



# PROGETTO DI INGEGNERIA DI INTERNET E DEL WEB

Web Server con Adattamento Dinamico di Contenuti Statici

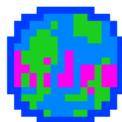
Giulia Cassara

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*.

Le scelte progettuali effettuate hanno avuto come obiettivo primario la minimizzazione dei tempi di risposta del server – senza tralasciare l'ottimizzazione nell'uso e rilascio delle risorse primarie di sistema.

Il codice sorgente è disponibile al seguente indirizzo:

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

# 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 essenzialmente 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 affatto il problema delle interrogazioni parallele

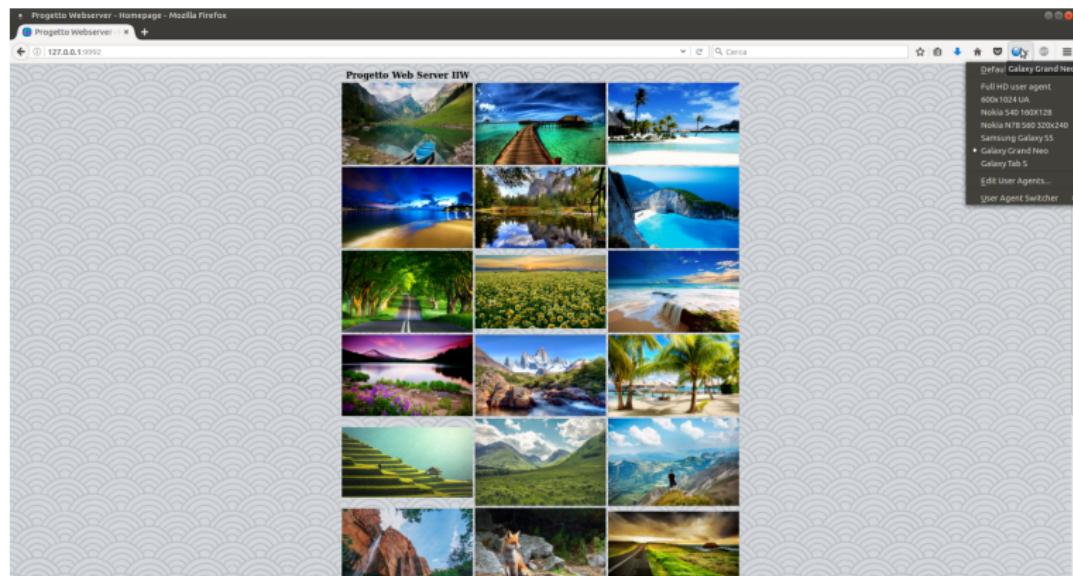
# WURFL

## Limitazioni Wurfl

- ▶ Il database di WURFL fornитоci 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 quindi 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

