



PROGETTO DI INGEGNERIA DI INTERNET E DEL WEB

Web Server con Adattamento Dinamico di Contenuti Statici

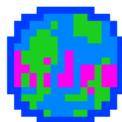
Giulia Cassarà

Giorgio Iannone

Emanuele Savo

Introduzione

Hideo



Il progetto **Hideo** è un web server con supporto minimale del protocollo HTTP/1.1 realizzato in linguaggio C usando le API della socket Berkeley.

Il suo scopo è offrire agli utenti la possibilità di scaricare immagini adattandole alle caratteristiche ed alle necessità del dispositivo richiedente.

- ▶ Quando un client si connette viene reindirizzato sulla pagina principale in cui sono presenti le miniature delle immagini disponibili per il download.
- ▶ Cliccando su un'immagine, essa viene convertita automaticamente dal server e poi consegnata all'utente nella risoluzione e qualità ottimale per il suo device.
- ▶ L'immagine convertita viene poi salvata dal server in una cache, risparmiando il carico di lavoro e diminuendo i tempi di risposta in caso di successive richieste.

Introduzione

Hideo

Il web server ha un'architettura **multithread** con pool di thread statico, quindi è capace di gestire più connessioni simultaneamente ed adotta un sistema di logging basato su livelli per discriminare i vari tipi di messaggi: *information, warning, error*.

Il codice sorgente è disponibile al seguente indirizzo:

<https://github.com/v2-dev/hideo>

Comportamento dell'applicazione

Inizializzazione

All'avvio del programma viene eseguita la lettura ed il parsing del file di configurazione *server.cfg*, contenente i seguenti parametri:

- ▶ porta di ascolto
- ▶ numero di thread
- ▶ lunghezza del backlog
- ▶ livello di logging

L'eseguibile inizializza la socket d'ascolto e le strutture dati relative al modulo WURFL (caricamento del database di *User Agent*), alla cache ed al logger con i parametri scelti.

Comportamento dell'applicazione

Multithreading

Terminata l'inizializzazione, vengono spawnati i *worker thread* che si occupano di servire i client. L'eseguibile entra in un loop non terminante nel quale:

- ▶ Accetta le connessioni dalla socket d'ascolto.
- ▶ Accoda il file descriptor ad una lista FIFO acceduta in lettura dai worker thread in stato di IDLE.
- ▶ Ai thread viene segnalato l'inserimento di nuovo file descriptor alla lista, e mediante l'utilizzo di un **mutex** uno dei thread prende l'esclusività su tale socket, rimuovendola dalla lista e servendo il client ivi collegato mantenendo aperta e sfruttando tale socket (permanenza della connessione del protocollo HTTP/1.1).

Comportamento dell'applicazione

Processamento della richiesta

Il thread, terminata la copiatura della socket ad un buffer interno, effettua un parsing del messaggio per:

- ▶ Controllarne l'effettiva validità ed aderenza alle specifiche HTTP/1.1
- ▶ per discriminare tra i metodi **GET** ed **HEAD** (di cui veniva richiesta l'implementazione).

Comportamento dell'applicazione

Processamento della richiesta

- ▶ Si ottiene dal modulo WURFL la risoluzione del dispositivo richiedente, e vengono memorizzati il percorso del file richiesto ed i parametri specificati nel campo HTTP *Accept* quali il fattore di qualità e l'estensione richiesta per l'immagine.
- ▶ Nel caso la richiesta riguardi un'immagine interviene il modulo cacher, che si occupa di inoltrare immediatamente il file se questi è presente in cache, senza dover così eseguire nuovamente alcuna operazione di conversione.
- ▶ In caso di fallimento della ricerca l'immagine viene convertita dal programma **ImageMagick**, inviata al client e salvata nella cache.

Comportamento dell'applicazione

Inoltro del contenuto & Chiusura della connessione

- ▶ Una volta intervenuto il cacher (nel caso delle immagini) od appurato che non vi siano errori di apertura della *index.html* l'applicativo si occupa di inoltrare una risposta al client, allegando o meno il file richiesto.
- ▶ In caso di errori con il file richiesto viene inoltrato un messaggio di errore **HTTP 404** mentre, nel caso di un metodo non implementato, l'errore **501**.
- ▶ La connessione verrà chiusa all'esplicita disconnessione del client od, in sua assenza, allo scadere di un timeout di connessione altrimenti resettato ad ogni nuova richiesta.

WURFL

Introduzione



WURFL è un *DDR* (Device Description Repository) sviluppato da **ScientiaMobile**, ed è composto da un set di **API** e librerie che permettono di consultare il proprio database e fornire, dato in input lo *User Agent*, il maggior numero di informazioni sul dispositivo che sta navigando su un sito web.

- ▶ Nel nostro caso abbiamo sfruttato le API di WURFL per ottenere i valori di risoluzione dei dispositivi
- ▶ Per ottenere una Trial di WURFL bisogna fare richiesta su <https://www.scientiamobile.com/>
- ▶ Documentazione ufficiale:
<https://docs.scientiamobile.com/documentation/infuze/infuze-c-api-user-guide>

WURFL

Funzionamento & Multithreading

Il funzionamento di **WURFL** consiste nel caricamento in memoria del database del programma (un file *.xml* rappresentato in C da una variabile opaca *wurfl_handle*) ed alla sua interrogazione riguardo i vari *User Agent*.

Dalla documentazione ufficiale si legge che:

"WURFL engine is thread safe and the intended design is that multiple threads share the same wurfl_handle."

- ▶ Quindi *wurfl_handle* è progettato per rispondere ad interrogazioni da più thread.
- ▶ Essendo il nostro Web Server **multi-thread**, questo ci ha permesso di non gestire il problema delle interrogazioni parallele.

