Web Server

con adattamento di contenuti statici

a cura di

Giorgia Marchesi

Università di Tor Vergata a.a 2017-2018

Indice

1. Introduzione

Un *web server* è un applicativo software che gira su un server fisico, in ascolto su porte dedicate, avente il compito di comunicare con i client - browser web - che richiedono i suoi contenuti e i suoi servizi. Il server utilizza il protocollo HTTP e le pagine servite sono documenti HTML che possono includere riferimenti a immagini e testo.

La richiesta effettuata dal client viene tradotta in una richiesta HTTP e viene consegnata al server indicato. Il server, dopo aver elaborato la richiesta, invia una risposta HTTP e un file HTML che viene poi tradotto sotto forma di testo e immagini.

L'obiettivo di questo progetto è la realizzazione di un *web server* con supporto minimale del protocollo HTTP/1.1 implementato in linguaggio C utilizzando le API della socket Berkeley, avente il fine di offrire agli utenti la possibilità di visualizzare immagini - scelte tra quelle elencate nella home page del server - adattandole dinamicamente in base alle caratteristiche del dispositivo richiedente.

Sarà il server ad occuparsi della conversione in base al fattore di qualità specificato dal client nell'header Accept e ad inviare poi l'immagine così ottenuta salvandola in cache consentendo di diminuire i tempi di risposta in caso di successive richieste della stessa nonché l'overhead computazionale dovuto alla conversione.

1.1 Scelte progettuali

Per il server è stata scelta un'architettura *monoprocesso-multithread* capace di gestire più connessioni simultaneamente creando un nuovo thread per ogni richiesta di connessione. Il server inoltre offre i servizi di logging per tracciare le richieste prese in carico e le relative risposte da parte del server e un servizio di caching per gestire più velocemente le richieste di adattamento dei file rispetto al dispositivo.

Le decisioni effettuate in fase progettuale hanno avuto come obiettivi principali quelli di garantire la semplicità implementativa e la minimizzazione dei tempi di risposta nel limite delle possibilità dell'architettura scelta.

Note: all'interno del modulo *threadmanager.h* sono presenti in parte le funzioni per la creazione di un pool dinamico, che tuttavia per questioni di tempo non sono state effettivamente integrate. Lo stesso vale per i moduli per l'installazione di WURFL, che offre delle API per reperire informazioni sul dispositivo mobile a partire dallo User-Agent, anch'esse non testate a causate del mancato riconoscimento di alcuni header file nella libreria libxml2 (in particolare sax2.h e gli header relativi al parsing dell'xml).

1.2 Caratteristiche di rilievo

Configurazione: l'header file *configuration.h* consente di impostare manualmente i valori di riferimento del server – numero di porta e numero massimo di connessioni – e i path di destinazione per il salvataggio delle risorse - immagini per il web server, cache e log - nonché ulteriori impostazioni secondarie.

Caching: servizio offerto per il salvataggio su disco delle immagini su cui è stata attuata la conversione, consentendo di velocizzare un eventuale futuro reperimento delle stesse, senza passare per i servizi di ImageMagick. Avviene utilizzando una *hashmap* la cui chiave viene costruita sulla base di alcuni valori (qualità, altezza, lunghezza, nome originale del file) manipolati secondo un algoritmo personalizzabile e il cui valore associato è il nuovo path del file normalizzato sulla base dei valori per cui è stata eseguita la conversione. Dunqua la presenza o meno del percorso del file serve ad indicare se l'immagine o no e se quindi sarà necessario o meno ripetere l'operazione di conversione, guadagnando quindi in prestazioni.

Loggin: tiene traccia del flusso di richieste e risposte che circolano tra i client e il server.

1.3 Moduli

Il web server è suddiviso in vari moduli, ciascuno dei quali svolge un compito specifico. I vantaggi di questa metodologia di sviluppo sono vari: i servizi dei singoli moduli sono facilmente utilizzabili all'interno di altri moduli tramite import degli stessi; inoltre si ha una maggiore leggibilità del codice e semplicità di manutenzione, consentendo agevolmente di aggiungere nuove funzionalità in base alle necessità.

I moduli possono essere suddivisi in 3 gruppi principali:

Moduli per le *funzionalità e i servizi del server*:

- **init_server.c:** si occupa della configurazione del server, del settaggio della porta di ascolto e del tipo di connessione da instanziare;
- **http_request.c, http_response.c:** il primo si occupa del parsing delle richieste ricevute, il secondo dell'invio delle risposte;
- **log.c:** implementa il servizio di logging;
- **cache.c:** implementa il servizio di caching.

Moduli per la *gestione della concorrenzialità* per la programmazione multithread.

- **thread_manager.c:** implementa le strutture dati e i metodi necessari alla creazione delle risorse necessarie ai thread del server per eseguire i loro compiti (include anche i metodi per la creazione del pool);
- **thread_connection_job.c:** include i metodi necessari ai thread per gestire effettivamente la connessione e portare a compimento le richieste dei client.

