS
Specificano i vincoli sulla risposta e sono di 5 tipi:

1. `authorization`: include credenziali dell'user_agent per la risorsa. Il valore basic si riferisce ad uno schema che utilizza user_id e passwd codificati in Base64 e presenti nella linea successiva a quella della definizione di basic.
2. `from`: è l'indirizzo email e serve ad identificare il responsabile del programma. Il suo uso è sconsigliato.
3. `if_modified_since`: è un esempio di header condizionale e indica che la richiesta può essere gestita in un modo diverso specificato nell'header. Se il last_modification_date (RESPONSE) è precedente (più fresca) all'if_modified_since allora il server deve rimandare la risorsa, altrimenti si usa la copia in cache.
4. `referer`: è usato dal client per includere l'URI da cui è stata richiesta la risorsa, in pratica rappresenta l'indirizzo del link uilizzato per accedere alla risorsa. Questo header crea problemi di sicurezza, in quanto usato dai server per tracciare il pattern di un utente. Un server può usare referer per negare l'accesso alle risorse da pagine che non sono sotto il suo controllo.
5. `user_agent`: include info sulla versione del browser, sul OS e può essere utilizzato dal server per rispondere adeguatamente alle capacità del browser.

RESPONSE HEADERS
La sintassi della status line è fissa e i RESPONSE headers non possono aggiungere info addizionali. Se un response header non è riconosciuto assume il ruolo di entity header:

1. `location`: utilizzato per smistare la richiesta dove la risorsa può essere trovata. È utile per la ridirezione. È un modo per identificare una delle risorse replicate in più formati. Se una risorsa è stata creata come risultato di una richiesta, allora l'header location identifica la risorsa appena creata.
2. `server`: è il corrispondente dell'user_agent, il valore può essere utilizzato a scopo di una migliore conoscenza dell'ambiente dove si possono avere errori, ma può essere utilizzato anche in modo malizioso andando ad agire sulla vulnerabilità di quel tipo di server. È un header opzionale.
3. `www-authenticate`: usato per l'accesso a risorse che necessitano di autorizzazione. Se il client non è autorizzato è restituito "401 – Unauthorized".

ENTITY HEADERS
È usato per includere informazioni sul corpo dell'entità o sulla risorsa nel caso in cui non ci sia l'entity body. Non è una proprietà ma deve essere richiesta o inviata. Inoltre tutti gli header che non sono riconosciuti vengono interpretati come entity header. In tal modo si definisce una gerarchia tra gli header ed è possibile sempre aggiungerne di nuovi.
HTTP 1.0 definisce 6 tipi di entity header:

1. `allow`: indica la lista di metodi che possono essere applicati alla risorsa. Se il metodo richiesto non è "allow" il server risponde con la lista dei metodi "allow".
2. `content_type`: indica il tipo di file multimediale nell'entity body.
3. `content_encoding`: indica la docifica applicata alla risorsa e come si può decodificare per ottenere il formato descritto nel content_type (di solito è usato xgzip).
4. `content_length`: indica la lunghezza dell'entity body in byte ed è usato per controllare l'integrità della risorsa. Può essere usato come validatore per comparare la copia in cache con quella corrente. Senza content_length il client dovrebbe chiudere la connessione per indicare la terminazione della richiesta. In caso di risposta generata dinamicamente il content_length non è conosciuto, ma il suo inserimento comporta overhead di calcolo e quindi si omette. È possibile inserirlo solo se le risposte dinamiche vengono prima bufferizzate.

5. `expires`: serve al mittente per capire se l'entità può essere considerata obsoleta dopo il tempo specificato nell'header. Un client non può cacheare una risposta una volta superato l'expiration time. Una risposta può essere messa in cache dopo il tempo dell'header, ma richiere necessariamente la validazione da parte dell'origin server. È di tipi entity perché si riferisce all'entità e non al messaggio.
6. `last_modified`: indica la data dell'ultima modifica. Se la risposta è dinamica segna il tempo di quando è stata generata dal server. È utilizzata per fare confronti in cache.

HTTP 1.0 – Classi di risposta

Ogni response message inizia con una status_line che ha tre campi:

1. versione protocollo del server
2. codice di risposta
3. descrizione in linguaggio naturale

I vari tipi di risposta sono raggruppati a classi. Ci sono 5 classi di risposta, ed ogni classe ha 3 cifre intere:

1xx Informativo
2xx Successo
3xx Redirezione
4xx Client error
5xx Server error

L'idea dell'utilizzo di classi di risposta è stata presa da SMTP. In ogni caso tra SMTP e HTTP ci sono differenza tra i codici. Infatti in SMTP le tre cifre rappresentano un livello di descrizione sempre più granulare e dettagliato, mentre in HTTP seconda e terza cifra non rappresentano categorie. In oltre la descrizione è customizzabile. Non tutti i codici di errore devono essere interpretati da tutte le applicazioni ma deve essere interpretabile la loro classe di appartenenza. Infatti se un codice xxx non è compreso da un'applicazione allora viene utilizzato x00.

1xx - Informational class

È una classe informativa che non è utilizzata in HTTP 1.0 e può essere customizzata per applicazioni sperimentali.

2xx - Success class

Serve ad indicare che il server ha compreso la richiesta e sa come gestirla.

200 – OK

generico, per quando la richiesta ha avuto successo. È usato anche per HEAD

201 – Created

Una risorsa è stata creata con successo dopo una POST. Può essere usato anche dopo PUT se una risorsa non esisteva ed è stata creata.

202 – Accepted

Richiesta ricevuta ma non ancora completata, serve a far continuare il proprio lavoro all'user_agent anche se non è ancora stata portata a termine la richiesta.

204 – No content

Indica che l'evento non può essere ancora gestito dal server.

3xx - Redirection class

Sono usati per informare l'user_agent che deve essere compiuta un'altra azione prima che la richiesta possa essere completata. Il numero di redirezioni è limitato in modo da evitare loop ed attualmente è scelto dal server.

300 – Multiple choice

Ad esempio se la risorsa è disponibile in più locazioni, l'user_agent può cercarla direttamente da un'altra locazione.

301 – Moved permanently

La risorsa è stata spostata permanentemente. Utile per GET e HEAD che vengono inoltrate direttamente alla nuova locazione. Con POST non funziona allo stesso modo poiché è richiesta una conferma diretta da parte dell'utente, in quanto non è un metodo "sicuro".

