FACOLTÀ DI INGEGNERIA INFORMATICA

INGEGNERIA DI INTERNET E DEL WEB

A.A. 2015–2016

PROGETTO A: WEB SERVER CON ADATTAMENTO DINAMICO DI CONTENUTI STATICI

G. CASSARÀ – G. IANNONE – E. SAVO

Indice

1. Introduzione

1.1 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, presenti nella pagina principale del server, adattandole alle caratteristiche ed alle necessità del dispositivo richiedente.

Quando un *client* si connette viene reindirizzato sulla pagina principale, index.html, 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.

1.2 Scelte progettuali

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.

1.3 Codice Sorgente

Il codice sorgente è disponibile al seguente indirizzo:

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

1.4 Licenza d'uso

Il codice sorgente del progetto Hideo viene rilasciato secondo la Licenza MIT:

2. Comportamento dell'applicazione

2.1 Inizializzazione

All'avvio del programma, attraverso la libreria *parse_conf_file.h,* viene eseguita la lettura ed il parsing del file di configurazione *server.cfg* - contenente i parametri riguardanti la porta d'ascolto, il numero di thread, lunghezza del backlog e livello di logging richiesto. Avvenuta l'eventuale sostituzione dei parametri di default, 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.

2.2 Multithreading

Terminata con successo l'inizializzazione, vengono *spawnati* i worker thread che si occuperanno di servire i client: l'eseguibile entra in un loop non terminante nel quale accetta le connessioni dalla socket d'ascolto ed accoda il file descriptor della connessione ad una lista FIFO acceduta in lettura dai worker thread in stato di IDLE. Ai thread viene infatti segnalato che un nuovo file descriptor è stato aggiunto nella lista perciò, mediante l'utilizzo dei Pthread Mutex, uno dei thread prende l'esclusività su tale file descriptor (rappresentante la socket), rimuovendolo dalla lista e servendo il client ivi collegato mantenendo aperta e sfruttando tale socket (permanenza della connessione del protocollo HTTP/1.1).

2.3 Processamento della richiesta

Il thread, terminata la copiatura dalla socket ad un buffer interno, effettua un parsing del messaggio sia per controllarne l'effettiva validità ed aderenza alle specifiche HTTP/1.1, sia per discriminare tra i metodi GET ed HEAD (di cui veniva richiesta l'implementazione).

In caso di esito positivo viene invocato il modulo WURFL che, ricevuto in input il campo *User-Agent*, restituisce ulteriori valori quali la risoluzione del dispositivo richiedente, nonché 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.

2.4 Inoltro del contenuto

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.

2.5 Chiusura della connessione

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.

3. WURFL

3.1 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 attualmente navigato un sito web od un servizio. Nel nostro caso abbiamo sfruttato le API di WURFL per ottenere i valori di risoluzione del dispostivo richiedente.

3.2 Ottenere WURFL

Wurfl è un Software proprietario: per ottenere una prova gratuita del programma bisogna visitare il sito di ScientiaMobile (url: https://www.scientiamobile.com/) e scaricare la versione di prova (Trial) di "WURFL InFuze C API". Durante la procedura si verrà contattati da un operatore dell'azienda che si assicurerà dell'assenza di finalità commerciali nella richiesta, fornendo un pacchetto .deb contenente gli header ed un database WURFL.xml in versione comunque limitata.

3.3 Funzionamento

Il funzionamento dell'ambiente WURFL, limitato alle funzioni da noi utilizzate, è ristretto al 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 dati estrapolabili dai vari User Agents.

Rimandiamo alla documentazione originale domande sulle ulteriori potenzialità delle API

(https://docs.scientiamobile.com/documentation/infuze/infuze-c-api-user-guide)

La consultazione della documentazione ci ha rassicurato che

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

ovvero che *wurfl_handle* è progettato per rispondere a più interrogazioni provenienti da più thread: essendo il Webserver multi-thread non abbiamo dovuto gestire il problema diversamente, ad esempio creando una coda FIFO di interrogazione per poter utilizzare un servizio monothread bloccante.

3.4 Osservazioni

Utilizzando la libreria WURFL non abbiamo potuto ignorare come nella quasi totalità dei dispositivi desktop testati la risoluzione comunicata fosse il valore di default di 800x600 pixel (valore fornito da WURFL in caso di identificazione fallita). Aprendo il file .xml abbiamo constatato che il database fornitoci è limitato e non aggiornato, in particolare sono presenti principalmente User Agent di dispositivi mobili. In questo modo sarebbe stato impossibile fare (in maniera rapida e naturale) test sull'adattamento dinamico delle immagini del nostro Webserver, perciò abbiamo ovviato utilizzando alcuni add-on per i nostri browser che ci permettevano di cambiare a nostro piacimento gli User Agent – ad esempio per Mozilla Firefox abbiamo utilizzato il seguente:

https://addons.mozilla.org/it/firefox/addon/user-agent-switcher/

4. Adattamento dell'immagine

4.1 ImageMagick

Per convertire le immagini abbiamo utilizzato il comando `convert` della suite di manipolazione di immagini **ImageMagick**. Il comando viene lanciato mediante la chiamata di sistema `system` che essenzialemente è un wrapper delle chiamate di sistema `fork+execve+wait`: viene creato un nuovo processo figlio che eseguirà il comando specificato, ed il thread che lancia la `system` rimane bloccato *in attesa* che il figlio termini il suo lavoro, ovvero attende che la conversione dell'immagine venga completata.