Moduli di supporto per la *gestione dei file e della memoria*

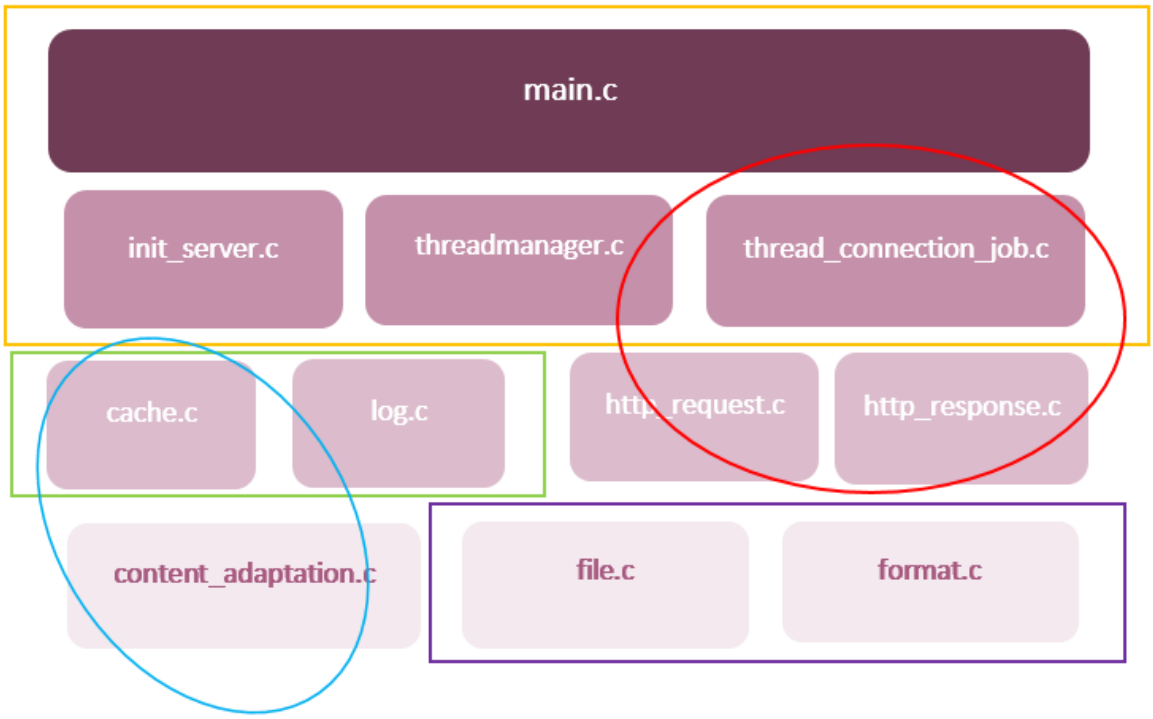
- **file.c** : gestione dei file
- **format.c:** gestione dell'allocazione della memoria dinamica e conversioni di formato

- util.c_ : ideata per la gestione di wurfl (non funzionante)

Infine:

- main.c: modulo di avvio che coordina l'inizializzazione di tutte le risorse utili al sistema e dei servizi e che si occupa dell'istanziazione delle connessioni e l'accettazione delle stesse.

Nell'immagine sottostante viene presentato il grafico raffigurante la gerarchia dei servizi dell'applicazione, in particolare si vuole mostrare come i livelli sottostanti offrano servizi a quelli di livello superiore, mettendo in evidenza alcune delle particolari interconnessioni tra i vari moduli (le spiegazioni si possono trovare nella leggenda).



Legenda:

- moduli che si occupano dell'esecuzione del server: inizializzazione e gestione delle connessioni. Il main richiama init_server per inizializzare la socket di ascolto e threadmanager.c per inizializzare i dati su cui lavoreranno i thread. Il modulo thread_connection_job.c viene invocato alla creazione dei thread non appena viene stabilita una nuova connessione.
- servizi offerti dal web server
- thread_connection_job.c invoca i moduli sottostanti per poter parsare le richieste e inviare i file.
- content_adaptation.c viene utilizzato da cache.c per convertire l'immagine sfruttando i servizi di ImageMagick.
- moduli per le operazioni di base

2. Comportamento del sistema

2.1 Inizializzazione

All'avvio del programma vengono istanziate tutte le risorse necessarie al corretto funzionamento dell'applicazione, comprese le strutture dati per la cache e per il log. Infine viene inizializzata la socket, che rimane in ascolto sulla porta indicata nell'header file *configuration.h* e riconfigurabile manualmente a piacere.