302 – Moved temporarily

L'unica differenza con 301 è che la risposta non può essere messa in cache.

304 – Not modified

La risorsa non è stata modificata.

4xx - Client error class

Identifica gli errori che sono generati da errate richieste client.

400 – Bad request

La sintassi della richiesta o è scorretta, o irriconoscibile.

401 – Unauthorized

Se la richiesta non include le autorizzazioni necessarie per l'accesso alla risorsa.

403 – Forbidden

Quando il server non accetta la richiesta. In tal caso il serve comprende la richiesta ma deliberatamente non la esegue. Il motivo può essere incluso nell'entity body.

404 – File not found

Quando il server non trova la risorsa richiesta, ma non esiste un modo per sapere se lo diverrà in futuro.

5xx - Server error class

Per errori legati al server o quando il server sa di non poter eseguire una richiesta. La differenza con 4xx è che l'errore è nel server e non è possibile risolverlo.

500 – Internal server error

Errore generico. Neanche il server è in grado di comprenderlo.

501 – Not implemented

La risorsa richiesta non è implementata e non può essere gestita. Ad esempio il server riceve una richiesta per un metodo definito in una vecchia versione del protocollo.

502 – Bad gateway

Quando un server che funziona da proxy o da gateway è incapace di processare la richiesta da parte di un altro server. Serve ad indicare che il server che ha risposto non è responsabile dell'errore.

503 – Service unavalaible

Quando un server è temporaneamente non disponibile ma crede di poter gestire la richiesta in un secondo momento (per esempio quando è temporaneamente occupato).

Estendibilità di HTTP

Uno dei principi iniziali nel design di HTTP è stata la nozione di estendibilità, infatti non c'è una lista di dimensione fissata per quanto riguarda i metodi o le classi di risposta. Questo aiuta le nuove applicazioni nate dopo il design di HTTP 1.0.

Sicurezza

HTTP 1.0 ha un suo meccanismo di sicurezza. Però piuttosto che dipendere dal protocollo HTTP o da altre applicazioni WEB è stata creata sicurezza a livello applicazione. Questo è il meccanismo di SSL (*Secure Socket Layer*).

HTTPS: scambi web con protocollo SSL

HTTPS utilizza SSL per i messaggi HTTP. Tramite questo meccanismo, la request, la response e la request-URI sono cifrate al fine che i packet monitor non riescano a interopretare a quali risorse si vuole accedere. HTTPS gira sulla porta 443 di TCP ed è del tutto trasparente all'utente se non per lo schema utilizzato nell'address bar.
Con HTTPS non si perdono le info quando si accede ad altre risorse che non fanno uso di SSL (senza handshake). Inoltre HTTP 1.0 fornisce un suo meccanismo di sicurezza basato sull'autenticazione. Infatti un client che vuole accedere ad una risorsa protetta deve rispondere con un'insieme di credenziali inserite in un'authorization header. Se le credenziali non sono appropriate, allora il server risponderà con 401 o 403. La pecca di questo sistema di autenticazione si trova nel fatto che l'informazioni sono inviate in chiaro.
In HTTP 1.1 la gestione della sicurezza è migliore.

Interoperabilità e compatibilità del protocollo

L'interoperabilità è una delle nozioni primarie da tener conto durante la specifica di un protocollo. L'utilizzo di nuove versioni deve sempre implicare la compatibilità con le vecchie versioni, e questo comporta che vecchie e nuove versioni utilizzino stesse regole sintattiche.

7 HTTP 1.1

I problemi riscontrati in HTTP 1.0 sono:

- mancanza controllo sulla cache;
- incapacità di scaricare solo porzioni del file richiesto e incapacità di continuare i download interrotti;
- uso povero di TCP;
- assenza di garanzia sulla ricezione delle risposte dinamiche;
- sfruttamento esagerato degli indirizzi IP;
- livelli poveri di sicurezza;
- alcuni problemi sui metodi;
- ambiguità di regole per la gestione di cache di proxy.

Concetti nuovi in HTTP 1.1

Meccanismo hop-by-hop

A causa della presenza di cache-proxy il meccanismo di transazione end-to-end, per questo sono inseriti headers hop_by_hop che servono per assicurare un invio sicuro tramite gli intermediari scelti.
Ad esempio, l'header `transfer_encoding` abilita la compressione hop-by-hop. Questi header non possono essere memorizzati in cache o inoltrati da proxy. Il tag `connection`, di tipo hop-by-hop, viene modificato ad ogni intermediario, e rappresenta una lista di header HBH. Ad ogni intermediaro si tolgono i vecchi header hop-by-hop e aggiunti i nuovi. Questi tag però non possono essere inoltrati direttamente. Questo meccanismo serve ad analizzare il traffico sulla rete, e se tra gli intermediari c'è qualcuno che non utilizza il protocollo HTTP 1.1 allora si usa la versione precedente, in modo che la connessione hop-by-hop non utilizzi HTTP 1.1.

Trasnfer-coding

In HTTP 1.1 si distingue chiaramente tra messaggio ed entità. Infatti il messaggio rappresenta l'unità di comunicazione su protocollo HTTP con header e body, mentre l'entità è il contenuto di un messaggio diviso in entity header ed entity body.

Il valore di transfer-coding di distingue dal valore content-coding di HTTP 1.0 in quanto ora in HTTP 1.1 c'è la distinzione tra codifica applicata all'entity e codifica applicata all'entity body.
Sono stati introdotti due tipi di header:

- `TE` request header che indica quali transfer-coding comprende il mittente
- `Transfer_encoding` general header che indica quale codifica può essere applicata al messaggio.

Entrambi sono di tipo hop-by-hop.

7.2 Metodi, header e codici di risposta

7.2.1 Metodi vecchi e nuovi

PUT e DELETE sono stati mantenuti, di cui è strata fornita anche una specifica formale.
Sono invariati GET, HEAD e POST e sono stati aggiunti tre nuovi metodi: OPTION, TRACE, CONNECT. Inoltre è stato inserito il concetto di metodo di livello *MUST*, ossia metodi che devono necessariamente essere implementati. GET e HEAD sono di livello must.