4.2 Funzionamento

Supponiamo di essere in questa situazione: un thread riceve, tramite una socket, 10 richieste di immagini, da parte di un client. Il thread leggerà la prima richiesta, eseguirà la prima `system` (cioè la prima conversione), poi leggerà la seconda richiesta ed eseguirà la seconda `system`, e così via. Seguendo questo approccio, il thread aspetta sempre che la conversione *i-esima* sia completata, prima di lanciare un nuovo processo per la conversione dell'immagine *i+1-esima*.

4.3 Osservazioni

Il vantaggio di questo approccio è che le conversioni non sono a carico del processo server, ma di altri processi creati *ad hoc,* tuttavia il thread che lancia la `system` deve comunque *aspettare* che la conversione termini, prima di poter continuare la sua esecuzione.

Abbiamo anche studiato approcci migliori che, tuttavia, abbiamo deciso di abbandonare in fase progettuale per mantenere in primis un codice semplice, lineare e facile da controllare in fase di debug e, soprattutto, per poter dedicare maggiore spazio all'implementazione, al debug ed al perfezionamento delle finalità principali del progetto.

4.4 Un diverso approccio

Una delle possibili implementazioni consisteva nei seguenti punti cardine:

- 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.
- Una singola conversione non blocca il thread, che risulterebbe così decisamente più veloce nel servire le richiese 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).

5. Struttura ed Architettura dell'Applicazione

5.1 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 dovrà 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.

5.2 Server main

Inizializzate le varie componenti di Cacher, Logger, WURFL, la funzione principale del server passa all'inizializzazione e gestione della socket TCP sulla porta specificata nel file di configurazione.

Questa funzione difatti accetta una connessione in entrata, salvando la socket relativa in una lista chiamata *list_sock* e *sveglia* mediante una signal uno dei thread del pool in attesa sulla condition.

5.3 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.

Pseudocodice relativo al loop di thread_job():

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

5.4 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.

- 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.

6. Cache

6.1 Funzionamento della Cache

La cache da noi implementata 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, implementando così una cache di tipo *LRU* (Least Recently Used).

6.2 Strutture dati

Il modulo Cacher implementa al suo interno due differenti 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.

Se esiste un *ramNode* per un dato file certamente esiste l’*hashNode* a lui associato (il quale è collegato al *ramNode* tramite puntatore). Se esiste un *hashNode* ma non il relativo *ramNode* allora il puntatore ha valore *NULL.*

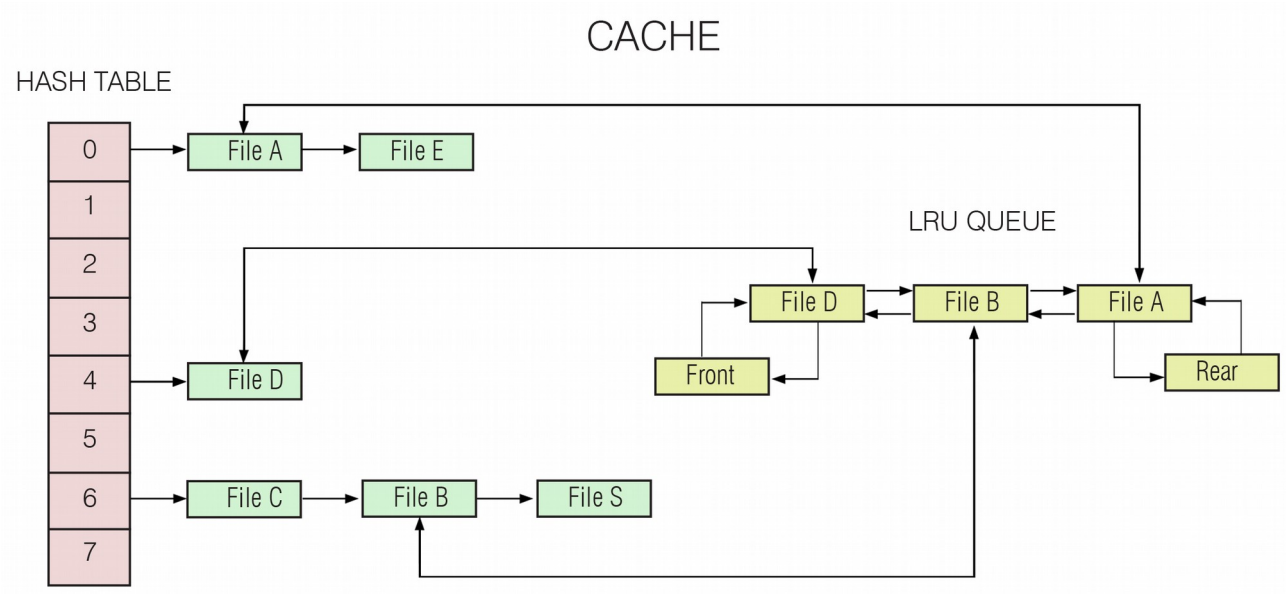


Figura 1: rappresentazione di una cache LRU.