2.2 Multithreading

Terminata l'inizializzazione l'eseguibile entra in un loop infinito nel quale accetta le connessioni dalla socket d'ascolto e per ognuna di queste spawna un worker thread che si occuperà di gestire effettivamente la connessione con i clients, mantenendola aperta fino alla scadenza del timeout o finché il client non effettuerà più richieste.

2.3 Processamento della richiesta

Il thread dopo aver inizializzato la sua struttura dati *thread_node* continua a leggere dalla socket in attesa di richieste da parte del client. All'arrivo dei dati, effettua il parsing del messaggio di richiesta sia per controllarne l'effettiva validità sia per discriminare i metodi GET e HEAD.

In caso la richiesta non sia valida il thread procede all'invio del messaggio di errore, altrimenti invoca i metodi necessari a prelevare i dati utili al proseguimento del processamento della richiesta tra cui:

- File richiesto
- User agent - - > nome dispositivo
- Accept - - > fattore di qualità

In questo progetto non è stato effettivamente utilizzato il risultato ottenuto parsando il messaggio contenuto nello User-Agent header poiché non è stato avviato wurfl. Invece dall'Accept è stato prelevato il fattore di qualità desiderato dal client.

Qualora il file richiesto fosse l'*index.html* il client verrà immediatamente reindirizzato alla home page. Qualora invece la richiesta riguardasse un'immagine, interverrà il modulo cache che controllerà se nella cache sia già presente il file procedendo all'invio, altrimenti effettuerà la conversione e inserirà il path del file in cache. Il thread procederà quindi all'invio del file presso il client e tornerà in ascolto di nuove richieste.

2.3 Inoltro del contenuto

Dopo aver terminato le operazioni in cache, l'applicativo si occupa di inoltrare una risposta al client: verrà inviato il solo header nei casi in cui si verifichi:

- Un errore nella richiesta → 405 Method not allowed
- Il file non sia presente → 404 Not Found
- Il client specifichi nell'HTTP request header il metodo HEAD

In tutti gli altri casi viene inviato lo status header 200 OK e il body contenente i dati richiesti dal client.

2.4 Chiusura della connessione

La connessione verrà chiusa o alla esplicita disconnessione del client o allo scadere del timeout di connessione settato all'avvio di ogni thread worker tramite il metodo setsockopt().

3 Adattamento dell'immagine – ImageMagick

Per convertire le immagini è stato utilizzato il comando *convert* della raccolta di programmi per la manipolazione di immagini fornita da ImageMagick.
Il comando viene lanciato tramite la *popen()* che si incarica di aprire un processo creando una pipe, eseguendo fork() e invocando la shell cui vengono passati sia il comando da eseguire sia i relativi argomenit:

convert –quality /path/originalimage /path/newimage

Lo stream I/O creato dalla popen viene in seguito chiuso tramite *pclose()* che assume un comportamento simile alla wait(), infatti il thread invocante rimane bloccato in attesa che il processo associato termini prima di poter riprendere il controllo.

3.1 Osservazioni

Il vantaggio dell'utilizzare la popen() è che le conversioni non sono a carico del thread worker ma di altri processi creati 'ad hoc'.

Lo svantaggio è che il thread che lancia il comando dovrà comunque rimanere in attesa del completamento della conversione prima di poter continuare la sua esecuzione, tuttavia questa scelta ha consentito di evitare complicazioni nel codice mantenendolo semplice, lineare e facile da controllare.

4 Architettura dell'applicazione e implementazione

4.1 Server

Il server è organizzato in due componenti principali:

- Il main che si occupa della gestione della socket TCP accettando le nuove connessioni;
- N thread che si occupano della gestione di ogni connessione, uno per ogni nuova connessione che arriva.

Osservazioni

La creazione di un thread per ogni nuova richiesta di connessione è dispendioso in termini di tempo di risposta e di connessione poiché ogni volta devono essere allocate tutte le risorse necessarie al thread. Tuttavia è essendo un ottimo compromesso per facilità implementativa e manutentiva.

Una possibile implementazione a cui si potrebbe pensare al fine di migliorare i tempi di risposta, consiste nel pre-allocare le risorse: si potrebbe costruire un pool di thread - statico o dinamico - ed eseguire prethreading all'avvio dell'applicazione, guadagnando tempo per la gestione delle connessioni. In tal caso diventa importante la taglia del pool: bisognerebbe infatti tenere conto delle caratteristiche del calcolatore ed eseguire dei test per decidere la dimensione ideale, che nel caso dinamico potrà essere modificata in base al carico del sistema.

4.2 Main

Inizializzate le varie strutture per il logging e il caching e tutte le risorse utili, si passa all'inizializzazione e gestione della socket TCP sulla porta specificata nel file di configurazione. Per ogni connessione in entrata il modulo principale istanzia e riempie coi dati necessari la struttura dati del thread che può iniziare la sua esecuzione.