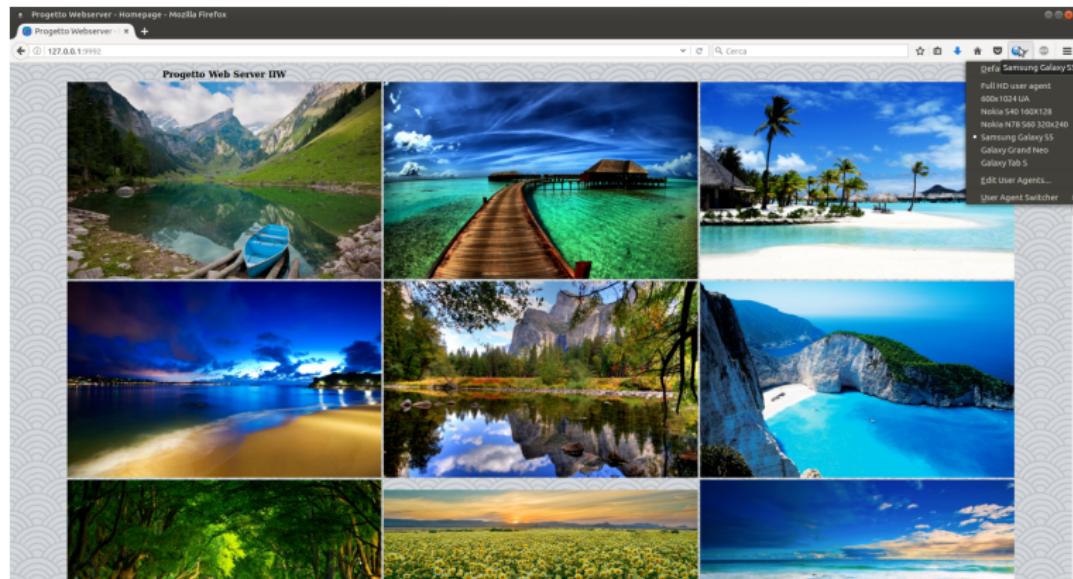
# WURFL

Samsung Galaxy Grand Neo 800x400



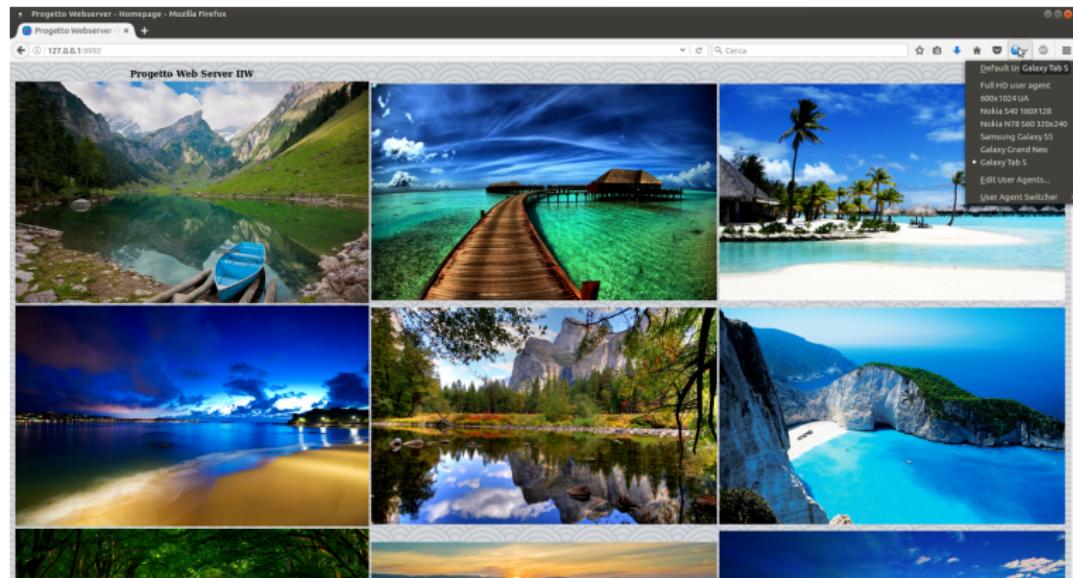
# WURFL

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 studiato approcci migliori che, tuttavia, abbiamo deciso di abbandonare in fase progettuale 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;

Per migliorare il tempo di risposta si è deciso di utilizzare la tecnica del *prethreading*, ovvero impostando staticamente il numero di thread del pool, specificandolo nel file di configurazione.

Nello scegliere il numero di thread ottimale si deve tenere conto delle caratteristiche fisiche del calcolatore che eseguirà l'applicazione; la trattazione di questo argomento esula dallo scopo di questo progetto, tuttavia alcuni test preliminari hanno individuato come ottimali i valori contenuti nel range 10-30.

# Struttura e Architettura dell'Applicazione

## Server Main

- ▶ 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, controllando sommariamente la presenza 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, il quale interrogherà il suo database interno per restituire i valori di altezza e larghezza ottimali per lo User-Agent ricevuto.