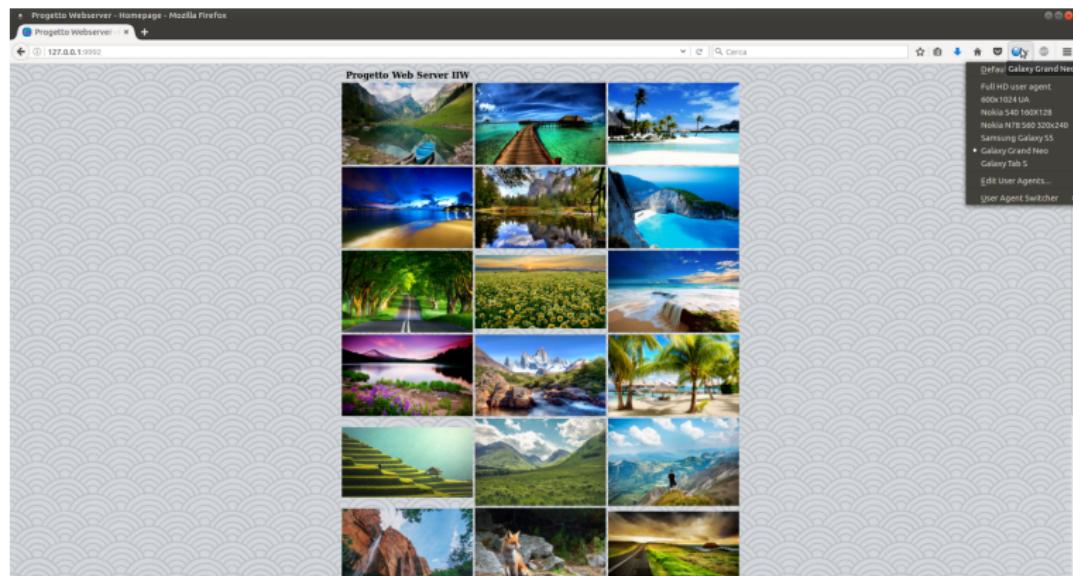
WURFL

Limitazioni Wurfl

- ▶ Il database di **WURFL** fornитoci è limitato e non aggiornato, contiene per lo più informazioni di User Agent di dispositivi mobili.
- ▶ Per effettuare correttamente i test sull'adattamento dinamico delle immagini del nostro Web Server abbiamo utilizzato un addon per Firefox:
<https://addons.mozilla.org/it/firefox/addon/user-agent-switcher/>
- ▶ **User Agent Switcher** permette di cambiare temporaneamente l'User Agent del browser con uno compatibile con il database di WURFL.

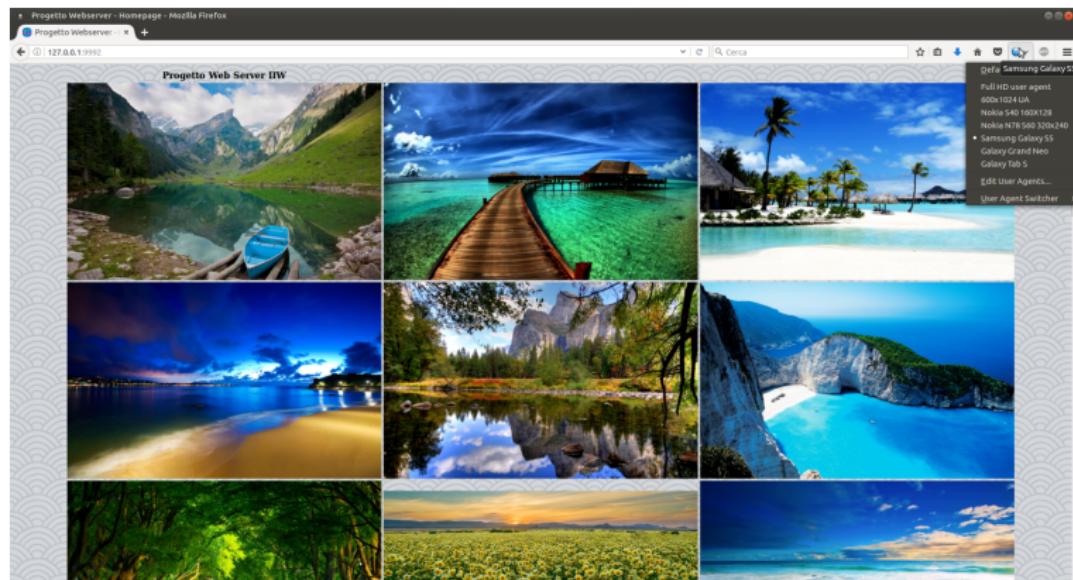
WURFL

Samsung Galaxy Grand Neo 800x400



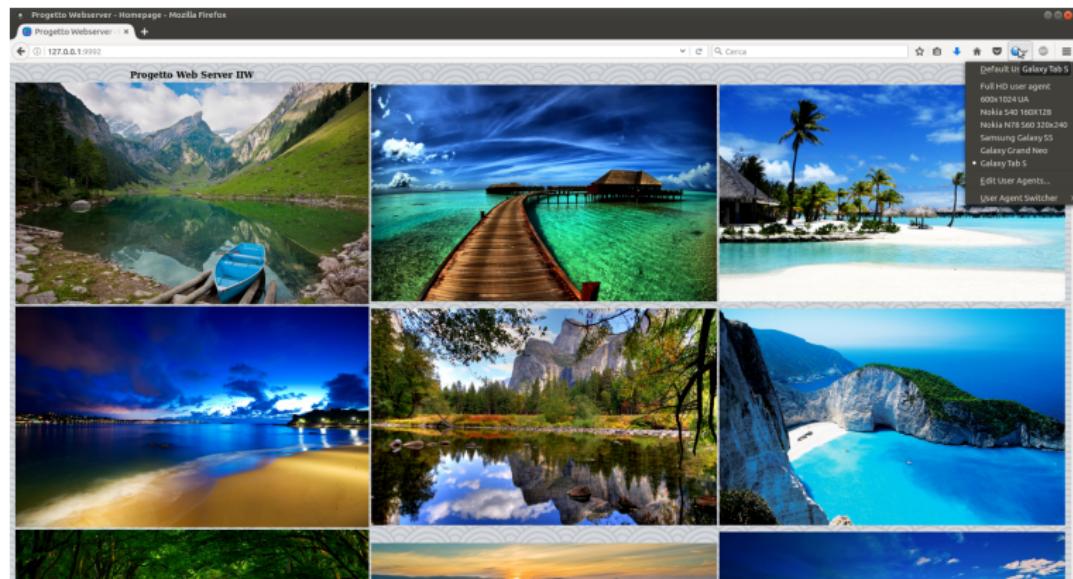
WURFL

Samsung Galaxy S5 1920x1080



WURFL

Samsung Galaxy Tab Pro 2560x1600



Adattamento dell'Immagine

ImageMagick



Per convertire le immagini abbiamo utilizzato il comando `convert` della suite di manipolazione delle immagini **ImageMagick**.

