Ingegneria di Internet e Web

Anno accademico: 2015/2016

WEB SERVER STATICO CON ADATTAMENTO DINAMICO DEI CONTENUTI

Giovanni D'Agostino – Marco Giorgi – Lorenzo Paris

Indice

1 Introduzione

Come funziona Internet ? Una semplice domanda la cui risposta svela spesso dinamiche complesse. Il comune utilizzatore della Rete non ha bisogno di conoscere le meccaniche che regolano il funzionamento del Web, infatti un browser per la navigazione con il quale accedere alle risorse desiderate gli è più che sufficiente.
Lo stesso discorso non può essere applicato a coloro che vogliono fare della creazione e della gestione di pagine web il loro mestiere, per essi è necessario conoscere quali meccanismi entrano in gioco durante la normale attività di navigazione che milioni di persone mettono in atto ogni giorno. Cerchiamo allora di rispondere alla nostra domanda iniziale: come funziona Internet ? Il Web si basa sul principio del trasferimento di informazioni da un terminale (Host) all'altro attraverso dei sistemi di trasmissione detti protocolli, tra questi HTTP (Hyper Text Transfer Protocol) è quello più diffuso. Questo protocollo prevede che l'utente navigatore digitando una URL o cliccando su un link richieda l'accesso a determinate risorse (input); queste risorse, quando disponibili, gli vengono inviate sotto forma di file, generalmente pagine html o immagini (output).
In pratica l'utente grazie al meccanismo delle URL e dei link, non fa altro che indicare attraverso il programma di navigazione (browser) del suo computer (client), il percorso da seguire per raggiungere determinate informazioni contenute in un altro computer (server). Un server è quindi un elaboratore che contiene e fornisce file. Nel caso specifico dei server Web si parla di computer destinati ad ospitare siti internet, questi ultimi possono essere composti da pagine statiche (semplici pagine HTML) o dinamiche (ad esempio siti che prevedono un adattamento dinamico dei loro contenuti).
Un Web server non è altro che un software installato in un server con la funzione di elaborare pagine web e generare dinamicamente contenuti. Le semplici pagine HTML non necessitano di particolari interventi da parte del Web server, il loro codice viene interpretato dal browser del computer client e per questo l'HTML è definito come un linguaggio client side. Altre pagine Web, che generano i propri contenuti in base a determinati criteri, hanno bisogno della mediazione di un Web server. Esemplificando, quando si invia ad un Web server la richiesta di una pagina HTML statica, esso:

1. Riconosce la richiesta;
2. Cerca, e se presente, trova la pagina nel computer server;
3. Invia la pagina al browser.

Nel caso di una pagina dinamica invece, il Web server:

1. Riconosce la richiesta;
2. Cerca e trova i contenuti richiesti all'interno del server Web;
3. Esegue le istruzioni previste producendo dinamicamente contenuti;
4. Invia i risultati al browser.

In questa relazione è presentata l'implementazione e la progettazione di un web server in grado di adattare dinamicamente immagini sulla base di richieste specifiche fornite.

1.1 SWWS

SWWS è l'acronimo di *Seven World's Wonders Server*, ossia un Web Server con adattamento dinamico di immagini rappresentanti le 7 (nuove) Meraviglie del Mondo.
Il Web Server è stato ideato e progettato per piattaforme Linux, utilizzando il linguaggio C e le API del socket di Berkeley.
L'utilizzo di tale Server è stato reso semplice fornendo all'utente un'interfaccia di supporto in linguaggio HTML; nella schermata principale sono infatti elencati i nomi delle 7 "Nuove Meraviglie del Mondo", che l'utente può selezionare e quindi visualizzare nella maniera opportuna in base alle caratteristiche del proprio dispositivo; in particolare l'elenco è stato ordinato in base alla dimensione (in MegaByte) delle relative immagini, in modo tale da poter analizzare velocità di download e di adattamento su immagini diverse tra loro.

Le principali caratteristiche del Web Server includono:

- Funzionalità di base: apertura del socket di ascolto su una porta configurabile all'avvio (o di default), instaurazione della connessione, elaborazione delle richieste HTTP/1.1, invio delle risposte e chiusura della connessione;
- Gestione concorrente delle connessioni;
- Parsing di richieste di tipo HEAD o GET;
- Logging delle principali attività del server;
- Adattamento on-the-fly di immagini;
- Caching della versione adattata di immagini;

2 Scelte Progettuali

2.1 TCP

Lo scambio di messaggi richiesta/risposta tra client e server, il controllo della trasmissione e della sua affidabilità è gestito dal protocollo di trasporto TCP.
TCP (Trasmission Control Protocol) è un protocollo orientato alla connessione che garantisce il trasporto affidabile per un flusso di dati bidirezionale fra due host remoti. Tale protocollo ha cura di tutti gli aspetti del trasporto, come l'acknoweledgment, i timeout, la ritrasmissione, ecc. Alla base della sua progettazione non stanno semplicità e velocità, ma la ricerca della massima affidabilità possibile nella trasmissione dei dati.
L'alternativa sarebbe stata utilizzare UDP (User Datagram Protocol), un protocollo senza connessione e non affidabile ma che ha il pregio di una maggiore velocità di trasferimento dati rispetto a TCP, qualità che si prestano bene ad esempio per applicazioni di tipo streaming e multicast.

2.2 HTTP 1.1

HTTP è l'attuale protocollo del livello di applicazione utilizzato nelle architetture Web Server.
Tale protocollo funziona attraverso un meccanismo di richiesta-risposta, il primo eseguito dal client e il secondo dal server.
In particolare il web server è stato progettato in modo tale da gestire richieste con metodi GET e HEAD: il metodo GET è usato per ottenere il contenuto delle risorse presenti nell'URI, mentre il metodo HEAD è usato per ottenere informazioni presenti nell'header, come ad esempio la data di ultima modifica.
Nella progettazione si è fatta particolare attenzione alla creazione del messaggio di risposta da parte del server come stabilito dallo standard del protocollo; la risposta è infatti suddivisa in tre sezioni:

1. Riga di stato;
2. Sezione di Header;
3. Contenuto (body).

Nella riga di stato sono riportati i seguenti codici a tre cifre:

- 200 Ok: indica che il server ha fornito correttamente il contenuto della richiesta;
- 400 Bad Request: informa il client che la richiesta non è stata compresa da parte del server;
- 404 Not Found: inviata dal server quando il contenuto richiesto dal client non è stato trovato nel percorso specificato;

- 408 Timeout: inviata dal Server allo scadere del timeout;
- 500 Server Error: il server non ha potuto soddisfare le richieste del client per un malfunzionamento del server stesso.

La Sezione di Header invece, riporta il nome del server e il tipo di contenuto che il server sta inviando, in particolare nel nostro caso, il tipo text/html o image/jpeg.
Il Server è stato progettato per operare mediante un protocollo di tipo HTTP 1.1, il quale, a differenza della precedente versione 1.0, presenta delle caratteristiche di maggior interesse ai fini dell'esecuzione del Server progettato:

- Connessioni persistenti e Pipelining;
- Chunked Encoding;

Una connessione persistente consiste nel trasferire coppie multiple di richiesta e risposta entro una stessa connessione TCP. Questo tipo di connessioni permettono una riduzione dei costi di instaurazione e abbattimento delle connessioni TCP.
Il Pipelining permette inoltre una trasmissione di più richieste senza attendere l'arrivo della risposta alle richieste precedenti. Con questa tecnica si riduce ulteriormente il tempo di latenza ed ottimizzazione del traffico di rete, soprattutto per richieste che riguardano risorse molto diverse per dimensioni o tempi di elaborazione.
Mediante il Chunked Encoding è stato invece possibile suddividere la risposta da inviare al client in sottoparti (ognuno chiamato per l'appunto "chunk") specificando la dimensione in esadecimale di ogni chunk ed inviando al termine un chunk di dimensione nulla.

2.3 Politica di Cache

Al fine di garantire una gestione della memoria limitata è stato scelto di implementare una politica di caching all'interno del progetto web server.
Nel momento in cui un client richiede una determinata immagine, il server ha il compito di convertire on-the-fly l'immagine in base alla qualità che il client richiede. Allo scopo di ridurre l'overhead computazionale dell'adattamento, tale immagine viene memorizzata all'interno della directory contenente le versioni originali, in modo da esaudire velocemente richieste future della stessa immagine.
Senza una determinata politica di gestione della memoria a disposizione, ad ogni connessione essa verrebbe ridotta fino a renderla insufficiente per eventuali nuove richieste. Si è scelto dunque di implementare una cache di tipo LFU (Least Frequently Used): vengono mantenute in cache le immagini richieste con maggiore frequenza.
L'utilizzo di LFU all'interno del web server sembra essere migliore rispetto ad altre politiche come ad esempio FIFO e LIFO, le quali non prendono in considerazione la frequenza delle richieste di una particolare immagine, ma tengono conto solamente della quantità di tempo trascorsa in memoria.

2.4 WURFL

WURFL (Wireless Universal Resource FiLe) è un progetto OpenSource fondato e gestito dall'italiano Luca Passani. Tale progetto ha sviluppato (tra le altre cose) un file XML nel quale sono presenti numerose informazioni sulle caratteristiche di dispositivi mobili diffusi nel mercato globale.
Mediante l'utilizzo di questo file è stato possibile conoscere le specifiche dei vari dispositivi mobili e permettere dunque l'adattamento dinamico delle immagini.