# Struttura e Architettura dell'Applicazione

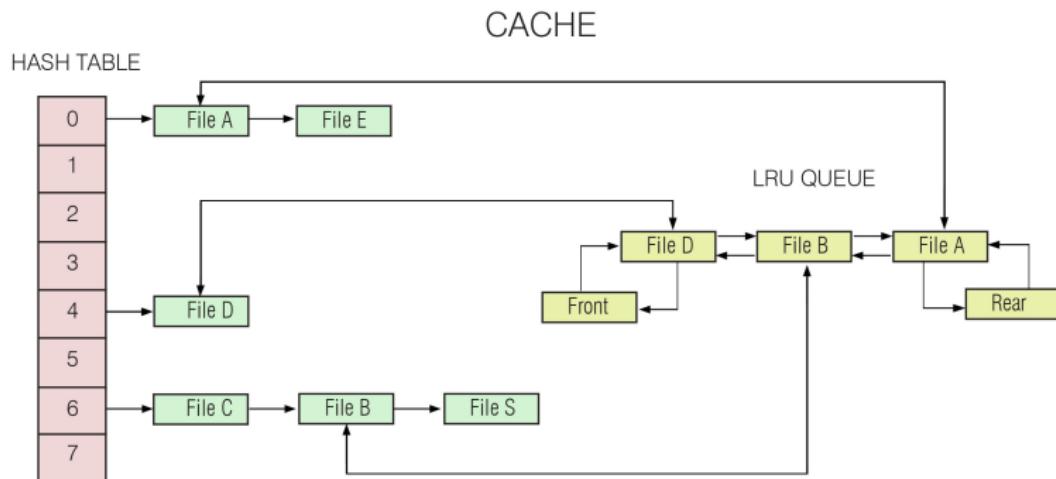
## Funzione serve\_request

- ▶ Il thread controlla la presenza di parametri riguardanti l'estensione ed il fattore di qualità del contenuto richiesto presenti nel campo Accept.
  - ▶ Qualora nella transazione HTTP 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.
- ▶ Prima di effettuare la conversione di un'immagine si interroga la 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, i cui parametri forniti sono gli stessi forniti alla cache. 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

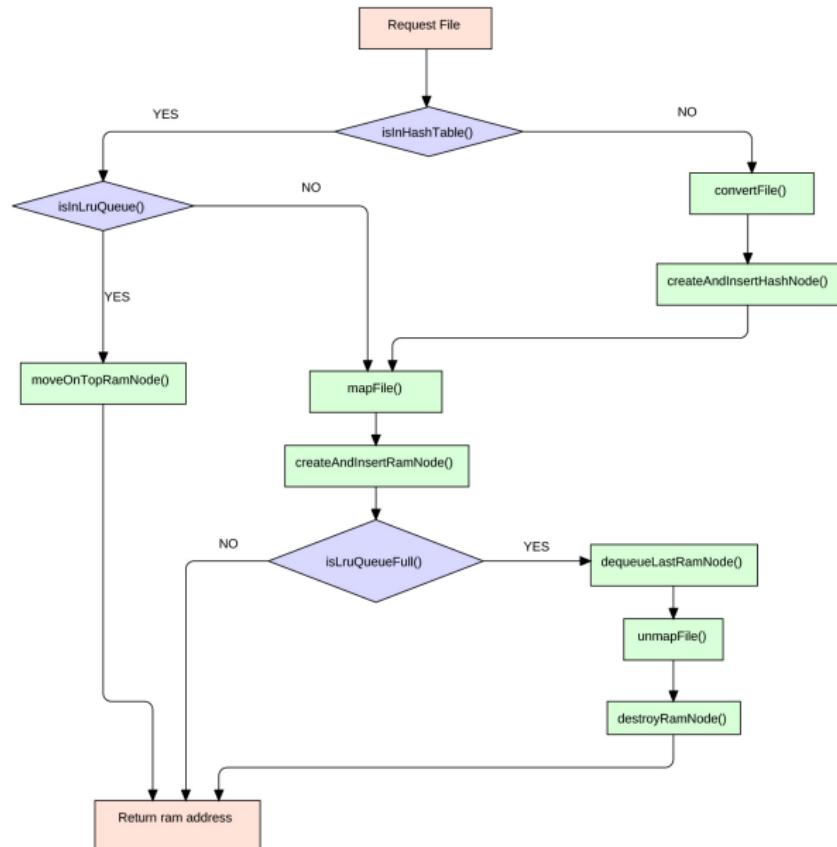
## Richiesta di un file

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()`, chiamata di sistema 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.
- ▶ Per velocizzare tale operazione abbiamo aggiunto un puntatore gemello che collega il *ramNode* all'*hashNode* associato, quindi anche in questo caso riusciamo immediatamente a trovare 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* riusciamo a recuperare 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  $\alpha$  si mantiene "stabile", ossia se il numero di collisioni tra gli elementi si mantiene abbastanza basso, 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 essenzialmente per 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:

