Project Heimdall

Proposta di implementazione per un web switch concorrente two-way di livello 7 (OSI) con politiche di bilanciamento del carico state less e state aware

Alessio Moretti - 0187698
Andrea Cerra - 0167043
Claudio Pastorini - 0186256

Corso di Ingegneria di Internet e del Web - A.A. 2014/2015
Università degli studi di Roma Tor Vergata
Facoltà di Ingegneria Informatica

Roma, 19 febbraio 2016

Indice

1 Example section

Yggdrasil, l'albero del mondo, che congiunge i nove regni del cosmo con Asgard, la dimora degli dei.

Heimdall, custode del Bifröst

Sample text and a reference[1]. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec at lorem varius, sodales diam semper, congue dui. Integer porttitor felis eu tempor tempor. Proin molestie maximus augue in facilisis. Phasellus eros dui, blandit eu nibh ut, pharetra porta enim. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam ullamcorper risus pretium est elementum, eget egestas lorem fermentum. Etiam auctor nisi purus, vitae scelerisque augue vehicula sed. Ut eu laoreet ex. Mauris eu mi a tortor gravida cursus eget sit amet ligula.

Figura 1: Thor di Asgard, *figlio di Odino*

2 Introduzione

2.1 Perché Heimdall?

Heimdall è il personaggio dell'universo Marvel, ispirato all'omonimo dio della mitologia norrena, egli è il guardiano del regno di Asgard e del Bifröst. Quest'ultimo è il ponte che unisce la Terra alla dimora degli dei ed Heimdall, come suo custode, ha il compito di aprirlo ed indirizzarlo verso gli altri mondi, permettendo solamente a chi è degno di attraversare le distese dello spazio. Ci piace pensare che questo sia un po' il ruolo del software nato dal nostro progetto: che sia in grado di scegliere come meglio indirizzare le connessioni in arrivo, ponendosi come "guardiano" di un cluster di server che fa ad esso capo. Quindi un **web switch** che sia funzionale sia per ricevere o trasmettere pacchetti di un regolare traffico HTTP, che per bilanciare il carico dello stesso traffico in arrivo sulle varie macchine.

2.2 Web switch di livello 7

Nella terminologia delle reti informatiche uno **switch** è un commutatore a livello datalink, ovvero un dispositivo che si occupa di instradare opportunamente, attraverso le reti LAN, selezionando i frame ricevuti e reindirizzandoli verso la macchina appropriata a seconda di una propria tabella di inoltro. Un **web switch**, a livello applicativo, è capace di reindirizzare i dati in funzione dei pacchetti che riceve, analizzandone il contenuto e decidendo opportunamente la destinazione, occupandosi allo stesso tempo di reinoltrare anche l'eventuale risposta della macchina selezionata verso il client che l'ha generata.
Le applicazioni sono molteplici per l'implementazione a livello applicativo: può essere considerato un **proxy**, oppure, selezionando opportunamente la macchina con più velocità di risposta o con minore pressione, può agire come **bilanciatore di carico**. Infatti ognuno dei client che fa richiesta, ad esempio, per uno specifico sito web, invia un pacchetto ad un indirizzo IP pubblico che corrisponde a quello del nostro switch applicativo. Questi, dopo aver correttamente letto il pacchetto, si occupa di consultare una tabella di inoltro generata con una determinata **politica di scheduling** e quindi

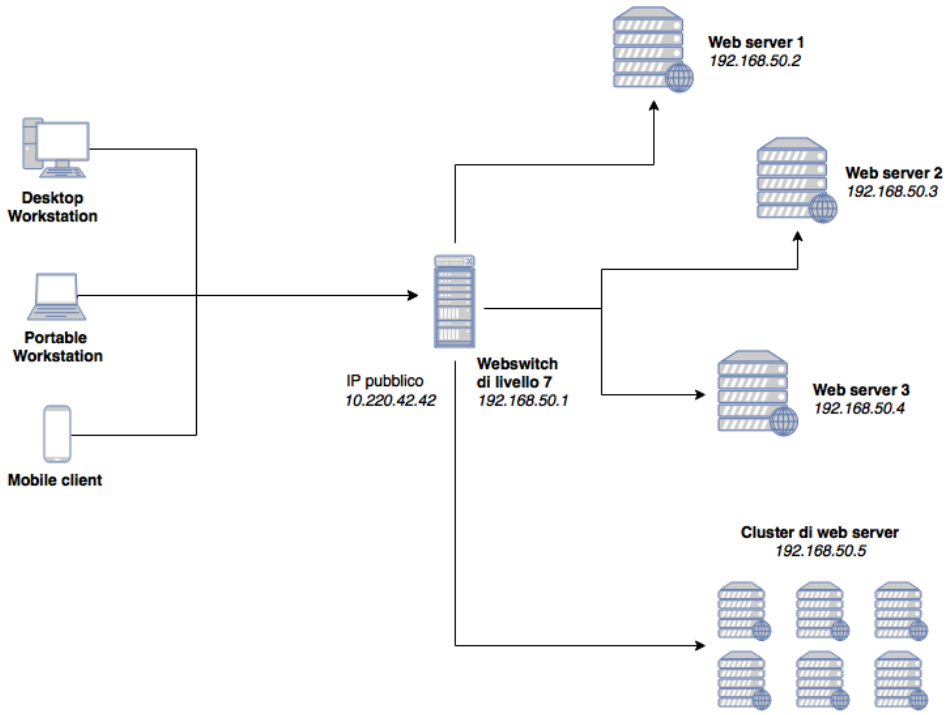

Figura 2: Esempio di uno *switch di livello 7 (OSI)*

gestire l'inoltro della richiesta ed il reinoltro della risposta del webserver. Tutto questo in maniera totalmente trasparente al client, qualsiasi sia la macchina che ha effettivamente risposto, che sia un web server oppure un cluster di macchine associate ad un ulteriore switch.

2.3 Assunzioni progettuali sul cluster

Nella fase di progettazione e realizzazione sono state definite le seguenti assunzioni:

- Ognuna delle macchine del cluster dispone di un web server Apache[2] in ascolto sulla porta 80
- Ognuna delle macchine monta il modulo ApacheStatus come monitor di carico

3 Architettura

3.1 Overview

3.2 Il processo principale

Il cuore di Heimdall sta ovviamente nel file principale del programma, il *main.c*, una volta avviato il programma, il processo principale prenderà vita nel sistema ed eseguirà le inizializzazioni dei componenti principali, tra questi abbiamo:

- Config
- Log
- Scheduler
- Pool Manager

Possiamo dividere questi 4 componenti in 2 categorie, i primi due infatti sono semplici strutture statiche all'interno del programma, sviluppate utilizzando il design pattern *Singleton*. Questo per garantire che questi due componenti vengano inizializzati una sola volta all'avvio del programma. I successivi due componenti, lo *Scheduler* e il *Pool Manager* sono invece thread del processo principale, questo perché come sarà più chiaro nei relativi paragrafi, i due componenti hanno da svolgere alcuni compiti dedicati che altrimenti sarebbero dovuti essere eseguiti dal programma principale intaccando le prestazioni del sistema dovute al fatto che non veniva sfruttata alcun tipo di concorrenza.

3.2.1 I processi figli e la memoria condivisa

Fatte le prime inizializzazioni il processo principale si avvia ad eseguire il suo compito ma prima di questo è necessario creare i processi figli che saranno incaricati di servire le connessioni HTTP. Tramite la funzione *do_prefork()* presente nel file *main.c* il processo principale esegue una serie di operazioni per creare i processi figli e inizializzare una memoria condivisa che verrà utilizzata appunto per la comunicazione tra il processo padre e i processi

figli, per motivi che saranno più chiari in seguito chiameremo da adesso in poi i processi figli: *Worker*.

Evidenziamo qui i tratti principali della funzione:

```
ConfigPtr config = get_config();

int n_prefork = 0;
ThrowablePtr throwable = str_to_int(config->pre_fork, &n_prefork);
if (throwable->is_an_error(throwable)) {
    get_log()->t(throwable);
}
```

Dal Config viene recuperato il numero di processi figli da creare, come verrà spiegato nel paragrafo relativo, questo valore potrà essere specificato dall'utente all'interno del file di configurazione. Con questo parametro oltre ad eseguire un numero finito di volte la chiamata di sistema *fork()* viene inizializzato un blocco di memoria condivisa di grandezza fissa, all'interno del quale vengono salvati 4 array, vediamoli nel dettaglio:

- ***worker_array***: In questo array vengono memorizzati i *pid_t* dei processi figli creati.
- ***worker_busy***: Questo è un array di flag e viene utilizzato per tenere traccia dello stato del lavoro di un worker. Nello specifico alla posizione i-esima che viene identificata a partire dal primo array possiamo trovare due valori distinti, un 1 per indicare che il worker è occupato a gestire una connessione e uno 0 per indicare che il worker è libero e in attesa di nuove connessioni.
- ***worker_counter***: Questo array serve a contare quante connessioni sono state assegnate al worker, questo parametro sarà chiarito nel paragrafo del Pool Manager e servirà alla schedulazione dei worker.
- ***worker_server***: Questa è una struttura di passaggio utilizzata dal thread Scheduler, che ricordiamo essere uno dei componenti inizializzati all'avvio di Heimdal, la sua funzionalità sarà descritta in maniera più adeguata nel relativo paragrafo.

Di seguito il codice che si occupa dell'inizializzazione della memoria condivisa:

```
    int total_size = 0;
total_size+=sizeof(THPSharedMem);
total_size+=sizeof(pid_t)*n_prefork; // Array worker
total_size+=sizeof(int)*n_prefork; // Array busy
total_size+=sizeof(int)*n_prefork; // Array counter
total_size+=sizeof(Server)*n_prefork; // Array server

// Initializes Shared memory
void *start_mem = init_shm(WRK_SHM_PATH, total_size, WRK_SEM_PATH);
if (start_mem == NULL)
    exit(EXIT_FAILURE);

get_log()->i(TAG_THREAD_POOL, "Shared␣memory␣start␣from␣%p", start_mem);

// Mappgin shared memory segment
THPSharedMemPtr worker_pool = start_mem;
worker_pool->worker_array   = start_mem+sizeof(THPSharedMem);
worker_pool->worker_busy    = start_mem+sizeof(THPSharedMem)+sizeof(pid_t)*n_prefork;
worker_pool->worker_counter = start_mem+sizeof(THPSharedMem)+sizeof(pid_t)*n_prefork+sizeo
worker_pool->worker_server  = start_mem+sizeof(THPSharedMem)+sizeof(pid_t)*n_prefork+sizeo

int i = 0;
for (i = 0; i < n_prefork; ++i){
    worker_pool->worker_array[i]    = 0;
    worker_pool->worker_busy[i]     = 0;
    worker_pool->worker_counter[i]  = 0;
}
```

La porzione di codice qui sopra mette in evidenza le righe che si occupano della creazione, della mappatura e dell'inizializzazione della memoria condivisa. La memoria condivisa utilizzata è quella appartenente allo standard POSIX, l'implementazione della wrapper proprietaria *init_shm()* è presente all'interno dei file *shared_mem.c* e *shared_mem.h*.

Fatto questo siamo finalmente giunti alla creazione dei processi figli, ancora una volta evidenziamo qui la porzione di codice interessata:

```
int children;
for (children = 0; children < n_prefork; ++children){
```

```
        pid_t child_pid;
        errno = 0;

        child_pid = fork();
        if (child_pid == -1)
            get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "do_

        // Child
        if (child_pid == 0){

            // see worker.c
            start_worker();
            break;

        }else{

            if(sem_wait(sem) == -1)
                return get_throwable()->create(STATUS_ERROR, "sem_wait", "do_prefork");

            // Scan array and set worker to first position available
            int i, flag = 0;
            for (i = 0; i < n_prefork; ++i){
                if (worker_pool->worker_array[i] == 0){
                    worker_pool->worker_array[i] = child_pid;
                    flag = 1;
                    break;
                }
            }

            if(sem_post(sem) == -1)
                return get_throwable()->create(STATUS_ERROR, "sem_post", "do_prefork");

            if (flag == 0){
                return get_throwable()->create(STATUS_ERROR, "Cannot␣add␣worker_pid␣to␣array",
            }
        }
    }
```

Come vediamo viene eseguito un for *n_prefork* volte dove *n_prefork* è il numero di processi che si vuole creare. Il processo figlio nasce e avvia l'esecu-

zione della sua funzione main e cioè la funzione *start_worker()*, il processo padre invece si occupa di accedere in memoria condivisa per aggiornare le strutture dati che abbiamo spiegato poco fa.

Facciamo notare che l'esecuzione della chiamata di sistema *fork()* non viene casualmente eseguita in questo momento, bensì si sfrutta il fatto che la *fork()* copia l'intera memoria del processo padre verso il figlio e grazie a questo il figlio si ritroverà i componenti come *Config*, *Log* e *Memoria Condivisa* già inizializzati e referenziati.

3.2.2 Server in ascolto

Una volta completata l'inizializzazione dei componenti di base di Heimdall il processo principale si avvia a svolgere il suo incarico cioè quello di mettersi in ascolto su una porta (configurabile nel file di configurazione) in attesa di nuove connessione.

```
        int port = 0;
throwable = str_to_int(config->server_main_port, &port);
if (throwable->is_an_error(throwable)) {
    log->t(throwable);
}

// Creates a new server
int sockfd;
throwable = create_server_socket(TCP, port, &sockfd);
if (throwable->is_an_error(throwable)) {
    log->t(throwable);
    exit(EXIT_FAILURE);
}

log->i(TAG_MAIN, "Created␣new␣server␣that␣is␣listening␣on␣port␣%d", port);

// Starts listening for the clients
int backlog = 0;
throwable = str_to_int(config->backlog, &backlog);
if (throwable->is_an_error(throwable)) {
    log->t(throwable);
}
```

```
throwable = listen_to(sockfd, backlog);
if (throwable->is_an_error(throwable)) {
    log->t(throwable);
    exit(EXIT_FAILURE);
}

log->i(TAG_MAIN, "Ready to accept incoming connections...");

int count = 0;

// Starts to listen incoming connections
while(TRUE) {

    // Accepts new connection
    int new_sockfd;
    throwable = accept_connection(sockfd, &new_sockfd);
    count++;
    if (throwable->is_an_error(throwable)) {
        log->t(throwable);
        exit(EXIT_FAILURE);
    }

    while(TRUE){
        throwable = th_pool->add_fd_to_array(&new_sockfd);
        if (throwable->is_an_error(throwable)) {
            //get_log()->i(TAG_THREAD_POOL, "No space for FD available, wait for space.");
            throwable->destroy(throwable);
        }else{
            break;
        }
    }

    log->i(TAG_MAIN, "New connection accepted on socket number %d - total %d", new_sockfd,
}
```

Questo è il cuore di Heimdall, come possiamo notare da queste poche righe il processo principale recupera dal config la porta dove mettersi in ascolto, crea un nuovo "server socket" e si mette in attesa dell'arrivo di nuove connessione in un loop infinito. Quando un nuovo client si collega la funzione accept_connection ritorna e il processo principale prosegue la sua strada. Qui vogliamo far notare che lo sviluppo della funzione main è stato pensato

per poter eseguire il minor numero di linee di codice, questo per permettere al processo principale di tornare immediatamente in ascolto di nuove connessioni, infatti dopo aver salvato il file descriptor della socket in una apposita struttura dati (che sarà descritta nel paragrafo worker pool), il processo principale termina il primo ciclo del loop infinito e torna in ascolto di nuove connessioni.