2.5 ImageMagick

ImageMagick è un programma open source disponibile per tutti i principali sistemi operativi scaricabile dal sito internet: www.imagemagick.org. Attraverso un'interfaccia a riga di comando, è stato possibile modificare e convertire in diversi formati le immagini richieste dai diversi dispositivi.
Infatti grazie al comando *convert* e le opzioni *-resize* e *-quality*, è possibile ridimensionare e cambiare la qualità delle immagini in modo tale da soddisfare le richieste del client.

2.6 SQLite 3

SQLite (versione 3) è una libreria software scritta in linguaggio C. L'acceso ai dati dal database di Wurfl è stato reso possibile tramite l'utilizzo di questa libreria, che ne permette l'apertura e l'esecuzione di comandi SQL. La scelta di utilizzare SQLite a discapito di altre librerie verte sul fatto di essere di facile utilizzo e implementato nello stesso linguaggio di programmazione del Web Server.
Tale libreria è stata importata nel progetto attraverso la dicitura *#include "sqlite3.h"*.
Vengono quindi utilizzati nel progetto comandi SQL di tipo SELECT per ricevere le informazioni dal DB corrispondenti allo UserAgent o ai nomi commerciali dei dispositivi connessi.

2.7 Siege

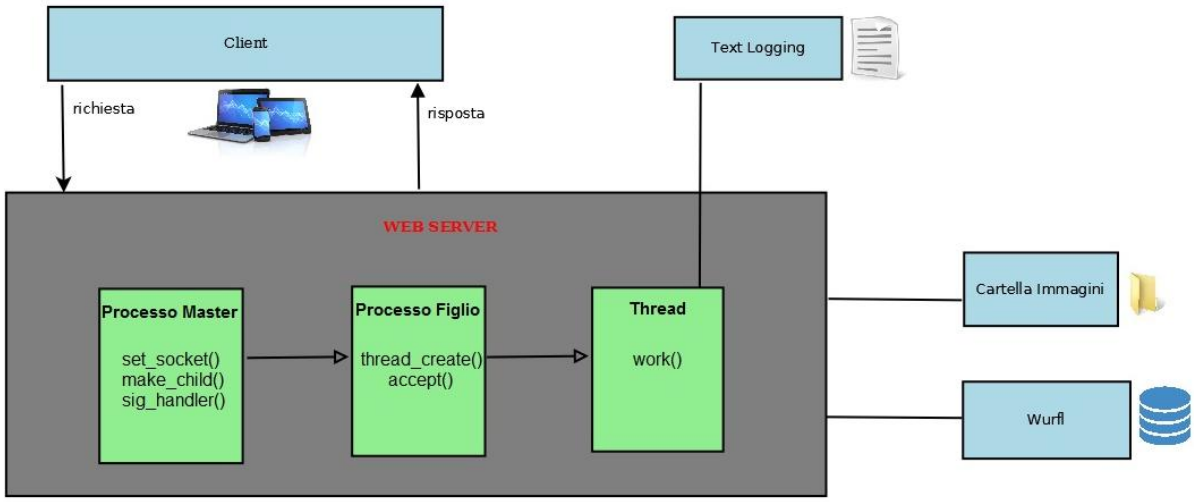
Per la valutazione delle prestazioni del Web Server è stato scelto di utilizzare il tool a riga di comando Siege.
Siege consente di misurare le performance del Web Server in base a determinati parametri di simulazione. Permette quindi di far connettere, in modo regolare o casuale, più client allo stesso web server per analizzare il livello di stress che quest'ultimo è in grado di supportare. Questo tool effettua una simulazione mediante linea di comando, al termine della quale restituisce una serie di risultati tra i quali:

- Tempo trascorso del test;

- Quantità di dati trasferiti (compresi header);
- Tempo di risposta del server;
- Throughput;
- Livello di concorrenza;
- Numero di transazioni totali completate con successo.

L'istallazione più semplice di Siege per Linux è l’esecuzione su linea di comando della seguente operazione: *sudo apt-get install siege*.

3 Struttura Server



Tra i vari modelli di progettazione di un web server, è stata utilizzata una struttura basata sul modello ibrido: le richieste di connessione giunte al web server vengono gestite attraverso l'uso di processi e threads.
In questo caso abbiamo un processo Master che crea molteplici processi figli, ciascuno dei quali genera diversi threads di esecuzione. L'organizzazione di base è la seguente:

- Il processo padre manda in esecuzione i processi figli;
- Ciascun processo figlio crea un numero fissato di threads worker e svolge la funzione di listener;
- Quando arriva una richiesta, il listener la assegna ad un thread worker che la gestisce.

Tale architettura permette di ottenere un'alta scalabilità e un minor consumo di risorse di sistema rispetto ad un modello "prefork", in cui la gestione delle richieste è affidata direttamente ai processi figli senza l'ausilio di threads aggiuntivi. La stabilità invece è simile ad un server di tipo multi-process puro. La nota negativa è data dal fatto che questo approccio implica un Server dal codice più complesso ed un supporto multi-threading che dipende dal tipo di SO utilizzato.
Avendo scelto questo modello architetturale, il Server presenta delle direttive di base per la gestione dei processi e dei threads specificate nel file *basic.h*; tra queste troviamo:

- *StartServer*: contiene il numero processi figli da creare in fase di inizializzazione;
- *MinIdleThreads* e *MaxIdleThreads*: rappresentano il numero minimo e massimo di threads idle (complessivo per tutti i processi);
- *ThreadsPerChild*: rappresenta il numero di threads creati da ciascun processo child in fase di inizializzazione;

- *ServerLimit*: imposta il limite sul numero di processi figli attivi contemporaneamente sul Server;
- *ThreadLimit*: imposta il limite sul numero di threads gestiti contemporaneamente da ogni processo figlio.

Passiamo adesso alla descrizione dettagliata del ruolo svolto dai principali componenti del Web Server implementato.

3.1 Processo Master

Il processo Master rappresenta il punto di partenza da cui viene avviato il Server, e in quanto tale si occupa dell'apertura di un nuovo socket e della creazione dei processi figli che andranno poi a gestire le connessioni in arrivo su di esso. Le sue funzionalità sono descritte all'interno del *main()* ed i compiti principali svolti possono essere riassunti nella maniera seguente:

- Inizializzazione delle variabili globali e dei semafori POSIX per la sincronizzazione dei vari flussi di esecuzione condivisi dai processi e dai threads;
- Creazione e apertura del file di Log, dove vengono memorizzate le informazioni sulle transazioni avvenute;
- Creazione e apertura passiva del socket di ascolto, per ricevere/inviare messaggi da/a un altro processo applicativo (remoto o locale);
- Creazione e avvio di un thread specifico per la gestione della cache;
- Registrazione di un handler che gestisce l'arrivo di segnali provenienti dai processi figli;
- Creazione e avvio di un numero iniziale di processi figli specificato nelle direttive;
- Gestione della richiesta di creazione di un processo aggiuntivo in seguito alla ricezione dell'apposito segnale da parte di un processo figlio.

Vediamo ora con maggior dettaglio come vengono implementate alcune delle principali funzionalità accennate.

Creazione e apertura passiva del socket di ascolto

Per poter instaurare connessioni tra host remoti, il server deve essere in grado di ricevere connessioni in arrivo. Questo viene implementato tramite la chiamata alla funzione *set_socket()*, che utilizza le funzioni *socket()*, *bind()* e *listen()* messe a disposizione dalla libreria *sys/socket.h*. Questo procedimento prende il nome di "apertura passiva del socket di ascolto".
Un socket costituisce in sostanza un canale di comunicazione fra due processi su cui si possono leggere e scrivere dati analogo a quello di una pipe ma, a differenza di

questa, i socket non sono limitati alla comunicazione fra processi che girano sulla stessa macchina, ma possono realizzare la comunicazione anche attraverso la rete.
La creazione del socket di ascolto avviene ad opera della funzione *socket()*, la quale, dopo aver specificato la famiglia di protocolli utilizzata (*AF_INET*), il tipo di socket scelto (*SOCK_STREAM*) e il protocollo che si vuole adottare, in caso di successo restituisce il suo file descriptor. Il campo protocollo è stato posto a 0 per selezionare quello di default indotto dalla coppia *domain* e *type* (*AF_INET + SOCK_STREAM* determinano una trasmissione TCP).
In seguito la chiamata alla funzione *bind()* assegna un indirizzo locale ad un socket, specificando l'indirizzo del server e il numero di porta su cui ci si vuole porre in ascolto. Per indicare un indirizzo IP generico (0.0.0.0) è stata utilizzata la costante *INADDR_ANY*.
La successiva chiamata della funzione *listen()* pone il socket specificato in modalità passiva e predispone una coda per le connessioni in arrivo di lunghezza pari a *BACKLOG*; quest'ultimo indica il numero massimo di connessioni pendenti accettate: se esso viene ecceduto al momento della richiesta di connessione, il client riceverà un errore di tipo *ECONNREFUSED*, oppure, se il protocollo supporta la ritrasmissione (come accade nel caso del TCP), la richiesta sarà ignorata in modo tale da poter ritentare la connessione.
A questo punto la funzione *set_socket()* termina restituendo il file descriptor del socket appena creato.