4.3 Thread connection job

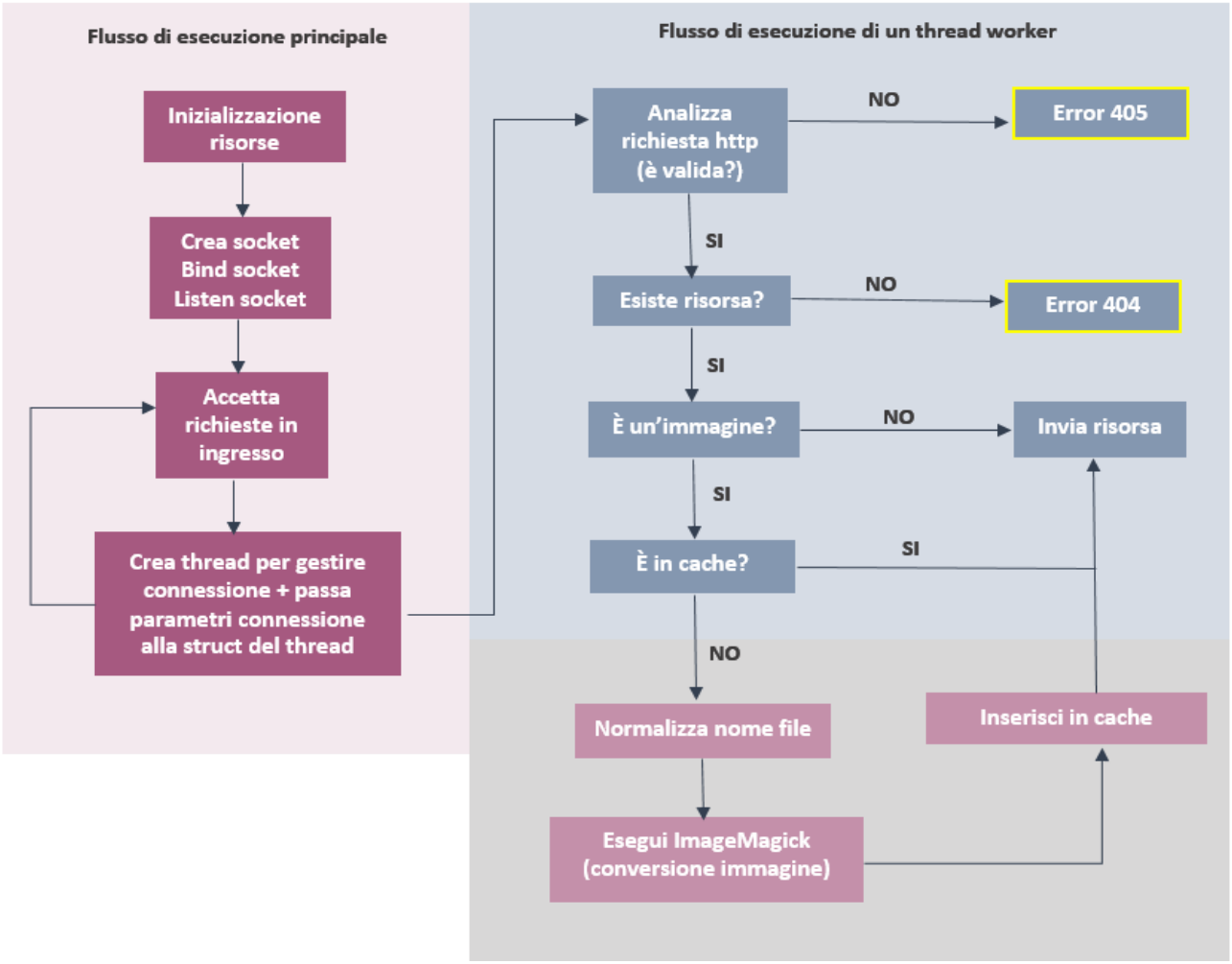
Ogni thread una volta creato passa ad eseguire la funzione *accept_connection()*, dove serve in loop il client connesso. Non appena questo si disconnette o scade il timeout, il thread viene distrutto.

Pseudo codice accept_connection()

```
while(client is connected)
{
        parse_request();
        if( file is not valid) send_not_implemented;
        else if(file is index.html) send_file;
        else{
                if(!image is in cache){
                        normalize_name();
                        convert();
                        insert_in_cache();

                } send_file;
        }
}
```

- All'arrivo di una nuova richiesta si scrive un messaggio nel file del log contenente informazioni sul chi ha inviato la richiesta, quale thread se ne sta occupando e l'header ricevuto.
- Si individua il metodo della richiesta HTTP accettando solo i metodi GET e HEAD;
- Una volta individuato il file richiesto, se è l'index.html si procede immediatamente al suo invio, altrimenti, trattandosi di un'immagine, si prelevano gli ulteriori dati necessari alla conversione;
- Vengono parsati i campi User-Agent e Accept e prelevati i valori di interesse;
- Prima di effettuare la conversione si controlla se il file è presente nella cache chiamando la funzione *hashmap_search_item()* specificando il nome del file, il fattore di qualità e se presenti la larghezza e l'altezza dell'immagine;
- Se l'immagine è presente in cache si procede all'invio del file;
- Se l'immagine non è presente di procede si converte l'immagine standardizzando il nome originale del file , si inserisce il file in cache ed infine si invia il file;
- Al termine della connessione il thread viene distrutto.