3.3 File di configurazione

I file di configurazione sono gli unici file accessibili all'utente che installa sulla propria macchina Heimdall WebSwitch. Tramite questi file è possibile impostare alcuni parametri per adattare il WebSwitch alle proprie esigenze.

I file di cui stiamo parlando sono due e adesso ne illustreremo i dettagli.

3.3.1 File dei parametri

Questo è il file di configurazione generale di Heimdall. Al suo interno possiamo trovare i seguenti parametri:

- *algorithm_selection*: specifica il tipo di algoritmo che si vuole usare per Heimdall, con il valore 0 si sceglie di usare l'algoritmo STATELESSRR, con il valore 1 si sceglie di usare l'algoritmo STATEFUL.
- *pre_fork*: specifica il numero di proessi figli che verranno creati all'avvio di Heimdall, i processi figli ricordiamo sono i processi che gestiranno le connessioni HTTP. Se il valore di questo parametro viene impostato a 0 Heimdall creerà automaticamente 1 processo figlio per gestire le richieste.
- *print_enable*: abilita le stampe a video se impostato a 1, altrimenti le disabilita.
- *log_level*: specifica il livello di stampe che si vogliono avere sul terminale, i livelli di log saranno spiegati bene nella sezione dedicata.
- *write_enable*: abilita la scrittura dei file di log.

- *log_file_req*: specifica il percorso del file di log per le connessioni richieste a heimdall.
- *log_file_resp*: specifica il percorso del file di log per le risposte fatte alle connessioni.
- *timeout_worker*: specifica il
- *update_time*: questo parametro indica il
- *server_config* specifica il percorso del file di configurazione dovengono specificati i server del cluster collegati a Heimdall.
- *server_main_port*: specifica la porta di ascolto del processo padre di Heimdall per accettare nuove connessioni.
- *backlog*: specifica il valore di backlog passato alla funzione accept.
- *max_fd*: specifica il numero massimo di connessioni persistenti che Heimdall è in grado di gestire. Questo parametro può variare in base al numero massimo di file descriptor che possono essere contemporaneamente aperti nel sistema.
- *sockets_path*: specifica una directory del sistema dove Heimdall potrà creare socket AF_UNIX, necessarie per permettere la comunicazione tra il processo padre e i suoi figli. Heimdall dovrà avere permessi di lettura e scrittura su questa directory.
- *max_thread_pchild*: specifica il numero massimo di thread_request che ogni Worker può creare, e cioè il massimo numero di richieste che possono essere accettate contemporaneamente su una singola connessione persistente.

Rappresentiamo qui con un esempio un file dei parametri:

```
# STATELESSRR 0 STATEFUL 1
algorithm_selection 1

# The number of active processes to handle requests
# If prefork is 0, the system create anyway 1 child process
```

```
pre_fork 10

# If 1 log print to shell is enabled
print_enable 1

# Specify the log level desired
log_level 2

# Write on log file enabled, see log_file_req and log_file_resp variable
write_enable 0

...
...
```

3.3.2 File dei server

Il file dei server è il file dove vengono specificati i server del cluster collegati a Heimdall, questo file è specificato (all'interno del file di configurazione) nel parametro *server_config*. Al suo interno vengono specificati un nome per il server associato ed il relativo indirizzo ip.

Rappresentiamo qui con un esempio un file dei server:

```
# Server 1
Name:bifrost.asgard
IP:192.168.1.4
```

3.3.3 Implementazione dei file di configurazione

La lettura del file dei parametri spetta al parser Heimdall Config, la sua implementazione è presente nei file heimdall_config.c e heimdall_config.h

All'interno del file .h sono presenti alcuni valori necessari al funzionamento del parser, tra questi troviamo:

- Una costante *CONFIGFILE* dove va specificato il percorso del file dei parametri.

```
/*
 * ------------------------------------------------------------------------
 * Relative position to config file
 * ------------------------------------------------------------------------
 */
#define CONFIGFILE "../code/config/heimdall_config.conf"
```

- Una struttura dati di tipo *ConfigPtr*

```
/*
 * ------------------------------------------------------------------------
 * Structure        : Config struct
 * Description      : This struct collect all config value from config file.
 * ------------------------------------------------------------------------
 */
typedef struct config{
    char *algorithm_selection;
    char *pre_fork;
    char *print_enable;
    char *log_level;
    char *write_enable;
    char *log_file_req;
    char *log_file_resp;
    char *timeout_worker;
    char *killer_time;
    char *update_time;
    char *server_config;
    char *server_main_port;
    char *backlog;
    char *max_fd;
    char *sockets_path;
    char *max_thread_pchild;
} Config, *ConfigPtr;
```

Come possiamo notare i campi della struttura sono esattamente le chiavi che abbiamo elencato precedentemente nel paragrafo relativo al file dei parametri. Come si vede dall'esempio i campi all'interno del file sono disposti in un ordine preciso, dettato da una regola ti tipo *chiave:valore*. Il parser esegue la lettura del file una riga per volta, estrae la chiave, il suo valore e passa il tutto ad una funzione di callback.

```
/*
 * ----------------------------------------------------------------------------
 * Function      : _get
 * Description   : Used for get key and value from a line.
 *
 * Param         :
 *   array[]     : line string.
 *   from        : from value start.
 *   to          : escape character where function stop.
 *
 * Return        : string.
 * ----------------------------------------------------------------------------
 */
char *_get(char array[], int from, char escape){

    int from_cpy = from;
    int total = 0;

    while (1) {
        if (array[from_cpy] == escape)
            break;
        ++from_cpy;
        ++total;
    }

    total++; // add \0 space

    char *subset = malloc(sizeof(char) * total);
    if (subset == NULL) {
        fprintf(stderr, "Error␣in␣_get␣config_parser.␣\n");
        return NULL;
    }

    int j;
    for(j = 0; j < total - 1; ++j, ++from) {
        subset[j] = array[from];
    }

    subset[total-1] = '\0';

    return subset;
```

```
}

/*
 * ------------------------------------------------------------------------
 * Function     : init_config
 * Description  : Parse config file and call callback function for return values.
 *
 * Param            :
 *   path           : Path to file be parsed.
 *   config_handler : Callback function, return to main key, value and config reference.
 *   ptr_config     : Pointer to struct.
 *
 * Return       :
 * ------------------------------------------------------------------------
 */
int init_config(const char *path, int config_handler(char *key, char *value, void *p_config),

    singleton_config = ptr_config;

    // open file
    FILE *config_file = fopen(path, "r");
    if (config_file == NULL) {
        fprintf(stderr, "Error␣while␣trying␣to␣open␣config␣file.␣\n");
        return -1;
    }

    // while each line
    char string[MAX_LENGTH];

    while(fgets(string, MAX_LENGTH, config_file)) {

        // skip, line comment or empty line
        if (string[0] == '#' || string[0] == '\n')
            continue;

        char *key = _get(string, 0, ESCAPE_CHARACTER);
        if (key == NULL) {
            fprintf(stderr, "Error␣in␣init_config␣config_parser.␣\n");
            return -1;
        }
```

```
        char *value = _get(string, strlen(key)+1, '\n');
        if (value == NULL) {
            fprintf(stderr, "Error␣in␣init_config␣config_parser.␣\n");
            return -1;
        }

        if(config_handler(key, value, ptr_config) == -1) {
            fprintf(stderr, "Error␣config␣parser,␣no␣key␣'%s'␣found␣in␣Config.␣\n", key);
            return -1;
        }

        free(value);
        free(key);
    }

    fclose(config_file);

    return 0;
}
```

La funzione config_handler riceve la chiave e il valore estratto, ed effettuando un semplice confronto con la chiave esegue l'inserimento del valore all'interno della struct.

```
/*
* ------------------------------------------------------------------------
* Function     : config_handler
* Description  : Callback function, see heimdall_config.c for more information.
*
* Return       : 0 if ok, -1 if error.
* ------------------------------------------------------------------------
*/
int config_handler(char *key, char *value, void *p_config) {

    Config* config = (Config *)p_config;

    if (strcmp(key, "algorithm_selection") == 0) {
        if (asprintf(&config->algorithm_selection, "%s", value) == -1)
            return -1;

    }else if (strcmp(key, "pre_fork") == 0) {
        if (asprintf(&config->pre_fork, "%s", value) == -1)
```

```
            return -1;

            ...
            ...

    }else
        return -1;  /* unknown key, error */

    return 0;
}
```

Il funzionamento del parser per il file dei server è a grandi linee lo stesso, il codice relativo può essere trovato nei file server_config.c e server_config.h. Unica differenza è che questo parser non ritorna una struct con tutti i valori del file config bensì ritorna una struttura contenente questi valori:

```
/*
 * ------------------------------------------------------------------------
 * Structure        : Config struct
 * Description      : This struct collect all config value from config file.
 * ------------------------------------------------------------------------
 */
typedef struct server_config{
    char **servers_names;
    char **servers_ip;
    int total_server;
} ServerConfig, *ServerConfigPtr;
```

Ossia un array di puntatori ai nomi, un array di puntatori ai relativi indirizzi ip e un intero che indica il numero totale dei server identificati nel file, quest'ultimo utile per scansionare gli array descritti.

Come ultima caratteristica di implementazione facciamo notare che la lettura dei file e quindi la relativa esecuzione dei parser viene eseguita solamente una volta all'avvio del programma, questo per ridurre gli evidenti accessi di I/O che sarebbero necessari per estrarre di volta in volta il valore richiesto. Per fare ciò è stato quindi utilizzato un approccio alla programmazione chiamato *Singeleton*, infatti una volta eseguito il parser dei file, il riferimento alla struttura viene salvato nella variabile globale:

```
/*
```

```
 * ------------------------------------------------------------------------
 * Description  : Global variable, singleton instance of Config
 * ------------------------------------------------------------------------
 */
void *singleton_config = NULL;
```

Un semplice controllo viene comunque eseguito ed è lo stesso controllo che avvia l'esecuzione del parser la prima volta, se la variabile è NULL viene eseguita la lettura del file ed inizializzata la relativa struttura. Grazie poi alla chiamata fork() che viene eseguita qualche momento dopo tutti i figli riceveranno "gratuitamente" in eredità la struttura Config.

3.4 Logging

Per poter gestire meglio l'output su console e su file abbiamo deciso di implementare una nostra "classe" di log. Prima di descrivere il codice vogliamo far notare che abbiamo sviluppato il *logger* seguendo un approccio orientato agli oggetti e seguendo il design pattern *Singleton*. Infatti questo è uno dei componenti che viene inizializzato nel processo principale all'avvio del programma, nello specifico tramite la chiamata *get_log()* eseguita nel *main()*:

```
        Log *new_log() {

    ConfigPtr config = get_config();

    Log *log = malloc(sizeof(Log));
    if (log == NULL) {
        fprintf(stderr, "Memory␣allocation␣error␣in␣new_log!");
        exit(EXIT_FAILURE);
    }

    // retrieving log file pointer or allocating it
    req_log = fopen(config->log_file_req, "a+");
    if (req_log == NULL) {
        fprintf(stderr, "Error␣in␣log␣file␣opening!\n");
    }

    // retrieving log file pointer or allocating it
    resp_log = fopen(config->log_file_resp, "a+");
```

```
    if (resp_log == NULL) {
        fprintf(stderr, "Error in log file opening!\n");
    }

    // Set "methods"
    log->d = d;
    log->i = i;
    log->e = e;
    log->r = r;
    log->t = t;

    return log;
}

Log *get_log() {

    if (singleton_log == NULL) {
        singleton_log = new_log();
    }

    // return singleton
    return singleton_log;
}
```

Abbiamo già incontrato una simile struttura, infatti ricordiamo che anche il parser del file di configurazione segue lo stesso approccio. Tornando al codice vediamo che la funzione *get_log()* non fa altro che richiamare la funzione *new_log()* nel caso in cui il riferimento al singleton sia NULL, e questo in generale avviene soltanto all'avivo del programma. La funzione *new_log()* è quella che si occupa dell'inizializzazione vera e propria dell'oggetto *Log*, viene infatti eseguita una *malloc()* per allocare una struttura di tipo Log, e vengono inizializzati i puntatori ai file di logging e alle funzioni. In questo modo abbiamo organizzato tutti i "metodi" all'interno di un unico oggetto Log:

```
        typedef struct log {
    int (*d)(const char* tag, const char *format, ...);
    int (*i)(const char* tag, const char *format, ...);
    int (*e)(const char* tag, const char *format, ...);
    int (*r)(int type, void *arg, char *host, int pid);
```

```
    void (*t)(ThrowablePtr throwable);
} Log, *LogPtr;
```

In questo modo per poter stampare a video sarà sufficiente richiamare la funzione:

```
get_log()->i(TAG, "Hello World!");
```

Le funzioni *d()* (debug), *e()* (error) e *i()* (informazioni) sono state implementate per organizzare le stampe a video. Infatti nel file di configurazione è possibile impostare il livello di stampe che si desidera vedere tramite il parametro *log_level*. Le funzioni per la loro implementazione si equivalgono a meno del livello di log che trattano, ne riportiamo per questo solamente una:

```
static int i(const char* tag, const char *format, ...) {

    ConfigPtr config = get_config();

    int byte_read = 0;

    int level = 0;
    ThrowablePtr throwable = str_to_int(config->log_level, &level);
    if (throwable->is_an_error(throwable)) {
        t(throwable);
    }

    int print_enable = 0;
    throwable = str_to_int(config->print_enable, &print_enable);
    if (throwable->is_an_error(throwable)) {
        t(throwable);
    }

    if (INFO_LEVEL >= level && print_enable == 1) {

        char *formatted_str;

        va_list arg;
        va_start (arg, format);
        byte_read = vasprintf(&formatted_str, format, arg);
        va_end (arg);
```

```
        char *output;
        byte_read = asprintf(&output, "%s: I/%s: %s", timestamp(), tag, formatted_str);

        free(formatted_str);

        printf("%s \n", output);
        fflush(stdout);
        free(output);
    }

    return byte_read;
}
```

Come vediamo viene dapprima recuperato dal file di configurazione il livello impostato per il log, inoltre viene anche recuperato il valore del parametro *print_enable* (ricordiamo che è possibile disabilitare completamente le stampe), eseguito un semplice controllo il codice all'interno del terzo if non è altro che una copia della funzione *printf()*, l'unica modifica che viene fatta è solo quella di formattare l'output in modo che possa essere facilmente leggibile. Infatti l'aggiunta del parametro tag e del timestamp è stato di fondamentale importanza per identificare nella console quale componente stava effettivamente stampando.