Gestore dei segnali ed eliminazione dei processi zombie

Il Server implementa un handler per gestire l'arrivo di segnali provenienti dai processi figli in seguito al verificarsi di determinate situazioni.
Innanzitutto, quando un processo figlio termina prima del padre, non è garantito che il padre possa ricevere immediatamente lo stato di terminazione. Il sistema operativo è quindi obbligato a conservare una certa quantità di informazioni riguardo ai processi che stanno terminando. I processi che sono terminati, ma il cui stato di terminazione non è stato ancora ricevuto dal padre, sono dunque chiamati "zombie" e rimangono nella tabella dei processi del sistema. La possibilità di avere degli zombie deve essere sempre tenuta presente quando si scrive un programma che deve essere mantenuto in esecuzione a lungo e che può generare molti figli, in particolare nel caso di un Web Server. I processi zombie in realtà non consumano risorse di memoria o processore, ma occupano comunque una voce nella tabella dei processi che a lungo andare potrebbe esaurirsi. L'eliminazione di questi viene dunque affidata al gestore dei segnali *usr_handler()* che viene invocato appena il kernel invia il segnale *SIGCHLD* al processo Master al fine di comunicare l'avvenuta terminazione di un processo figlio. A questo punto il gestore invoca la funzione *waitpid()* definita nell'header file *sys/wait.h*, la quale ritorna appena individua un processo figlio terminato. Nel caso in cui più di un figlio è terminato, l'inserimento di *waitpid()* all'interno di un ciclo while permette la terminazione di processi multipli.
Il gestore dei segnali viene invocato anche quando il processo Master riceve il segnale *SIGUSR1* proveniente da uno dei suoi processi figli, richiedendo la creazione

di un nuovo processo. Dunque, dopo aver controllato se è effettivamente possibile creare un nuovo processo figlio in quanto non è stata ancora raggiunta la soglia massima, viene invocata la funzione *Make_Child()* che si occupa di creare e avviare un nuovo processo. A questo punto viene incrementato il contatore dei processi totali e si riprende la normale esecuzione, in cui il processo Master si mette in attesa di ulteriori segnali.

3.2 Processi figli

Ciascun processo figlio, una volta creato dal processo Master, ha il compito di creare un numero prefissato di threads worker e di mettersi in ascolto sul socket per eventuali connessioni in arrivo. Il suo ciclo di vita viene descritto nella funzione *Child_Main()* la quale fornisce principalmente i seguenti servizi:

1. Inizializzazione delle strutture dati e dei semafori POSIX riservati ad ogni processo figlio e condivisi con i threads ad esso associati;
2. Creazione e avvio di un numero iniziale di threads indicato nelle direttive;
3. Utilizzo della funzione *select()*, per mettere in attesa il processo se non vengono individuate nuove connessioni;
4. Chiamata della funzione *accept()*, che blocca l'esecuzione fin quando non viene rilevata una connessione pronta sul socket di ascolto;
5. Segnalazione ai threads worker della presenza di una connessione accettata;
6. Creazione di un thread sostitutivo oppure, in base al verificarsi di determinate condizioni, invio di un apposito segnale al processo master.

Di seguito troviamo una descrizione più dettagliata di alcuni dei servizi specificati.

Funzione *accept()* e segnalazione ai threads worker

Ogni processo figlio si occupa di prelevare le connessioni pronte sul socket di ascolto tramite la funzione *accept()*. In genere essa viene chiamata dal Server per gestire la connessione una volta completato “l’handshake a 3 vie” e restituisce quindi un nuovo socket descriptor su cui si potrà operare per effettuare la comunicazione. La *accept()* è una funzione bloccante: se non ci sono connessioni completate il processo viene messo in attesa. Una volta ottenuto un nuovo socket descriptor, esso viene inserito nell'array *ConnectionList[]* la quale contiene l'insieme dei file descriptor delle connessioni pronte; viene quindi segnalato ai threads worker la presenza di una nuova connessione da gestire. Ciascun thread preleverà uno dei file decriptor dall'array e provvederà ad esaudirne le relative richieste HTTP.
L'accesso al socket di ascolto da parte dei processi figli è protetto mediante l'uso di un semaforo POSIX condiviso fra tutti i processi, utile ad evitare fenomeni di race condition dovuti a invocazioni simultanee alla funzione *accept()*. Se un processo non troverà la risorsa disponibile, si metterà in pausa per un breve periodo e riproverà in un momento successivo.

Esiste un array *ConnectionList[]* separato per ogni processo figlio, ma ogni processo ne condivide il proprio con i threads ad esso associati. Per permettere la sincronizzazione dei vari flussi di esecuzione, ogni array è protetto da un semaforo POSIX che ne permette l'accesso esclusivo.
La funzione di segnalazione ai threads, che serve per comunicare la presenza di una nuova connessione pronta, viene implementata mediante l'utilizzo della *pthread_cond_signal()*, la quale fa ripartire uno dei threads bloccati sulla *pthread_cond_wait()* in attesa che si verifichi l'apposita condizione. Tali funzioni vengono messe a disposizione dalla libreria di sistema *pthread.h*.

Creazione thread sostitutivo e invio segnali al processo Master

In seguito alla segnalazione ai threads worker di una nuova connessione pronta, il numero dei processi idle totale riferito all'intero Server e quello relativo al singolo processo figlio, andrà a diminuire. Per mantenere il numero di threads idle sempre maggiore o uguale a *MinIdleThread*, ogni processo figlio si occupa di creare un nuovo thread idle che andrà a sostituire quello che si sta occupando della gestione della nuova connessione, dunque non più "idle". In questo modo viene garantita la presenza del numero minimo di threads idle costante specificato nelle direttive del server. Ma non sempre tutto questo è possibile: nel caso in cui ci sono molti client connessi contemporaneamente, e il numero di threads per ogni processo figlio raggiunge il limite massimo specificato in precedenza, non è consentito crearne uno nuovo. Queste situazioni vengono gestite mediante l'invio del segnale *SIGUSR1* al processo Master, la quale creerà un nuovo processo figlio per ristabilire il numero minimo di threads idle e soddisfare richieste aggiuntive. Ci si assicura comunque che il numero di processi figli in esecuzione sul Server non superi mai il limite prestabilito dalla costante *ServerLimit*.
A questo punto il processo ricomincia il ciclo con una nuova chiamata alla funzione *accept()*, proseguendo se ci sono connessioni disponibili o rimanendo in attesa che esse arrivino.

3.3 Threads Worker

Ogni thread worker si occupa della gestione di una singola connessione precedentemente accettata dal processo figlio a cui il thread è associato. Il suo ciclo di vita è descritto dalla funzione *Thread_Main()* e le principali funzionalità svolte da ogni thread sono le seguenti:

- Attesa di un segnale proveniente dal processo figlio, la quale comunica la disponibilità di una nuova connessione;
- Gestione delle richieste HTTP tramite la chiamata alla funzione *work()*;
- Creazione di eventuali threads idle aggiuntivi;
- Terminazione del processo figlio associato al thread al verificarsi di determinate condizioni;
- Terminazione del thread worker.

Descriviamo adesso con maggior dettaglio l'implementazione di alcune delle funzionalità sopra citate.

La funzione *work()* per la gestione delle richieste HTTP

La gestione delle richieste HTTP è affidata alla funzione *work()*, che riceve come parametri in ingresso il file descriptor che fa riferimento al client connesso, l'indirizzo del file di Log su cui trascrivere le informazioni sui trasferimenti avvenuti, e un semaforo per sincronizzare le operazioni di conversione delle immagini.
Dopo aver istanziato le diverse strutture dati e i buffer che serviranno per poter elaborare le richieste in arrivo, il thread chiama la funzione *select()*, che blocca l'esecuzione fin quando non vi è effettivamente una richiesta pronta per essere gestita; viene inoltre impostato un TIMEOUT che indica la quantità di tempo in cui il thread rimarrà in attesa di nuove richieste prima che venga segnalato al client un avviso di timeout e che la funzione *work()* ritorni.
Dopo essersi accertati della presenza di una nuova richiesta da parte del client, il thread esegue una *read()* la quale legge dal file descriptor specificato e ne trascrive il contenuto in un apposito buffer. Quest'ultimo viene poi "spezzettato" in singole richieste HTTP ed analizzate una ad una. Se la richiesta ricevuta non viene riconosciuta, in quanto contiene metodi diversi da GET o HEAD, la funzione termina e il thread segnala al client di non essere stato in grado di riconoscere la richiesta.
Il metodo HEAD viene gestito mediante un semplice tentativo di apertura del file richiesto: se il file è presente nell'apposita directory, viene inviata al client una risposta affermativa; se invece il tentativo di apertura fallisce, il thread segnala di non aver trovato il file specificato. In ogni caso l'esecuzione prosegue e si passa all'elaborazione della prossima richiesta.
La gestione del metodo GET invece è un po' più complessa. Se il file richiesto non viene trovato, il thread si comporta come è stato descritto per il metodo HEAD; se invece il file cercato è presente nell'apposita directory, l'esecuzione procederà diversamente a seconda se la richiesta si riferisce alla pagina iniziale del Web Server, avente formato HTML, oppure ad una delle immagini in formato JPEG disponibili:

- Nel primo caso, il file viene semplicemente inviato tramite la funzione *send_file()* per poi passare alla prossima richiesta dopo aver memorizzato le informazioni riguardanti il trasferimento sul file di Log;
- Nel caso in cui il file richiesto è un'immagine JPEG, vengono prelevate dal buffer varie informazioni quali lo UserAgent, la qualità e la risoluzione, necessari a creare il comando per la conversione dell'immagine mediante il programma ImageMagick. Viene dunque aggiornata la lista delle immagini presenti nella cache segnalando all'apposito thread l'arrivo di una nuova richiesta e il nome dell'immagine a cui essa fa riferimento; a questo punto viene nuovamente fatta una ricerca sulle immagini memorizzate nella cache per trovare quella che soddisfa le caratteristiche del dispositivo connesso. Se l'immagine, strutturata in base alle caratteristiche del dispositivo client, non è presente, verrà allora creata lanciando, tramite la *system()*, il comando di

ImageMagick precedentemente composto “ad hoc”. In definitiva, avendo a disposizione l’immagine cercata, essa verrà inviata con la funzione *send_file()*, verrà aggiornato il file di Log e si passerà alla richiesta successiva.