5 Cache

La cache memorizza nella directory CACHE_DIR (il cui path è configurabile in configuration.h) i file convertiti tramite ImageMagick. In RAM viene mantenuto il path del file convertito, qualora presente in cache, in modo tale da non ripetere nuovamente la procedura di conversione dell'immagine ogni volta che viene richiesta.

5.1 Strutture dati

Il caching è stato implementato utilizzando una struttura HashMap:

- Una tabella Hashmap contiene entry di tipo HashmapNode
- Un HashmapNode rappresenta un file presente su disco: contiene informazioni sulla sua locazione in memoria secondaria e sul numero di accessi.

5.2 Inserimento di una entry

Inizialmente la tabella hash è vuota: la prima richiesta da parte di uno dei thread genererà un cache miss e di conseguenza non sarà presente nemmeno il file fisico, pertanto si eseguiranno le seguenti operazioni:

- Si normalizza il nome del file da inserire in base ai parametri specificati per poter poi inserire il file nella directory della cache e controllarne agevolmente la presenza. Il formato previsto è:

  ./CACHE_DIR/filename_q[_w_h].format (i parametri tra parentesi quadre possono non essere definiti)

- Si converte quindi il file invocando i metodi presenti di *content_adaptation.c* che richiama i servizi offerti da ImageMagick per la manipolazione delle immagini, passando come path di destinazione quello appena generato tramite standardizzazione.

- Si crea la chiave associata al file secondo un algoritmo personalizzabile. Nel progetto essendo poche le immagini e tutte con nomi diversi e quindi dovendo teoricamente essere assai raro il verificarsi di una collisione, si è deciso di calcolare la chiave in questo modo:

  valore numerico associato al primo carattere del nome del file + [(q x w) / h]

  Questa decisione è stata presa per evitare collisioni in caso w = h per lo stesso file.
- In seguito si crea un HashmapNode, avente come indice la chiave realizzata tramite la funzione *hashmap_func()* e come valore il nome normalizzato del file, inserendolo poi nella tabella Hashmap.

5.3 Richiesta di un file al gestore della cache

Supponiamo di richiedere un file X:

- Il thread worker esegue una ricerca nella tabella hash per controllare se sia presente il nodo con indice hash passando come parametri di ricerca il nome del file originale e i valori di q, w e h;
- Ricostruisce quindi la chiave a partire dai parametri passati;
- Se si verifica un cache hit si preleva il valore associato, ovvero il path del file nella cache directory. In tal caso non sarà necessario ripetere l'operazione di conversione e si potrà procedere direttamente all'invio del file;
- Se invece si verifica un cache miss allora l'HashNode relativo al file X non esiste, il che implica che nemmeno il file sia presente su disco, pertanto si effettuano le operazioni di inserimento di una entry (vedi paragrafo 5.2).

Osservazioni

Scegliere come struttura dati della cache una tabella hash ha consentito di garantire l'inserimento e la ricerca di un nodo in tempo O(1). Tuttavia si penalizza la ricerca non per chiave che richiede la scansione di tutti gli HashNode, compromesso accettabile essendo un'operazione eseguita molto meno frequentemente.

Inoltre essendo una struttura dati memorizzata in RAM consente di sapere in tempi rapidi quali file siano o meno memorizzati su disco evitando ad ogni richiesta una ricerca del file su disco secondario o la conversione tramite ImageMagick.

5.4 Pulizia della cache

Essendo la cache una risorsa limitata viene pulita a determinati intervalli di tempo attuando una politica di cancellazione del file meno utilizzato grazie all'utilizzo di un contatore *usage* e di primitive di sincronizzazione. Schematicamente se ne illustra il funzionamento:

- All'inserimento di un nuovo nodo nella Hashmap, il contatore usage viene impostato ad 1;
- In caso di accesso al file il contatore viene incrementato di una unità;
- Ogni X secondi un thread creato 'ad hoc' all'avvio dell'applicazione per la pulizia della cache, controlla quale file sia il meno utilizzato, quindi lo rimuove dalla directory dove sono salvati i file convertiti tramite ImageMagick e resetta i valori del nodo hash. Se non ci sono più file da eliminare, torna in stato di attesa.

6 Logging

6.1 Formato messaggi

Il log tiene traccia delle operazioni del server, in particoalre le richieste effettuate dai client, le risposte inviate dal server e file rimossi dalla cache durante le operazioni di pulizia. I messaggi vengono salvati nella directory LOG (la cartella di destinazione è modificabile accedendo al file configuration.h).