Per concludere descriviamo la funzione *r()* (response) è quella utilizzata per scrivere sui file di logging, in questo modo abbiamo raccolto da una parte il codice di formattazione del file di log. Invece la funzione *t()* (throwable) non è altro che un'estensione della funzione *e()*, questa viene utilizzata per stampare a schermo gli oggetti Throwable.

3.5 Gestione degli errori

Come detto in precedenza, Heimdall, pur essendo scritto in C (linguaggio per definizione e per storia di tipo procedurale), è stato concepito avendo in mente la programmazione orientata agli oggetti e sopratutto al linguaggio Java. Questo ci ha portato alla creazione della struttura dati Throwable, che ricorda molto la class omonima di Java[3]. Questa struttura rappresenta il

valore di ritorno di ogni funzione del sistema, tale che essa sia l'unico modo per poter gestire gli errori.

```
/*
 * ------------------------------------------------------------------------
 * Structure    : typedef struct throwable
 * Description  : This struct collect all functions pointers and attributes for messages.
 *
 * Data:
 *  status      : Error level, 0 or -1 see macro in this .h.
 *  message     : The error message.
 *  stack_trace : Contains all function name where error occurs.
 *
 * Functions:
 *  create  : Pointer to create_throwable function.
 * ------------------------------------------------------------------------
 */
typedef struct throwable {
    int status;
    char *message;
    char *stack_trace;

    struct throwable* (*create)(int status, char *msg, char *stack_trace);
    struct throwable* (*thrown)(struct throwable* self, char *stack_trace);
    int (*is_an_error)(struct throwable* self);

    void (*destroy)(struct throwable *self);
} Throwable, *ThrowablePtr;
```

Il *Throwable* è definito tramite le variabili **status**, **message** e **stack_trace**.
La variabile **status** può assumere uno dei seguenti valori:

- *STATUS_OK*, se non ci sono errori;
- *STATUS_ERROR*, se si è verificato qualche errore.

La variabile **message** contiene la descrizione del problema (che può derivare da una chiamata di sistema e quindi è utile la funzione *get_error_by_errno()*).
La variabile **stack_trace** contiene la lista delle funzioni che il Throwable ha attraversato ed è di fondamentale importanza per comprendere la dinamica dell'errore.
Il Throwable è istanziabile tramite la chiamata *get_throwable().create()*, a cui

vengono passati i parametri sopra descritti. In particolar modo, stack_trace conterrà solamente il nome della funzione che sta invocando la chiamata. Avendo reso omogenee tutte le chiamate a funzioni, risulta semplificato il controllo, l'inoltro o la stampa degli errori.
Per esempio, all'interno della funzione *get_data()*, utile per il parsing delle risposte ricevute dal modulo di Apache Status (si veda 4.3.1), abbiamo:

```
 if (strcmp(text, "idleworkers") == 0) {
  throwable = str_to_int(text_data, &(self->idle_workers));
  if (throwable->is_an_error(throwable)) {
    return throwable->thrown(throwable, "get_data.idle_workers");
  }
}

return get_throwable()->create(STATUS_OK, NULL, "get_data");
```

In questo caso viene chiamata la funzione *str_to_int()*, che ritorna un throwable (tale funzione è una wrapper della funzione *strotol()*, che effettua la conversione di una stringa in un intero). Il throwable viene quindi controllato tramite la chiamata *is_an_error()*, che, verificandone lo status, ritorna un booleano. Nel caso il valore di ritorno sia positivo, viene eseguito il thrown dell'errore, specificando da dove lo si sta effettuando (in questo caso, nella funzione *get_data_idle_workers*).
La struttura è allocata dinamicamente tramite la funzione *create()* e, di conseguenza, è necessario liberare lo spazio utilizzato: se la funzione *is_an_error()* non rileva alcun errore, provvederà essa stessa alla deallocazione; in caso contrario, lo sviluppatore o la chiamata a *log->t()* libererà la memoria, dopo aver stampato un report dell'errore.

3.6 Pool manager

Abbiamo già nominato questo componente nel paragrafo sul processo principale. Il *Pool Manager* è un thread del processo principale creato all'avvio del programma. Il compito principale di questo thread è di gestire la schedulazione dei worker, vediamo in dettaglio questo come avviene.

Come già detto l'inizializzazione del Pool Manager viene eseguita dal main con queste righe di codice:

```
    // Initializes Thread Pool
    ThreadPoolPtr th_pool = get_thread_pool();
    if (th_pool == NULL)
        exit(EXIT_FAILURE);
```

La funzione *get_thread_pool()* è presente all'interno dei file *thread_pool.c* e *thread_pool.h* ed è definita come segue:

```
ThreadPoolPtr get_thread_pool() {

        if (singleton_thdpool == NULL) {
                singleton_thdpool = init_thread_pool();
    }

    // return singleton
    return singleton_thdpool;
}

static ThreadPoolPtr init_thread_pool() {

        get_log()->i(TAG_THREAD_POOL, "Thread␣pool␣start.");

        pthread_t t1;
        int born;

        born = pthread_create(&t1, NULL, init_pool, NULL);
        if (born != 0) {
                get_log()->e(TAG_THREAD_POOL, "Error␣in␣pthread_create");
                return NULL;
        }

        ThreadPoolPtr th_pool = malloc(sizeof(ThreadPool));
    if (th_pool == NULL) {
        get_log()->e(TAG_THREAD_POOL, "Memory␣allocation␣error␣in␣init_thread_pool!");
        return NULL;
    }

    th_pool->thread_identifier  = t1;
    th_pool->add_fd_to_array    = add_fd_to_array;
    th_pool->print_fd_array     = print_fd_array;
```

```
    return th_pool;
}
```

Ancora una volta l'approccio utilizzato per la creazione del Pool Manager è il *Singleton*, infatti esiste una sola istanza del Pool Manager all'interno di tutto il programma. La funzione *init_thread_pool()* conferma quando detto fin ora, infatti questa funzione che viene eseguita dal main non fa altro che creare un thread e una struttura *ThreadPoolPtr*.

La *pthread_create()* specifica come funzione di start per il thread la funzione *init_pool()* che riportiamo di seguito:

```
        static void *init_pool(void *arg){

        // detach itself
        pthread_detach(pthread_self());

        ConfigPtr config = get_config();

    max_fd = 0;
    ThrowablePtr throwable = str_to_int(config->max_fd, &max_fd);
    if (throwable->is_an_error(throwable)) {
        get_log()->t(throwable);
        exit(EXIT_FAILURE);
    }

    // Init static array and save pointer to global variable
    fd_array = malloc(sizeof(int) * max_fd);
    if (fd_array == NULL) {
        get_log()->e(TAG_THREAD_POOL, "Memory␣allocation␣error␣in␣init_pool!");
        exit(EXIT_FAILURE);
    }

        int i;
    for (i = 0; i < max_fd; ++i){
                fd_array[i] = 0;
    }

    // enter in thread pool loop
    thread_pool_loop();
```

```
    // Never reached
    return arg;
}
```

Queste sono le prime linee di codice che vengono eseguite dal thread Pool Manager. La prima cosa che viene fatta è quella di recuperare dal file di configurazione il numero massimo impostato di file descriptor disponibili per Heimdall, come già spiegato nel paragrafo del file di configurazione questo parametro indica sostanzialmente il numero massimo di connessioni che Heimdall può gestire concorrentemente. Infatti ogni volta che un client si collega viene generato un file descriptor per la socket di comunicazione, questi file descriptor sono però un numero limitato che dipende dalla configurazione del sistema operativo, quando Heimdall raggiungerà questo limite non accetterà più connessioni finche non verrà liberato spazio. Una volta inizializzato questo array il thread si mette in loop chiamando la funzione *thread_pool_loop()*. Per non annoiare la lettura con troppo codice evidenziamo solamente i tratti principali della funzione.

```
for (;;) {

        int fd = 0;
throwable = get_fd(&fd);
if (throwable->is_an_error(throwable)) {
    //get_log()->i(TAG_THREAD_POOL, "No fd to serve.");
    throwable->destroy(throwable);
    continue;
}

...
...
```

All'interno della funzione *thread_pool_loop()* è presente, come è di facile intuizione dato il nome, un loop infinito. Questo perché il thread eseguirà una costante verifica dell'esistenza di un file descriptor presente all'interno dell'array, infatti la funzione *get_fd()* è definita come segue:

```
        static ThrowablePtr get_fd(int *fd_ptr){

        int s = 0;
        s = pthread_mutex_lock(&mtx_wait_request);
        if (s != 0)
                get_log()->e(TAG_THREAD_POOL, "Error in pthread_mutex_lock");

    // Scan array and get the first fd != 0
    int i, flag = 0;
    for (i = 0; i < max_fd; ++i){

        if (fd_array[i] != 0){
            *fd_ptr = fd_array[i];
            fd_array[i] = 0;
            flag = 1;
            break;
        }
    }

        s = pthread_mutex_unlock(&mtx_wait_request);
        if (s != 0)
                get_log()->e(TAG_THREAD_POOL, "Error in pthread_mutex_unlock");

    if (flag == 0){
        return get_throwable()->create(STATUS_ERROR, "Cannot get fd", "get_fd");
    }else{
        return get_throwable()->create(STATUS_OK, NULL, "get_fd()");;
    }
}
```

Come vediamo la funzione si occupa dell'accesso all'array dei file descriptor che abbiamo inizializzato prima, facciamo notare che l'accesso all'array è gestito da un semaforo mutex, questo perché anche il main accede a questa struttura quando arrivano nuove conessioni, infatti se ricordiamo quanto detto nel paragrafo del processo principale quando un client si collega la funzione *accept_connection()* ritorna il file descriptor associato alla socket, questo viene poi messo all'interno dell'array dei file descriptor tramite la funzione *add_fd_to_array()*. Ora che abbiamo spiegato anche l'ultimo passaggio della funzione main capiamo che il thread pool è l'incaricato di verificare che ci siano nuovi file descriptor da servire. La funzione *get_fd()* è di facile

lettura, viene semplicemente eseguito un for sull'array dei file descriptor e viene ritornato il primo file descriptor disponibile.

Una volta recuperato il file descriptor della connessione è necessario individuare un worker che la gestisca, questo compito è sempre svolto dalla funzione *thread_pool_loop()* che esegue il seguente codice:

```
int i, min = -1, position = -1;
for (i = 0; i < n_prefork; ++i){
        if (worker_pool->worker_busy[i] == 0){
                if(min == -1){
                        min = worker_pool->worker_counter[i];
        position = i;
        }

                if (worker_pool->worker_counter[i] <= min){
                        min = worker_pool->worker_counter[i];
                        position = i;
                }
        }
}
```

Qui incontriamo le strutture dati che abbiamo descritto nel paragrafo del processo principale, ossia le strutture dati che si trovano in memoria condivisa. L'accesso a questa zone di memoria è eseguito come anche nel main tramite l'utilizzo di semafori POSIX, nello specifico i *named semaphore*. Prima di tutto viene individuato un worker libero, ricordiamo che l'array worker_busy è un array di flag, lo 0 indica un worker libero. Una volta identificato un possibile worker viene valutato il suo stato di lavoro eseguendo il controllo sull'array worker_counter, dal quale riusciamo a risalire al numero di connessioni che il worker ha gestito in totale. Tramite queste due ricerche identifichiamo il worker che ha lavorato meno e lo selezioniamo per la nuova connessione. Tutto questo viene fatto per distribuire il carico di lavoro in maniera equa su tutti i worker in modo tale da non avere dei processi annoiati e evitando così di sovraccaricarne altri.

```
        // Error no worker available
    if(position == -1) {

        get_log()->i(TAG_THREAD_POOL, "No Worker available, wait for space.");

        if(sem_post(sem) == -1){
            get_log()->e(TAG_THREAD_POOL, "Error in sem_wait - thread_pool_loop");
            exit(EXIT_FAILURE);
        }

        continue;

    }else{

        worker_pool->worker_busy[position] = 1;
        worker_pid = worker_pool->worker_array[position];
        worker_pool->worker_counter[position] = worker_pool->worker_counter[position]

        get_log()->i(TAG_THREAD_POOL, "Get Worker %ld", (long)worker_pid);

        // Retrieving server from scheduler
        ServerPtr server = get_scheduler()->get_server(get_scheduler()->rrobin);
        // Storing server in shared memory
        worker_pool->worker_server[position] = *server;

        if(sem_post(sem) == -1){
            get_log()->e(TAG_THREAD_POOL, "Error in sem_wait - thread_pool_loop");
            exit(EXIT_FAILURE);
        }

        break;
    }
```

All'interno della variabile *position* viene salvato l'indice del worker selezionato, se questo è -1 vuol dire che tutti i worker sono occupati perché stanno gestendo altre connessioni, in questo caso il ciclo di schedulazione ricomincia e continua finche non ci sarà un worker pronto a gestire la connessione. In caso di successo le strutture dati in memoria condivisa vengono aggiornate, il flag all'interno del'array worker_busy passa a 1 per indicare che il worker non è più disponibile, dal worker_array viene recuperato il pid del processo,

il contatore delle connessioni gestite dal worker viene aggiornato nell'array worker_counter e infine come ultimo passaggio richiamiamo il thread Scheduler per richiedere di fornirci un server dove il worker andrà a eseguire le richieste del client, sblocchiamo il semaforo, visto che abbiamo terminato l'utilizzo della memoria condvisa, e ci avviamo ad eseguire le ultime linee di codice della funzione:

```
            while (TRUE){

                    throwable = send_fd(fd, worker_pid);
                    if (throwable->is_an_error(throwable)) {
                            get_log()->e(TAG_THREAD_POOL, "Failed␣attempt␣%d␣to␣send␣file␣
                            throwable->destroy(throwable);
                            attempt++;
                    }else{
                            break;
                    }
            }

            // close fd from main side
            throwable = close_connection(fd);
    if (throwable->is_an_error(throwable)) {
        get_log()->t(throwable);
    }
```

Abbiamo quindi recuperato dall'array un file descriptor associato a una socket, abbiamo selezionato il worker per servire la connessione ci manca solamente di dire al worker quale sia questo file descriptor. Per fare questo viene utilizzato un altro componente di Heimdall, il *message controller*.