La fase di conversione dell’immagine è protetta grazie all’utilizzo del semaforo *sem_convert*, per evitare che diversi threads provino ad eseguire contemporaneamente il comando di conversione tramite la *system()*; viene inoltre utilizzato un mutex condiviso da tutti threads di ogni processo per rendere esclusiva la fase di aggiornamento della lista cache.
La *work()* termina quando non ci sono più richieste da soddisfare e il TIMEOUT definito nella *select()* è scaduto.

Terminazione di un processo figlio e creazione di threads idle multipli

Dopo aver terminato di gestire le richieste HTTP provenienti da un client e aver chiuso la connessione, il thread worker verifica se può terminare l'intero processo a cui appartiene se ritenuto “superfluo”. Questo è un procedimento molto delicato in quanto la terminazione del processo figlio provoca anche la chiusura di ogni thread ad esso associato. Per questo motivo, prima di eseguire questa operazione, vengono fatti i dovuti controlli per stabilire se effettivamente tutti i suoi threads sono idle, se il numero di processi figli totali presenti sul Server sia maggiore della soglia minima, oppure semplicemente se il numero dei threads idle totali rimanga superiore al limite consentito anche dopo la terminazione del processo. Se tutte queste condizioni si verificano vengono quindi aggiornati i contatori opportuni e viene terminato l'intero processo, la quale determina l'invio del segnale *SIGCHLD* al processo Master. Quest'ultimo, informato della presenza di un processo zombie, si occuperà di eliminarlo anche dalla tabella dei processi del sistema.
Se la terminazione del processo figlio non avviene, in quanto non si verificano le condizioni sopracitate, il thread prosegue la sua esecuzione verificando che il numero totale di threads idle del sistema sia ancora maggiore o uguale del minimo consentito. In caso contrario si occupa di ristabilirne il numero prestabilito creando tanti threads quanti ritenuti necessari allo scopo.
A questo punto il thread termina aggiornando le opportune variabile globali e rilasciando tutte le risorse ad esso associate.

3.4 Comunicazione tra processi

Uno degli aspetti fondamentali nella programmazione di un Web Server è la gestione della comunicazione e della sincronizzazione tra i vari processi in esecuzione.
Nonostante nei sistemi UNIX esistano diversi meccanismi per lo scambio di messaggi come le pipe, le named pipe (o fifo), o le code di messaggi SystemV, il Web Server creato non necessita di un vero e proprio sistema di comunicazione, ma più precisamente di un modo per avere accesso a diverse strutture dati condivise contenenti informazioni utili a tutti i processi, evitando così di creare stati di

inconsistenza o fenomeni di *race condition*. Per questo motivo si è scelto di fare ricorso a tecniche per la gestione della memoria condivisa mediante la funzione *mmap()* e di gestire la sincronizzazione dei flussi di esecuzione attraverso l'uso di semafori POSIX.
Attraverso l'utilizzo della *mmap()* è possibile mappare il contenuto di un'area di memoria: mediante il flag *MAP_SHARED* infatti, le modifiche effettuate sulla sezione specificata vengono viste da tutti i processi che lo hanno mappato. Generalmente utilizzare questa tecnica per creare una memoria condivisa fra processi diversi è estremamente inefficiente, in quanto occorre passare attraverso il disco, incidendo in modo negativo sulle prestazioni del Web Server; tuttavia, nel caso in cui si eseguisse la mappatura aggiungendo il flag *MAP_ANONYMOUS*, la regione mappata non verrebbe associata a nessun file e, se acceduta o modificata, rimarrebbe sempre in memoria, pronta per essere riletta; in questo modo, dato che ciascun processo figlio mantiene nel suo spazio degli indirizzi anche le regioni mappate dal processo padre, ognuno di essi sarà anche in grado di accederne velocemente. Con questo procedimento diventa possibile creare una memoria condivisa fra processi diversi, purché questi abbiano almeno un progenitore comune che abbia effettuato il memory mapping anonimo: nel caso di un Web Server con architettura ibrida, questo ruolo è svolto dal processo Master.
Per sincronizzare gli accessi alle regioni di memoria condivise ed evitare quindi problemi di race condition, è stato scelto di utilizzare una serie di semafori POSIX condivisi tra i processi. La loro creazione è anch'essa affidata alla funzione *mmap()* mediante l'utilizzo dei flag *MAP_SHARED* e *MAP_ANONYMOUS*, mentre la sua inizializzazione avviene tramite la chiamata a *sem_init()*. La funzione inizializza un semaforo all'indirizzo puntato dall'argomento, consente di impostare un valore iniziale e, specificando un valore non nullo in *pshared*, renderlo condiviso tra i vari processi.
Una volta ottenuto l'indirizzo di un semaforo ed averlo inizializzato, è possibile utilizzarlo con *sem_wait()* e *sem_post()*:

- La funzione *sem_wait()* cerca di decrementare il valore del semaforo indicato: se questo ha un valore positivo, ossia la risorsa è disponibile, la funzione ha successo ed il valore del semaforo viene diminuito di 1 ed essa ritorna immediatamente; se il valore è nullo, la funzione si blocca fintanto che il valore del semaforo non torni positivo così che poi essa possa decrementarlo con successo e proseguire;
- La funzione *sem_post()* incrementa di uno il valore corrente del semaforo indicato dall'argomento: se questo era nullo la relativa risorsa risulterà sbloccata cosicché un altro processo, eventualmente bloccato in una *sem_wait()* sul semaforo, potrà essere svegliato e rimesso in esecuzione.

La scelta di utilizzare la combinazione di memoria condivisa e locking sui semafori POSIX, nonostante richieda di attendere il rilascio del semaforo per accedere a determinate regioni di memoria, permette di sfruttare appieno diversi vantaggi: la memoria condivisa è la forma più veloce di comunicazione fra due processi in quanto permette agli stessi di vedere nel loro spazio di indirizzi una stessa sezione di

memoria; pertanto non è necessaria nessuna operazione di copia per trasmettere i dati da un processo all'altro, in quanto ciascuno può accedervi direttamente con le normali operazioni di lettura e scrittura dei dati in memoria.

4 Adattamento Dinamico Immagini

4.1 Introduzione

Il Web Server prevede, tra le sue funzionalità, l'adattamento dinamico (on the fly) di immagini.
Nella pagina *Index.html* l'utente seleziona una tra le 7 immagini disponibili; a quel punto il browser invierà al server una richiesta HTTP nella quale verrà specificato, tra le altre cose, l'URI relativo al file selezionato e lo User-Agent del dispositivo destinatario dell'immagine; mediante queste due informazioni il server sarà in grado di inviare al client l'immagine prescelta secondo determinati caratteristiche tra le quali: dimensioni dell'immagine adattata allo schermo del client e qualità immagine idonea al dispositivo del client.

4.2 WURFL

Per l'adattamento delle immagini su dispositivi mobili si è dovuto far uso di un file XML (reperibile all'indirizzo internet http://wurfl.sourceforge.net/) contenente svariate informazioni su tali dispositivi.
Tale file, *WURFL.xml*, è un file di tipo XML, e in quanto tale, per poter essere utilizzato secondo i nostri scopi, si è dovuto decodificare in altri formati "leggibili" dal nostro server.
Si è visto che tale file XML aveva al suo interno una struttura ben precisa riguardo le informazioni sui dispositivi mobili; in particolare i dati più rilevanti utili ai nostri scopi erano quelle riguardanti lo User-Agent associato al nome commerciale del dispositivo e le corrispondenti dimensioni dello schermo (altezza e larghezza), quindi un'infima quantità rispetto all'enorme contenuto di informazioni riguardanti ogni dispositivo.
Avendo inoltre tale file dimensioni non trascurabili (circa 20 MB), sembrava non molto performante il suo utilizzo diretto durante la normale esecuzione del server, ossia effettuare una scansione di tutti i dispositivi presenti al suo interno per estrapolare i dati (per lo più inutili ai nostri scopi).
Si è deciso allora di creare un Database che contenesse solo ed esclusivamente i dati a noi utili di questo macro-file.
La creazione di questo DB consiste principalmente in 2 passaggi:

1. La prima parte, svolta dal nostro programma *DB_Create.c*, è costituita dall'estrapolazione dal file *WURFL.xml* dei soli dati a noi utili e della scrittura degli stessi in un file apposito in formato txt chiamato *databaseWURFL.txt*; tale file prevede una notevole leggibilità delle informazioni essendo strutturato in maniera semplice:

   nome_device; user_agent_associato; larghezza_schermo; altezza_schermo

In questo modo abbiamo un file pronto per poter essere decodificato velocemente per scrivere successivamente le informazioni nel DB;

2. La seconda parte è costituita quindi dalla scrittura di ogni informazione contenuta nel file *databaseWURFL.txt* all'interno di un Database; tale operazione è stata effettuata mediante un software free (SQLiteman) che, attraverso un opportuno tool, ha permesso la decodifica del suddetto file e l'inserimento dei dati in un Database chiamato *DevicesDB.db* contenente i campi desiderati: device, user_agent, width, height.

A questo punto siamo in grado di effettuate una scansione veloce e performante su DB tramite comandi SQL.