- ▶ Link: <https://wwwimagemagick.org/script/convert.php>
- ▶ Il comando viene lanciato mediante la chiamata di sistema `system`, che è essenzialmente una `fork+execve+wait`.
- ▶ Il thread che sta servendo il client crea quindi un processo figlio per eseguire il comando specificato, e rimane bloccato *in attesa* che il figlio termini il suo lavoro, ovvero attende che la conversione dell'immagine sia completata.

Adattamento dell'Immagine

Osservazioni

Vantaggi:

- ▶ le conversioni non sono a carico *diretto* del processo server, ma di processi creati *ad hoc*.
- ▶ implementazione relativamente facile.

Svantaggi:

- ▶ il thread deve comunque aspettare che la conversione *i-esima* sia completata, prima di poter lanciare un nuovo processo per la conversione *i+1-esima*.

Adattamento dell'Immagine

Un diverso approccio

Abbiamo anche pensato ad **approcci migliori** che, tuttavia, abbiamo poi *abbandonato* per mantenere un codice più semplice e facile da debuggare. L'approccio alternativo consisteva nei seguenti passi:

- ▶ Il thread che serve il client legge la prima richiesta, e fa partire la prima conversione.
- ▶ Senza aspettare che la conversione termini, legge la seconda richiesta, e fa partire la seconda conversione, e così via.
- ▶ Tra una lettura di una richiesta e l'altra, il thread *controlla* quali sono le immagini che sono state convertite completamente (ovvero i processi figli che hanno terminato il proprio lavoro) ed *evade* le relative richieste.

Adattamento dell'Immagine

Un diverso approccio

- ▶ Una singola conversione *non* blocca il thread, che risulterebbe così decisamente più veloce nel servire le richieste del client.
- ▶ Per creare il processo per la conversione anziché la chiamata system venivano utilizzate fork + execve.
- ▶ Per controllare se un processo è terminato basta eseguire la waitpid con parametro WNOHANG (per rendere la verifica non bloccante).

La **complessità** di questo approccio sta nel fatto che il thread deve *ricordarsi* della storia passata, ovvero delle richieste ricevute ma non ancora evase.

Struttura e Architettura dell'Applicazione

Server

Il Server è organizzato in due componenti principali:

- ▶ Una funzione main che si occupa della gestione della socket TCP accettando nuove connessioni;
- ▶ Un pool statico di thread worker che si occupa della gestione di ogni singola connessione;

Il server è **prethreaded** con un numero di thread statico fissato nel file di configurazione.

Struttura e Architettura dell'Applicazione

Server Main

La funzione **main** del server svolge i seguenti compiti:

- ▶ Inizializzazione delle componenti Cacher, Logger, WURFL.
- ▶ Inizializzazione e gestione della socket TCP sulla porta specificata nel file di configurazione.
- ▶ Il main entra in un ciclo in cui accetta continuamente richieste di connessione in entrata, salvando la relativa connessione in una lista chiamata *list_sock* e sveglia mediante una signal uno dei thread del pool in attesa sulla condition.

Struttura e Architettura dell'Applicazione

Thread job

- ▶ Ogni thread del pool esegue la funzione `thread_main`.
- ▶ Se non vuota, il worker estrae una socket dalla lista `list_sock`, altrimenti si mette in attesa di una nuova signal dal thread main.
- ▶ Il thread che è stato risvegliato chiama la funzione `thread_job()` in cui serve in loop il client connesso alla socket:

```
infinite loop
{
    serve_request(socket_client);
    if (some_error)
    {
        free(socket_client);
        close(socket_client);
        break loop;
    }
    else
        continue loop serving client;
}
```

Struttura e Architettura dell'Applicazione

Funzione serve_request

Nella funzione `serve_request(socket_client)` viene effettuato il parsing della richiesta HTTP:

- ▶ Viene fatto il parsing del metodo, accettando esclusivamente richieste **GET** o **HEAD**.
- ▶ Viene fatto il parsing del path richiesto, verificando l'assenza di errori.
- ▶ Viene fatto il parsing dei campi HTTP e vengono posti in buffer, se presenti, i campi *Accept* ed *User Agent*.
- ▶ Se gli header sono corretti, viene passato lo User Agent al modulo **WURFL** per ricavare valori di altezza e larghezza ottimali per lo User-Agent ricevuto.

Struttura e Architettura dell'Applicazione

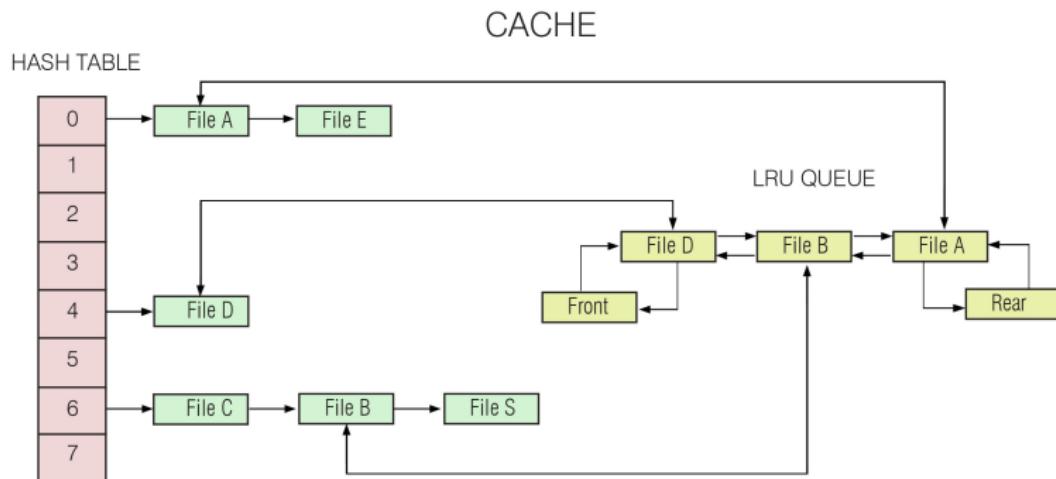
Funzione serve_request

- ▶ Il thread controlla la presenza di parametri riguardanti l'estensione ed il fattore di qualità del contenuto richiesto presenti nel campo *Accept*.
 - ▶ Se il client non specifica alcun campo *Accept* (o specifica un generico *Accept*: **/**) vengono assunti come valori di default il formato PNG ed un fattore di qualità $q = 1.0$.
- ▶ Si richiede l'immagine alla cache richiamando la funzione `obtain_file(params)` specificando nome, risoluzione, fattore di qualità ed estensione del file richiesta.
- ▶ Se l'immagine richiesta non è presente in cache essa viene convertita chiamando la funzione esterna `nConvert` di ImageMagick. Dopo la conversione il file viene aggiunto nella cache e inviato al client.