Il message controller non è altro che una serie di funzioni che utilizzano le socket locali AF_UNIX per lo scambio di file descriptor. Infatti la funzione *send_fd()* che è definita all'interno del file *message_controlller.c* e *message_controller.h* non fa altro che creare un socket AF_UNIX per l'invio del file descriptor al worker. Vengono utilizzate le socket perché non è possibile scambiare il file descriptor senza scambiare anche i permessi di lettura di su quel file, infatti al di fuori del contesto di un processo un file descriptor non è altro che un semplice intero, facciamo notare anche che

vengono eseguiti dei tentativi di invio perché nel momento dell'invio del file descriptor il worker, causa problemi di schedulazione da parte del sistema operativo, potrebbe non essere pronto a ricevere il dato. Una volta effettuato lo scambio il Pool Manger ha terminato la sua esecuzione e può tornare a schedulare altri worker, come ultima cosa non ci dimentichiamo di chiudere il file descriptor dal lato del processo padre infatti una volta inviato, un altro riferimento al file viene creato nella tabella dei file descriptor del worker e quindi quello presente nel processo principale non è più necessario.

3.7 Scheduler

Lo scheduler è un componente fondamentale di un sistema informatica: si occupa di stabilire un ordinamento temporale l'esecuzione di un set di richieste di accesso ad una risorsa. Nel caso di un web switch di livello 7, lo scheduler va a garantire che ognuna delle richieste in arrivo possa essere inoltrata immediatamente alla prima macchina disponibile, secondo una politica di scheduling che sia *state-less*, quindi che non consideri l'attuale carico di lavoro delle macchine del cluster, oppure *state-aware*, che monitori costantemente tale carico e modifichi di conseguenza l'assegnazione delle richieste (verrà spiegato nel dettaglio come lavorano e quando sono disponibili tali politiche in *4*).

In questa implementazione lo scheduler, che come vedremo va a sfruttare un algoritmo di selezione *Round Robin* la cui struttura verrà esplicitata più avanti, viene definito come segue.

```
/*
 * ------------------------------------------------------------------------
 * Structure        : typedef struct scheduler_args
 * Description      : This struct represents the arguments necessary to run the
 *                    scheduler properly
 * ------------------------------------------------------------------------
 */
typedef struct scheduler_args {
    RRobinPtr     rrobin;                                   // Round Robin struct
    ServerPoolPtr server_pool;                              // Server Pool struct

    ServerPtr (*get_server)(RRobinPtr rrobin);              // to retrieve a server
} Scheduler, *SchedulerPtr;
```

In particolare la **pool dei server** altro non è che un *lista collegata* formata da strutture dati elementari per la gestione dei server indicati nel file di configurazione come appartenenti al cluster, definite come segue

```
/* ------------------------------------------------------------------------
 * Structure        : typedef struct server_node
 * Description      : This struct represents a single server node in order to
 *                    manage a pool of remote machines
 * ------------------------------------------------------------------------
 */
typedef struct server_node {
    char *host_address;                                     // machine canonical name
    char *host_ip;                                          // machine ip address
    int  status;                                            // machine status
    int  weight;                                            // machine weight

    struct server_node *next;                               // next server_node
} ServerNode, *ServerNodePtr;
```

mentre le strutture dati che vengono elaborate ed utilizzate come valore di ritorno della schedulazione e che sono alla base della costituzione del buffer su cui opera Round Robin, non sono altro che una versione semplificata e costituita dalle sole informazioni di base per la connessione.

Nella **fase di inizializzazione** viene quindi popolata la pool recuperando gli indirizzi delle macchine del cluster, che vengono settate come disponibili e con peso minimo. Quindi a seconda che si sia configurato il web switch in modalità *state-aware* o *state-less*, rispettivamente viene o non viene istanziato un thread che si occuperà di aggiornare periodicamente, con gestione degli accessi concorrenti al buffer del Round Robin, lo stato delle macchine. Ogni volta che una connessione viene accettata viene recuperato un server valido da passare al processo che gestirà la connessione tramite memoria condivisa. Lo scheduler è recuperabile tramite un *singleton*.

```
\* inside thread_pool.c ... *\
// Retrieving server from scheduler
ServerPtr server = get_scheduler()->get_server(get_scheduler()->rrobin);
// Storing server in shared memory
worker_pool->worker_server[position] = *server;
```

Viene sempre selezionato un server che sia disponibile, quindi viene sempre effettuato un controllo sullo *status* dello stesso server, nel caso in cui sia abilitato il controllo sullo stato della macchina: l'unico caso in cui questi risulta *BROKEN* e non *READY* è nella circostanza in cui ogni server del cluster risulta non disponibile per cui il worker (che analizzeremo in *3.8*) non avvierà nessuna connessione di inoltro della richiesta.

Dalla necessità progettuale di garantire uno **scheduling adattabile** a condizioni di stress da carico, quindi per soddisfare specifiche di *state-awareness*, nascono i parametri relativi a status e peso nei nodi della pool di server e nasce un adattamento *pesato* dell'algoritmo di Round Robin.

3.8 Worker

Il **worker** è il componente principale di Heimdall: serve le richieste dei client, smistandole ai vari server presenti nel cluster e inoltrandone le risposte.
Nell'attuale implementazione, il worker è un processo composto da quattro thread: il *thread di lettura*, il *thread di scrittura*, il *thread di richiesta* e il *thread di watchdog*.
Heimdall è implementato in modo tale da effettuare il **prefork** di un numero configurabile di processi (si veda 3.3). Ciò è utile per evitare di rallentare l'esecuzione a causa del ritardo dovuto al tempo di creazione dei processi.

3.8.1 Gestione delle richieste

L'applicazione soddisfa le specifiche **HTTP 1.1**[4], gestendo **connessioni persistenti**[5] e supportando il **pipelining**[6] delle richieste. Per ottenere il supporto alle connessioni persistenti, Heimdall chiude la connessione con il client allo scadere di un timer, al fine di ricevere più richieste tramite la stessa connessione. Ricevute, queste vengono inserite in una coda, per essere poi servite nello stesso ordine con cui sono state ricevute.
All'avvio del WebSwitch, tramite prefork, vengono creati i vari worker, che rimangono in pausa finché non ricevono una connessione attraverso cui gestire le richieste. Tale passaggio è realizzato tramite il *pool manager*, in particolare

dal message controller. Ricevuta una connessione da poter servire, il worker crea i thread necessari per la ricezione, l'inoltro e la risposta alla richiesta.

Coda delle richieste

Per poter supportare il pipeling, è stato quindi necessario creare una coda, che contenesse, in ordine, tutte le richieste effettuate da un client tramite la stessa connessione HTTP, come specificato dalla RFC: "A server MUST send its responses to those requests in the same order that the requests were received."[6]

La coda è molto semplice ed espone le seguenti operazioni:

```
void (*enqueue)(struct request_queue *self, RequestNodePtr node);
struct request_node*(*dequeue)(struct request_queue *self);
int (*is_empty)(struct request_queue *self);
struct request_node*(*get_front)(struct request_queue *self);
int (*get_size)(struct request_queue *self);

void (*destroy)(struct request_queue *self);
```

La coda contiene elementi di tipo RequestNode, cioè la struttura dati che incapsula la richiesta, la risposta, un riferimento al nodo precedente, un riferimento al nodo successivo, una struttura (il chunk) e altre variabili di supporto per il multithreading e per il watchdog.

```
pthread_t thread_id;
HTTPRequestPtr request;
HTTPResponsePtr response;
time_t request_timeout;
struct request_node *previous;
struct request_node *next;
ChunkPtr chunk;
int *worker_status;
pthread_mutex_t mutex;
pthread_cond_t condition;
```

Chunk di dati

Un chunk è un'ulteriore struttura dati, introdotta al fine di rispondere in modo più efficiente e veloce possibile al client: non è necessario ricevere la

risposta completa, poiché questa potrebbe essere corposa e, quindi, occupare "molto" spazio e richiedere molto tempo per essere completamente ricevuta, aggiungendo ritardo per il seguente inoltro.
La struttura è molto semplice e contiene un'area di memoria fissa, di dimensione pari a 4096 byte, che viene allocata al momento della creazione della struttura stessa.
Questa scelta è finalizzata all'alleggerimento massimo del carico su Heimdall: facendo da "passa carta" tra server e client, la memoria avrà il minor quantitativo di risorsa possibile, in ogni istante.

3.8.2 Gestione delle connessioni

Connessione

Il sistema è stato concepito per essere il più modulare possibile con valori di ritorno il più possibile uniformi. La gestione delle connessioni ne è un esempio: è stato realizzato un wrapping delle API Socket di Berkley in modo tale da gestire tutti gli errori allo stesso modo, tramite l'utilizzo dei Throwable (si veda 3.5). L'implementazione delle chiamate è stata resa minimale, in modo tale da concentrarsi solo sulla logica dell'applicazione e non sulla sua effettiva implementazione.
Le funzioni per creare una nuova connessione di tipo server e di tipo client sono esplicative.
Tipo server:

```
/*
 * Function    : create_server_socket
 * Description : This function creates a TCP or a UDP server
 *               bound at specified port.
 *
 * Param       :
 *   type   : The type of the socket, 0 for TCP, 1 per UDP.
 *   port   : The number of the port where bind the server.
 *   sockfd : The pointer of the int where save the file
 *            description.
 *
 * Return      : A Throwable.
 */
```

```
ThrowablePtr create_server_socket(const int type, const int port, int *sockfd) {

    ThrowablePtr throwable = create_socket(type, sockfd);
    if (throwable->is_an_error(throwable)) {
        return throwable->thrown(throwable, "create_server_socket");
    }

    struct sockaddr_in addr;

    memset((void *) &addr, 0, sizeof(addr));    // Set all memory to 0
    addr.sin_family = AF_INET;                  // Set IPV4 family
    addr.sin_addr.s_addr = htonl(INADDR_ANY);   // Waiting a connection on all server's IP add
    addr.sin_port = htons(port);                // Waiting a connection on PORT

    if (bind(*sockfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
        return get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "create_server
    }

    return get_throwable()->create(STATUS_OK, NULL, "create_server_socket");
}
```

Tipo client:

```
/*
 * Function    : create_client_socket
 * Description : This function creates a TCP or a UDP client
 *               that connects itself at specific IP thorough
 *               a specific port.
 *
 * Param       :
 *   type   : The type of the socket, 0 for TCP, 1 per UDP.
 *   port   : The number of the port where bind the server.
 *   sockfd : The pointer of the int where save the file
 *            description.
 *
 * Return      : A Throwable.
 */
ThrowablePtr create_client_socket(const int type, const char *ip, const int port, int *sockfd)

    ThrowablePtr throwable = create_socket(type, sockfd);
    if (throwable->is_an_error(throwable)) {
        return throwable->thrown(throwable, "create_client_socket");
    }
```

```
    struct sockaddr_in addr;

    memset((void *) &addr, 0, sizeof(addr));            // Set all memory to 0
    addr.sin_family = AF_INET;                          // Set IPV4 family
    addr.sin_port = (in_port_t) htons((uint16_t) port); // Set server connection on specified

    if (inet_pton(AF_INET, ip, &addr.sin_addr) == -1) {
        return get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "create_client
    }

    if (connect(*sockfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
        return get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "create_client
    }

    return get_throwable()->create(STATUS_OK, NULL, "create_client_socket");
}
```

Questo espediente ha dato la possibilità, per esempio, di modificare più volte il codice che permette l'invio delle richieste e la successiva ricezione delle risposte, senza stravolgere, per quanto possibile, la logica del sistema.

Richieste HTTP

Alla base della capacità del web switch di inoltrare ad una macchina del cluster la richiesta che gli viene effettuata dal client, vi è la capacità di poter analizzare tale richiesta e poterne trarre le corrette informazioni.
Per questo sono state definite macro degli *header* più importanti come da (**) per poter *parsare* il pacchetto HTTP una volta che questo è correttamente giunto (completo) al web switch. E' stata quindi definita la struct seguente.

```
/*
 * ------------------------------------------------------------------------
 * Structure        : typedef struct http_request
 * Description      : This struct collect all attributes and functions pointers to
 *                    manage, retrieving or creating, HTTP requests
 * ------------------------------------------------------------------------
 */
typedef struct http_request {
    char *status;                                       // whether the request can be handled
    char *req_type;                                     // type of the request
```

```
    char *req_protocol;                                 // accepted only HTTP/1.1
    char *resp_code;
    char *resp_msg;
    char *req_resource;                                 // resource locator
    char *req_accept;                                   // accepting content info
    char *req_from;                                     // client and request generic info
    char *req_host;
    char *req_content_type;                             // content type info
    int   req_content_len;
    char *req_upgrade;                                  // no protocol upgrade are allowed

    char *header;

    ThrowablePtr (*get_header)(struct http_request *self, char *req_line);
    ThrowablePtr (*get_request)(struct http_request *self, char *req_line, int len);
    ThrowablePtr (*read_headers)(struct http_request *self, char *string, int type);
    ThrowablePtr (*make_simple_request)(struct http_request *self, char **result);
    ThrowablePtr (*set_simple_request)(struct http_request *self, char *request_type, char *re
    void (*destroy)(struct http_request *self);
} HTTPRequest, *HTTPRequestPtr;
```

Per cui una volta inizializzata può essere utilizzata per memorizzare, attraverso la lettura degli *header* e dei parametri ad essi associati le informazioni necessarie per poter inoltrare alla macchina del cluster selezionata. Sono state utilizzate le convenzioni previste del protocollo già viste nell'RFC precedentemente indicato: in particolare la corretta lettura di *<carriage return><newline>* singolo e doppio rispettivamente a fine riga e come separatore fra headers e contenuto del pacchetto, per poter determinare gli estremi di lettura del *parser*.
Osserviamo come non siano presenti tutti gli header definiti dal protocollo e come alcuni siano definiti per quanto riguarda la risposta HTTP (su questo aspetto torneremo fra poco).

Particolarmente importanti per questa implementazione sono i parametri che definiscono *tipo di richiesta*, *protocollo* e *risorsa richiesta*. All'interno della struct è altresì presente un set di puntatori a funzione che consente di impostare i parametri per costruire una semplice richiesta HTTP, funzionalità che viene usata per inoltrare la richiesta al cluster.

Risposte HTTP

La modularità della struct vista per quanto riguarda l'analisi della richiesta HTTP ha consentito di includere in essa anche i parametri, come il *messaggio ed il codice di risposta*, nonchè la *lunghezza della risorsa* contenuta nel pacchetto, necessari per la corretta lettura di un completo pacchetto HTTP di risposta. Per cui è bastato utilizzare le funzionalità già implementata ed "estendere" la struttura precedente come segue.

```
        /*
 * ------------------------------------------------------------------------
 * Structure        : typedef struct http_response
 * Description      : This struct collect all attributes and functions pointers to
 *                    manage, retrieving or creating, HTTP responses
 * ------------------------------------------------------------------------
 */
typedef struct http_response {
    struct http_request  *response;
    int    http_response_type;
    char  *http_response_body;                                  // the body of the message

    ThrowablePtr (*get_http_response)(struct http_response *self, char *buffer);
    ThrowablePtr (*get_response_head)(struct http_response *self, char *head);
    ThrowablePtr (*get_response_body)(struct http_response *self, char *body);
    void (*destroy)(struct http_response *self);
} HTTPResponse, *HTTPResponsePtr;
```

L'unica differenza con l'implementazione già descritta è il mantenimento in un'area di memoria dell'originale contenuto del pacchetto inviato dal server, necessario per poter correttamente inoltrare al client la risposta senza dover rilavorare anche con il contenuto della risposta HTTP. Questo ha portato ad una riduzione dell'overhead soprattutto in condizioni in cui tale contenuto è un file di testo consistente od un'immagine.