4.3 Ricerca Dispositivo

Una volta ricevuta la richiesta HTTP dal client, il nostro server è in grado di catturare le informazioni contenute al suo interno e spedire un'immagine che sia conforme alle caratteristiche del dispositivo client stesso in termini di qualità e dimensioni.
Entrambe le caratteristiche vengono decodificate a partire dalla richiesta HTTP.
Tali richieste hanno una struttura ben definita, e mediante la funzione *get_quality()* si va infatti a cercare all'interno di tale richiesta il punto in cui viene specificata la qualità dell'immagine adottata dal client, ossia un numero decimale compreso tra 0 e 1; in caso di mancata presenza di tale campo nella richiesta HTTP, il server adotterà un valore di default pari a 1.
L'operazione di "resizing" dell'immagine è invece affidata a due funzioni che lavorano l'una in soccorso dell'altra:

- In primo luogo viene eseguita la funzione *get_resolution_fromDB1()* il suo quale scopo è quello di cercare all'interno del Database le caratteristiche di altezza e larghezza relative allo UserAgent richiedente. Allo stesso modo dell'operazione precedente, dalla richiesta HTTP ricevuta viene catturato lo UserAgent del client connesso al server e passato come parametro alla funzione suddetta; a questo punto vengono utilizzate funzioni contenute all'interno della libreria SQLite per aprire il Database, ricevere i dati richiesti mediante query SQL, e restituire i valori alla funzione chiamante.

Qualora però la query di ricerca in base allo UserAgent fallisse, ossia non esiste nessun dispositivo nel DB con lo UserAgent indicato nel pacchetto di richiesta, viene allora effettuata una ulteriore ricerca in base al nome (commerciale) del dispositivo.

- Il passo successivo è stato quindi quello di "provare" a risalire al sistema operativo del dispositivo client (tramite lo stesso UserAgent), per poi estrapolare il nome del device secondo precise decodifiche della stringa, specifiche per ogni diverso Sistema Operativo;

in particolare, all'interno del file *UserAgentAnalyzer.h* sono presenti funzioni per la decodifica dello UserAgent di ben 4 sistemi operativi diversi: Linux (Android), Windows Phone, Symbian, BlackBerry.

Quindi nel caso in cui la funzione *get_resolution_fromDB1()* fallisse, viene eseguita la *get_resolution_fromDB2()* la quale appunto, estrapola il nome del dispositivo mediante caratteristiche comuni dello UserAgent dello stesso sistema operativo, ed esegue ancora una volta una query SQL dal nostro Database provando in questo caso a trovare le caratteristiche del dispositivo non più attraverso il campo UserAgent, ma mediante il *nome_device*.
In questo modo abbiamo una probabilità di riuscita dell'operazione più elevata (circa 50% rispetto al 3-5% di *get_resolution_fromDB1()*).

4.4 ImageMagick

Una volta ottenute le informazioni riguardanti le caratteristiche del client, possiamo effettuare una conversione dell'immagine mediante l'esecuzione di un comando direttamente da shell.
La funzione "*system*" infatti permette di ricevere come parametro una stringa ed eseguirla come comando su shell .
Il comando passatogli è una funzione del programma ImageMagick che prende come parametri il path dell'immagine da convertire, la qualità dell'immagine con cui convertirla e le dimensioni (altezza, larghezza); tale funzione creerà un nuovo file con l'immagine convertita e rinominata secondo le caratteristiche di conversione.
A questo punto l'immagine è pronta per poter essere inviata al client.

5 Caching Immagini

La politica di caching delle immagini è suddivisa in due diverse fasi ognuna delle quali ha un compito ben preciso e fondamentale riguardo le performance e la scalabilità del server.

FASE 1

La prima fase ha lo scopo di ridurre l'overhead del periodo di:

richiesta immagine - spedizione immagine

basandosi sul principio di località temporale.
Nella prima esecuzione del server sarà presente in locale una directory che conterrà le 7 immagini indicate nella pagina HTML *Index*; nel momento in cui un client sceglie di visualizzare un'immagine, essa verrà processata e modificata in base alle caratteristiche del dispositivo client: verranno modificate la dimensione (altezza e larghezza) e/o l'indice di qualità; a questo punto il file immagine è pronto per essere spedito al client. L'immagine "editata" non viene scartata, ma ne viene salvata una copia (modificata e rinominata in base alle caratteristiche di editing) all'interno della stessa directory. In questo modo, all'eventuale richiesta della stessa immagine con gli stessi attributi di editing da parte di un altro client, l'immagine non avrà bisogno di un'ulteriore processamento, ma verrà spedita la copia già modificata precedentemente, risparmiando così il tempo di conversione della stessa immagine.

FASE 1 - Livello Software

Quello che accade al livello software è sostanzialmente l'esecuzione del comando "system", il quale esegue tramite linea comando la stringa passatagli come parametro; tale stringa conterrà il comando per la modifica dell'immagine desiderata tramite il programma "ImageMagick" e la contemporanea creazione all'interno della directory dell'immagine stessa.

FASE 2

Le operazioni svolte nella "Fase 1" però, quantificate per tempi molto lunghi nel quale entrano in gioco dispositivi client con caratteristiche diverse tra loro, porta nel tempo ad un incremento massiccio del numero di immagini modificate all'interno della directory, con una conseguente riduzione sulle performance di progetto per tale tipo di cache.
A questo scopo avviene quindi una seconda fase, la quale ha il compito di mantenere costante nel tempo il numero di tali immagini.
In questa fase si fa uso di una lista gestita all'interno del file *CacheList.h*. All'interno di questa lista vi sono un susseguirsi di nodi contenenti, per ognuno di essi, una stringa relativa al nome dell'immagine processata e gestiti secondo una politica LFU (Least Frequently Used), e il puntatore al nodo successivo nella lista.

Tale lista ha una dimensione limitata a 15 elementi, quindi fino alla 15° richiesta (diversa dalle altre) i nodi verranno disposti secondo una struttura a pila; una successiva richiesta non presente all'interno della lista fa sì che venga eliminata l'immagine meno richiesta per essere rimpiazzata dalla nuova.

FASE 2 - Livello Software

A livello software, ad ogni nuova richiesta da parte di un client, vengono passati dal *Thread_Cache* alla funzione *refresh_list()* due parametri: la stringa contenente il nome dell'immagine e il puntatore al primo nodo della lista. All'interno di tale funzione avverrà quindi una scansione della lista per verificare se tale immagine fosse già presente: in caso di esito negativo, il nodo viene aggiunto in coda alla lista, altrimenti viene tolto dalla posizione corrente e reinserito in coda.
La *refresh_list()* ritornerà quindi un valore booleano il quale indicherà se tale immagine fosse già presente nella lista: in caso affermativo, l'effetto che si ha avuto è stato un semplice riordinamento dei nodi; nel caso in cui invece il nodo non fosse presente nella lista, verrà incrementato e controllato il numero di elementi presenti all'interno di essa; se tale limite fosse ecceduto, si provvederà ad eliminare il nodo in cima alla lista (cioè il meno utilizzato) mediante la funzione *delete()*, ed eliminare il file relativo all'immagine in esubero all'interno dell'apposita directory ritornando il comando da far eseguire mediante la funzione *system()* nel *Thread_Cache*.

6 Logging

Per la gestione del Logging si è scelto di salvare su un file di tipo txt, nominato *log.txt*, le informazioni sulle transazioni avvenute tra Client e Server.
La creazione, o la semplice apertura di tale file, è implementata nel processo Master, la quale si occupa di verificare l'esistenza del file all'interno della directory principale.
Il compito di scrivere i messaggi sul log è affidata ad ogni singolo thread worker all'interno della funzione *work()*, luogo in cui viene chiamata la funzione *writeLog()* che si occupa di definire il formato delle informazioni memorizzate e di implementare la sincronizzazione delle scritture sul file. In particolare, attraverso l'utilizzo della funzione *flock()* è stato possibile sincronizzare il lavoro di scrittura sul file da parte dei thread. La funzione *flock()*, dunque, è usata per acquisire o rilasciare il lock sul file descriptor *LogFd* a seconda di quanto specificato nei suoi parametri di invocazione:

- LOCK_EX: richiede un lock esclusivo sul file;
- LOCK_UN: rilascia il lock sul file .

Le informazioni all'interno del file di Log sono strutturate nella seguente maniera:

Host [data] " metodo immagine protocollo" stato "UserAgent"

- *Host*: indirizzo IP del client connesso;
- *Data*: data ed ora in cui è stata ricevuta la richiesta HTTP;
- *Metodo:* metodo utilizzato all'interno della richiesta;
- *Immagine:* percorso completo dell'immagine richiesta;
- *Protocollo:* protocollo utilizzato dal client;
- *Stato:* codice riferito allo stato della risposta da parte del server;
- *UserAgent:* UserAgent del client connesso.

7 Configurazione ed Uso

7.1 Installazione

Il Software è stato implementato per funzionare su piattaforme Linux. Ai fini dell'installazione è sufficiente effettuare una copia della directory del programma in una qualsiasi directory del proprio sistema. Il Web Server inoltre fa uso di alcuni free software necessari per la corretta esecuzione del codice. Tali software sono facilmente installabili mediante i seguenti comandi da shell:

- ImageMagick (tool di conversione immagine): *sudo apt-get install imagemagick*
- SQLite3 (libreria di supporto per database): *sudo apt-get install sqlite3 libsqlite3-dev*

7.2 Compilazione

Al fine di facilitare la corretta compilazione di tutti i sorgenti necessari, il codice viene fornito corredato di un makefile che provvede autonomamente alla compilazione e al linking dei vari file oggetto.
Per avviare la compilazione è sufficiente posizionarsi nella directory in cui si è installato il webserver e lanciare il comando *make*. A seguito dell'esecuzione di tale comando infatti, verrà creato nella directory corrente un file eseguibile di nome *SWWS* pronto per poter essere lanciato da linea comando mediante la consueta sintassi *./SWWS*.
Qualora si avesse il bisogno di ricompilare nuovamente codice, basterà lanciare il comando *make clean* il cui effetto sarà quello di cancellare tutti i file oggetto.