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

# Logger

## Tabella Livelli

Ogni messaggio viene trascritto con il rispettivo flag associato. 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:

<b>Log level</b>	<i>Info</i>	<i>Warning</i>	<i>Error</i>
<b>0</b>			
<b>1</b>			*
<b>2</b>		*	
<b>3</b>		*	*
<b>4</b>	*		
<b>5</b>	*		*
<b>6</b>	*	*	
<b>7</b>	*	*	*

# Logger

## Approccio

In seguito ad una discussione all'interno del gruppo di lavoro è emerso che il logger avrebbe dovuto rispettare essenzialmente due punti:

- ▶ **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

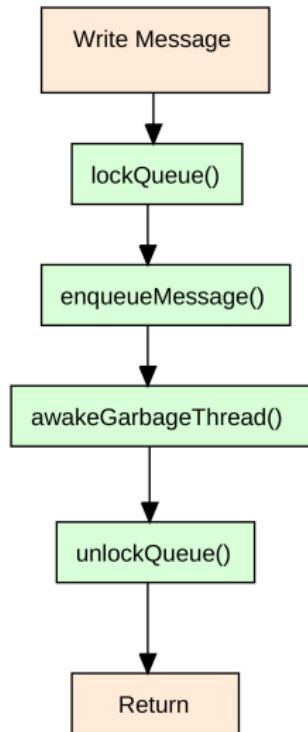
## Osservazioni

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

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

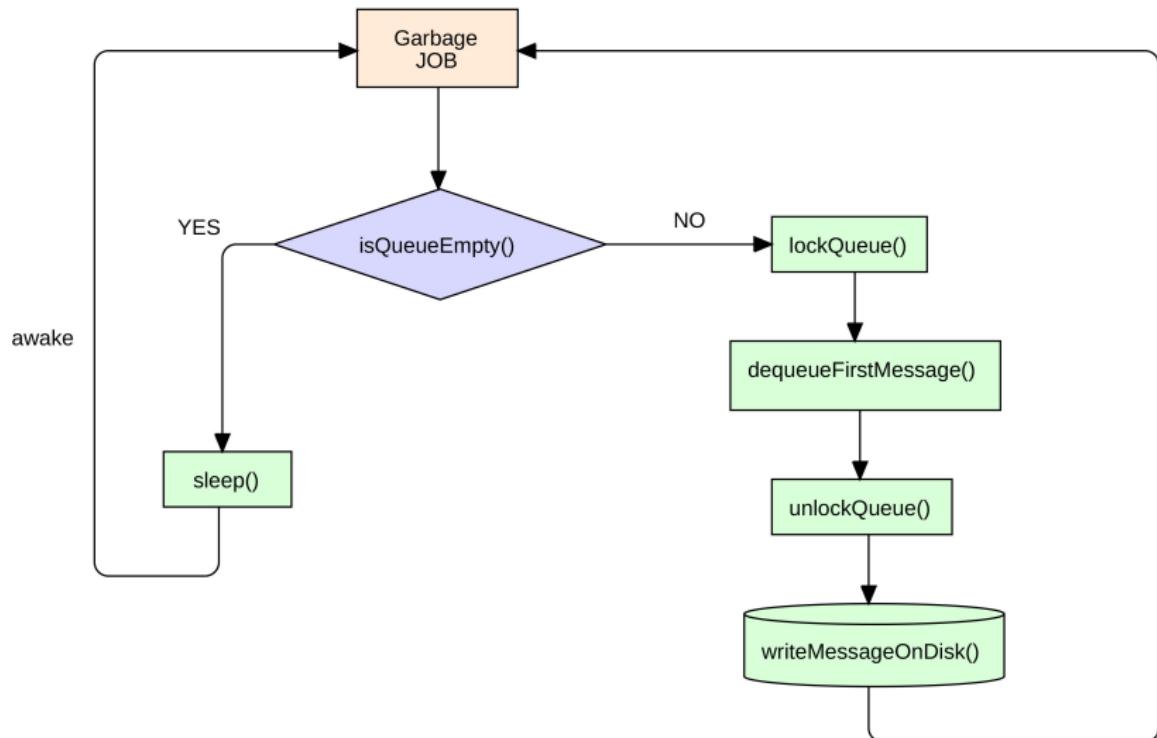
# Logger

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 ).