3.8.3 Thread di lettura

Appena il worker riceve una nuova connesione da gestire, il thread di lettura ha il compito di leggere le richieste dalla socket, tramite una read bloccante. Successivamente, il thread accoda le richieste del client, per un numero massimo di richieste configurabile (si veda 3.3), creando, per ogni richiesta,

un thread di richiesta che, dialogando con un server nella pool, si occuperà di gestirla. Il thread di lettura sarà bloccato costantemente in attesa di nuove richieste, aggiornando il timer adibito alla verifica dello stato di vita del worker (si veda il paragrafo 3.8.6), alla ricezione di ogni nuova richiesta.

```
// Gets queue
RequestQueuePtr queue = worker->requests_queue;

while (TRUE) {
    // Waits
    while (max_thr_request > 100) {
      if (pthread_cond_wait(&cond_thr_request, &mtx_thr_request) != 0) {
        return get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "receive_http_
      }
    }

    // Updates timer
    worker->watchdog->timestamp_worker = time(NULL);

    // Creates the node
    RequestNodePtr node = init_request_node();
    if (node == NULL) {
      get_log()->e(TAG_WORKER, "Malloc error in init_request_node");
      worker->reader_thread_status = STATUS_ERROR;
      return NULL;
    }
    node->worker_status = &worker->request_thread_status;

    // Enques the new node
    queue->enqueue(queue, node);

    // Receives request
    ThrowablePtr throwable = receive_http_request(worker->sockfd, node->request);
    if (throwable->is_an_error(throwable)) {

      get_log()->t(throwable);
      worker->reader_thread_status = STATUS_ERROR;

      // if we get some error on cliebt socket
      worker->worker_await_flag = WATCH_OVER;
      pthread_cond_signal(&worker->await_cond);
```

```
      pthread_exit(NULL);
    }

    request_counter++;

    // Creates the request thread
    int request_creation = pthread_create(&(node->thread_id), NULL, request_work, (void *) nod
    if (request_creation != 0) {
      get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "read_work
      worker->reader_thread_status = STATUS_ERROR;
      return NULL;
    }

    // Gets mutex
    if (pthread_mutex_lock(&mtx_thr_request) != 0) {
      return get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "receive_http_ch
    }

    max_thr_request++;

    // Sends signal to condition
    if (pthread_cond_signal(&cond_thr_request) != 0) {
      return get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "receive_http_ch
    }

    // Releases mutex
    if (pthread_mutex_unlock(&mtx_thr_request) != 0) {
      return get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "receive_http_ch
    }
}
```

Il thread eseguirà questo compito fino alla morte del worker.

3.8.4 Thread di richiesta

Il thread di richiesta è adibito all'invio ad un server della richiesta ricevuta dal *thread di lettura* e alla successiva ricezione della risposta. Il thread ottiene il server a cui deve inoltrare la richiesta, accedendo ad uno spazio di memoria condiviso con tutti i worker, tramite un semaforo:

```
// Asks which host use
```

```
ServerPtr remote = NULL;

// retrieving remote host from the shared memory

if(sem_wait(sem) == -1){
  get_log()->e(TAG_WORKER, "Error in sem_wait - start_worker");
  exit(EXIT_FAILURE);
}

int i;
for (i = 0; i < number_of_worker; ++i){
  if (worker_pool->worker_array[i] == (long)getpid()){
    remote = &(worker_pool->worker_server[i]);
    break;
  }
}

if(sem_post(sem) == -1){
  get_log()->e(TAG_WORKER, "Error in sem_wait - start_worker");
  exit(EXIT_FAILURE);
}

// checking for remote host status
if (remote->status == SERVER_STATUS_BROKEN) {
  *node->worker_status = STATUS_ERROR;

  // if we get the server status as broken
  if (pthread_mutex_unlock(&node->mutex) != 0) {
    get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_wor
    *node->worker_status = STATUS_ERROR;
    exit(EXIT_FAILURE);
  }

  return NULL;

}
```

Successivamente, apre una nuova connessione verso il server, riceve la risposta nel chunck e si pone in attesa della liberazione di questo da parte del *thread di scrittura*, per poi rieseguire questa serie di operazioni, fino alla completa ricezione della risposta:

```
// Creates a new client
int sockfd;
throwable = create_client_socket(TCP, host, 80, &sockfd);
if (throwable->is_an_error(throwable)) {
  get_log()->t(throwable);
  *node->worker_status = STATUS_ERROR;

  // Releases mutex
  if (pthread_mutex_unlock(&node->mutex) != 0) {
    get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_wor
    *node->worker_status = STATUS_ERROR;
    exit(EXIT_FAILURE);
  }

  return NULL;
}

// Logging - request
HTTPRequestPtr request = node->request;
get_log()->r(RQST, (void *)request, host, sockfd);

get_log()->d(TAG_WORKER, "io␣Dott.␣%ld␣del␣%ld␣Faccio␣richiesta␣a␣%s␣su␣socket:␣%d", (long) pt

if (asprintf(&(node->request->req_host), "%s:80", host) < 0) {
  get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_work"
  *node->worker_status = STATUS_ERROR;

  // Releases mutex
  if (pthread_mutex_unlock(&node->mutex) != 0) {
      get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_w
      *node->worker_status = STATUS_ERROR;
  }

  return NULL;
}

// Sends request
throwable = send_http_request(sockfd, node->request);
if (throwable->is_an_error(throwable)) {
  get_log()->t(throwable);
  *node->worker_status = STATUS_ERROR;
```

```
  // Releases mutex
  if (pthread_mutex_unlock(&node->mutex) != 0) {
      get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_w
      *node->worker_status = STATUS_ERROR;
  }

  return NULL;
}

// Receives header into http_response
throwable = receive_http_response_header(sockfd, node->response);
if (throwable->is_an_error(throwable)) {
  get_log()->t(throwable);
  *node->worker_status = STATUS_ERROR;

  // Releases mutex
  if (pthread_mutex_unlock(&node->mutex) != 0) {
      get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_w
      *node->worker_status = STATUS_ERROR;
  }

  return NULL;
}

// Logging - response
HTTPResponsePtr response = node->response;
get_log()->r(RESP, (void *)response, host, sockfd);


// Sends signal to condition
if (pthread_cond_signal(&node->condition) != 0) {
  get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_work"
  *node->worker_status = STATUS_ERROR;

  // Releases mutex
  if (pthread_mutex_unlock(&node->mutex) != 0) {
      get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_w
      *node->worker_status = STATUS_ERROR;
  }
```

```
  return NULL;
}

// Releases mutex
if (pthread_mutex_unlock(&node->mutex) != 0) {
  get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "request_work"
  *node->worker_status = STATUS_ERROR;
  return NULL;
}

// Receives the response in chunks
throwable = receive_http_chunks(sockfd, node->response, node->chunk);
if (throwable->is_an_error(throwable)) {
  get_log()->t(throwable);
  *node->worker_status = STATUS_ERROR;
  return NULL;
}
```

Concluso il suo lavoro, il thread morirà, chiudendo la connessione verso il server.

3.8.5 Thread di scrittura

Infine, il thread di scrittura è adibito all'inoltro della risposta ottenuta tramite il *thread di richiesta*. Dovendo rispondere in ordine, il thread si trova in loop sulla coda delle richieste, chiedendo costasntemente il suo primo elemento. Dopo aver inviato l'header di risposta, i due thread iniziano a cooperare per mezzo di una condition, andando a scrivere e a leggere nella stessa area di memoria, il *chunk*.

```
// Gets queue
RequestQueuePtr queue = worker->requests_queue;

while(TRUE) {

    // Gets node
    RequestNodePtr node = queue->get_front(queue);

    if (node != NULL) {

        // Gets mutex
```

```
if (pthread_mutex_lock(&node->mutex) != 0) {
    get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "wri
    worker->writer_thread_status = STATUS_ERROR;
    return NULL;
}

while (node->response->response->header == NULL) {
    if (pthread_cond_wait(&node->condition, &node->mutex) != 0) {
        get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno),
        worker->writer_thread_status = STATUS_ERROR;
        return NULL;
    }
}

// Sends the response header
ThrowablePtr throwable = send_http_response_header(worker->sockfd, node->response);
if (throwable->is_an_error(throwable)) {
    get_log()->t(throwable);
    worker->writer_thread_status = STATUS_ERROR;

    // unlock if error
    if (pthread_mutex_unlock(&node->mutex) != 0) {
        get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno),
    }

    return NULL;
}

// Releases mutex
if (pthread_mutex_unlock(&node->mutex) != 0) {
    get_log()->t(get_throwable()->create(STATUS_ERROR, get_error_by_errno(errno), "wri
    worker->writer_thread_status = STATUS_ERROR;
    return NULL;
}

// Gets the chunk
ChunkPtr chunk = node->chunk;

// Sends the response chunks
throwable = send_http_chunks(worker->sockfd, chunk, node->response->response->req_cont
if (throwable->is_an_error(throwable)) {
```

```
            get_log()->t(throwable);
            worker->writer_thread_status = STATUS_ERROR;
            return NULL;
        }

        // Dequeues the request and it destroys that
        node = queue->dequeue(queue);
        node->destroy(node);
    }
}
```

3.8.6 Thread di watchdog

Il thread di *watchdog*, letteralmente di sorveglianza, è adibito al controllo della **corretta esecuzione del worker**,in particolare al controllo che tale esecuzione, nell'occupare per eccessivo tempo le risorse del sistema, non vado a creare un collo di bottiglia che via via porta al collasso del programma. Per far questo esso viene eseguito in modalità *detached* dal thread principale del worker ed è legato dalle operazioni che vengono eseguite dal processo da una serie di variabili:

- *pthread_condition* su cui è in attesa il thread principale del worker
- *flag* legata alla condition di cui sopra
- *timestamp* settato dal thread di lettura della richiesta in arrivo

Come appare da una prima analisi della struct del watchdog.

```
/*
 * ------------------------------------------------------------------------
 * Structure        : typedef struct watchdog_thread
 * Description      : this struct helps to manage and set attributes for the thread
 *                    which watch over the remote connection thread termination
 * ------------------------------------------------------------------------
 */
typedef struct watchdog_thread {
    int status;
    pthread_cond_t *worker_await_cond;
    int *worker_await_flag;
```

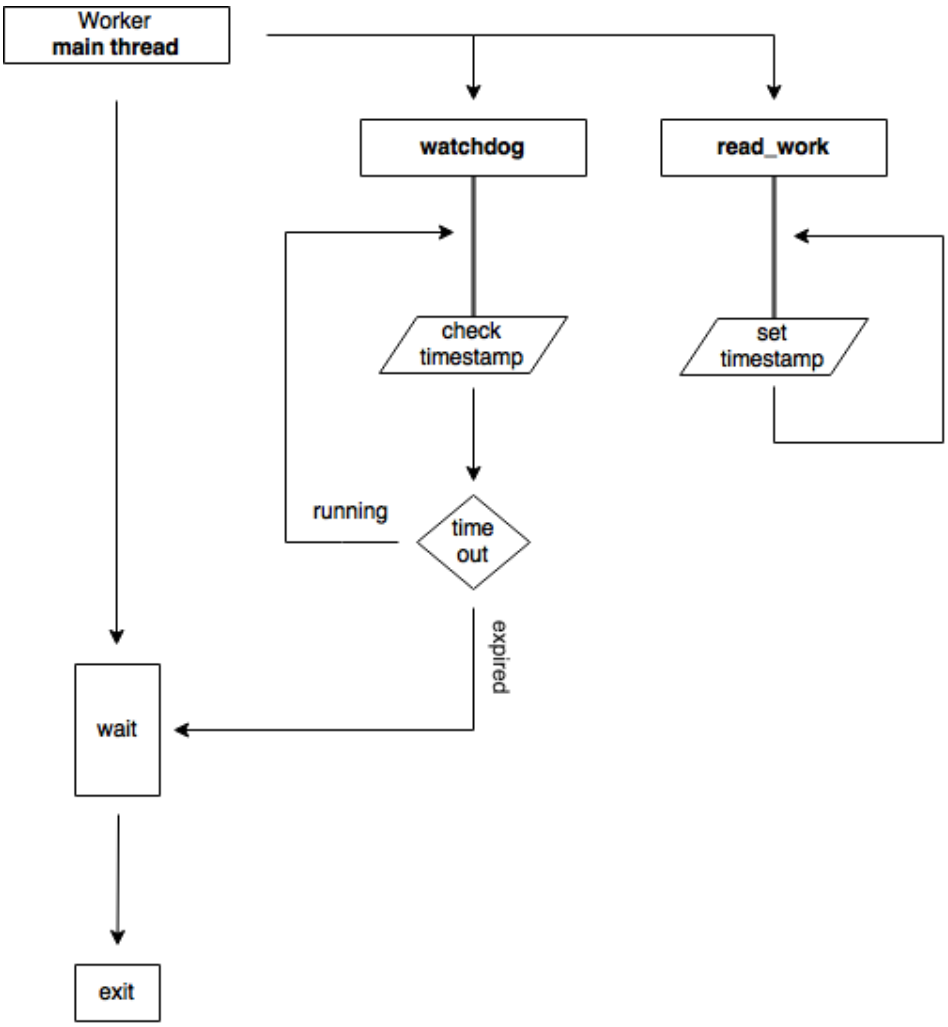

Figura 3: Schema della procedura di controllo sul tempo di esecuzione

```
    time_t killer_time;                    // to schedule the watchdog wakeup
    time_t timeout_worker;                 // to abort a thread run
    time_t timestamp_worker;               // timestamp last worker operation
} Watchdog, *WatchdogPtr;
```

Dove il *killer_time* ed il *timeout_worker* sono definiti dal file di configurazione per dare modo all'operatore di modellare l'implementazione sulle caratteristiche della macchina su cui gira il web switch.

Una più schematica rappresentazione del flusso di esecuzione è possibile

vederla in *3*. In particolare quando vengono distaccati i thread ausiliari, il thread principale del worker si mette in attesa della fine di una delle condizioni di termine del servizio di cui si è già discusso sopra, fra cui anche lo scadere del massimo tempo di esecuzione disponibile. In particolare, ad ogni iterazione del thread di lettura, avremo un aggiornamento del timestamp del worker, per cui ogni volta che si ha l'arrivo di una risposta da reinoltrare o di una richiesta da soddisfare si assume il server come operativo od il client in ascolto.
Contemporaneamente il watchdog rimane in *nanosleep* per un lasso di tempo pari a quello configurato, a meno dell'arrivo di segnali che vengono gestiti e dopo i quali il watchdog ritorna in attesa, scaduto il quale la variabile di timestamp viene controllata. Se risulta *scaduta* viene aggiornato il flag di attesa del worker e gli viene segnalato di riattivarsi e di mettere in pratica le procedure di *clean up* per scollegarsi dal client e dal server assegnato.