7.3 Esecuzione

Una volta compilato, il programma è pronto per poter essere eseguito. In fase di avvio dell'esecuzione è possibile utilizzare due opzioni:

-p numero_porta -d

- La prima riguarda la possibilità di impostare un numero di porta idoneo (compreso tra 1024 e 65535) su cui si desidera che il server si ponga in ascolto; in assenza di tale campo, il server si porrà in ascolto sulla porta di default 9091;
- Nella seconda opzione è possibile avviare il server in modalità debug, utile per stampare a schermo alcuni dei messaggi significativi nel corso dell'esecuzione

L’ordine delle due opzioni è intercambiabile, è possibile cioè inserire prima l’opzione di debug e poi quella relativa alla porta, l’importante è inserire il **numero_porta** immediatamente dopo la relativa opzione **–p** seguita da uno spazio.
Ad esempio è possibile lanciare il seguente comando:

./SWWS –d –p 3030

7.4 Terminazione

Una volta avviato il server è possibile terminarne l’esecuzione mediante la combinazione di tasti **ctrl+c**. Tale azione provocherà il corretto rilascio di tutte le risorse acquisite durante l’esecuzione, e successivamente terminerà il programma.

8 Prove di Esecuzione

8.1 Richiesta GET di un documento HTML

Richiesta espressa mediante l'utilizzo del tool a linea di comando CURL

GET / HTTP/1.1
Host: 127.0.0.1:9091
User-Agent: curl/7.43.0
Accept: */*

8.2 Richiesta HEAD di un'immagine JPEG

Richiesta espressa mediante l'utilizzo del tool a linea di comando CURL

HEAD /Pictures/Petra.jpg
Host: 127.0.0.1:9091
User-Agent: curl/7.43.0
Accept: */*

8.3 Richiesta GET di un'immagine JPEG

Richiesta espressa mediante browser su dispositivo mobile GT-I9100

GET /Pictures/Petra.jpg HTTP/1.1
Host: 10.200.182.69:9091
Connection: keep-alive
Referer: http://10.200.182.69:9091
Accept: text/html,aplication/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Linux; U; Android 4.1.2; it-it; GT-I9100 Build/JZO54K) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile Safari/534.30
Accept-Encoding: gzip,deflate
Accept-Language: en-GB, en-US
Accept-Charset: utf-8, iso-8859-1, utf-16, *;q=0.7

8.4 Risposta 200OK all'immagine JPEG

HTTP/1.1 **200OK**
Mon, 30 Nov 2015 14:56 24 CET
Transfer-Encoding: chunked
Content-Type: image/jpeg
Data (102774 bytes)

9 Test

Siege è un'utilità di benchmark e load test HTTP eseguita da linea di comando. Si differenzia da altri tools per il fatto che dà la possibilità di lavorare con quasi tutti i Web Server disponibili sul mercato e per l'opportunità di testare contemporaneamente più URL, permettendo così di simulare l'uso del server con maggiore accuratezza e migliore aderenza alla realtà operativa.
Con Siege si può simulare la navigazione di più utenti contemporaneamente e verificare subito performance, tempi di risposta e capacità del Server di servire correttamente ogni richiesta, evidenziando eventuali problematiche ed errori.
Nelle misurazioni effettuate si è deciso di impostare un tempo di testing costante pari a 60 secondi osservando il comportamento del server al variare del numero di client connessi contemporaneamente, da un minimo di 10 a un massimo di 1000. In particolare si è scelto di far variare due parametri fondamentali riguardo alle performance e al carico di lavoro del Web Server: le costanti STARTSERVER e THREADSPERCHILD definiti nel file *Basic.h*.

9.1 Web Server Progettato

9.1.1 Scelta di STARTSERVER

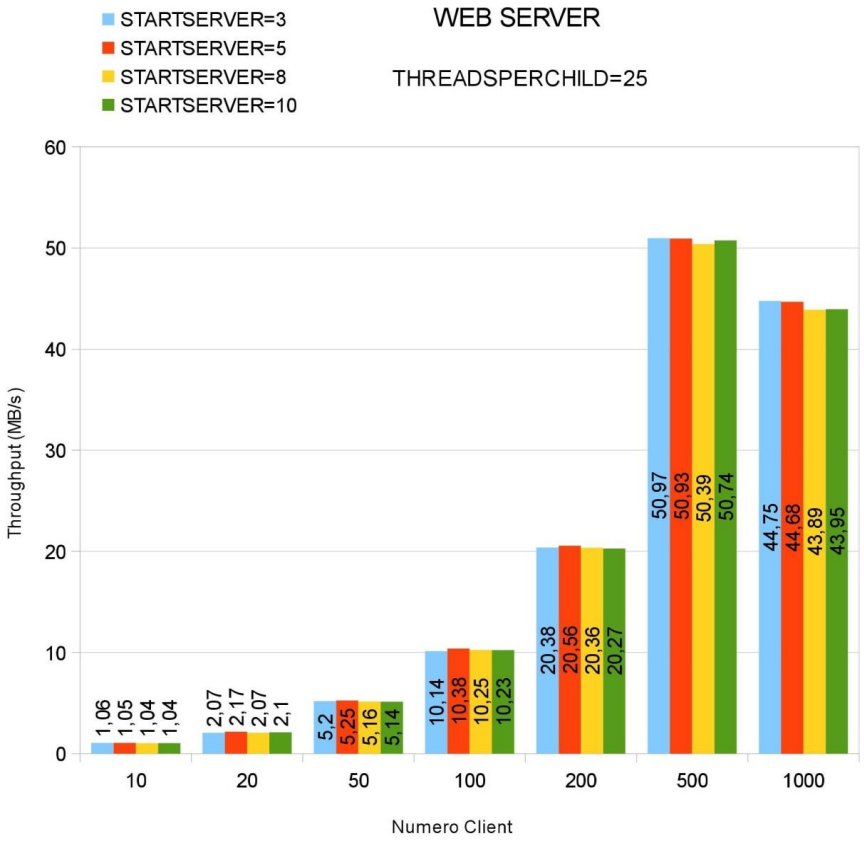


Grafico 1

Dal Grafico 1 è possibile osservare come la scelta del numero iniziale di processi figli pari a 5, sia da considerare la migliore in termini di prestazioni. Infatti, all'aumentare del numero di client connessi in modo concorrente al web server, il throughput risulta mediamente superiore rispetto ad altri valori come 3, 8 e 10. I test sono stati effettuati mantenendo costante il numero di threads gestiti da ogni processo pari a 25, valore di default del server Apache.

9.1.2 Scelta di THREADSPERCHILD

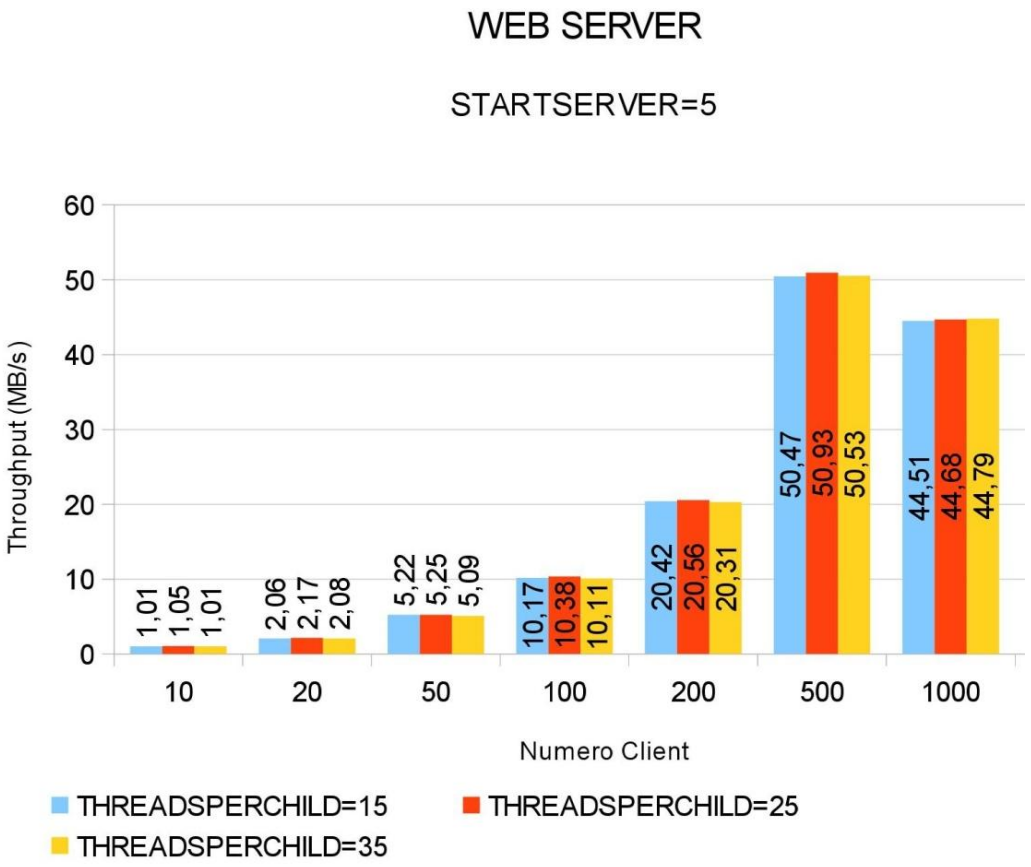


Grafico 2

Stabilito il numero iniziale di processi pari a 5, si è deciso di far variare la costante THREADSPERCHILD in modo tale da trovare la configurazione più adatta alle esigenze del web server. Dal Grafico 2 si evince che, impostando la costante THREADSPERCHILD pari a 25, le prestazioni del web server risultano essere migliori rispetto a scelte come ad esempio 15 e 35. Insieme al valore di THREADSPERCHILD è stato fatto variare anche il numero minimi e massimo di threads idle in accordo con il numero di threads di partenza stabilito.