7.2.2 Headers vecchi e nuovi

GENERAL HEADERS: passano da 2 a 9. `Data e pragma` non hanno subito modifiche.
REQUEST HEADERS: passano da 5 a 19, in 4 classi diverse, che sono:

1. response preference
2. information sent with request
3. conditional request
4. constraint on server

RESPONSE HEADER: passano da 3 a 9 in 4 classi, che sono:

1. redirection
2. information related
3. security reladed
4. caching related

ENTITY HEADERS: mantengono la stessa semantica di HTTP 1.0 e passano da 6 a 10.
HOP-BY-HOP HEADERS: da 0 a 8. I vecchi headers end-to-end non possono essere utilizzati in quelli hop-by-hop.

7.2.3 Classi di risposta vecchie e nuove

Passano da 16 a 41. non sono state definite nuove classi, ma solo ampliate le vecchie.
Classe 1xx: mentre era usata poco in HTTP 1.0 perché generale, in HTTP 1.1 sono stati introdotti il *100 – Continue* e il *101 – switching protocols*.
Classe 2xx: i codici per la classe di successo passano a 7.
Classe 3xx: viene utilizzata dal client per cercare un modo per completare la richiesta. Alcuni codici della versione 1.0 hanno cambiato significato ed in tutto sono diventati 8.
Classe 4xx: sono diventati 14 codici, divisi in 5 sottoclassi:

1. HTTP 1.0 error code (400, 401, 403, 404)
2. Motivazione dell'errore di tipo 1 (405, 407, 408, 410)
3. Capacità di negoziazione in HTTP 1.1 (406, 415, 413)
4. legate alla lunghezza (411, 414)
5. nuovi codici per altre funzionalità (402, 409, 412, 416, 417)

Classe 5xx: solo 2 codici nuovi aggiunti (*502 – gateway timeout, 505 – HTTP version protocol not supported*).

7.3 Caching

7.3.1 Termini legati al caching

Una risposta ottenuta da un server può essere messa in cache per un certo periodo di tempo. La response cacheata sarà restituita alla prossima richiesta inerente alla stessa risorsa. I termini legati al caching sono:

- *age*: tempo trascorso da quando l'entità è stata inviata dal server o rivalidata nella cache;
- *expiration time*: quanto tempo la risorsa può restare in cache prima di dover essere per forza rivalidata;
- *freshness/staleness lifetime*: il server deve decidere se una risposta è fresh o stale a seconda dell'age o dell'expiration time;
- *validità*: per sapere se una copia è stale o fresh;
- *cacheability*: una risposta è cacheabile se rispetta alcuni vincoli ad essa legati, come ad esemio l'expiration time;
- *cache manteinance*: una risposta cacheabile è memorizzata in cache e restituita più tardi. La gesione delle richieste in cache necessita della gestione di problemi come cacheabilità di una risposta, durata di cacheabilità, decisioni sulla rivalidazione di una risorsa.

7.3.2 Caching in HTTP 1.0

La necessità di fare caching è legata alla bassa velocità di connessione e alla possibilità di rendere la stessa risorsa disponibile a diverse richieste dello stesso tipo. In HTTP 1.0 sono presenti 3 controlli sulla cache, basati sugli header:

- direttive sulla richiesta: `pragma: no cache`
- modificatore per la richiesta get: `if-modified-since`
- response header: `expires`

Se la copia in cache è fresh allora il server può restituire un codice di risposta *304 – not modified* senza body.
Alcune implementazioni di HTTP 1.0 utilizzano male l'header expires, forzando la morte immediata della risorsa. Questo fenomeno è detto *cache busting* ed è usato dal server quando crede che la risorsa sia stale. Se il client ha fatto una richiesta di una risorsa (con una corretta locazione) ed essa è presente in cache (*cache-hit*) il server di origine non è in grado di sapere che la richiesta è stata servita dal proxy e allora deve inviare al proxi le modifiche (se ci son state).

7.3.3 Caching in HTTP 1.1

Per fare in modo che una cache restituisca il valore della response assicurandone la consistenza con la copia sul server HTTP 1.1 segue queste line guida:

1. il ruolo del protocollo è assicurare la correttezza senza preoccuparsi di quanto tempo possa essere cacheato, dove e come debba essere sostituito il contenuto;
2. assicurare correttezza, anche se comporta costo: una cache non può restituire inconsapevolmente un valore vecchio;
3. dare al server più informazioni sulla cacheabilità di una risorsa;
4. non deve dipendere da timestamp assoluti o da sincronizzazioni tra client e server
5. deve fornire un supporto di negoziazione del tipo di risposta cacheata.

In HTTP 1.0 era già permesso usare la cache con *expires* e così è stato possibile aggiungere nuovi header per dare più controllo sulla cache. La combinazione di tutti gli header possibili è esponenziali, per questo è stato introdotta una linea guida: le componenti dovrebbero essere propense ad accettare nuove richieste e dovrebbero cercare di inviare quanti meno dati possibili (ricevere tutto ma inviare solo in modo significativo).
Esistono 4 header in HTTP 1.1 relativi alla cache, e sono:

1. *age*: il server lo usa per indicare quanto tempo fa è stata generata la risposta sul server. Se il server in esame è un proxy cache allora il valore di age è il valore dell'ultima rivalidazione.

I proxy sono obbligati a generare questo header ed è espresso in secondi come numero non negativo.

2. *Cache control header*: HTTP 1.1 fornisce più controllo sulla cache, sia per chi invia che per chi riceve. Sono specificate diverse direttive attraverso questo header e sono anche estensibili. Le direttive possono essere estese a tutta la catena di connessioni a livello hop-by-hop e sono di livello *must*, quindi la direttiva (header) deve essere implementata per tutta la catena. Anche se req e res hanno stesse direttive, non c'è un modo per cui req e res se ne accorgano, e quindi sono trattate separatamente.
   Le direttive applicabili sulla richiesta si dividono in:
   - *no-cache*: è come pragma: no-cache e dice al server che non possono essere utilizzate risposte cacheate, è detto end-to-end reload
   - *only-if-cached*: si vuole una risposta solo se presente in cache
   - *no-store*: fa in modo che né la richiesta né la risposta siano cacheate. Si può ottenere un po' di privacy.
   - *max-age*: serve a decidere se una risposta in cache è stale e può modificare l'expiration settato di default. Se max-age=0 allora viene forzata la rivalidazione end-to-end.
   - *max-stale*: è il limite temporale massimo con cui il client accetta risposte vecchie
   - *min-fresh*: è il limite temporale minimo con cui il client accetta risposte nuove
   - *no-transform*: specifica che la cache non deve modificare in alcun modo il contenuto. Potrebbe capitare che siano applicate delle conversioni per migliorare il throughput. No-transform da il controllo completo al mittente
   - *extention-token*: permette di aggiungere nuove direttive.