Cache

Introduzione

- ▶ Cache di tipo **LRU** (Least Recently Used).
- ▶ La cache memorizza in memoria secondaria tutti i file convertiti, senza cancellarli e mantiene in RAM i file utilizzati più di recente, in modo da poterli servire più velocemente ai thread che ne fanno richiesta.



Cache

Strutture dati

- ▶ **Tabella hash con liste di collisione:**

- ▶ Una tabella contenente entry chiamate *hashNode*.
- ▶ Un *hashNode* rappresenta un file presente su disco, contenendo informazioni sul file quali il percorso completo.
- ▶ Un file rappresentato da un *hashNode* non necessariamente è presente anche in RAM.

- ▶ **Coda LRU:**

- ▶ Una coda che memorizza entry chiamate *ramNode*.
- ▶ Un *ramNode* rappresenta un file caricato in RAM.
- ▶ Un *ramNode* contiene l'indirizzo del file caricato in RAM (tramite mmap)
- ▶ I *ramNode* utilizzati più recentemente vengono "spinti" verso la testa, quelli meno utilizzati recentemente rimangono accodati.

Cache

Inserimento iniziale di una entry

L'inserimento iniziale di un' immagine nella cache consiste nel:

- ▶ Eseguire la conversione del file originale con i parametri specificati.
- ▶ Salvare il file convertito nella cartella cache, secondo lo schema: “./cache/fileA/x/y/q/fileA.jpg”.
- ▶ Una volta terminata la conversione viene creato l'*hashNode* ed inserito nella tabella hash.
- ▶ Viene infine creato il *ramNode*, che viene inserito in testa della coda LRU.

Cache

Richiesta di un file

Supponiamo di richiedere un file A al gestore della cache:

Il gestore esegue una ricerca nella tabella hash per vedere innanzitutto se il file A è presente su disco:

- ▶ Se non esiste l' $hashNode$ relativo al file A , significa che non esiste quel file su disco, perciò si esegue l' **inserimento iniziale di una entry**.
- ▶ Se la ricerca ha successo significa che il gestore ha trovato l' $hashNode$ relativo al file A , quindi certamente il file A è presente su disco.

Cache

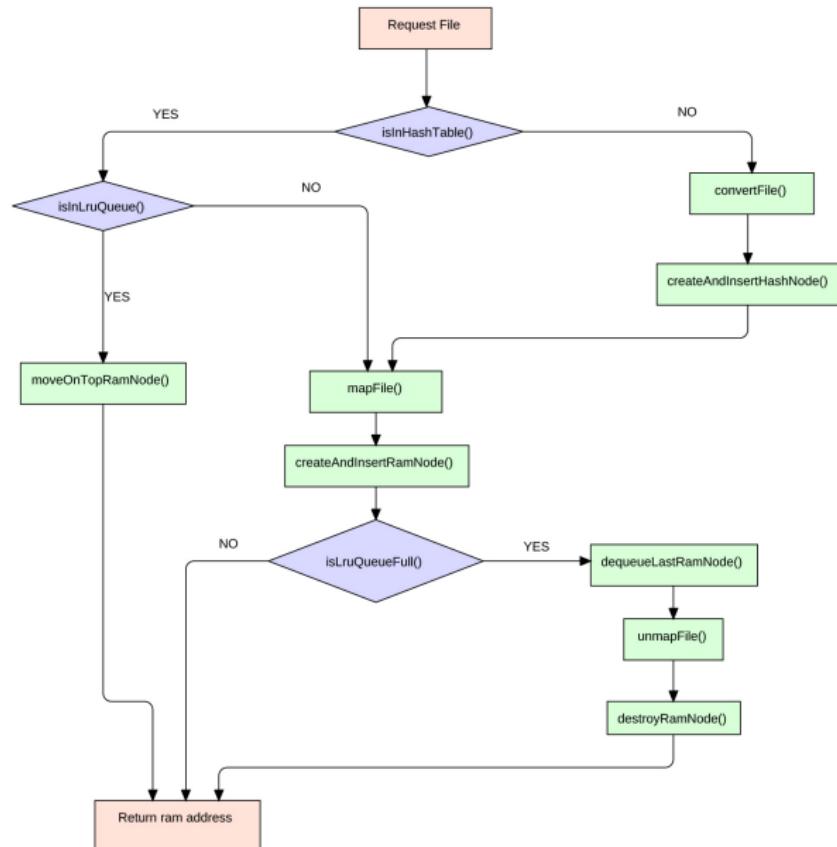
Richiesta di un file

Se l'*hashNode* è stato trovato, si valuta il contenuto del suo puntatore al *ramNode*:

- ▶ Se questo campo è diverso da NULL, allora il file A è anche caricato in RAM:
 - ▶ Il gestore ottiene immediatamente il *ramNode* del file A.
 - ▶ Sposta il *ramNode* del file A in testa alla coda LRU, in quanto ultimo file richiesto.
 - ▶ Restituisce l'indirizzo in RAM del file.
- ▶ Se invece il campo puntatore è uguale a NULL, vuole dire che il file A non è caricato in RAM:
 - ▶ Bisogna fare la open del file mediante il nome completo indicato.
 - ▶ Eseguire una mmap del file descriptor ottenuto.
 - ▶ Inizializzare un nuovo *ramNode* con i giusti parametri.
 - ▶ Inserire il *ramNode* alla testa della coda LRU.
 - ▶ Restituire al thread l'indirizzo del mapping.