6.3 Inserimento iniziale di una entry

La tabella hash e la coda appena inizializzate saranno vuote: qualsiasi sia la prima richiesta del thread vedrà un *miss*. Oltre a non esser presente una entry di tipo *ramNode* o di tipo *hashNode* non è presente il file fisico, perciò sarà necessario:

- Eseguire la conversione del file originale con i parametri specificati.
- Salvare il file convertito nella cartella cache, secondo il seguente schema:

  “./cache/fileA/x/y/q/fileA.jpg".

  - fileA Nome del file
  - x, y Valori indicanti la risoluzione
  - q valore numerico [1~100] indicante la qualità

- 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.

6.4 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'h*ashNode* relativo al file A, significa che non esiste quel file su disco, perciò si esegue il punto **6.3 Inserimento 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.
- Dato l'hashNode, si valuta il contenuto del 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 (probabilmente perché non è stato utilizzato per un certo lasso di tempo ed è stato rimosso).
    - 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 relativo.
  - Le successive richieste dello stesso file saranno servite più velocemente, in quanto il file sarà già caricato in ram.

6.5 Rimozione di una entry ramNode

Se il file A non viene utilizzato per un certo lasso di tempo il relativo *ramNode* si posiziona al termine della coda LRU, e quindi è a "rischio cancellazione". Il numero di *ramNode* presenti nella coda LRU è limitato ad un valore preimpostato

N, perciò all'inserimento dell' N+1-esimo *ramNode* viene rimosso l'elemento utilizzato meno recentemente.

La "cancellazione" consiste nelle seguenti operazioni:

- *munmap* dell'indirizzo.
- Settaggio del puntatore al *ramNode* presennte nell'*hashNode* al valore NULL.
- Liberazione della memoria occupata mediante una *free*.

Il file, ovviamente, è ancora presente su disco, ma non è più caricato in ram.

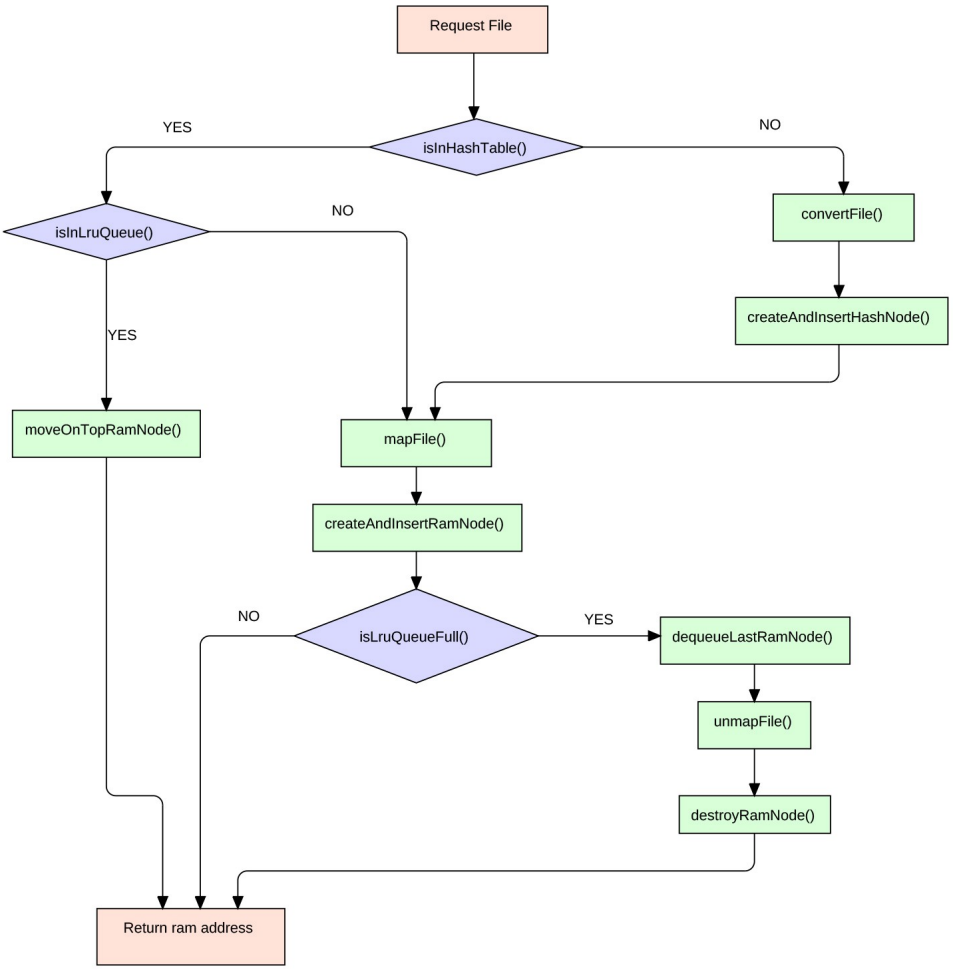


Figura 2: Flowchart indicativo di richiesta al gestore della cache.

6.6 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 uno stringente (se non il più significativo) 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 (od evitando l'implementazione di un diverso metodo di ricerca più efficace).

6.7 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.

Cancellazione ramNode

Le uniche cancellazioni che vengono eseguite sono quelle dei *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).

Ricerca ramNode

Il costo per la ricerca del *ramNode* è dato dal costo della ricerca dell'*hashNode:* 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 rappresenta il numero di *hashNode* presenti nella tabella hash
- m rappresenta il numero di liste di collisione della tabella

ed il rapporto di questi due valori rappresenta il fattore di carico α. Se il fattore di carico α 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).