Ogni volta che un thread riceve una richiesta da parte di un client, crea un nuovo messaggio da inserire nel log avente questo formato:

[REQUEST]
Client [IP_NUMBER] on port [PORT_NUMBER] to thread [THREAD_T TID] on socket [SOCK_NUMBER]
asks:
HEADER DI RICHIESTA

```
[REQUEST]
client [127.0.0.1] on port [62108] to thread [139652471838464] on socket [4]
asks:
GET /santorini.jpg HTTP/1.1
Host: 127.0.0.1:5231
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
If-Modified-Since: Sat, 22 Sep 2018 15:08:40 GMT
```

In questo modo è possibile sapere chi ha inviato il messaggio, quale thread si sta occupando della sua gestione e su quale socket è arrivato, per completezza si include anche l'header di richiesta.

Quando il thread processa effettivamente la richiesta viene evaso un ulteriore messaggio con struttura simile al precedente:

[ANSWER] to client [IP_NUMBER][PORT] on socket [4]
File richiesto
[date]
[header status]

```
[ANSWER] to client [127.0.0.1][62108] on socket [4]
file /santorini.jpg
Date: Sat, 22 Sep 2018 15:11:32 GMT
HTTP/1.1 200 OK
```

Per quanto riguarda la cache invece il formato è il seguente: [CACHE] file rimosso

6.2 Implementazione

Per quanto riguarda la trascrizione dei messaggi sul log, si è preferito non affidare il compito ai thread che si occupano della connessione ma bensì ad un thread creato ad hoc, per non rallentare ulteriormente la gestione delle connessioni: infatti le operazione di scrittura su disco sono onerose e lente e andrebbero ad inficiare sulle prestazioni dei thread.

Pertanto i thread si appoggiano ad un buffer statico precaricato in memoria all'avvio dell'applicazione, che server ad immagazzinare i messaggi in attesa di essere effettivamente scritti su file.

Il buffer è implementato come una lista collegata ed è gestito secondo la logica del produttore-consumatore (tanti produttori – i thread lavoratori - ed un solo consumatore – il thread del log -), regolato da primitive di sincronizzazione per evitare corruzione dei dati.

Il thread del log rimane in attesa per X secondi, dopodiché si sveglia per prelevare i dati dalla lista e scriverli sul log.txt.

Quindi all'avvio dell'applicazione:

1. Viene inizializzato il mutex per la scrittura nel buffer, utile per evitare che l'indice del buffer non venga aggiornato correttamente;
2. Viene creato il buffer e settato l'indice a 0;
3. Viene spawnato il thread che si occupa della scrittura su file.

6.3 Inserimento e sincronizzazione

Supponiamo ora che il thread A voglia inserire un messaggio e chiamiamo TL il thread che si occupa di scrivere su disco. Supponiamo inoltre che TL sia ancora nello stato di sleep():

- Il thread A procede a formattare il messaggio (vedi paragrafo 6.1)
- Il thread A chiama la funzione *write_on_buffer()*:
    - Prende il lock
    - Inserisce il messaggio
    - Aggiorna l'indice del buffer
    - Rilascia il lock
- Passati X secondi, il TL si sveglia:
    - Prende il lock
    - Scorre il buffer dall'inizio fino all'indice del buffer
    - Riazzera l'indice
    - Rilascia il lock

Osservazioni

Se il carico della rete diventa elevato, l'inserimento dei messaggi nel buffer può diventare un collo di bottiglia. Per questo motivo si è deciso di annullare sia l'inserimento nel buffer sia la scrittura del file del log quando la lista è piena oltre l'80%. Infatti si ritiene che oltre questa soglia il thread del log non riuscirebbe a smaltire abbastanza velocemente le scritture e che quindi anche i thread che si occupano della connessione rimanendo per un tempo maggiore in attesa del lock, non riuscirebbero a gestire in modo efficiente le richieste.

7 Ambiente di sviluppo

Il progetto è stato sviluppato su sistema operativo Ubuntu (64-bit) montato su macchina virtuale Oracle VM VirtualBox.

Il computer ha le seguenti caratteristiche:

- Sistema operativo: Windows 8.1 Pro
- Processore: Intel® Core™ i7-3632QM CPU @ 2.20GHz
- RAM: 8.00 GB
- Tipo sistema: 64 bit

L'utilizzo della macchina virtuale è stato costretto a causa della difficoltà di installazione della partizione, resettata a ogni avvio del sistema. L'utilizzo della macchina virtuale ha comportato un generale rallentamento del sistema nonché un calo delle prestazioni in fase di test.

L'IDLE di programmazione utilizzato è CLion.