Cache

Richiesta di un file



Cache

Osservazioni

L'implementazione della cache poteva *limitarsi* alla sola coda LRU, tuttavia l'aggiunta di una tabella hash per i file non caricati in memoria ha apportato notevoli vantaggi:

- ▶ Essendo una struttura dati memorizzata in RAM ci permette di sapere in tempi rapidi quali file sono memorizzati su disco, evitando ad ogni richiesta una `open`, che *rappresenterebbe* un collo di bottiglia nelle prestazioni generali del server.
- ▶ La tabella hash ci informa dell'eventuale presenza di tale file in RAM, *evitando* una scansione della coda LRU.

Cache

Prestazioni

Inserimento hashNode: L'inserimento di un *hashNode* nella tabella hash consiste nel trovare la lista di collisione a lui "prefissata" mediante la funzione hash sul nome del file, presentando quindi un costo $O(1)$.

Inserimento ramNode: Anche l'inserimento di un *ramNode* ha costo $O(1)$, in quanto consiste nella modifica di puntatori nella testa della coda.

Cancellazione hashNode: Gli *hashNode* non vengono mai cancellati.

Cache

Prestazioni

Cancellazione *ramNode*: Vengono cancellati solo i *ramNode* che si trovano in fondo alla coda LRU:

- ▶ Per trovare questi nodi è sufficiente accedere al nodo “Rear” della coda LRU e vedere all’indietro a quale *ramNode* punta.
- ▶ Una volta eliminato il *ramNode* bisogna notificare all’*hashNode* associato che il suo *ramNode* non esiste più, bisogna cioè modificare il campo puntatore dell’*hashNode* che punta al *ramNode*, settandolo al valore NULL.
- ▶ Grazie ad un puntatore gemello che collega il *ramNode* all’*hashNode* associato troviamo immediatamente l’*hashNode* che ci interessa. Di conseguenza, anche la cancellazione presenta un costo costante $O(1)$.

Cache

Prestazioni

Ricerca ramNode: il costo per la ricerca del *ramNode* è dato dal costo della ricerca dell'*hashNode*, infatti una volta trovato l'*hashNode* recuperiamo immediatamente il *ramNode* associato, accedendo semplicemente al campo puntatore. Il costo per la ricerca di un *hashNode* è dato da:

$$T_{avg}(n) = O(1 + \alpha), \text{ con } \alpha = n/m$$

dove:

n: numero di *hashNode* presenti nella tabella hash

m: numero di liste di collisione della tabella

Se il fattore di carico α si mantiene “stabile” (numero di collisioni) il tempo di ricerca può essere considerato pari ad un costo costante $O(1)$.

Cache

Interfacce utilizzate

```
char * obtain_file(struct cache * web_cache, char * name,
                    char * ext, int x, int y, int q, int * size, int file_type);

void release_file(struct cache * myCache, char * name, char * ext,
                  int x, int y, int q, int file_type);
```

Dove:

- ▶ `web_cache`: indirizzo dell'istanza della cache
- ▶ `name`: nome del file da richiedere
- ▶ `ext`: estensione del file da richiedere
- ▶ `x,y`: risoluzione dell'immagine da richiedere
- ▶ `size`: puntatore alla dimensione del file
- ▶ `file_type`: flag immagine o file html

Cache

Interfacce utilizzate

- ▶ L'interfaccia `obtain_file` serve per richiedere un certo file al gestore della cache, e restituisce l'indirizzo al primo byte del file richiesto.
- ▶ L'interfaccia `release_file` serve per rilasciare il file precedentemente richiesto.

In altre parole, ad ogni file è associato un **contatore di utilizzo**. Serve ad evitare che un file finito in fondo alla coda LRU ma ancora utilizzato da qualche thread venga eliminato dal gestore della cache.

Logger

Livelli

Per differenziare la tipologia dei messaggi trascrivibili sul file di log abbiamo introdotto tre diversi flag ed abbiamo assegnato loro un valore numerico:

Error	1	Messaggi di errore, priorità massima
Warning	2	Messaggi di avviso
Error	4	Messaggi a puro scopo informativo

Logger

Tabella Livelli

Il logger viene inizializzato ad uno degli **otto livelli** disponibili ed utilizza lo stesso meccanismo dei permessi nei file system *Unix* per stabilire quale messaggi ignorare e quali copiare:

Log level	Info	Warning	Error
0			
1			*
2		*	
3		*	*
4	*		
5	*		*
6	*	*	
7	*	*	*

Logger

Approccio

Abbiamo progettato il logger per rispettare due punti essenziali:

- ▶ **Flush:** I messaggi di log devono essere scritti su file il prima possibile, evitando di trattenerli nei buffer. In caso di blackout infatti tutti i messaggi di log andrebbero perduti.
- ▶ **Bottleneck:** I messaggi di log *non* devono essere scritti dai singoli thread per non rallentare le prestazioni dei servizi, in quanto le scritture su file sono operazioni molto costose.

Logger

Implementazione

La nostra implementazione finale ha visto perciò l'inclusione di un ulteriore thread, il **garbage thread**:

- ▶ Il garbage thread attende in stato di IDLE il riempimento di una coda di messaggi.
- ▶ Un thread che vuole scrivere un messaggio sul file di log notifica ad un thread ad hoc, il garbage thread, il relativo messaggio
 - ▶ Il thread continua a servire il client, senza preoccuparsi del resto.
- ▶ Il messaggio viene inserito nella coda di messaggi ed al garbage thread viene notificato tale inserimento.
- ▶ Il garbage thread esce dalla fase di IDLE e svuota la coda di messaggi, scrivendo ogni messaggio sul file di log mediante una `write`.
- ▶ Dopo aver svuotato la coda, il garbage thread torna in stato di IDLE, in attesa di nuovi inserimenti.

Logger

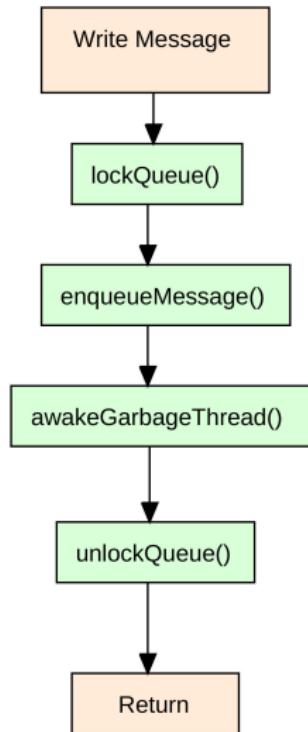
Osservazioni

Dal punto di vista di un thread la scrittura di un messaggio sul file di log risulta quindi abbastanza veloce, consistendo solo in operazioni effettuate in RAM:

- ▶ Acquisizione del lock della coda.
- ▶ Inserimento del messaggio della coda.
- ▶ Invio del segnale di "sveglia" al garbadge thread.
- ▶ Rilascio del lock della coda.