6.8 Interfacce utilizzate

Esaminiamo adesso le interfacce utilizzate dai thread per richiedere un file al gestore della cache:

*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);

web_cache:	Indirizzo dell'istanza della cache
name:	Nome del file da richiedere (path)
ext:	Estensione del file da richiedere
x, y:	Risoluzione dell'immagine da richiedere
size:	Dimensione del file (serve per inviare correttamente il file al client)
file_type:	Flag per capire se il file richiesto è una immagine o un file html, se è un file html i parametri x,y, ext saranno ignorati

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* deve essere chiamata dalla componente che ha richiesto precedentemente il file, non appena ha finito di lavorare con esso. Questa seconda interfaccia è stata creata in seguito ad un problema che ci siamo posti:

Supponiamo che un thread t richieda un file A. Il gestore della cache fornirà il file A al thread t, e sposterà il file A all'inizio della coda LRU. Il thread t comincierà quindi a lavorare con quel file, e, molto probabilmente, lo invierà ad un client che ne ha fatto richiesta.

> Supponiamo che mentre il thread t stia inviando il file al client, il server riceva un numero molto grande di richieste di altri file.
>
> Il gestore della cache servirà dunque l'elevata mole di richieste di file da parte dei vari thread, e, piano piano, il file A finirà in fondo alla coda LRU, fino ad arrivare al punto in cui il gestore deciderà di rimuoverlo, facendo anche la munmap dell'indirizzo in memoria.
>
> In questo frangente di tempo è possibile che il thread t stia ancora inviando il file al client, e che quindi stia utilizzando la mappatura del file A in RAM, con l'impossibilità di eseguire correttamente il trasferimento od, in casi estremi, generando un Segmentation Fault.

È quindi necessario mantenere, per ogni file, un contatore di utilizzo, per sapere quanti thread stanno utilizzando quel file in un preciso momento. Quando viene chiamata la *obtain_file*, il contatore viene incrementato, quando invece viene chiamata la *release_file* viene decrementato.

Per ogni file in fondo alla coda LRU viene controllato tale contatore: soltanto se il suo valore è nullo viene eliminato il ramNode e liberata la relativa memoria.

Nonostante tale comportamento non sia né usuale né probabile è tuttavia possibile, perciò abbiamo prontamente implementato questa funzione di controllo.

7. Logger

7.1 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, di priorità massima.
Warning	**2**	Messaggi di avviso.
Info	**4**	Messaggi a puro scopo informativo.

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:

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

7.2 Implementazione

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 (o soprattutto in caso di crash improvviso dell'applicazione) 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.

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
  - Per scrivere i messaggi abbiamo utilizzato la funzione *write* e non la *printf/fprintf* (od altre funzioni bufferizzate simili) per evitare appunto di avere una scrittura bufferizzata
- Dopo aver svuotato la coda, il garbage thread torna in stato di IDLE, in attesa di nuovi inserimenti

7.3 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: acquisizione del lock della coda, inserimento del messaggio nella coda, invio di un segnale di "sveglia" al garbage thread, rilascio del lock della coda.

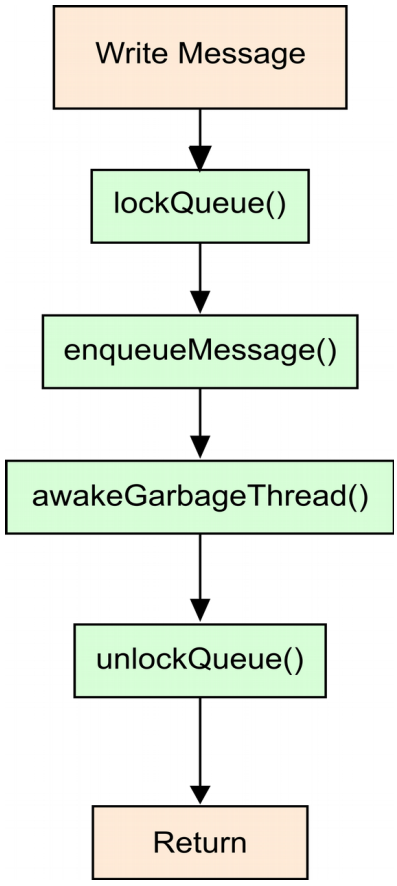


Figura 1: Scrittura sul log vista da un thread esterno.