Osserviamo come questo controllo viene fatto su una variabile settata dal thread di lettura, permettendoci con semplicità di valutare eventuali problemi sia sulla linea fra web switch e macchina del cluster che fra web switch e client, evitando lo stato di *hanging* che porterebbe ad uno stallo del programma.

4 Politiche di scheduling

La schedulazione permette la selezione della macchina predisposta a rispondere alla richiesta HTTP appena arrivata da parte del client, si basa su una tecnica nota come **bilanciamento del carico**, ovvero la distribuzione del carico, solitamente di elaborazione o di erogazione di uno specifico servizio, tra più server. Questo permette di poter **scalare** sulla potenza di calcolo del cluster dietro al web switch, lasciando che siano diverse macchine a rispondere a seconda di quella che è più veloce, più performante, oppure monitorando costantemente lo stato dei server e scegliendo quello meno sottoposto ad una pressione del carico di lavoro. Le macchine, specificando hostname ed indirizzi IP, sono date in un apposito file di configurazione.

Nella nostra implementazione **thread scheduler** si occupa di fornire, ogni volta che viene invocato, una macchina selezionata secondo una delle due politiche che andremo ora a spiegare nel dettaglio.

4.1 State-less: implementazione con Round-Robin

L'algoritmo di scheduling Round-Robin (da adesso RR, *n.d.r.*) è un algoritmo che agisce con prelazione distribuendo in maniera equa il lavoro, secondo una metrica stabilita in partenza. Vediamo quindi la struttura che si occupa di gestire la schedulazione tramite Round-Robin e che contiene le funzioni *wrapper* alle strutture dati che garantiscono il suo corretto funzionamento.

```
/*
 * ------------------------------------------------------------------------
 * Structure         : typedef struct round_robin_struct
 * Description       : This struct represents a Round Robin discipline that can
 *                       be used also a stateful discipline with minimum overhead
 *                     (weighted mode enabled)
 * ------------------------------------------------------------------------
 */
typedef struct round_robin_struct {
    CircularPtr circular;

    ThrowablePtr (*weight)(CircularPtr circular, Server *servers, int server_num);
```

```
    ThrowablePtr (*reset)(RRobinPtr rrobin, ServerPoolPtr pool, int server_num);
    Server *(*get_server)(CircularPtr circular);
}RRobin, *RRobinPtr;
```

Possiamo osservare come siano mantenuti i puntatori alle funzioni necessarie al caso di politica di scheduling *state-aware*, ma per ora l'unica vera funziona a cui si farà accesso è quella per il recupero del server correntemente selezionato.

L'algoritmo funziona utilizzando un **buffer circolare** come possiamo vedere in *figura 4*: questo permette di iterare la selezione su una lista di elementi precedentemente caricata. Possiamo osservare che, oltre alle funzioni e le variabili necessarie a garantire l'accesso atomico all'area di memoria che contiene il buffer, necessario come vedremo nel caso *state-aware* per evitare la concorrenza con il thread che si occupa dell'update dello stato, sono mantenuti:

- Un puntatore all'array di server
- La posizione attuale del puntatore di *testa*
- La lunghezza del buffer, necessaria anche per le operazioni di aggiornamento dei puntatori
- I puntatori di *testa* e *coda* per avanzamento e lettura dal buffer

Le funzioni restanti permettono di inizializzare il buffer (oltre che di liberare con sicurezza l'area di memoria occupata) e di aggiornare i puntatori sopra menzionati.

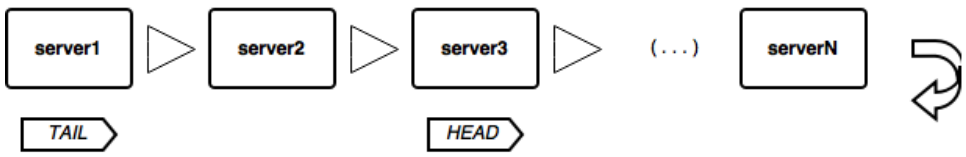

Figura 4: Schema di funzionamento del buffer circolare

4.2 State-less: implementazione con Round-Robin

L'algoritmo di scheduling Round-Robin (da adesso RR, *n.d.r.*) è un algoritmo che agisce con prelazione distribuendo in maniera equa il lavoro, secondo una metrica stabilita in partenza.
L'algoritmo funziona utilizzando un buffer circolare come possiamo vedere in *figura*: questo permette di iterare la selezione su una lista di elementi precedentemente caricata. E' necessario quindi specificare due passi per il corretto funzionamento, dopo aver dato un rapido sguardo alla struttura che lo rappresenta nella nostra implementazione.

```
/*
 * ------------------------------------------------------------------------
 * Structure        : typedef struct circular_buffer
 * Description      : This struct helps to manage a circular buffer of fixed length
 * ------------------------------------------------------------------------
 */
typedef struct circular_buffer {
    Server      *buffer;
    int         buffer_position;
    int         buffer_len;

    Server      *head;
    Server      *tail;

    pthread_mutex_t mutex;

    ThrowablePtr (*allocate_buffer)(CircularPtr *circular, Server **servers, int len);
    ThrowablePtr (*acquire)(struct circular_buffer *circular);
    ThrowablePtr (*release)(struct circular_buffer *circular);
    void         (*progress)(struct circular_buffer *circular);
    void         (*destroy_buffer)(struct circular_buffer *circular);
} Circular, *CircularPtr;
```

È necessario quindi specificare tre passi per il corretto funzionamento, dopo aver dato un rapido sguardo alla struttura che lo rappresenta nella nostra implementazione.

Inizializzazione del buffer in questa fase la struttura dati che rappresenta il buffer circolare, che abbiamo visto mantenere due puntatori di *testa* e *coda*, viene inizializzata associandovi un array di puntatori di strutture di tipo *Server*, precedentemente allocata ed il cui pattern è stato fissato, e viene eseguita la funzione di allocazione del buffer:

```
/* inside allocate_buffer ... */
// allocating the buffer
circular->buffer = *servers;
circular->buffer_len = len;
// setting params
circular->head = circular->buffer;
circular->tail = circular->buffer + (len - 1);
```

In un'ottica di *produttore vs consumatore*, chiaramente visibile nella figura precedente, è necessario che testa e coda non coincidano mai per evitare concorrenza. In questa implementazione si è deciso di separare l'accesso concorrente alla struttura, per il suo aggiornamento, e la lettura dei dati in essa contenuti. Quindi la *testa* conterrà il puntatore al prossimo server da selezionare per schedulare la richiesta, mentre la *coda* punterà all'area di memoria contenente il server attualmente selezione per la schedulazione.

Aggiornamento dei puntatori per poter sfruttare le peculiarità di questa struttura dati è necessario che i due puntatori vengano aggiornati secondo l'aritmetica del buffer circolare per cui, una volta raggiunta l'estremità dell'array, il valore successivo della posizione corrente ritorna ad essere quello del primo valore dello stesso array.
Nel dettaglio viene eseguito, secondo le specifiche sopra riportate, nella nostra implementazione, la seguente funzione:

```
void progress(CircularPtr circular) {
    // recomputing tail, head and buffer position
    circular->tail            = circular->head;
    circular->buffer_position = (circular->buffer_position + 1) % circular->buffer_len;
```

```
    circular->head            = circular->buffer + circular->buffer_position;
}
```

Selezione del server a questo punto, una volta che il thread chiamante invoca lo scheduler per recuperare il server che è stato selezionato dall'algoritmo, lo scheduler a sua volta invoca la funzione wrapper dalla struttura che gestisce la politica RR e questa esegue il codice ora riportato.

```
    /* inside get_server ... */
    // allocating server ready struct
    ServerPtr server_ready = malloc(sizeof(Server));

    /* ... */

    // stepping the circular buffer
    circular->progress(circular),
    // retrieving server from tail
    *server_ready = *(circular->tail);
    return server_ready;
```

In conclusione quello che stiamo attuando è un **bilanciamento del carico uniforme** su ognuna delle macchine del cluster. Infatti, senza condizioni sullo stato delle macchine, iterando semplicemente sull'array dei server, ad ogni nuova connessione verrà assegnata una macchina diversa, alleggerendo tutti i server e pareggiando per ciascuno il carico. Il cluster manterrà il carico complessivo ma ogni singola unità contribuirà equamente a soddisfare le connessioni in arrivo.

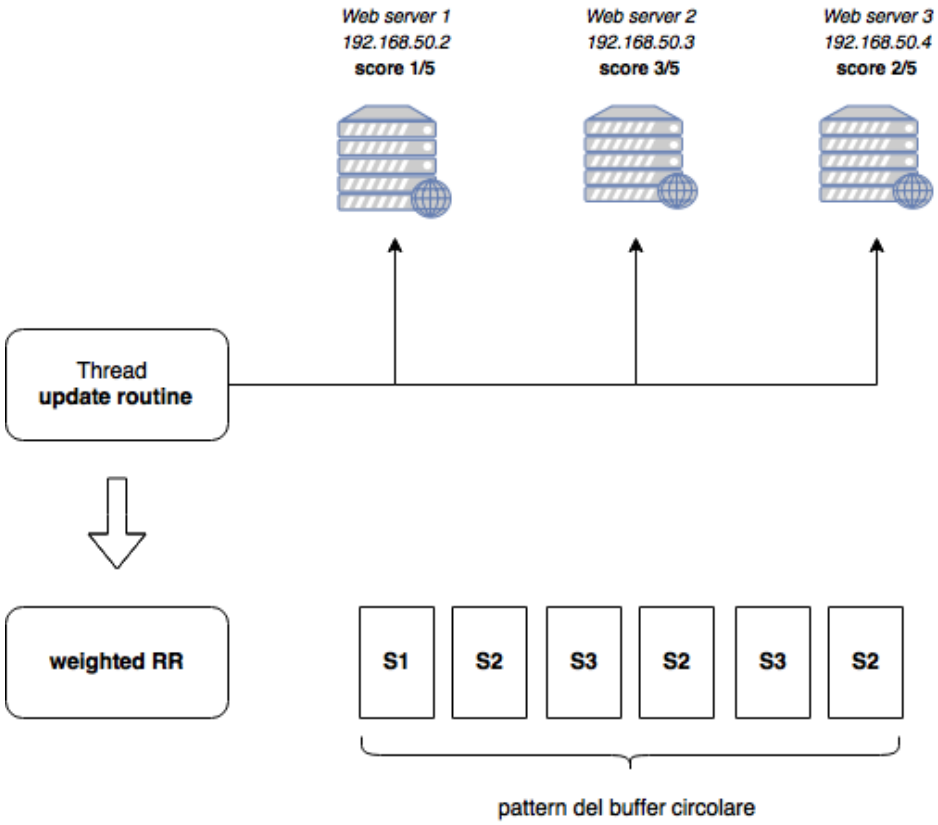

Figura 5: Schema della procedura di aggiornamento dello stato dei server

4.3 State-aware: implementazione con monitor di carico

Un algoritmo di schedulazione cosiddetto *state-aware* si occupa di selezionare la macchina a cui inoltrare la connessione basandosi non solo sulla conoscenza delle macchine presenti nel cluster ma anche sul loro status. In particolare, in questa implementazione, si è deciso di ricorrere all'analisi dei risultati di un **monitor di carico** presente su ciascuna delle macchine del cluster (in riferimento alle assunzioni progettuali, questi è i modulo *ApacheStatus* di cui si parlerà più avanti in *4.3.1*). Tale monitor, che ritorna una serie di parametri indici dell'attuale impiego di risorse della macchina, permette di definire un **algoritmo pesato** per la selezione del server che risponderà alla connessione in arrivo al web switch.
Anche in questo caso andremo a determinare una serie di passi che vengono seguiti, tenendo conto che in fase progettuale *si è deciso di sfruttare lo stesso*

algoritmo RR già utilizzato nel caso *state-less*, ma che ricordiamo essere stato predisposto per una ulteriore versione pesata. Per far questo si lavora sulla struttura Server

Detachment del thread di update nei file di configurazione dell'applicazione è possibile definire due livelli di lavoro:

- **AWARENESS_LEVEL_LOW** che corrisponde ad una versione state-less dell'algoritmo RR e si riporta al caso precedente
- **AWARENESS_LEVEL_HIGH** che corrisponde all'algoritmo state-aware e che necessiterà di una routine di aggiornamento dello stato delle macchine del cluster

Il secondo caso è proprio quello qui descritto e corrisponde a lavorare utilizzando, oltre al thread principale che si occupa di accettare le connessioni in arrivo, un **thread predisposto alla sola verifica dello stato dei server**. Tale thread viene istanziato nel momento in cui viene inizializzato lo scheduler e vengono allocate le strutture dati alla base di RR.
Il lavoro di tale thread, che ora vedremo nel dettaglio, è quello deducibile da *figura 5*.

Routine di score all'interno di questa routine, che viene eseguita da un thread distaccato e che viene eseguita una volta ogni *UP_TIME* secondi, tempo di update in secondi definito dall'utente nei file di configurazione, viene richiamata più volte la funzione che si occupa di recuperare e parsare l'interrogazione del modulo *ApacheStatus* e recupare da questa i **worker in *idle state* ed i worker in *busy state***. A questo punto si va a modificare il nodo della pool dei server precedentemente allocata (di cui si è già parlato in *3.7*). Viene quindi eseguita la sequente routine.

```
/* inside apache_score ... */
// retrieving status from remote Apache machine
throwable = apache_status->retrieve(apache_status);
//checking for errors or if server is currently down
if (throwable->is_an_error(throwable)) {
```

```
        server->weight = WEIGHT_DEFAULT;
        server->status = SERVER_STATUS_BROKEN;
        return throwable->thrown(throwable, "apache_score");
    } else {
        server->status = SERVER_STATUS_READY;
    }

    /* ... */
    int score;
    int IDLE_WORKERS  = apache_status->idle_workers;
    int TOTAL_WORKERS = apache_status->busy_workers + IDLE_WORKERS;

    // calculating and setting score - mapping in [w, W]
    score = (IDLE_WORKERS   - WEIGHT_DEFAULT)   *
            (WEIGHT_MAXIMUM - WEIGHT_DEFAULT)   /
            (TOTAL_WORKERS  - WEIGHT_DEFAULT)   +   WEIGHT_DEFAULT;
    server->weight = score;
```

Alla fine quello che ottengo è uno **score** che vado a settare nel nodo contenuto nella **pool dei server** che viene definito dalla relazione matematica che è così esplicitata:

$$score\Big(\frac{IDLE\_WORKERS}{TOTAL\_WORKERS}\Big) \ \in \ [w, W]$$

ottenendo quello che un *mapping* del rapporto fra i worker occupati nella macchina ed i worker totali a disposizione di Apache per rispondere ad una richiesta in arriva. Tale indice viene memorizzato come *peso del server nel cluster*.
Notiamo che nel caso ci siano problemi nel recuperare l'indice di score si supporrà che il server non è momentaneamente disponibile ed il suo status verrà segnalato come BROKEN, fino al prossimo aggiornamento.