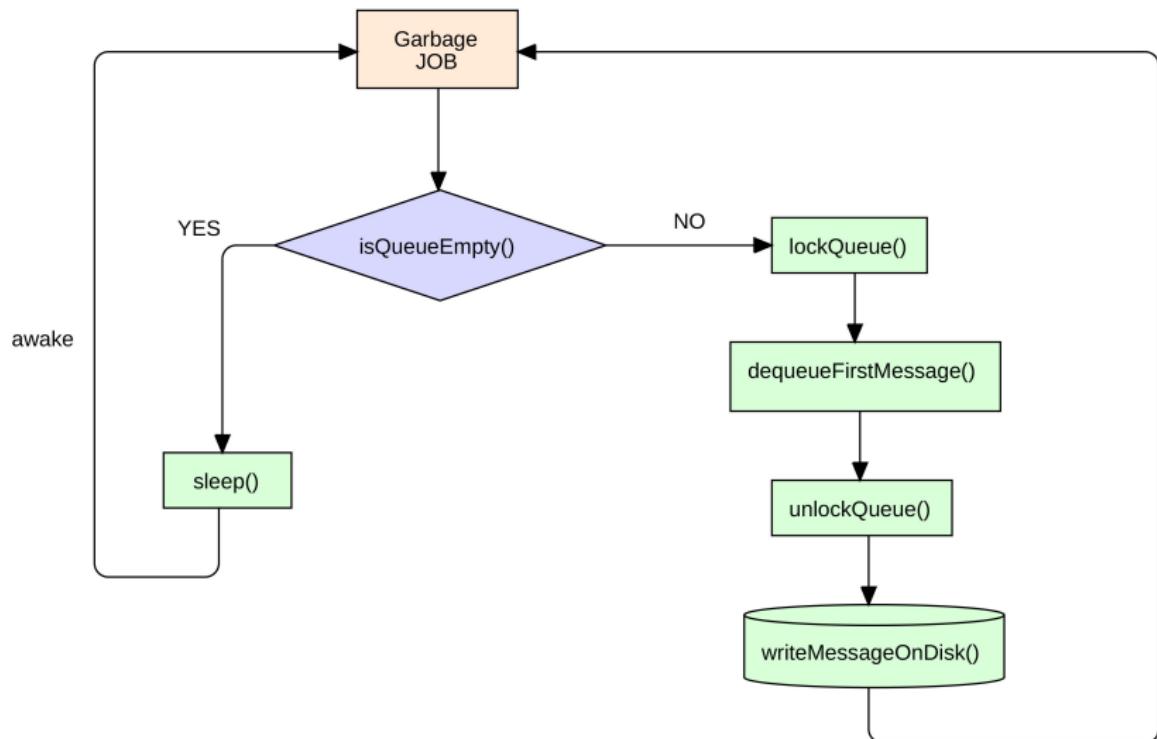
Logger

Scrittura da thread esterno



Logger

Scrittura del Garbage Thread



Logger

Interfacce

```
struct logger * create_logger(char * pathLog, int log_lvl);
```

con:

- ▶ pathLog: percorso dove creare il file di log.
- ▶ log_lvl: livello del log.
- ▶ restituisce un puntatore all'istanza logger creata.

```
void toLog(int type, struct logger * myLogger, char * buf, ...);
```

con:

- ▶ type: tipo di messaggio (Error, Warning o Info).
- ▶ myLogger: puntatore all'istanza del logger.
- ▶ buf: messaggio da scrivere (testo formattato, parametri illimitati).

Performance e benchmarking

Premesse

- ▶ Prima di confrontare **Hideo** con **Apache** abbiamo condotto alcuni test preliminari durante la fase di sviluppo per individuare un valore ottimale per il numero di worker thread, successivamente individuato nel range di valori compreso tra 10 e 30 – di qui la decisione di utilizzare il valore 20.
- ▶ I test, effettuati con **httpperf**, hanno evidenziato come il server riesca a servire senza rallentamenti, errori o warning fino 2305 connessioni al secondo.
- ▶ Aumentando il numero di connessioni/secondo il server continua a funzionare, tuttavia il degrado è significativo delle prestazioni (in particolare abbiamo riportato valori decisamente discostanti nei tempi di risposta e nel tempo di connessione medio).