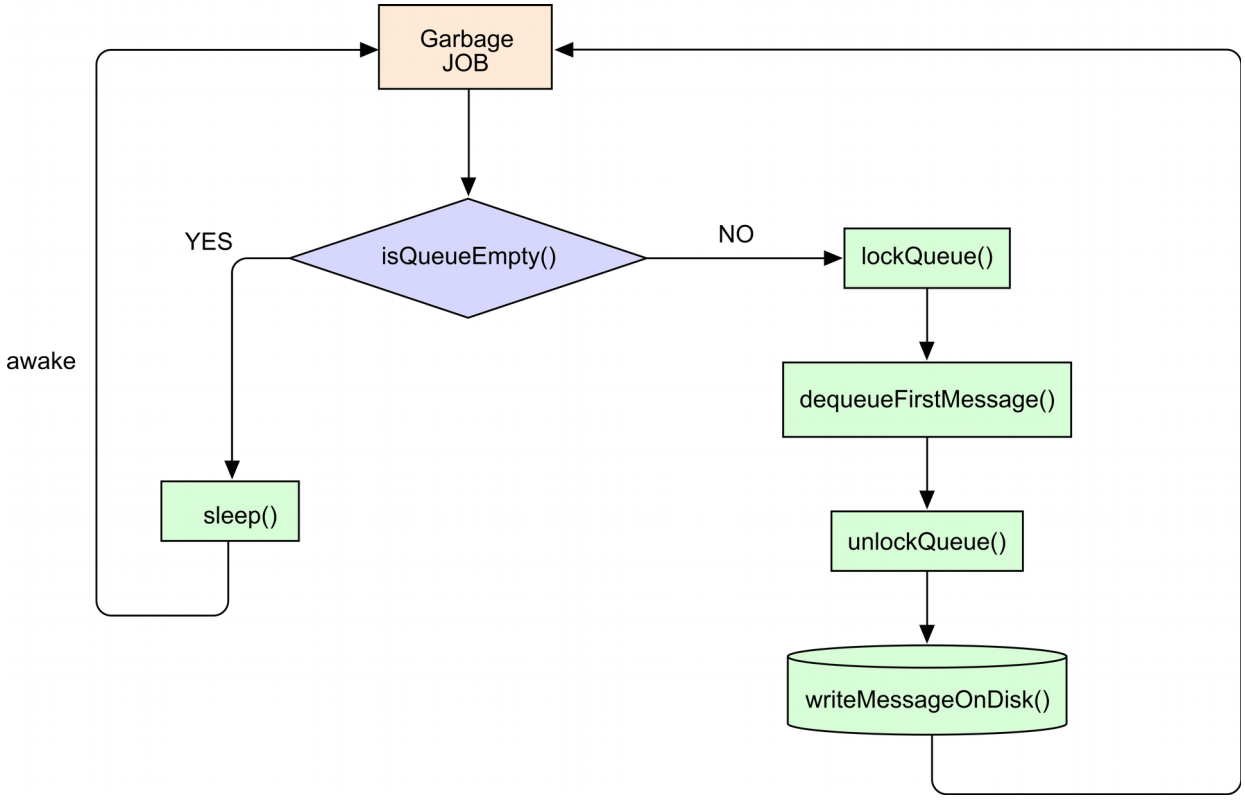


Figura 2: Ciclo infinito del garbage thread.

7.4 Interfacce

L'interfaccia utilizzata dai thread per scrivere sul file di log è la seguente:

*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, con un numero illimitato di parametri)

8. Performance e benchmarking

8.1 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.

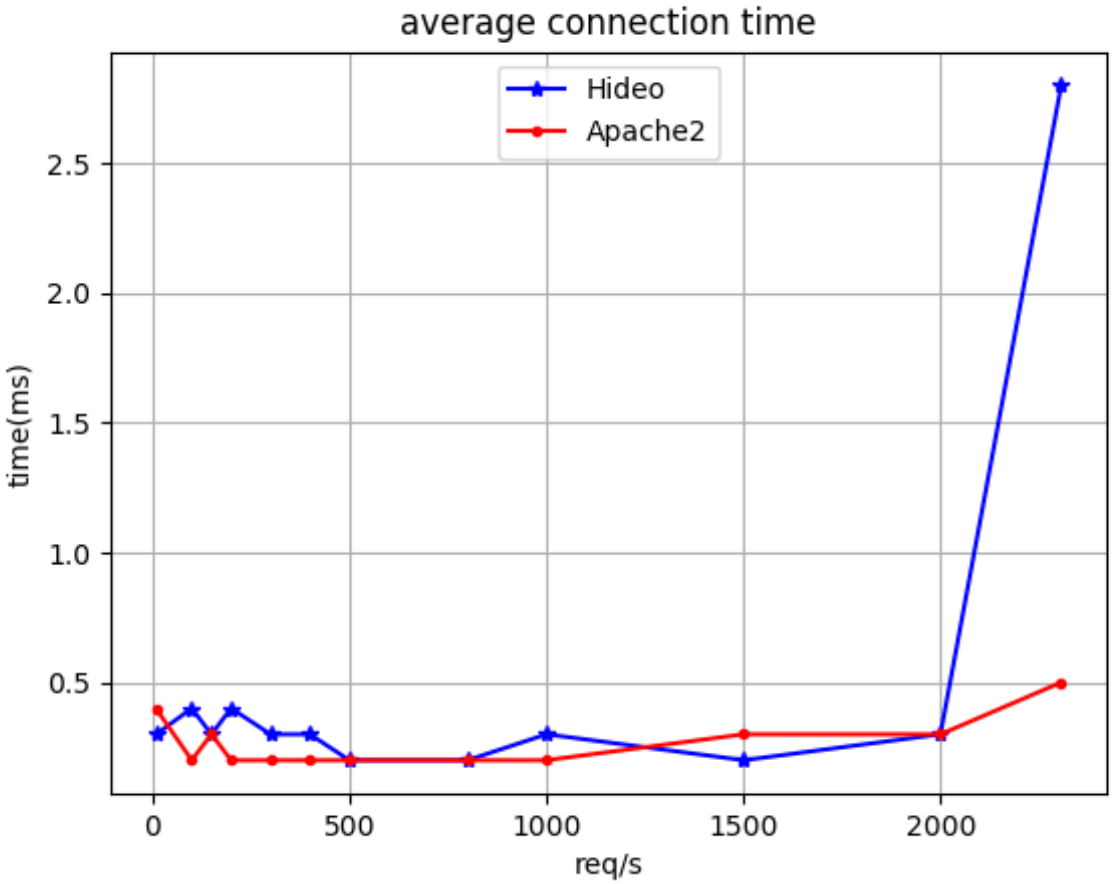
I test, effettuati con httperf, hanno evidenziato come il server riesca a servire senza rallentamenti, errori o warning fino 2305 connessioni al secondo.