   Direttive applicabili sulla riposta: assicurano al server che la cache utilizzata lungo una catena segua determinate regole. Direttive
   - *Public*: indica che la response debba essere cacheata
   - *Private*: può essere cacheata ma per un singolo utente
   - *No-store*: non deve essere cacheata
   - *No-cache*: costringe la rilavidazione ad ogni richiesta
   - *No-transform*: non fa alterare la risposta
   - *Must revalidate*: deve essere rivalidata ogni volta che è stale. Se il proxy non può rivalidare con l'origin server, restituisce un errore invece di una resp stale. La differenza con no-cache sta nel fatto che in must-revalidate la risorsa può esere cacheata
   - *Proxy revalidation*: meno restrittiva dal must revalidate e si applica a proxy o gateway per cache-user-agent solo condivise
   - *Max-age*: indica l'expiration-time di un entità
   - *S_max_age*: si applica solo alle cache condivise e il suo valore sovrascrive quello di expires e max-age
   - *Extention token:* permette di aggiungere nuove direttive
3. *Entity tag (ETag) response headers*: fu introdotto per confrontare le risorse cacheate con le nuove versioni. Una nuova versione di un'entità ha anche un diverso entity-tag, è specifico per una singola risorsa e non è utilizzabile per fare confronti tra risorse diverse. Si può utilizzare con i meccanismi di: *if-match* e *if-no-match* che sono tag condizionali.
   Strong e weak entity tag: strong fa il confronto byte a byte, weak fa il confronto semantico.
4. *Vary response header*: quando una risorsa è cacheata si usa l'URI come chiave dell'entity in cache (HTTP 1.0). Ma visto che una risorsa può avere più rappresentazioni, allora l'URI non può essere utilizzata come chiave ed è stato introdotto il response header *vary* che viene utilizzato per selezionare la variante appropriata dell'oggetto

7.4 Ottimizzazione della bandwidth

Motivi dell'incremento dell'uso di banda:

- risorse più grandi
- numero di immagini incluse in un documento
- più utenti con migliore connessione
- browser con connessioni multiple.

Interventi previsti per l'ottimizzazione:

- trasmissione solo delle parti del documento che cambiano (range request)
- scambio di info di controllo per eliminare trasmissione di bit non necessari (meccanismo di *expect-continue*)
- operazioni di compressione e decompressione prima di mandare e dopo aver ricevuto.

7.4.1 Range request

In pratica ci sono molti utenti con una bassa connessione e quindi capita che se la pagina ci mette molto a caricarsi allora l'utente abortisce la connessione e fa il reload (quindi sarà richiesta di nuovo tutta la pagina). Si deve evitare che si ricarichi tutto e dopo che la connessione è stata interrotta si riprenda. Inoltre capita che i client siano più interessati a zone specifiche del documento. Tutto ciò è possibile farlo tramite l'entity header *content-range*. Questo header aiuta nel posizionare porzioni dell'entity body nell'intero entity. Il client si accorge che non è stato completato il download della risposta perché il server invia il codice d'errore di risposta *206 – Download parziale*. Quindi il client sarà in grado di richiedere la risorsa dal punto in cui si è fermato fino alla fine, oppure effettuare una richiesta per uno o più porzioni (range-multirange). Quando si usa multirange nella risposta del server sarà utilizzata la stringa di separazione "-- rope --" per indicare le parti mancanti. Ovviamente se i range della richiesta non sono soddisfacibili allora sarà restituito il codice *416 – requested range not satisfiable*. Se invece è soddifsacibile solo un certo range, allora viene introdotto il response header *accept-ranges* che se ha valore *none* è simile al 416, ad eccezione che inserisce il codice *200 – ok* con accept range = none.
Per combinare le richieste di range con le condizioni, si introduce l'header *if-range* che permette al server di inviare al client la risposta solo se cambiata in quel range. Viene inoltre introdotto l'header *if-unmodified-since* che chiede una porzione della risorsa solo se non cambiata da una certa data, altrimenti si invia l'header *last-modified* con l'intera risorsa.

7.4.2 Meccanismo expect-continue

È utile conoscere lo stato del server, ossia sapere se può gestire le richieste di un client prima che inizi ad inviarle. I server hanno un proprio limite anche se il protocollo non lo definisce.
L'expect-continue informa il client sullo stato del server e sulla sua predisposizione a gestire le richieste. Infatti se il server è disponibile gli risponderà *100 – continue*. Altrimenti *413 – request entity too large* oppure *403 – forbidden* oppure *417 – expectation failed*.
Il meccanismo di expect è hop-by-hop e si riferisce di volta in volta solo al prossimo hop. L'expect-header è end-to-end.

7.4.3 Compression

Due header che sono il TE (il transfer encoding per indicare la compressione) e l'accept-encoding (per dire che il server accetta la compressione) in HTTP 1.1 è un meccanismo hop-by-hop che indica le trasformazioni applicabili all'entity body.

7.5 Gestione della connessione

HTTP utilizza TCP che non è ottimizzato per connessioni di breve durata comuni nello scambio HTTP. Infatti TCP utilizza un handshake a 3 vie più un quarto messaggio per la chiusura della connessione. In tal modo per una tipica richiesta HTTP di 10 pacchetti, 7 su 17 sono di controllo.

Una proposta per evitare questo problema è l'utilizzo di connessioni parallele. Con HTTP 1.0 per il download di una pagina sono eseguite una serie di richieste sequenziali, mentre con le connessioni parallele si fanno più richieste in contemporanea per gli elementi della pagina. Un'altra alternativa è quella delle connessioni persistenti che serve per evitare *setup* e *teardown* della connessione. Inoltre dando più tempo di vita alle connessioni esse sono in grado di individuare lo stato di congestione della rete. In poche parole l'idea è quella di utilizzare la stessa connessione e lasciarla attiva per più elementi. La gestione delle connessioni persistenti si sviluppa in 5 fasi:

1. keep alive
2. connessioni parallele (obiettivi delle connessioni persistenti)
3. connection header
4. pipeling e gestione della connessione durante il pipeling
5. decisioni da prendere prima di chiudere una connessione (lato c e lato s).