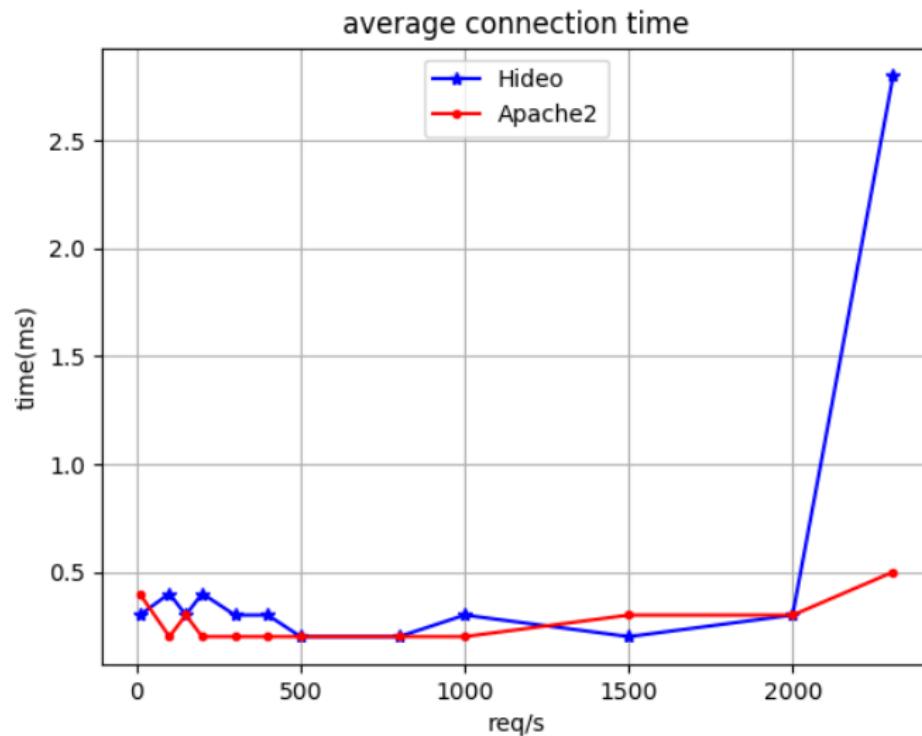
Performance & Benchmarking

Test

- ▶ Il plotting dei grafici è stato effettuato mediante la libreria Python **MatplotLib**.
- ▶ In ascissa è riportato il valore del tasso di arrivo, misurato in richieste al secondo (req/s); in ordinata, invece, è riportato il tempo corrispondente, misurato in millisecondi(ms); in blu le prestazioni di Hideo, in rosso quelle di Apache2.