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).

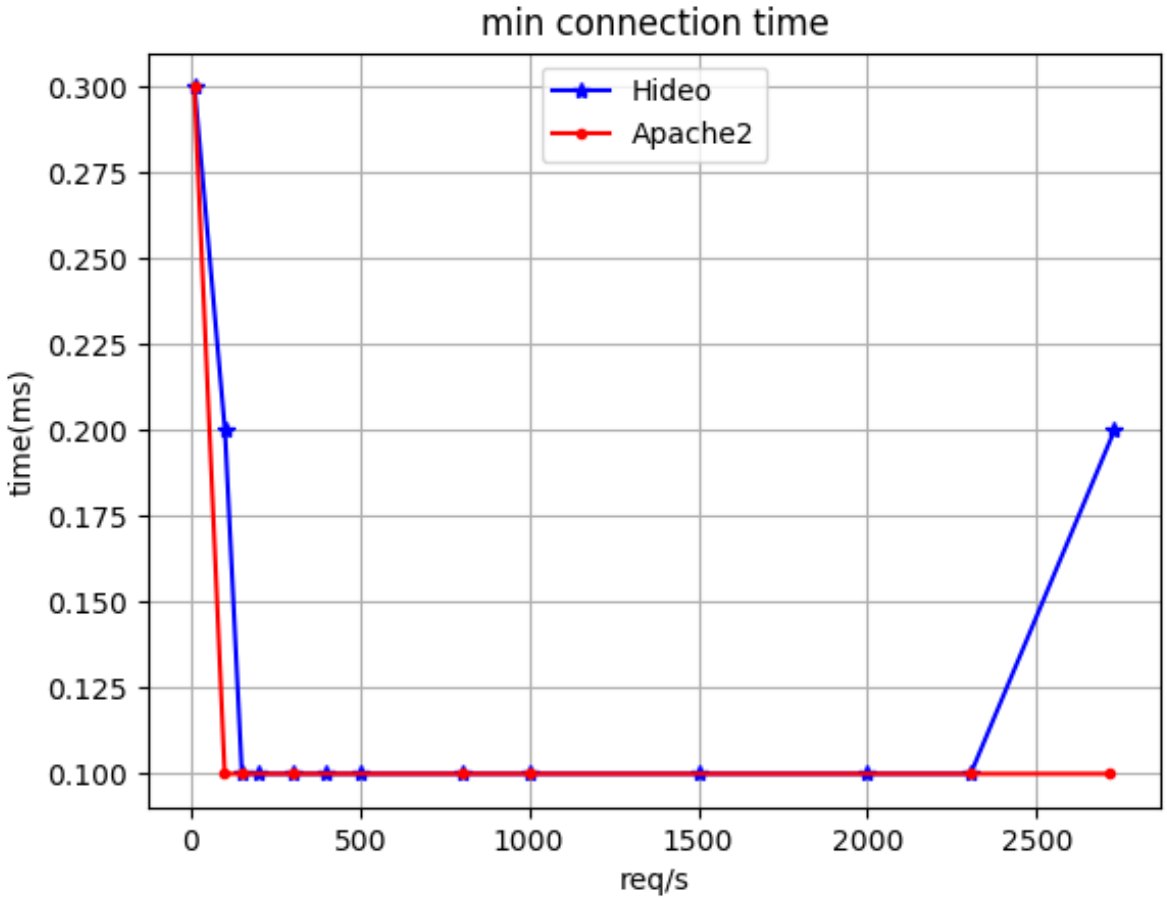
8.2 Test

Il plotting dei grafici è stato effettuato mediante la libreria Python MatplotLib e riassumono i risultati dei test, confrontandoli.