1. Meccanismo Keep-alive di HTTP 1.0

Il meccanismo di connessioni persistenti non è proprio di HTTP 1.0 ma alcuni browser hanno introdotto un nuovo header (di tipo request header) per soddisfare tali esigenza: `keep_alive`.
Il server lascia aperta la connessione e la vuole usare anche dopo aver inviato la risposta. Il client manda la richiesta con `connection: keep_alive`. Se il server è d'accordo risponde con *200 – OK* e *connection: keep-alive*. Con questo meccanismo il client non è in grado di sapere quando finisce la risposta, perché non viene chiusa la connessione. Una semplice estensione permette di gestire i tempi di chiusura (o quanto debba durare).

2. Evoluzione di HTTP 1.1 – connessioni persistenti

Esistono due ampi approcci:

a. nuovi metodi di HTTP che potrebbero essere usati per prelevare più di una risorsa (*variazione di GET)*
b. utilizzo di connessioni parallele per prelevare risorse

A. Variazioni di GET: MGET, GETLIST, GETALL (nessuna di queste tre è usata in 1.1)

MGET è simile al comando *mget* di FTP, usato per prelevare più file in una sola connessione FTP. HTTP propone di listare più risorse in una richiesta per evitare più connessioni TCP. La connessione dovrebbe essere gestita in modo seriale dal server e la risposta contiene tutte le risorse nella lista, rappresentate dalla loro URI e descrizione. Quando il browser trova una pagina con più immagini usa le info di layout presenti negli header per iniziare a fare il rendering della pagina. GETLIST e GETALL sono simili a MGET, solo che con GETLIST si specifica una lista di risorse, e GETALL effettua un'unica richiesta delle risorse incluse nel documento senza mandare tanti GET.

B. Approccio con connessioni simultanee-parallele

Un browser può aprire più connessioni in contemporanea e utilizzarle per scaricare le risorse in modo parallelo. Ovviamente il setup e teardown delle connessioni parallele introduce molto overhead sulla rete e se più utenti usano più connessioni parallele allora si può diminuire il throughput. Ovviamente se si abortisce la connessione allora saranno chiuse tutte.

Connessioni persistenti in HTTP 1.1

L'idea è di permettere a client e server di gestire le connessioni persistenti. Questa proposta può essere divisa in 2 parti:

1. riuso delle connessioni persistenti che ha 3 obiettivi:
    a. ridurre i costi delle connessioni TCP (minimizzando setup e teardown)
    b. ridurre la latenza evitando gli slow-start di TCP
    c. evitare spreco di banda e soprattutto congestioni

2. modifica a livello applicazione. Le modifiche da apportare sono piccole. Se richieste differenti sono inviate sulla stessa connessione persistenti senza attendere la risposta di ogni richiesta individuale si ha il fenomeno di pipeling.
   In HTTP 1.1 le connessioni sono trattate come persistenti per default, ma sono una specifica di livello *should*. GETALL e GETLIST hanno dato il loro proposal per essere realizzati come parte di HTTP 1.1. Le connessioni parallele sono poco comuni e si preferiscono le connessioni persistenti con pipeling. Un altro problema potrebbe essere la presenza di proxy o cache sul path.

3. Connection header

Alcune implementazioni di HTTP 1.0 usano gli header connection per dire che la connessione deve essere lasciata aperta. In HTTP 1.1 si vuole dare maggiore controllo sulle connessioni a client e server. L'header connection può anche indicare che una parte della richiesta vuole chiudere la connessione (`connection: close`). Questo header è stato introdotto per essere compatibile con gli altri header connection di HTTP 1.0. In HTTP 1.1 sono persistenti, e quindi è come se di default ci fosse sempre keep-alive. Problema: non tutti i proxy gestiscono le connessioni.

4. Pipeling su connessioni persistenti

L'idea è quella di utilizzare una sola connessione persistente per diminuire i tempi di setup e teardown, evitando di attendere la risposta per ogni richiesta prima di inviarne un'altra. Il pipeling è gestita dal server ma richiesta dal client. Si inviano le richieste senza attendere le singole risposte e le richieste vanno gestite in ordine, perché altrimenti il client non può ottenere il valore più recente della risorsa. Il protocollo suggerisce che solo i metodi idempotenti possano essere usati in sequenza.

Head of line blocking

Se la prima richiesta di una sequenza prende molto tempo, le altre accumulano ritardo; questo può essere alleviato se il proxy (il server contattato) smista le richieste a diversi origin server. Si deve comunque attendere che le richieste vengano risposte in ordine. Una soluzione sarebbe quella di mettere richieste in ordine temporale di calcolo (secondo la generazione della risposta), ma non è sempre possibile.

Chiusura inaspettata della connessione in uno stream di richieste in pipe

I motivi per cui una connessione può interrompersi sono diversi:

- Utente: Stop volontario
- Utente: Link ad un'atra pagina
- Server: che chiude pensando che ci sia un abuso delle risorse
- Rete: failure

Le connessioni persistenti, oltre a soffrire di tutti i problemi legati ad una caduta di connessione legata ad una singola richiesta aggiungono nuovi problemi: se viene abortita la richiesta l'utente può fare retry o ignorarlo. In situazioni di pipeling non ci sono info sullo stato tra le richieste e non sono introdotti problemi con i metodi sicuri come HEAD e GET, ma ce ne sono con PUT e POST, in quanto c'è possibilità di un cattivo aggiornamento sul server. Il protocollo specifica che il client dovrebbe essere in grado di fare recovery da qualsiasi situazione. La mancanza di comunicazione tra i livelli del protocollo fa sì che ci sia una gestione povera degli abort durante il pipeling.