8 Analisi delle performance

8.1 Premesse

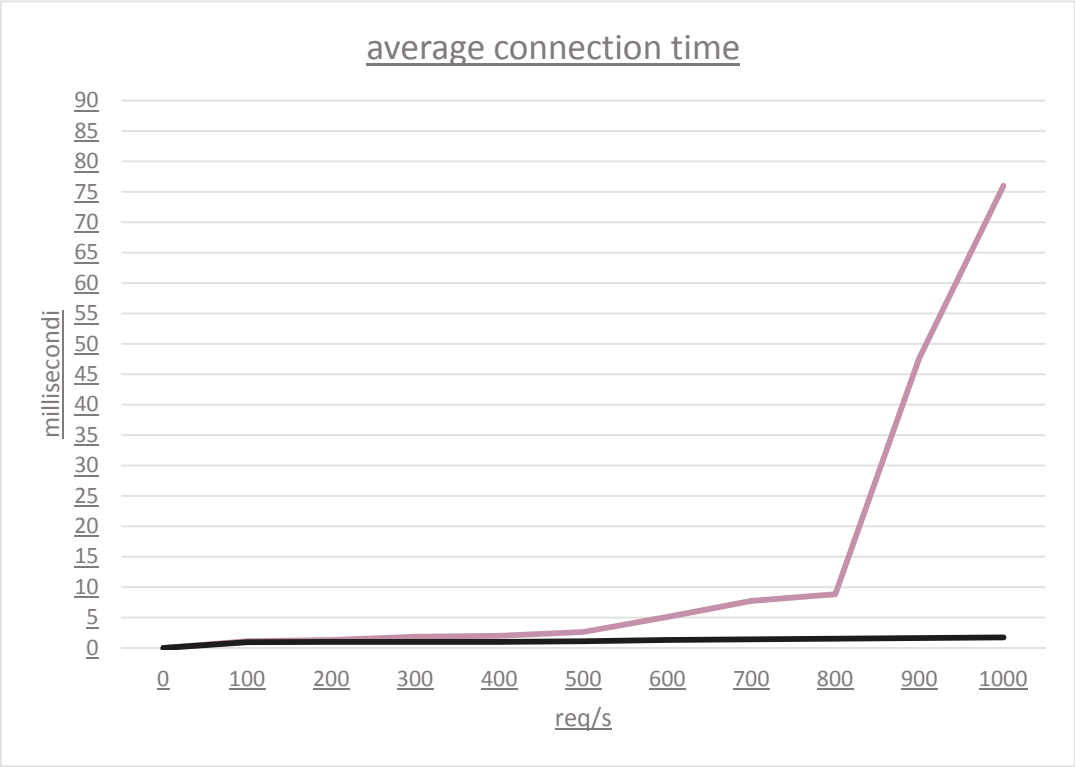
Sono stati eseguiti dei test preliminari per comprendere i limiti del server. Quello che si è notato è che oltre le 1015 connessioni al secondo il server inizia a generare errori e che alcune connessioni non vengano correttamente elaborate, vengano rifiutate o scandano in timeout.

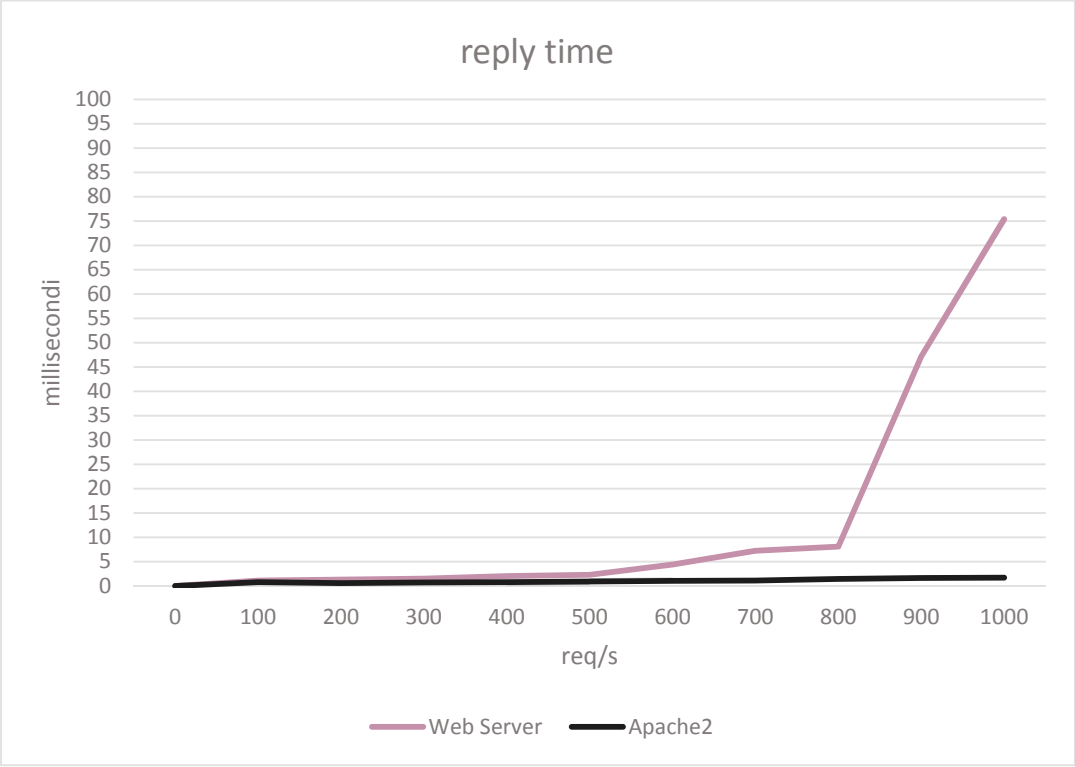
Si è arrivati a testare 30k connessioni e non si sono riscontrati errori fino alle 1009 req/s, pertanto il rate massimo supportato si aggira è attorno alle 1000 req/s, presentando tuttavia notevoli ritardi rispetto ad Apache2.