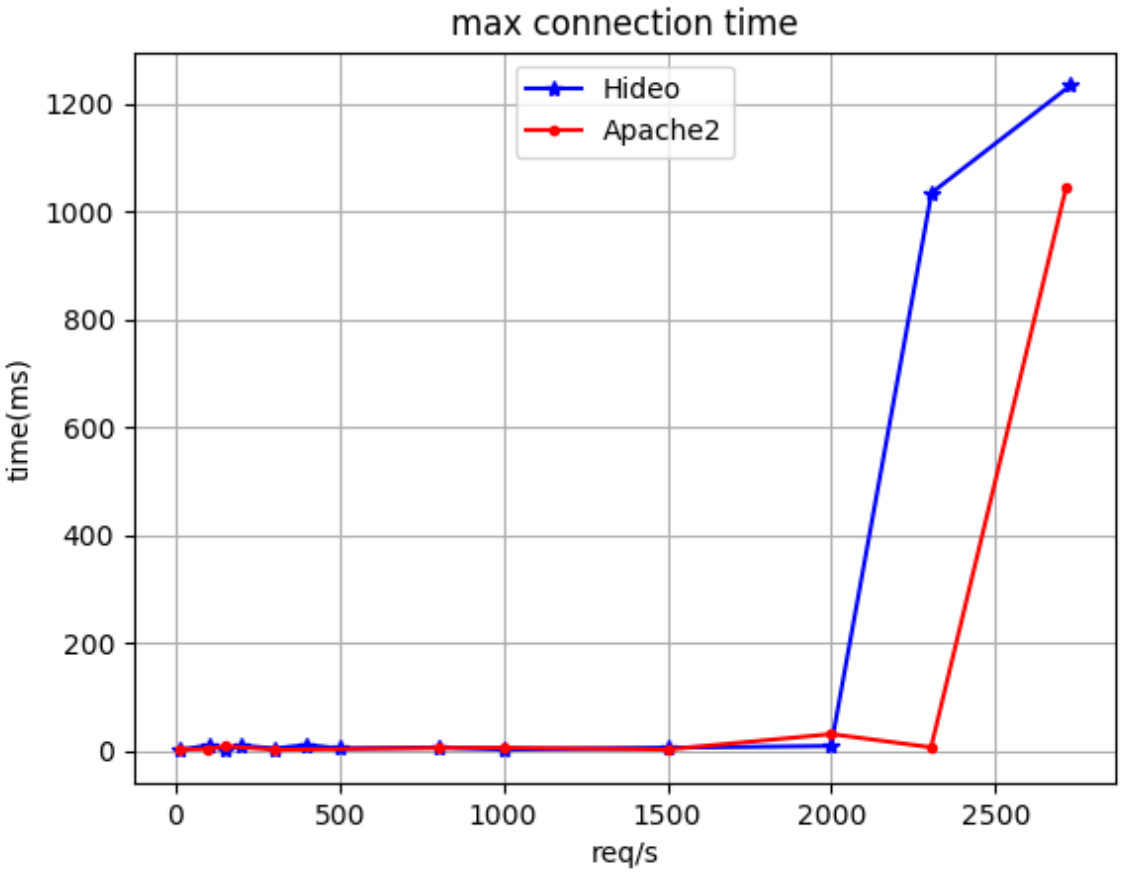
In ascissa è riportato il valore del tasso di arrivo, misurato in richieste al secondo (req/s); in ordinata, invece, è riportato il tempo corrispondente, misurato in millisecondi(ms); in blu le prestazioni di Hideo, in rosso quelle di Apache2.



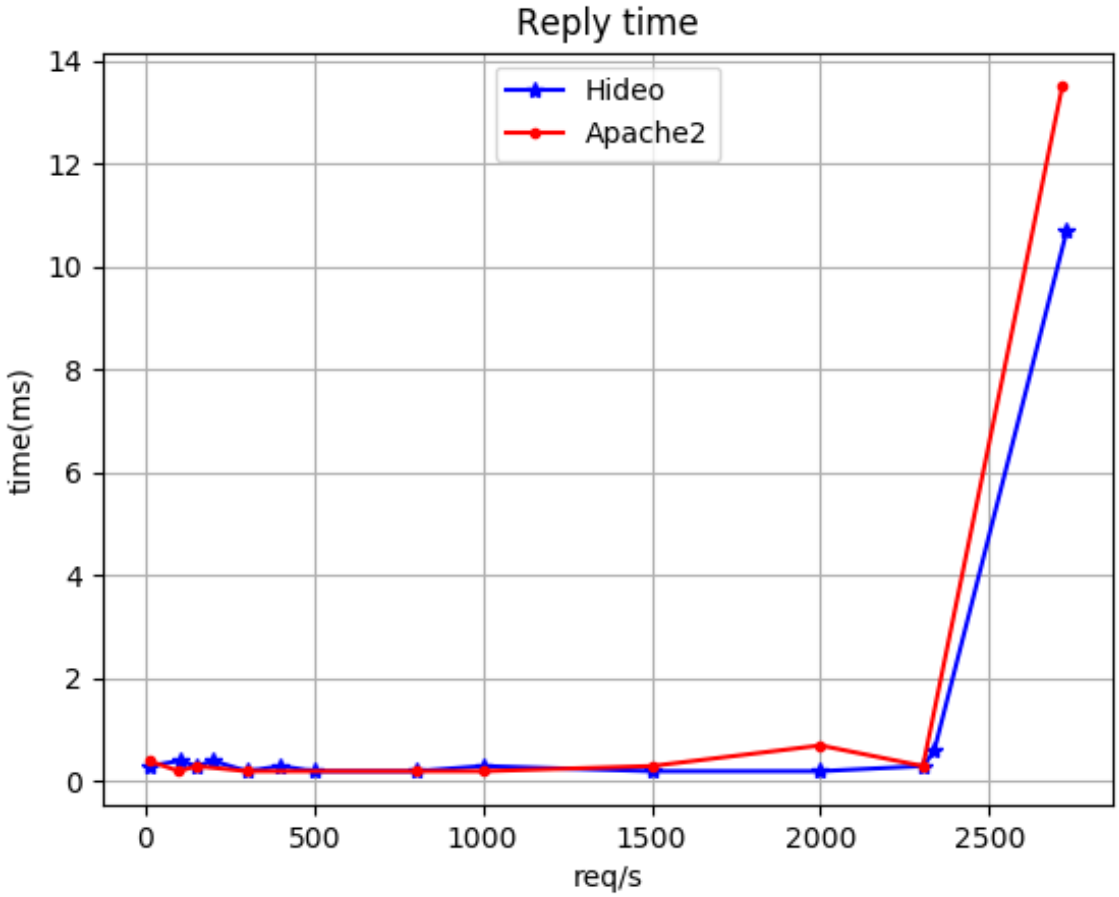
req/s	10	100	150	200	300	400	500	800	1000	1500	2000	2305
Hideo	**0.3**	**0.4**	**0.3**	**0.4**	**0.3**	**0.3**	**0.2**	**0.2**	**0.3**	**0.2**	**0.3**	**2.8**
Apache	**0.4**	**0.2**	**0.3**	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**	**0.3**	**0.3**	**0.5**