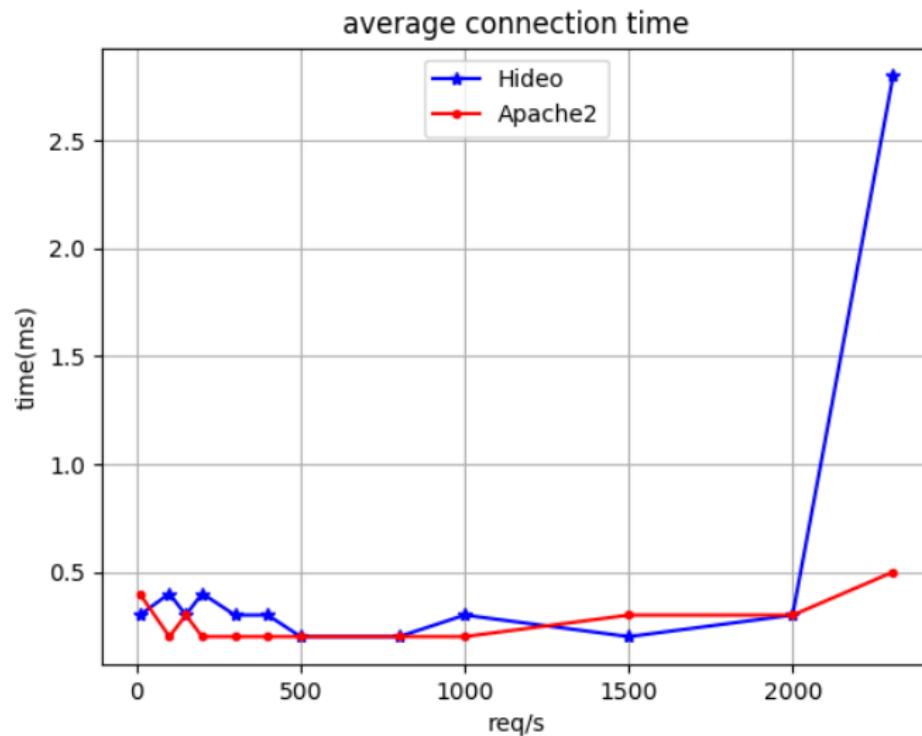
Performance & Benchmarking

Average Connection Time



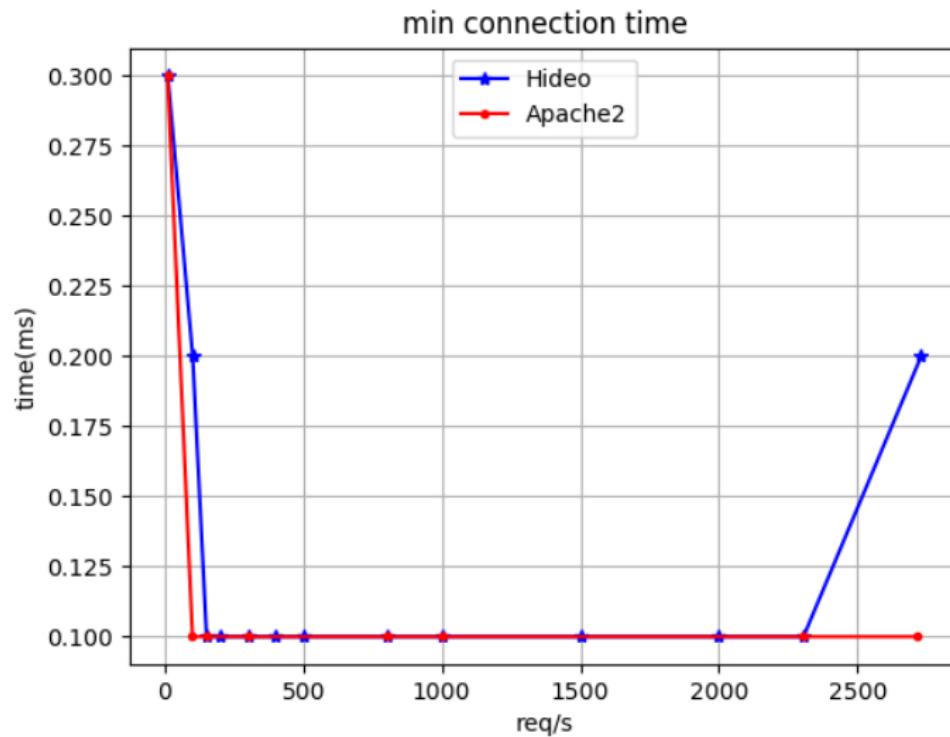
Performance & Benchmarking

Average Connection Time



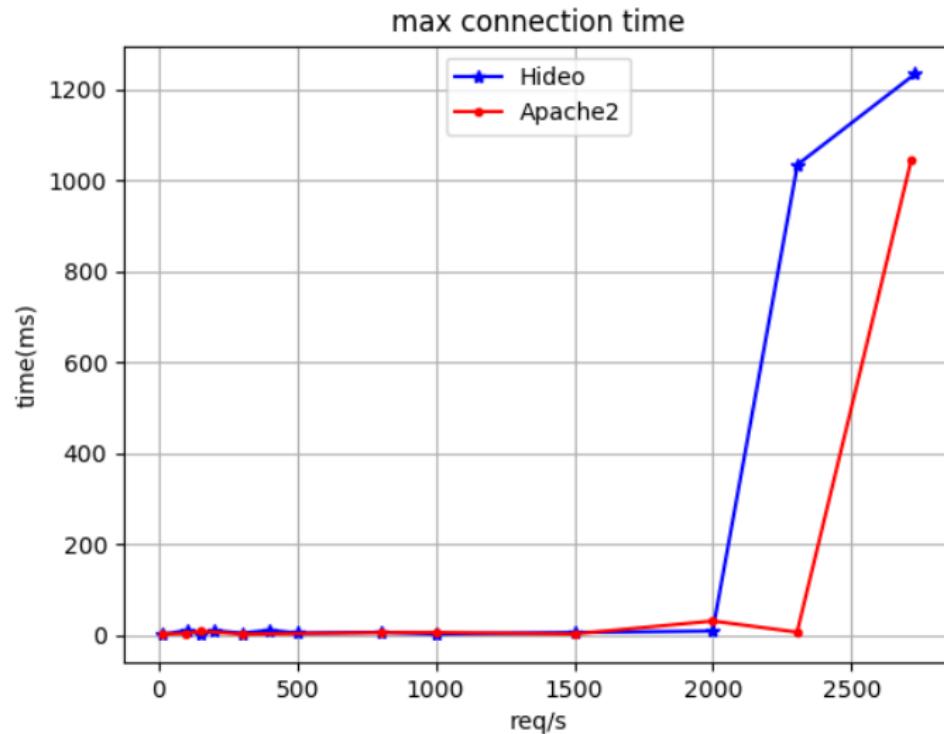
Performance & Benchmarking

Minimum Connection Time



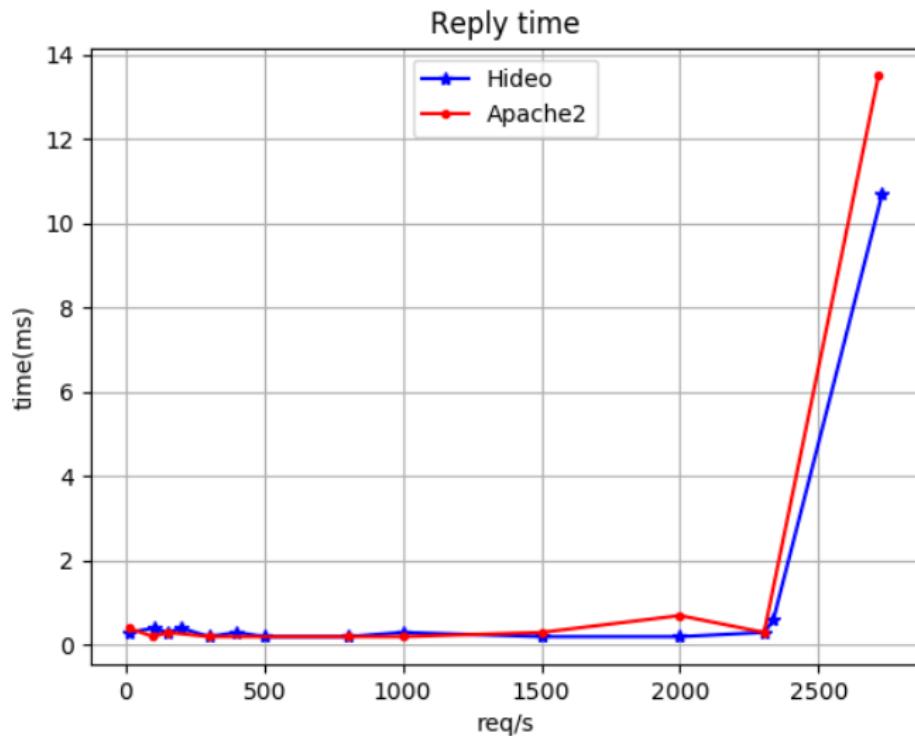
Performance & Benchmarking

Maximum Connection Time



Performance & Benchmarking

Reply Time



Performance & Benchmarking

Analisi del Test

- ▶ I test sono stati effettuati su macchina virtuale utilizzando la distribuzione *GNU/Linux Xubuntu*.
- ▶ Il motivo di tale decisione è dovuto al fatto che **ScientiaMobile** ci ha fornito un pacchetto *.deb* (quindi nativamente supportata nelle distribuzioni Debian e derivate), oltre alla volontà di fornire un ambiente pulito e minimale ad entrambi i server.
- ▶ Dai risultati dei test, abbiamo constatato che i tempi di risposta fino ad un certo *overhead* sono migliori di **Apache2**.

Performance & Benchmarking

Analisi del Test

- ▶ Una delle ipotesi per cui i tempi di risposta sono talvolta più stringenti è che l'architettura di **Apache2** è molto più complessa (essendo multi-processo e multi- thread) rispetto all'architettura di **Hideo**.
- ▶ Questa complessità, tuttavia, si traduce in robustezza ed affidabilità nella risposta, permettendo ad **Apache2** di mantenere prestazioni stabili molto oltre le 2000 connessioni simultanee.

Utilizzo dell'Applicazione

Prerequisiti

Per la compilazione del Web Server sono necessari:

- ▶ Compilatore **GCC** versione 4+.
- ▶ Header C di sistema.
- ▶ Pacchetto Header di WURFL.
- ▶ Database di WURFL.

Utilizzo dell'Applicazione

Compilazione & Configurazione

- ▶ Per la compilazione del Web Server utilizzare i comandi *make* e *make clean*.
- ▶ Per configurare il server modificare opportunamente il file *server.cfg* attraverso i parametri:
 - ▶ **server**: Numero di porta sulla quale rimanere in ascolto.
 - ▶ **threads**: Numero di thread del pool statico.
 - ▶ **backlog**: Numero massimo di connessioni accettate in coda.
 - ▶ **loglvl**: Tipologia di logging.

Utilizzo dell'Applicazione

Esecuzione

- ▶ Una volta compilato e configurato, si può eseguire il server attraverso il comando `./server`.
- ▶ Il server sarà disponibile all'indirizzo:
`http://127.0.0.1:porta_configurata`.
- ▶ Il log del server sarà accodato al file `server.log` (consultabile in tempo reale con il comando `tail -f server.log`).