9.2 Differenze con Apache

Il Web Server Apache è un software Open Source, questo vuol dire che il suo codice è pubblicamente disponibile ed aperto al contributo di chiunque desideri modificarlo agevolando così il suo continuo sviluppo; in questo modo Apache ha potuto usufruire dell'opera volontaria di migliaia di programmatori sparsi in tutto il mondo. Inoltre, Apache è un programma scaricabile gratuitamente e liberamente dalla rete (http://httpd.apache.org/), elemento che insieme alle sue indubbie qualità di stabilità e potenza ha contribuito non poco alla grande diffusione di questo Web Server. Infatti le funzionalità e le prestazioni offerte dal server Apache non hanno nulla da invidiare a quelle di numerosi server "commerciali", tanto che Apache risulta essere il software web server più usato su internet (nel novembre 1999 deteneva il 52% della quota di mercato a livello mondiale, risultando primo per adozione anche in Italia). Grazie a questa linea di pensiero, il progetto Apache e riuscito ad ottenere un gran numero consensi e simpatie da parte di molti programmatori sparsi per il mondo, evolvendosi e migliorando le sue prestazioni a ritmi sfrenati.
Oggi il Web Server Apache è considerato il server web più popolare (circa il 60% dei server web al mondo sono server Apache) ed affidabile al mondo per le sue caratteristiche sofisticate e le eccellenti performance, sia perché è gratuitamente reperibile, sia perché la sua estraneità da strategie di mercato, fanno sì che le varie correzioni e patch siano immediatamente disponibili appena corretti, garantendo in questo modo affidabilità e robustezza.
L'architettura di Apache è di tipo modulare, ossia è composto da un piccolo nucleo che realizza le funzionalità di base, mentre per estendere queste funzionalità di base si utilizzano i moduli. Questi moduli possono essere compilati staticamente nel nucleo oppure caricati dinamicamente a tempo di esecuzione. All'avvio del server, è possibile scegliere i moduli che devono essere caricati, indicandoli in un opportuno file di configurazione. Alcuni possono essere inclusi di default nel core del server e vengono chiamati moduli standard.
Apache è stato progettato per essere flessibile su ogni tipo di piattaforma e con ogni configurazione d'ambiente; per questo scopo, da Apache 2.0, sono stati inseriti i moduli MPM per la gestione di operazioni quali il binding, gestione dei processi/thread, connessioni.
Un modulo MPM è compilato nel server e permette a SO differenti di fornire moduli appropriati per
una maggiore efficienza. Permette infatti di applicare politiche di gestione diverse in base alle proprie esigenze. Solo un modulo MPM alla volta può essere caricato nel server; alcuni tra quelli disponibili sono: prefork, worker, event.
Per i test effettuati si è scelto di utilizzare il Web Server Apache configurato con il modulo MPM worker. La scelta è stata fatta principalmente tenendo in considerazione le analogie tra l'architettura prevista per questo modulo e la struttura generale del Web Server progettato.

9.2.1 Throughput

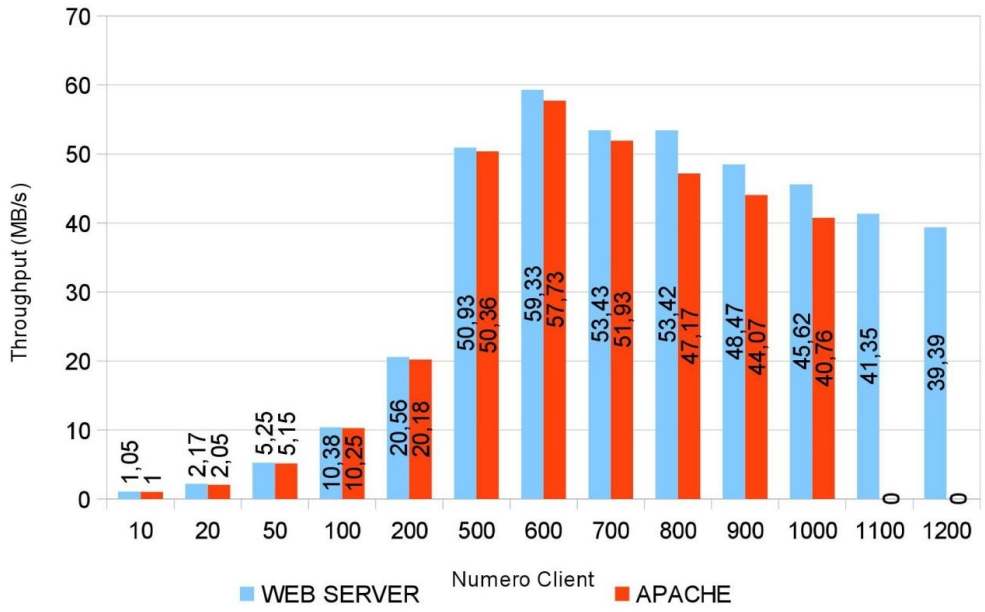


Grafico 3.1

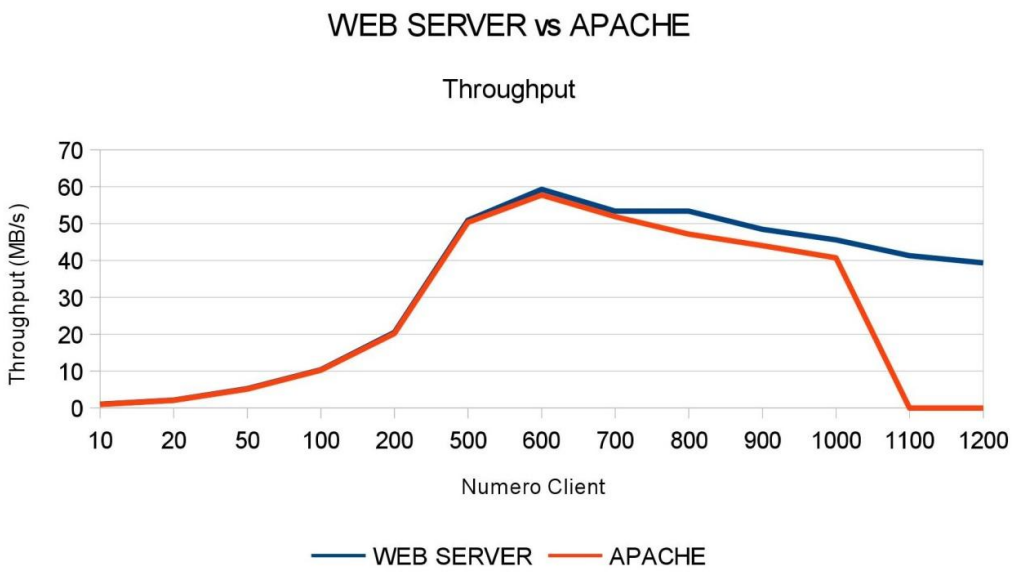


Grafico 3.2

Dai grafici è possibile notare la differenza di prestazioni in termini di throughput tra i due server; in particolare, nel primo grafico è possibile osservare come, studiando i risultati dei server per un numero ridotto di client connessi, la differenza appare minima; viceversa, la differenza appare più significativa quando il numero dei client connessi aumenta. Questa differenza viene evidenziata nel secondo grafico, dove è possibile notare come l'andamento del server Apache, dopo un valore di circa 700 client concorrenti, tende a ridursi notevolmente, fino ad arrestarsi per un numero maggiore di 1000 client, valore in cui tale server smette di funzionare. Al contrario, il

web server progettato, tende a mantenere un valore di throughput pressoché costante anche per valori pari a 1200 client.