req/s	10	100	150	200	300	400	500	800	1000	1500	2000	2305	2730
Hideo	**0.3**	**0.2**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.2**
Apache	**0.3**	**0.1**	**0.1**	**N/A**	**0.1**	**N/A**	**N/A**	**0.1**	**0.1**	**N/A**	**N/A**	**0.1**	**0.1**



req/s	10	100	150	200	300	400	500	800	1000	1500	2000	2305	2730
Hideo	**0.5**	**10.2**	**1.7**	**9.5**	**3.7**	**10.5**	**3.9**	**5.6**	**1.8**	**5.3**	**8.4**	**1035**	**1235**
Apache	**2.1**	**3.0**	**8.2**	**N/A**	**1.6**	**N/A**	**N/A**	**4.8**	**5.3**	**2.0**	**30.5**	**6.0**	**1047**



req/s	10	100	150	200	300	400	500	800	1000	1500	2000	2305	2340	2730
Hideo	**0.3**	**0.4**	**0.3**	**0.4**	**0.2**	**0.3**	**0.2**	**0.2**	**0.3**	**0.2**	**0.2**	**0.3**	**0.6**	**10.7**
Apache	**0.4**	**0.2**	**0.3**	**N/A**	**0.2**	**N/A**	**N/A**	**0.2**	**0.2**	**N/A**	**N/A**	**0.3**	**N/A**	**13.5**

8.3 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.

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.

9. Utilizzo dell'applicazione

9.1 Prerequisiti

Per la compilazione del webserver Hideo sono necessari, oltre ai tool essenziali quali un compilatore (GCC) e gli header C di sistema, il pacchetto di header di WURFL nonché un relativo database.

Come già ripetutamente sottolineato, le ultime due componenti sono ottenibili esclusivamente su esplicita richiesta presso il sito della ScientiaMobile che, probabilmente, fornirà un pacchetto diverso a seconda della distribuzione dichiarata nel questionario, oppure a priori fornirà un .deb che andrà installato mediante dpkg (nelle distribuzioni basate Debian), convertito in .rpm (per le distribuzioni basate RedHat) oppure decompresso ed installato manualmente.

Per quanto riguarda il nostro ambiente di lavoro, un'installazione vanilla di Ubuntu Server LTS, è stato sufficiente installare il pacchetto *build-essentials* attraverso il gestore di pacchetti APT ed il sovracitato .deb di WURFL mediante dpkg.

9.2 Compilazione e configurazione

Le frequenti ricompilazioni operate nel corso dello sviluppo dell'applicativo hanno reso necessario la stesura di script per velocizzare le operazioni di compilazione e pulizia. Invece di ricorrere a soluzioni autoprodotte abbiamo sfruttato le potenzialità di *make* ed abbiamo preparato un *Makefile* che rende disponibili le operazioni di *make* e *make clean.*

La configurazione del server può essere eseguita modificando opportunamente il file **server.cfg** attraverso i parametri (tra parentesi i valori di default in caso di mancata dichiarazione)

- **server** Numero di porta sulla quale rimanere in ascolto [5700]
- **threads** Numero di thread del pool statico [10]
- **backlog** Numero massimo di connessioni accettate in coda [64]
- **loglvl** Tipologia di logging richiesta (cfr Cap. 7.1) [7]

9.3 Esecuzione

Una volta compilato e configurato, il server è pronto per essere eseguito, sufficientemente eseguendo, preferibilmente da terminale, l'eseguibile *server* mediante il comando *./server.*

Il server sarà così 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*).

10. Conclusioni

Come team ci possiamo ritenere soddisfatti del lavoro svolto, in particolare per la scelta di soluzioni adottate per lo sviluppo del progetto, come la cache e il logging.

Abbiamo spoesso pensato ad alcune possibili ottimizzazioni implementabili, ed in particolare avremmo voluto codificare un metodo per la conversione parallela delle immagini all'interno di uno stesso thread, poiché attualmente il thread risulta bloccato in attesa della consegna dei file convertiti da parte del modulo conversioni.

Un'ulteriore miglioramento potrebbe consistere nel rendere il pool di thread dinamico, e quindi aumentare o diminuire all'interno di un range il numero di thread in base al carico di sistema, alle richieste, ed alle prestazioni percepite.