Durante i test si è notata inoltre una grande varianza sui risultati ottenuti, per questo motivo, non essendo agevole effettuare una media, si è preferito prendere come parametri i valori migliori rilevati durante le misurazioni (per ogni rate in media sono stati eseguiti 10 test).

8.2 Test

I grafici riportano in ascissa il valore del tasso di arrivo misurato in richieste al secondo (req/s) e in ordinata il tempo corrispondente misurato in millisecondi (ms). In viola viene tracciato l'andamento del server implementato, in nero quello di Apache2.





8.3 Analisi dei test

Come si può notare dai grafici l'andamento del server implementato tende ad essere lineare fino a 800 req/s per diventare poi esponenziale, a differenza di quello di Apache2 che rimane costante.

Il fatto che il computer in presenza di macchina virtuale presenti un notevole calo delle prestazioni e quindi un rallentamento generale del sistema è una delle possibili ipotesi sul perché di tale discrepanza.

Inoltre Apache2 è un server multi-processo multi-thread e quindi in grado di gestire un numero molto più consistente di connessioni, garantendo robustezza e stabilità, a differenza di quello implementato che essendo mono-processo impone un vincolo sul numero di thread generabili e quindi sul numero di connessioni gestibili.

Ulteriore motivo del gap tra i due server è dovuto al fatto che l'allocazione delle risorse per i thread, la creazione dei thread e l'inizializzazione delle loro struttura dati avvengono ad ogni nuova connessione, quindi ripetendosi un numero considerevole di volte, il tempo di completamento di tutte queste operazioni richiede molto tempo, inficiando sui tempi di gestione della connessione.

Si ritiene dunque che tramite pre-threading, quindi precaricando le risorse all'avvio, e col supporto di una lista di thread che può essere incrementata in base al carico (eseguendo quindi un minor numero di operazioni di creazione, allocazione e associazione) e snellendo il codice si possano ottenere risultati migliori nei test, più lineari e paragonabili ad Apache2.

9. Utilizzo dell'applicazione

9.1 Prerequisiti

Per la compilazione dell'applicazione sono necessari il compilatore gcc e gli header C di sistema.

9.2 Compilazione

È necessario cambiare i path di CACHE_DIR, LOG_DIR e ROOT, presenti nel file configuration.h: infatti i percorsi sono stati impostati per la macchina su cui sono stati implementati.

Per compilare:

1. Aprire il terminale nella directory dove è salvato il file
2. Aprire il file e modificare il configuration.h, in particolare i path per la cache, il log e la cartella resources.
3. Eseguire il comando (in caso copiare e incollare):

```
gcc –o webserver –pthread http_request.c http_response.c main.c init_server.c threadmanager.c thread_connection_job.c format.c log.c file.c cache.c content_adaptation.c
```

In alternativa, essendo presente il file oggetto già compilato all'interno della cartella contentente l'intero progetto, si prò procedere direttamente all'esecuzione (vedere paragrafo 9.3).

9.3 Esecuzione

Una volta compilato e configurato, il server è pronto per essere eseguito, inserendo da terminale il comando

./webserver

Il server sarà disponibile all'indirizzo **http://127.0.0.1:porta_di_configurazione** [default = 5231]

10 Conclusioni

Come prima esperienza di un progetto in C posso ritenermi soddisfatta del lavoro svolto, ho infatti cercato di pensare alle soluzioni migliori che col tempo a disposizione potessi implementare e soprattutto ho cercato di trovare soluzioni alternative attuabili in fasi successive.
Come già detto nei capitoli precedenti è possibile infatti fare delle migliorie, sia modificando le scelte architetturali sia ristrutturando e snellendo codice stesso; sono infatti convinta che in questo modo il server migliorerebbe le sue prestazioni.