RR pesato dai nodi della pool dei server aggiornati con il loro peso viene costruito, secondo lo schema in *5*, un pattern dei server secondo il loro peso, di modo da distribuire il carico secondo sempre un algoritmo RR, ma in cui per ogni sequenza il server viene selezionato un numero di volte pari al suo peso: comparirà massimo W in caso di basso carico di lavoro ed al minimo w volte in condizioni di forte stress. I due parametri sono, in questa implementazione, macro che possono essere modificate a seconda dei limiti

delle macchine del proprio cluster, di default $w = 1$ e $W = 5$, soggetti al tuning del web switch in fase di installazione ed ottimizzazione. Alla prima iterazione tutte le macchine so di default settata con peso minimo (pari a w)

In conclusione, con questa opzione abilitata, si ha la possibilità di ridistribuire equamente il lavoro, permettendo al web switch di adattare la distribuzione del carico a secondo dello stato attuale, evitando di sovraccaricare nodi sensibili allo stress in determinate condizioni o che sono stati sottoposti già ad uno stress eccessivo. Si è scelto di riadattare RR per ottenere una soluzione modulare e che fosse facile riadattare ed ottimizzare a seconda di entrambe le condizioni operative, sia senza che con conoscenza dello stato delle macchine. Osserviamo infatti che in entrambi i casi RR risulta pesato, nel secondo caso preso in esame tale peso non è più fisso e minimo ma variabile dipendentemente dalle condizioni delle macchine.

La ricerca di una soluzione modulare che possa essere presa poi in esame da futuri sviluppatore e possa essere oggetto di un *tuning* più approfondito, è stata intrapresa perseguendo il principio per cui *simplicity favours regularity*.

4.3.1 Modulo Apache Status

Il modulo Apache Status (mod_status)[8] è un modulo che fornisce informazioni sull'attività e le prestazioni del server in cui è installato. Questo modulo è disponibile nella versione base di Apache senza il bisogno di dover scaricare nessun altro componente aggiuntivo. Per utilizzarlo è necessario solo attivarlo nella configurazione del host. Il modulo formatta in una pagina HTML tutta una serie di statistiche e dati facilmente leggibili da un essere umano (oppure nella sua variante machine readble, versione che usiamo in questa applicazione).
I dettagli che fornisce sono:

- Il numero di worker che stanno servendo delle richieste
- Il numero totale di worker in pausa

- Lo stato di ciascun worker, il numero di richieste che tale worker ha eseguito e il numero totale di byte serviti dal worker
- Il tempo da quando il server è stato avviato/riavviato
- Il numero medio di richieste per secondo, il numero di bytes serviti per secondo e il numero medio di bytes per richiesta

Tramite questo modulo quindi siamo stati in grado di poter verificare lo stato di una macchina (di un server Apache) senza la necessità di installare nessun componente aggiuntivo o di valutare empiricamente il carico del server tramite per esempio il tempo di risposta di quest'ultimo.

4.3.2 Performance della politica state aware

E' interessante osservare l'effettivo lavoro dello scheduler e della politica di *state awareness* già commentata, in una condizione di forte stress delle macchine, nell'estratto del file di log che segue. La situazione proposta è stata ottenuta sovraccaricando una delle macchina del cluster e lasciando che il web switch si accorgesse di tale sovraccarico per bilanciare le richieste in arrivo.

```
[Wed Feb 10 11:31:44 2016] - 192.168.1.3 to: 192.168.1.4 - worker: 3032 - GET / HTTP/1.1
[Wed Feb 10 11:31:44 2016] - 192.168.1.3 to: 192.168.1.5 - worker: 3031 - GET / HTTP/1.1
[Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.6 - worker: 3040 - GET / HTTP/1.1
[Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.5 - worker: 3039 - GET / HTTP/1.1
[Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.6 - worker: 3038 - GET / HTTP/1.1
[Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.5 - worker: 3037 - GET / HTTP/1.1
[Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.6 - worker: 3036 - GET / HTTP/1.1
[Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.5 - worker: 3035 - GET / HTTP/1.1
[Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.6 - worker: 3034 - GET / HTTP/1.1
```

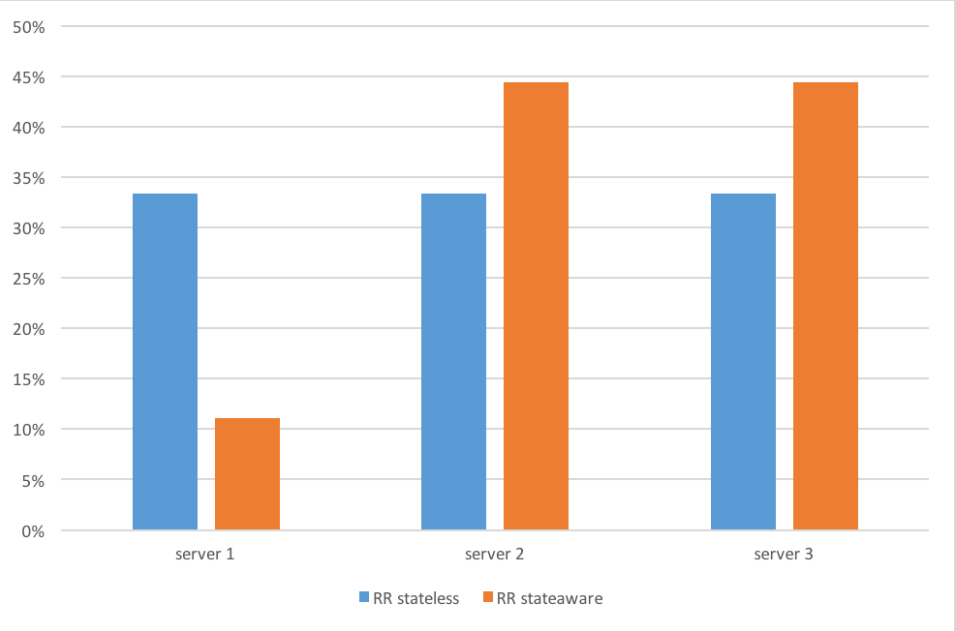

Figura 6: Ripartizione del carico di lavoro, *confronto fra politica state-less e state-aware*

In queste circostanze, considerato che secondo la nostra implementazione un server deve apparire almeno una volta nel buffer circolare, considerato che uno dei server era stato sottoposto a forte carico e che gli altri due server avevano un peso relativo superiore di tre unità, otteniamo che il server a venire meno sottoposto a connessioni in arrivo è proprio il server il cui carico di lavoro è già calcolato come particolarmente elevato.
E' possibile accertarlo sia dall'estratto del file di log riportato che alla *6* che ne quantifica i risultati.

5 Performance

Abbiamo visto più volte durante questa trattazione che è stato necessario sottoporre a *tuning* l'effettiva implementazione del server o che si consiglia di modellare quanti più aspetti possibili del web switch secondo la rete e le capacità della macchina. Riportiamo qui alcuni test che sono stati utilizzati in fase di progettazione per modellare la risposta del programma. I test sono stati condotti su:

- Macchina virtuale con Ubuntu Server 15.10 - 512 MB di RAM
- Macchina con Ubuntu Desktop 15.10 - 6 GB di RAM, processore Intel i3 (1,9 GHz)

Nel primo caso durante tutta la fase di sviluppo, nel secondo caso durante i test di carico per dare possibilità al web switch di non dover dipendere dall'hardware virtualizzato. Consideremo in tutti i casi la rete non saturata, in particolare non consideriamo la possibilità di influenze da parte di traffico generato da altre fonti essendo i test, per necessità, condotti o con una rete locale fra macchine collegate da uno switch di rete, oppure in locale su diverse macchine virtuali.

Numero di processi e limite dei file descriptor durante i test che seguono il numero di worker, variabile nelle configurazioni, è fissato a 10 come di default. Questo è stato giustificato da un'osservazione in fase sperimentale: non esiste una vera e propria variazione apprezzabile delle performance all'aumentare del numero di worker sopra i 10-15. Il valore di 10 unità è stato proposto per evitare sia un eccessivo utilizzo della memoria, sia per limitare il lavoro complessivo durante la creazione dei thread, sia per evitare ulteriore *overhead* dovuto al cambio di contesto.
Inoltre il numero di file descriptor utilizzabili per l'apertura delle socket è fissato, come da default nei file di configurazione, a 4096.

Numero di connessioni e di richieste per poter simulare l'approccio più "aggressivo" di un browser moderno, si è pensato di testare il programma

permettendo a *httperf* di inviare 10 richieste per ogni connessione instaurata. Quindi ogni worker avrà una coda di 10 richieste da smaltire.

5.1 Test di carico

Il **confronto dei tempi di risposta** è stato condotto unendo ad uno switch di rete tre macchine, una su cui girava il web switch sull'ambiente desktop di cui sopra, una da cui veniva condotti i *benchmark* sia via browser che sia attraverso il tool da linea di comando *httperf* di cui parleremo nell'appendice E.3.

Mantenendo *frequenza di richieste per secondo* costanti, osserviamo due situazioni diverse, riscontrate con l'aumentare di tale frequenza. In *figura 7* possiamo vedere come ci sia un valore intorno al quale è possibile registrare oscillazioni, di decimi di millesimi di secondo, dovute probabilmente allo stato della macchina ospitante il web switch. Tuttavia è significativo come all'aumentare del rate (per esempio a *50 req/s*) cominciamo ad osservare un nuovo livellamento di dei tempi di risposta, maggiormente significativo. Questo a causa del carico di lavoro che comincia a giungere alla macchina e che porta ad una situazione stabile di occupazione dei worker.
Questa osservazione è suffragata dall'analisi del grafico in *figura 8.* dove possiamo vedere come, con una frequenza di 100 req/s, si ha un aumento ripido dei tempi di risposta, dovuto al fatto che vengono fatte sempre più richieste, con un tempo almeno doppio di quanto occorre al web switch per liberare i propri worker per poter rispondere alle nuove connessioni. Andremo ora a vedere più nel dettaglio, nello *stress test*, questa peculiarità del nostro programma.

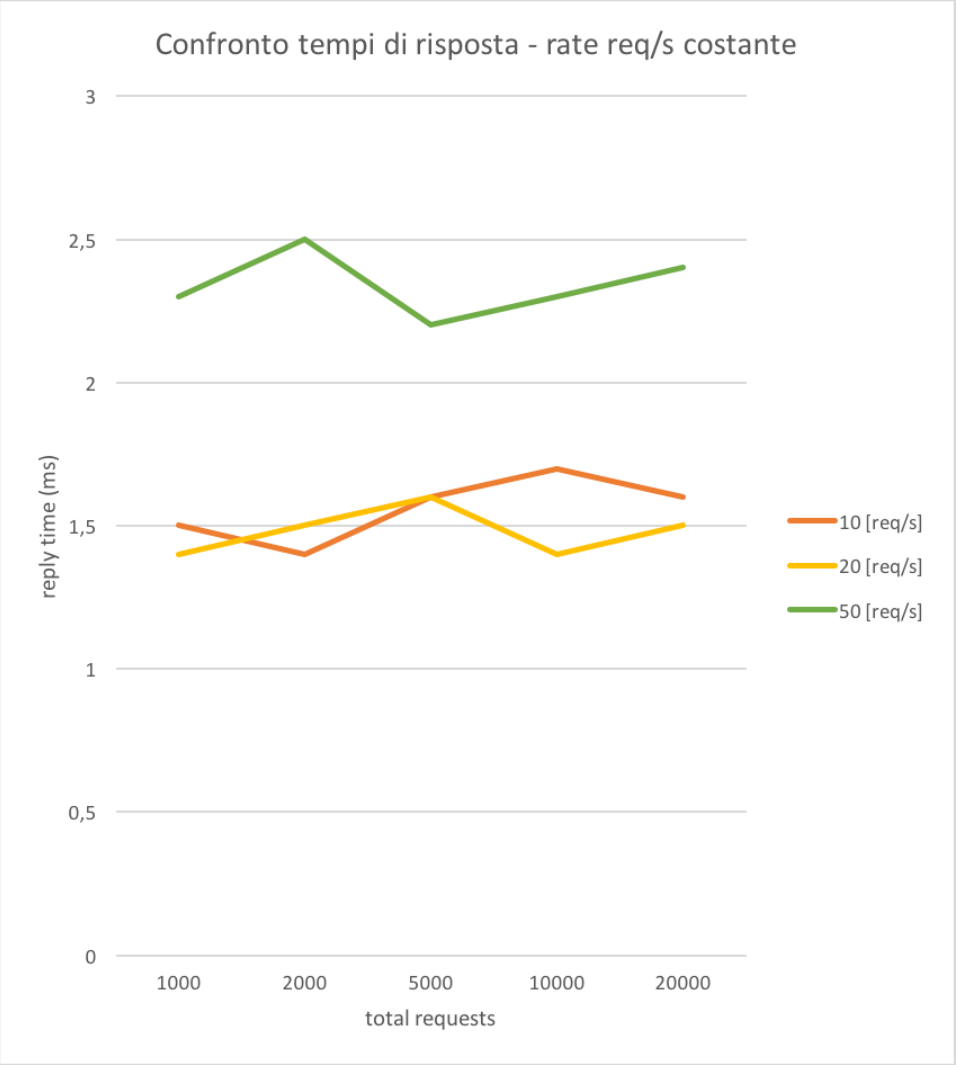

Figura 7: Confronto fra i tempi di risposta con diversi rate all'aumentare delle richieste. Si osserva lo stabilizzarsi intorno ad un livello medio.