5. Chiusura di connessioni persistenti

Il protocollo non specifica quanto debba durare la connessione e lascia la libertà di decidere a client e server purché una connessione sia chiusa al termine (quando non serve più). Devono essere risolti gli interessi competitivi tra client e server:

a. il server vorrebbe servire quanti più utenti possibili e le connessioni persistenti lo intralciano.

b. Il server potrebbe scegliere di gestire più richieste tramite la stessa connessione per lo stesso client perché esso stesso vorrebbe solo richiedere il rinnovo della connessione che è stata chiusa.
c. Un server potrebbe ristabilire in una finestra temporale che può essere occupata da una nuova connessione persistente per assicurare l'equità che si può basare sul tempo totale o Idle della connessione.
d. Un client potrebbe voler privileggiare client rispetto ad altri. Ad esempio se il client è un proxy potrebbe decidere di lasciare aperta la connessione per più tempo.

I problemi appena visti si aggiungono ad altri e trattano l'inserimento di parametri per connessioni persistenti tra cui:

- *timeout*: indica il tempo che deve trascorrere prima che la connessione sia chiusa
- *max*: numero massimo di richieste per connessione
- *state*: per indicare quali headers devono avere lo stato memorizzato

Questi parametri introducono dei problemi, infatti timeout e max possono introdurre dei tempi morti su una connessione che diventa Idle per un certo periodo di tempo, ad esempio quando il client non ha previsto bene i tempi necessati.
State è un parametro molto utile che viene usato per non riinviare gli headers come *accept*.
Purtroppo l'option-state introduce overhead significante e non c'è un modo per quantificare il vantaggio utilizzando state. Ogni server si basa su delle euristiche che tengano conto di diversi fattori per chiudere le connessioni. HTTP 1.1 richiede che ci sia il recovery da chiusure impreviste per alcuni casi come l'uso del metodo POST. Il client reinvierà la richiesta dopo un certo periodo di tempo e saranno gestiti gli errori 4xx o 5xx per chiusura parziale.

7.6 Trasmissione dei messaggi

L'obbiettivo è quello di assicurare che da entrambe le parti ci si accorga di aver ricevuto il messaggio per intero, senza alcuna perdita. Il trasporto sicuro attraverso la rete è essenziale per mantenere l'integrità delle transazioni. Ovviamente la lunghezza della risposta è utile per conoscere (al ricevente) quando la risposta sarà ottenuta per intero.
L'unico meccanismo in HTTP 1.0 è *content_length* che va bene per le risposte statiche. Nell'1.0 il server indica la fine di un contenuto dinamico chiudendo la connessione. Se questo è l'unico modo non è possibile gestire le connessioni persistenti. HTTP 1.1 dà un'alternativa introducendo un Trasnfer-coding detto *chunked* che permette al mittente di dividere il corpo del messaggio in record chunk spediti separatamente. Ogni chunk è preceduto dalla sua lunghezza per fare in modo che il receiver possa controllarne l'integrità. Alla fine del messaggio completo il server invia un chunk di lunghezza 0. Questa tecnica evita la necessità di utilizzo di grandi buffer, che può essere necessaria se il proxy sta memorizzando la risposta per un vecchio client HTTP 1.0. Inoltre la tecnica elimina i tempi di latenza per la generazione dell'intera risposta, che includono il calcolo e l'inserimento del content_length. Ovviamente request e response possono essere chunk. Il messaggio può essere seguito da una coda opzionale che è separato dal corpo della risposta, ed indica che il corpo è stato ricevuto per intero e la coda (trailer) può essere processata. Il trailer può includere solo entity-header. Nel trailer possono essere includi header opzionali come ad esempio per chiedere da parte del server se il proxy che ha ricevuto il messaggio ha inoltrato la richiesta. Se un messaggio chunked deve essere inoltrato ad un client HTTP 1.0, il proxy 1.1 deve parsare tutti i chunked in un buffer e creare il content_length valido.

7.7 Estendibilità

HTTP 1.1 deve assicurare che ogni cambio fatto sia compatibile con le implementazioni esistenti, per assicurare che browser o server compatibili con le vecchie versioni possano continuare a

lavorare con le nuove. In HTTP 1.1 sono state lasciate molte possibilità per future estensioni cercando di limitarne i problemi. Ci sono tre modi chiave per facilitare l'estendibiltà di HTTP:

1. introdurre metodi per riconoscere le capacità del server prima di effettuare richieste per imparare ciò che il server ha appena ricevuto
2. aggiungere nuovi header per conoscere i server intermedi lungo il path di una transazione web
3. aggiungere supporto per l'upgrade verso altri protocolli

7.7.1 Conoscere i server

Sono introdotti due nuovi metodi: OPTIONS e TRACE.
OPTIONS è introdotto per conoscere le capacità dei server e può considerare diverse situazioni:

1. volontà di conoscere se un server è in grado di gestire un metodo non di livello *must*
2. un client potrebbe voler conoscere se ci sono specifici requisiti necessari associati ad una risorsa
3. un user-agent vuole sapere che capacità hanno i proxy lungo il path
4. un proxy potrebbe voler indagare sull'upstream del server per sapere se il server può gestire comportamenti specifici di HTTP 1.1 come il meccanismo di expect-continue.

Per tutti questi motivi OPTIONS è diretto a qualsiasi server lungo il path ed è sicuro come GET ed HEAD. Ad esempio OPTIONS può essere usato per ottenere la lista dei metodi gestiti dal server (*allow*). Per i client che vogliono specificare il proxy in un path per una risposta, è introdotto un nuovo request header. Il client che richiede il *max-forward* request header (simile a TTL) causa che ogni proxy decrementi il valore fino a 0, dopo di ché viene riposto all'option request. In pratica il max-forward è stato creato sia per mirare ad un proxy, sia per determinare i loop nella catena dei proxy. Per questo è simile al TTL.
Sono disponibili ulteriori estensioni per questo tipo di metodo, come una risposta generica, in cui il corpo del messaggio può essere usato per estensioni future.
Per ottenere in modo non ambiguo le capacità del server è stato introdotto l'header `web-dav`.

TRACE introdotto in HTTP 1.1 permette al client di consegnare il contenuto del messaggio che è stato attualmente ricevuto. Diversamente da OPTIONS, TRACE vincola un'azione da parte del server che risponde con una copia del messaggio ricevuto. In pratica il request message ottenuto dal server è inserito nel response body. Questa tecnica è ispirata da *traceroute*.