9.2.2 Longest Transaction

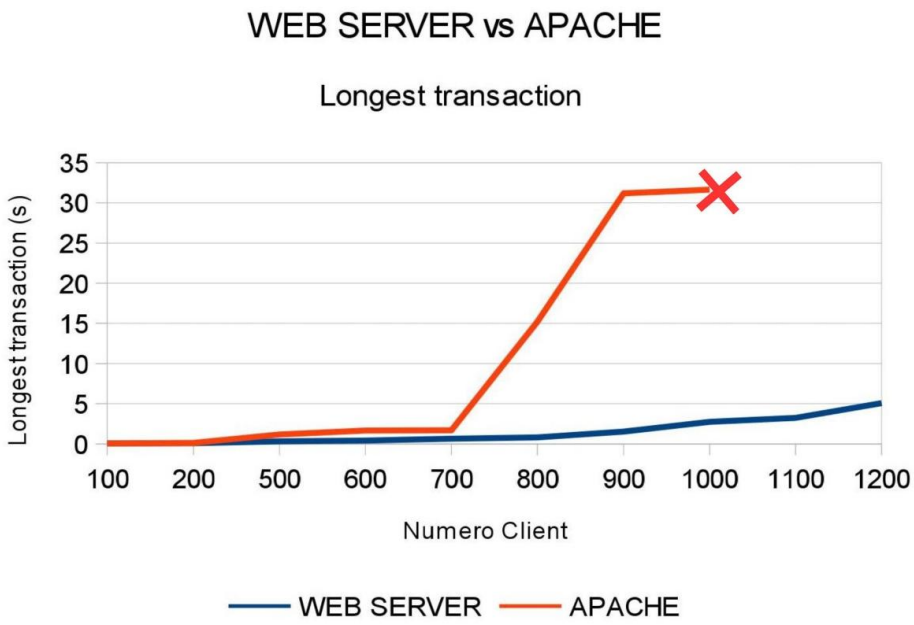


Grafico 4

Nel Grafico 4 sono visualizzati i risultati medi dei test relativi alla durata del più lungo trasferimento di una singola immagine, al crescere del numero dei client. Osserviamo infatti che, fino a valori pari a circa 700 client, l'andamento di entrambe le curve sembra essere analogo, seppur leggermente migliore nel web server progettato. Superato tale valore, i tempi divergono notevolmente fino al punto in cui il server Apache raggiunge valori di circa 30 secondi sino a fermarsi al raggiungimento di 1000 client, come precedentemente accennato. Diversamente avviene nel web server progettato nel quale la variazione di prestazioni cresce lentamente fino ad un valore massimo di circa 5 secondi, decisamente al di sotto del tempo massimo stimato con Apache.

9.2.3 Response Time

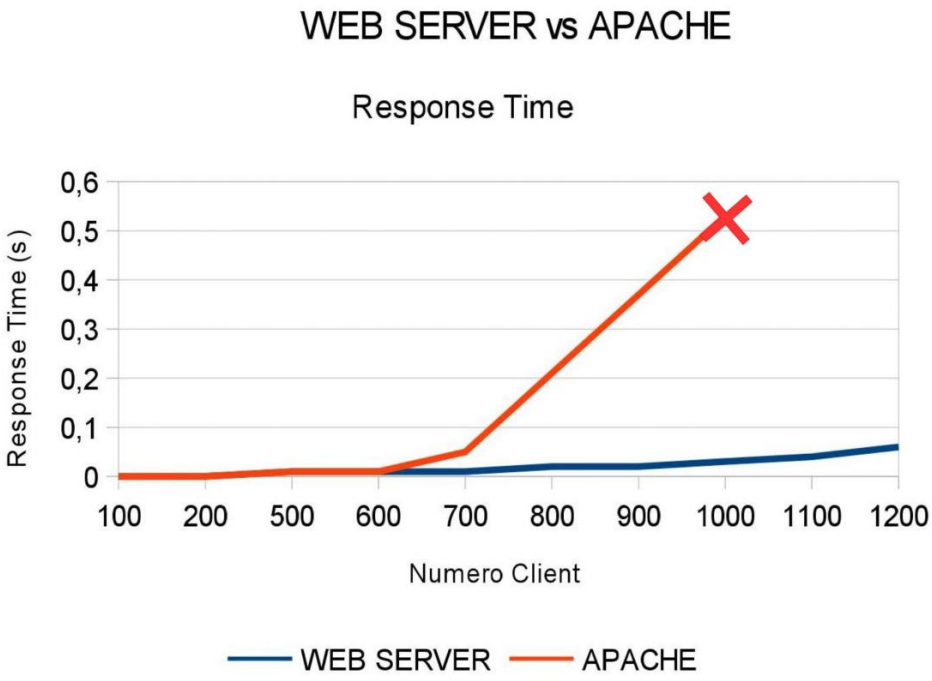


Grafico 5

In questo grafico sono rappresentati i risultati relativi ai tempi di risposta medi dei server al variare del numero dei client contemporaneamente connessi. E' possibile osservare come, fino al raggiungimento di circa 600 client, l'andamento delle prestazioni risulta pressoché uguale; superato tale valore, i risultati di Apache tendono ad aumentare notevolmente in proporzione al numero di client connessi, fino al valore critico di circa 1000 client in cui tale server si arresta. Il Web Server progettato invece, mantiene un andamento continuo anche per lo stesso numero di client, non superando mai un tempo di risposta pari a 0.1 secondi.

10 Limitazioni Riscontrate

10.1 WURFL

Il Web Server implementato è in grado di ridimensionare le immagini in base alle dimensioni del dispositivo client; per fare ciò, come già specificato nella sezione Scelte Progettuali, si è fatto uso del file *Wurfl.xml*.
Un problema però riscontrato nei vari test da noi effettuati riguarda il fatto di non trovare nel 90% (o forse più) dei casi la corrispondenza UserAgent_client - UserAgent_WURFL, ossia gli UserAgent ricevuti mediante richieste HTTP sembrano essere (quasi) sempre diversi da quelli dichiarati in WURFL. Questo inconveniente è sembrato in un primo momento irrisolvibile, ma in seguito, studiando a fondo la struttura degli UserAgent, si è notata una certa somiglianza tra UserAgent di dispositivi aventi lo stesso Sistema Operativo; qui di seguito vengono mostrati alcuni esempi nella quale si evidenzia il modello commerciale del dispositivo:

ANDROID

- Mozilla/5.0 (Linux; U; Android 4.1.2; it-it; GT-I9100 Build/JZO54K) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile Safari/534.30
- Mozilla/5.0 (Linux; U; Android 5.0; it-it; SM-N915P Build/LRX22C) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.96 Mobile Safari/537.36
- Mozilla/5.0 (Linux; U; Android 4.4; it-it; SM-G800F Build/KOT49H) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/30.0.0.0 Mobile Safari/537.36

WINDOWS PHONE

- Mozilla/5.0 (compatible; MSIE 9.0; Windows Phone OS 7.5; Trident/5.0; IEMobile/9.0; NOKIA; Lumia 505)
- Mozilla/5.0 (compatible; MSIE 10.0; Windows Phone 8.0; Trident/6.0; IEMobile/10.0; ARM; Touch; NOKIA; Lumia 822)
- Mozilla/5.0 (compatible; MSIE 9.0; Windows Phone OS 7.5; Trident/5.0; IEMobile/9.0; NOKIA; Lumia 900)

SYMBIAN

- Mozilla/5.0 (Symbian/3; Series60/5.2 NokiaE6-00/021.002; Profile/MIDP-2.1 Configuration/CLDC-1.1) AppleWebKit/533.4 (KHTML, like Gecko) NokiaBrowser/7.3.1.16 Mobile Safari/533.4 3gpp-gba
- Mozilla/5.0 (SymbianOS/9.3; U; Series60/3.2 NokiaE75-1/008.38.0; Profile/MIDP-2.1 Configuration/CLDC-1.1) AppleWebKit/413 (KHTML, like Gecko) Safari/413
- Mozilla/5.0 (SymbianOS/9.3; Series60/3.2 NokiaE72-1/100.000; Profile/MIDP-2.1 Configuration/CLDC-1.1) AppleWebKit/525 (KHTML, like Gecko) Version/3.0 BrowserNG/7.1.13126

BLACKBERRY

- Mozilla/5.0 (BlackBerry; U; BlackBerry 9900; en) AppleWebKit/534.11+ (KHTML, like Gecko) Version/7.1.0.346 Mobile Safari/534.11+
- Mozilla/5.0 (BlackBerry; U; BlackBerry 9800; en-US) AppleWebKit/534.1+ (KHTML, like Gecko) Version/6.0.0.201 Mobile Safari/534.1+
- Mozilla/5.0 (BlackBerry; U; BlackBerry 9850; en) AppleWebKit/534.11+ (KHTML, like Gecko) Version/7.0.0.254 Mobile Safari/534.11+

Per questo motivo si è deciso di implementare un set di funzioni, incluse nel file *UserAgentAnalyzer.h*, nella quale si cerca proprio di risalire al nome del dispositivo partendo dalla decodifica del sistema operativo incluso nello UserAgent.
Inoltre la scelta di effettuare ricerche all'interno del file di WURFL, dalle dimensioni di circa 20 MB, è sembrata piuttosto onerosa; la presenza poi, all'interno dello stesso file, di informazioni per lo più inutili ai nostri scopi, ha fatto sì che la presenza di un Database all'interno del nostro progetto rendesse le operazioni di ricerca più efficienti.
UserAgent di dispositivi Apple sembrano non essere presenti all'interno del file di WURFL; per questo motivo il Web Server non sarà in grado di rilevare la risoluzione di tali dispositivi.

10.2 Browser Dispositivi Mobili

Si consiglia infine di non testare il funzionamento del Web Server su dispositivi mobili attraverso l'utilizzo di browser preinstallati di default. Da varie ricerche effettuate sulla rete risulta infatti la presenza di numerosi bug all'interno di tali browser (soprattutto su Android) che potrebbero influire sulle prestazioni del Server implementato. Si consiglia quindi di installare sul dispositivo mobile uno dei browser commerciali attualmente a disposizione nei vari App Store: Opera, Chrome, Safari.
Si sconsiglia altresì l'utilizzo del browser Firefox: nonostante le sue ridotte dimensioni e la sua velocità di esecuzione, presenta il grave difetto, nel nostro caso, di spedire UserAgent obsoleti che rendono il parsing degli stessi molto complicato ai fini del rilevamento delle caratteristiche del dispositivo client.