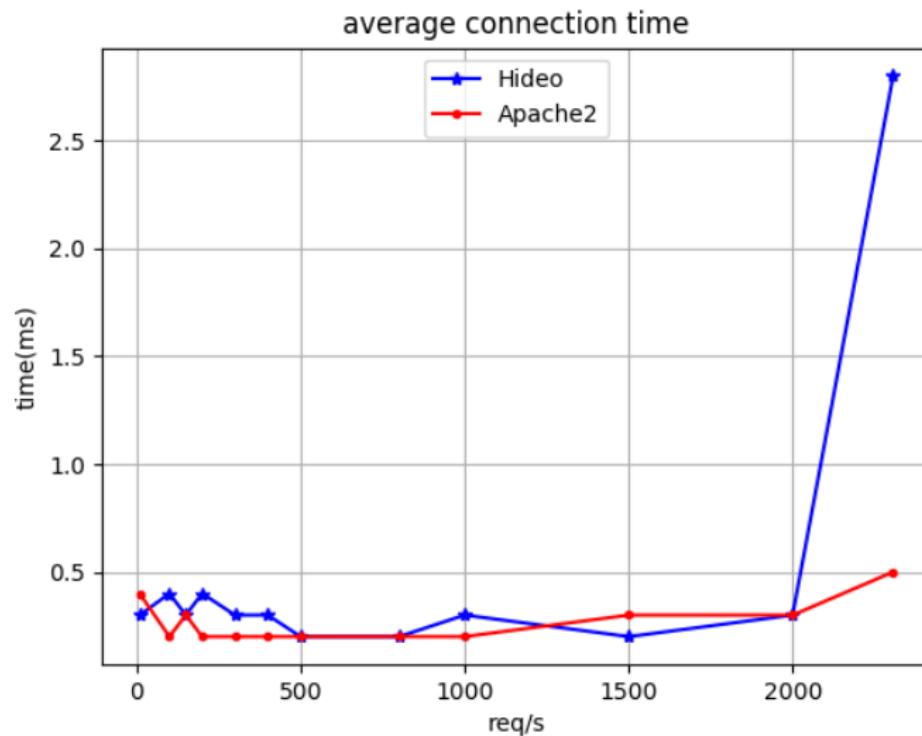
# Performance & Benchmarking

## Test

- ▶ Il plotting dei grafici è stato effettuato mediante la libreria Python **Matplotlib** e riassumono i risultati dei test, confrontandoli.
- ▶ 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.