7.7.2 Conoscere i server intermedi

HTTP 1.1 presta molta attenzione ai server intermediari, tramite il *via* general header. I server possono voler conoscere gli intermediari (parallelismo con traceroute). Tutti i proxy e gateway HTTP 1.1 sono obbligati a partecipare al meccanismo di via, identificandosi con hostname e numero di id del server, da cui è arrivata la richiesta o la risposta. Se si vuole beneficiare di privavy allora si può utilizzare uno pseudonimo inserito in append al *via* header. Non è difficile server HTTP 1.0 che non conoscono il via header. È di tipo end-to-end e tutti i proxy lo inoltrano.
L'utilizzo di via in combinazione con TRACE è usato per conoscere il path di un messaggio HTTP.

7.7.3 Upgrade ad altri protocolli

HTTP 1.1 ha un modo per far migrare le connessioni correnti verso quelle nuove. Infatti è stato introdotto l'header `upgrade` di tipi hop-by-hop utilizzato come segue:
si inserisce l'header *upgrade* e l'insieme dei protocolli supportati in modo che il ricevente possa passare ad uno di quelli presenti nella lista. La scelta del protocollo è indicata tramite *upgrade* e il codice *101 – switching protocol*. Essendo di tipo hop-by-hop, upgrade è tolto dal messaggio prima che venga inoltrato. Il server che riceve è libero di ignorare tale header.

7.8 Conservazione degli indirizzi

Internet è basato su ip v4. La popolarità del web ha introdotto l'outsourcing dei domini. Si pensa di passare a ip v6 perché gli indirizzi internet stanno finendo per cui si è pensato a varie tecniche per evitare che si consumino gli indirizzi attuali. Nella prima tecnica è quella di inserire nel messaggio http l'indirizzo del server passando

da GET /bar.html HTTP 1.0 a GET www.foo.com/bar.html HTTP 1.1

ma questo non è compatibile con la versione 1.0 di HTTP.
Per cui si è inserito un nuovo request header *Original -URI* che contiene il nome del server quindi si passa al msg

GET /bar.html HTTP 1.1
Host: www.foo.com

Tutte le richieste 1.1 devono avere questo campo e il numero di porta (che se omessa vale 80).

7.9 Negoziazione dei contenuti

Ci sono più rappresentazioni della stessa risorsa client e server devono mettono d'accordo su cosa scambiare. Il content negoziation può essere fatto dinamicamente infatti il client può ricercare le varie versioni presenti sul server e scegliere quella che preferisce.
Sono stati introdotti 5 Header in tutto: *accept*, *accept-char-set*, *accept-encoding*, *accept-language* (request header) e *content language* (responseheader).
Esistono due tipi di negoziazione:

1. Agent driver: il client riceve un insieme delle rappresentazioni alternative scegli la preferita e la richiede in una nuova request.
2. Server driver: il server sceglie la rappresentazione basandosi su ciò che è disponibile e sulle informazioni nei rquest header (unico meccanismo in HTTP 1.0)

Il server è libero di scegliere l'algoritmo per la generazione della lista dei formati e per la selezione. Se il formato richiesto dal client non è valido il server risponde con *406 – not acceptable*, introdotto per l'agent driver. Se il server ha una sola variante della risorsa la invia ignorando l'header. Esiste la risposta del server *300 – multiple choice* che indica al client la presenza di più varianti della risorsa anche su diverse locazioni. Un modo per specificare la preferenza di un formato è l'utilizzo del qvalue che è un valore tra 0.0 e 1.0.
La content negoziation è flessibile in modo tale che per ogni parametro specificato possa essere scelto un valore d'accettazione (di preferenza). Ovviamente l'uso di questo parametro aggiunge overhead, per questo va usato con cautela.
Nella negoziazione di risposte presenti in cache c'è bisogno di una gestione più attenta poiché una cache può rispondere con la locazione della richiesta, attuabile tramite gli header *vary* e *location*.
Una combinazione di agent e server driver CN detta *trasparent Content negoziation* che prevede una lista di varianti inviate dal server all'user agent che sceglie quella più appropriata. In questi casi l'user agnt non partecipa ogni volta alla scelta ma fa uso della cache lungo il path, in modo da alleggerire il server

7.10 Sicurezza autenticazione e integrità

Uno schema comune di autenticazione è il challenge response che prevede l'invio di una chllenge string dal server per una specifica risposta del client. La sfida è valida solo per uno specifico *realm* (regione di spazio in cui un utente può essere autenticato). Diversi server definiscono un *protection space* per una risorsa definendo in combinazione *realm* e *uri*; il vantaggio del protection space è che ci possono essere diversi schemi di autenticazione per diverse risorse.

7.10.1 Sicurezza ed autenticazione

HTTP 1.0 ha una minima nozione di sicurezza un client è abilitato a ricevere una risorsa dopo che il server ha verificato le credenziali (**basic autentication mecchanism**). Questo meccanismo non è

molto sicuro perché presuppone che la connessione tra client e server è fidata. È possibile cifrare le richieste infatti la password è cifrata in base 64.
HTTP 1.1 fa uso del **digest autentication** che permette di limitare l'uso delle credenziali ai singoli metodi e da un tempo limitato al calcolo del checksum, diversamente da http 1.0 in cui il tempo di vita è lungo e quindi abbassa la sicurezza sulle intercettazioni. In HTTP 1.1 il lifetime è stato accorciato restringendo le risposte ad una singola richiesta di un metodo. Ai client è richiesto di calcolare un checksum che coinvolge:
URI – metodo richiesto – username –password – valore nounce (in base 16 o 64 unico per ogni richiesta)
Se questo tipo di credenziali non sono accettate allo viene restituito il codice *401 – unauthorized*

7.10.2 Integrità

Il concetto chiave sull'integrità riguarda il contenuto che deve essere ricevuto intatto. È stato introdotto l'entity header *content-MD5* che fornisce un controllo di integrità primitivo nell'entity body (128 bit codificati in base 64). La generazione del content MD5 è end-to-end nessun intermediario sul path può modificarlo. L'unica operazione consentita a gateway e proxy è il controllo della presenza di errori sull'entity body.
Un attacco malizioso potrebbe essere quello di modificare l'header MD5 rendendo impossibile il recupero della risorsa.
Un altro tipo d'attacco è il *Denial Of Service* in cui sono sovraccaricati i buffer di memoria, ma questo attacco può essere evitato tramite il codice del server 411 – length required. Ma l'attacco può ricorrere all'utilizzo dei chunk per evitare il 411 ma se il server è in grado di fare la soma della lunghezza dei chunk può ancora rispondere con 411.
Si può attaccare inviando uri troppo lunghe ma il server può rispondere con un *414 – request uri too long*.