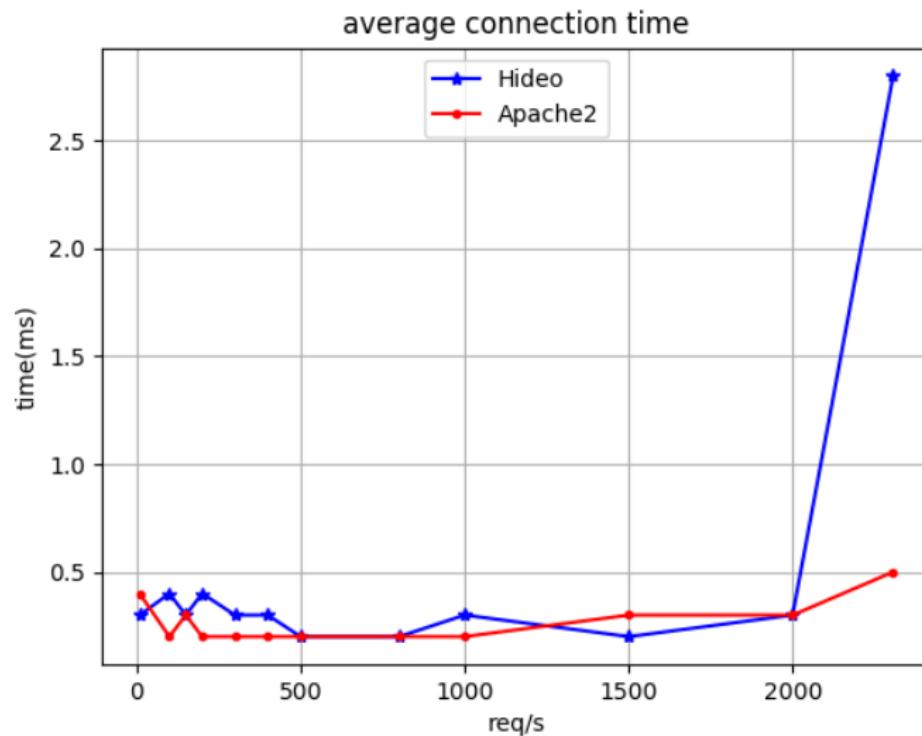
# Performance & Benchmarking

## Average Connection Time



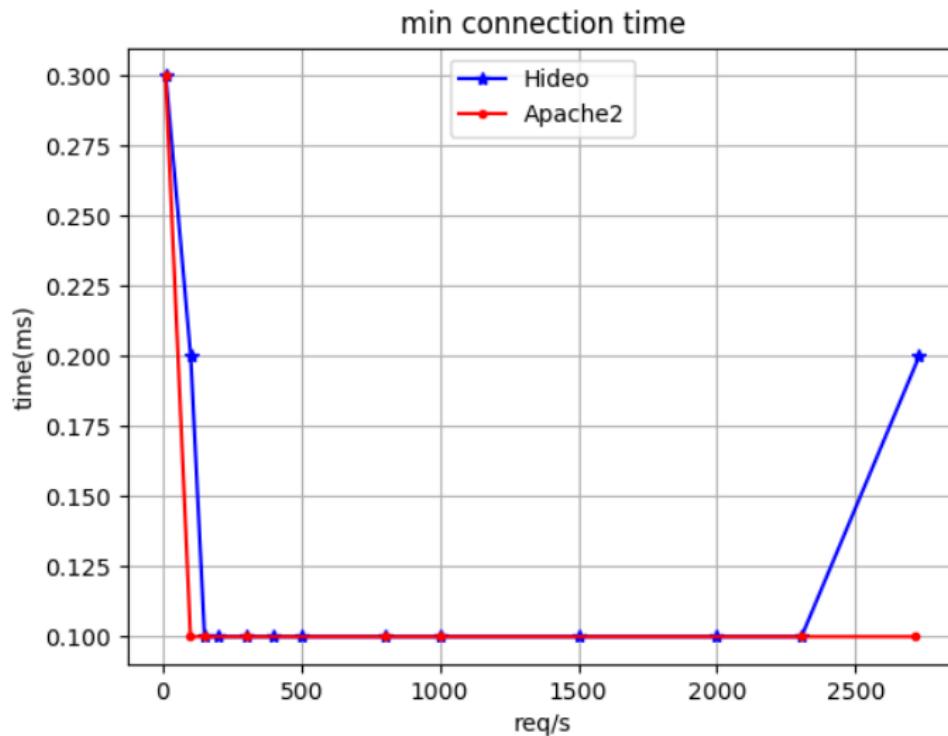
# Performance & Benchmarking

## Average Connection Time



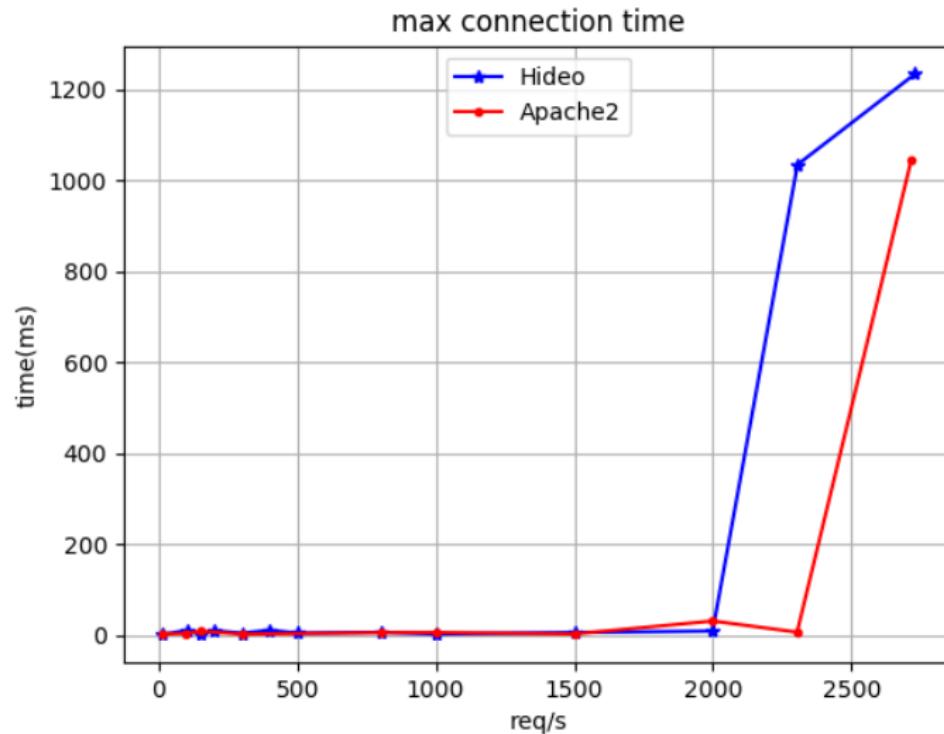
# Performance & Benchmarking

## Minimum Connection Time



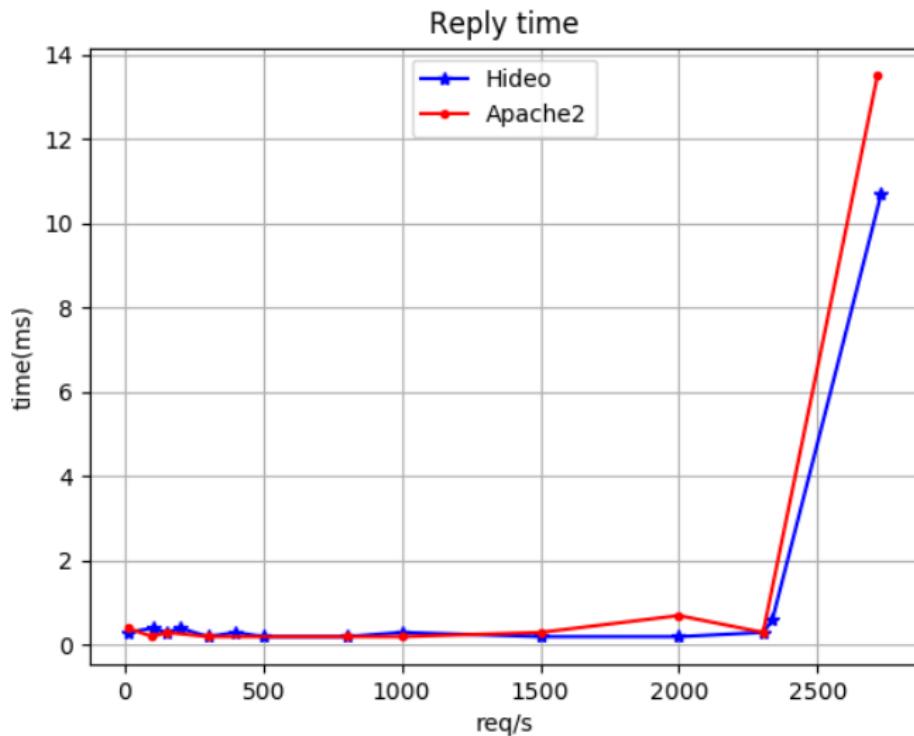
# Performance & Benchmarking

## Maximum Connection Time



# Performance & Benchmarking

## Reply Time



# Performance & Benchmarking

## Analisi del Test

- ▶ Per i test disponevamo di una macchina con preinstallata la distribuzione GNU/Linux Fedora sulla quale abbiamo hostato una macchina virtuale con un processore e due core virtuali
- ▶ Il motivo di tale decisione è dovuto in particolare 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.
- ▶ Nonostante Apache2 abbia alle spalle dieci anni di sviluppo da parte dei sviluppatori della Apache Foundation i risultati dei test ci hanno sorpreso, soprattutto i numerosi casi di prestazioni simili e migliori.

# 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 anche oltre le 2000 connessioni simultanee.