7.11 PROXY

7.11.1 TIPI DI PROXY

I proxy sono formati da un singolo host situato tra client e origin server, ne esistono tre tipi principali:

1- proxy agisce come intermediario (gateway) tra sistemi che usano diversi protocolli;
2- agisce intermediario tra due sistemi HTTP;
3- agisce come proxy HTTP, client, o Tunnel, la sua funzione varia nel tempo in base alla richiesta in pratica funziona da jolly.

1- I proxy che fanno da intermediari per altri sistemi funzionano come quelli per HTTP 1.0.
2- Nel secondo caso i proxy soddisfano la richiesta se garantiscono la freschezza della risorsa in chache, altrimenti la inoltrano all'origin server.
3- nell'ultimo caso il proxy può switchare in modatiltà tunnel se utilizzato con il metodo connect.

7.11.2 Requisiti sintattici

I proxy devono fare 2 cose :

- Inoltro dei messaggi
- Aggiunta e modifica degli header

Inoltro dei messaggi

Un proxy deve considerare la versione del protocollo dei server adiacenti. Se riceve da http 1.0 e inoltra a 1.1 riceverà un risposta 1.1 da convertire in 1.0. Ci sono regole specifiche per la gestione dei messaggi. Ad esempio il meccanismo di expect/continue tramite i codici 1xx, se questo meccanismo produce una risposta

negativa allora il serverrisponderà con *417 – Expectation failed*. Per l'invocazione di trace e optino i proxy devono controllare che ci sia l'header MAX-Forwards con un valore maggiore di zero e decrementarlo ad ogni hop. Allo stesso modo una richiesta senza host potrebbe introdurre un *400 – bad request*. Se un proxy non comprende un header allora lo inoltra direttamente a meno che non siano protetti dall'*header connection*, in tal caso per evitare di inoltrare header errati si controlla la lista dei connection tokens rimuovendo quelli non presenti nella lista.

Aggiunta e modifica degli header

I proxy HTTP devono essere in grado di aggiungere informazioni all'header VIA. I proxy all'estremità della rete devono nascondere l'identità delle macchine che proteggono e assicurare che non siano aggiunti al VIA. I proxy non possono alterare l'ordine dei campi nell'header altrimenti farebbero modifiche semantiche. Molti header end-to-end non possono essere modificati, ad esempio chache control: no trasform previene la modifica di alcuni header. Un proxy non dovrebbe modificare il *from* request header e il *server* response header, però può aggiungere un dominio ad un host name non pienamente qualificato.

7.11.3 Requisiti semantici

Esistono 4 campi d'interesse per i proxy:

1. *cache control*: il proxy deve interpretare gli header per gestire la propria chache
2. *gestione della connessione*: devono decidere quando chiudere una connessione persistente, l'unica limitazione è che gli utenti non possono avere più di due connessioni persistenti con lo stesso proxy.
3. *Gestione della banda*: deve controllare la capacità di down strema del server e ottimizzarla per le richieste dei client.
4. *Sicurezza* : tramite l'header *proxy-autenticate* e il codice di risposta *407 – proxy authorizzation required* Il 407 è simile a 401 che è end-to-end mentre 407 è hop-by-hop

7.12 ALTRE MODIFICHE

7.12.1 Modifiche ai metodi

- *Head*: in http 1.0 non era possibile modificarne il comportamento tramite costrutti condizionali mentre in http 1.1 è sempre possibile aggiungerne (ad es if_modified_since)
- *Put*: è stato definito in modo parziale in HTTP 1.0 e definito formalmente in 1.1. Un origin server esamina la request-uri per decidere se creare o modificare una risorsa. Ad esempio il server può rispondere con *201 – created* . Se una risorsa è associata ad una put allora la sua copia in cache è considerata stale.
    - *PUT vsPOST*: put crea e aggiorna mentre post gestisce una risorsa modificandola con i dati inviati.
- *Delete*: se associata ad una risorsa essa assume valore stale in cache
- *Connect*: serve per dire ad un proxy di funzionare da tunnel o per scopi futuri

7.12.2 Headers

- *General* : se un message body è modificato sono inserite informazioni nell'header trasfer encoding (hop-by-hop type)
- *Response*: sono stati aggiunti header warning ed è cambiato il retry after che in http 1.0 era informale, usato per chiarire il codice di risposta *503 – Service unavailable* e anche per l'errore *403 – request entity too large* e per altri della classe 3xx. L'header warning

aggiunge info attraverso i 7 warning code, che sono estendibili. Esempio *warning : 214 – trasformation Applied* quando il proxy modifica la risorsa.

- *Entity* : sono aumentati in http 1.1 e modificati, ad es il contetnt encoding può contenere valori multipli

7.12.3 Codici di risposta

- *2xx*
  - *203 – non authoritative information* : metadati diversi di quelli generati dall'origin server
  - *205 –reset content* : usato dall'origin server per aiutare i browser a dare feedback all'utente
  - *206 – partial content* : la cache che non supporta gli headers Range non può cacheare questo tipo di risposta
- *3xx*
  - *303–see other*: indica che si sta usando male la 302 – moved temporarily
  - *307 –temporarily redirect*: invia tramite location la nuova url temporanea della risorsa
  - *305– use proxy*: costringe a ripetere la richiesta passando per il proxy specificato in location
  - *306 – switch proxy*: variante di 305 usata solo da netscape
- *4xx* (3 codici nuovi di chiarimento, 2 di negoziazione e 2 per altri scopi)
  - *405 – not allowed* : utilizzato con l'header allow
  - *408 – request timeout* : il server chiude la richiesta del client che lo ha fatto aspettare troppo
  - *410 – GONE*: se la risorsa prima c'era e ora non c'è più
  - *413 – request entity too large*: il server non riesce a gestire richieste troppo grandi e risponde con retry atrer (meccanismo expect/continue)
  - *415 – unsupported media type*
  - *402 – payment required*
  - *409 – conflict*: quando si usano 2 put da 2 client diversi sulla stessa risorsa e allo stesso tempo
- *5xx*
  - *504 – gateway timeout*: per le direttive sul controllo della cache
  - *505 – HTTP VERSION PROTOCOL NOT SUPPORTED*