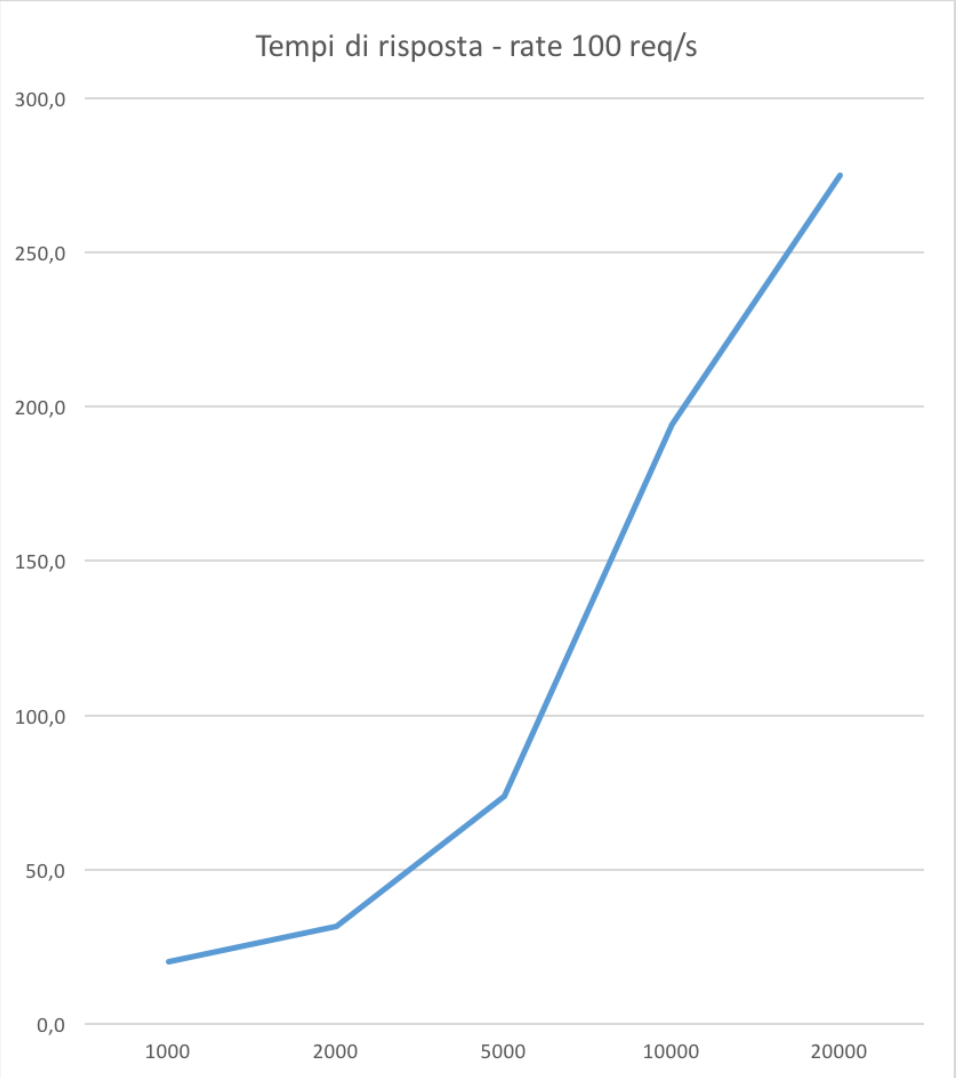

Figura 8: Crescita dei tempi di risposta all'aumentare del numero di richiesta, si osserva come il lavoro dei worker non è sufficiente è compensare la frequenza delle richieste.

5.2 Valutazione delle performance

In questo caso abbiamo deciso di testare il programma in un possibile scenario applicativo vicino a quello di uno sviluppatore indipendente o di uno studente. Valutando le dimensioni tipiche di un *Virtual Private Server* di fascia *entry level*, abbiamo utilizzato una macchina virtuale con 512 MB di RAM con

sistema operativo Ubuntu Server. Le condizioni del test includevano un numero di richieste fisso pari a 10000 richieste in arrivo, facendo variare la frequenza con cui queste giungevano. Si è deciso di prendere in considerazione sia il tempo di risposta che la frequenza con cui queste risposte giungevano al client che generava il carico di lavoro.

Quello che si è potuto constatare, relativamente alla *figura 9*, è il crescere dei tempi di risposta linearmente, con un andamento paragonabile a quello osservato ma in questo caso riconducibile ad un carico di richieste al secondo, piuttosto che all'aumentare delle connessioni. E' significativo osservare la diminuzione della pendenza per alte frequenze di richieste, riconducibile al raggiungimento della saturazione delle risorse del web switch che lavora ora ad un regime massimo.

In *figura 10*, invece, è possibile osservare come, ad un aumento dei tempi di risposta corrisponde, all'aumentare della frequenza di invio delle richieste, un aumento della frequenza di ricezione delle risposte fino un livello di assestamento. Difatti si ha, mantenendo le connessioni costanti, il raggiungimento di un livello medio attorno al quale rimane, a meno di oscillazioni, una apprezzabile frequenza di risposta. Si può dedurre l'esistenza di un livello di saturazione che non può essere ne abbattuto, a causa della frequenza elevata, ne innalzato a causa della frequenza fissa con il quale i worker si liberano e vengono rioccupati da nuove connessioni. Tale frequenza di lavoro è dovuta sia alla presenza di una coda delle richieste che garantisce una certa inerzia, sia al raggiunto limite delle capacità della macchina di operare il context switching e nel ricreare i thread.. Il che ci porta a concludere che a pieno regime il web switch riesce ad adeguarsi alla velocità con cui sono in arrivo le richieste e soddisfare la domanda, seppure con un ritardo lineare nei tempi di risposta.

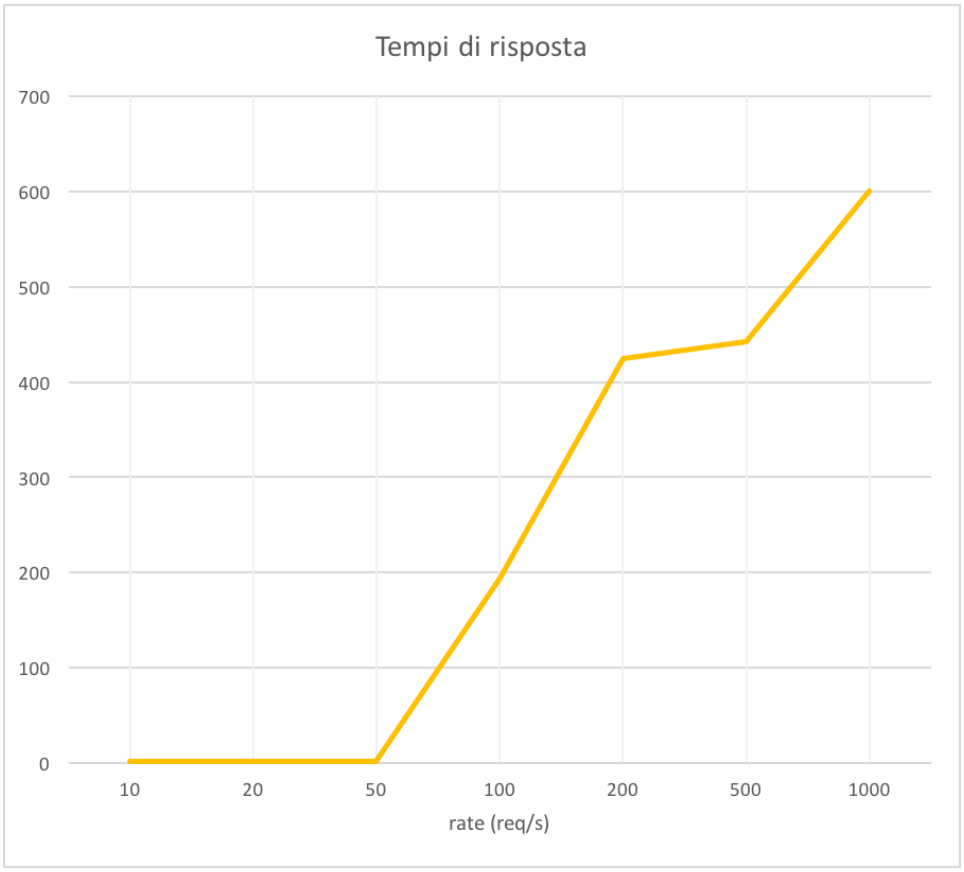

Figura 9: Analisi dei tempi di risposta all'aumentare del numero di richieste per secondo.

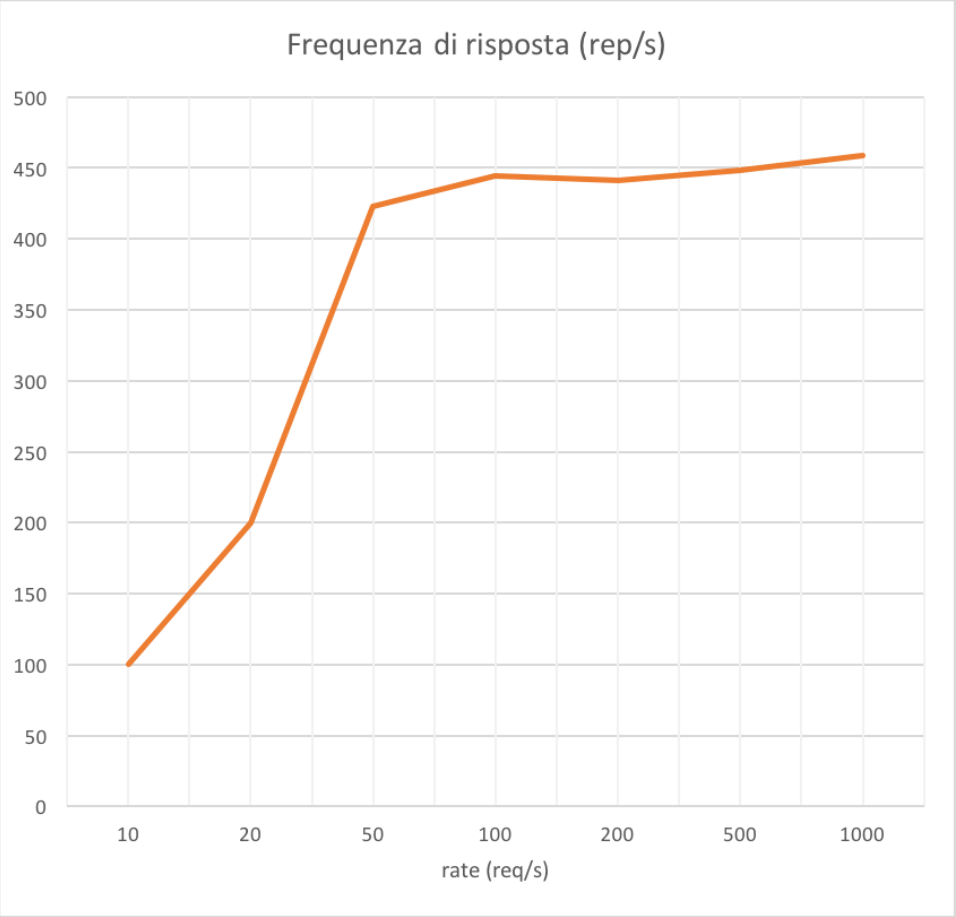

Figura 10: Analisi della frequenza di risposta.

5.3 Comparazione con Apache

Con le caratteristiche della macchina di test già riportate nella sezione precedente, si è proceduto a verificare, una volta valutate le performance a regime, quale fosse l'overhead aggiunto dal nostro programma ad una delle macchine del cluster montante un web server Apache.
Possiamo verificare le statistiche di questo test alla *figure 11*. Si osserva come, malgrado i tempi di risposta siano inferiori, tale differenza diventa poco apprezzabile nel momento in cui si ha una maggiore frequenza nelle risposte da parte del web server Apache. In conclusione si osserva come la durata del test è, in entrambi i casi, irrisoria e confrontabile. Per cui

l'overhead aggiunto, per quanto non possa essere eccessivamente abbattuto, risulta essere accettabile per un'applicazione effettiva del web switch.

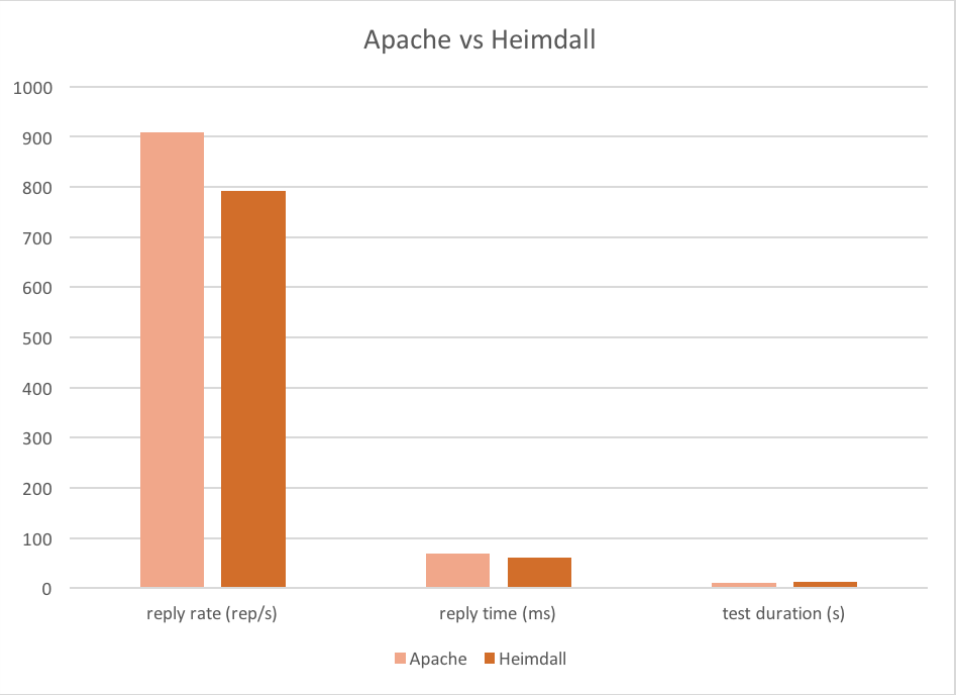

Figura 11: Confronto dei tempi di risposta di un web server Apache con il web switch Heimdall(10000 richieste per una frequenza di 100 req/s).

5.4 Limitazioni

Durante i test che sono appena stati descritti, in particolare quelli con macchina virtuale, monitorando anche le risorse della macchina, si è potuto constatare che era presente un'**occupazione della RAM** crescente e, in alcuni casi, eccessiva, al punto da non permettere alla macchina di proseguire ulteriormente nelle operazioni. Tale occorrenza, per quanto rara e verificabile solamente nel caso di un carico veramente elevato, è dovuta alla non corretta liberazione della memoria nei processi che agiscono da worker: un'analisi più accurata ha portato alla luce la presenza di una grande quantità di memoria virtuale allocata e liberata correttamente ma, con una frequenza elevata di richieste in arrivo, non viene liberata memoria con la stessa velocità con cui viene occupata, portando ad una situazione pericolosa di saturazione.

Ben altra saturazione è stata evitata, invece, limitando il **numero di file descriptor** associabili ad una socket, imponendo alla macchina di rispettare un certo limite ed evitando, quindi, di portare il programma al blocco nel caso di termine delle risorse a disposizione del processo.

E' possibile apprezzare in *figura 12* l'occupazione di memoria e di file descriptor durante uno *stress test* poco prima del raggiungimento del limite di memoria a disposizione.

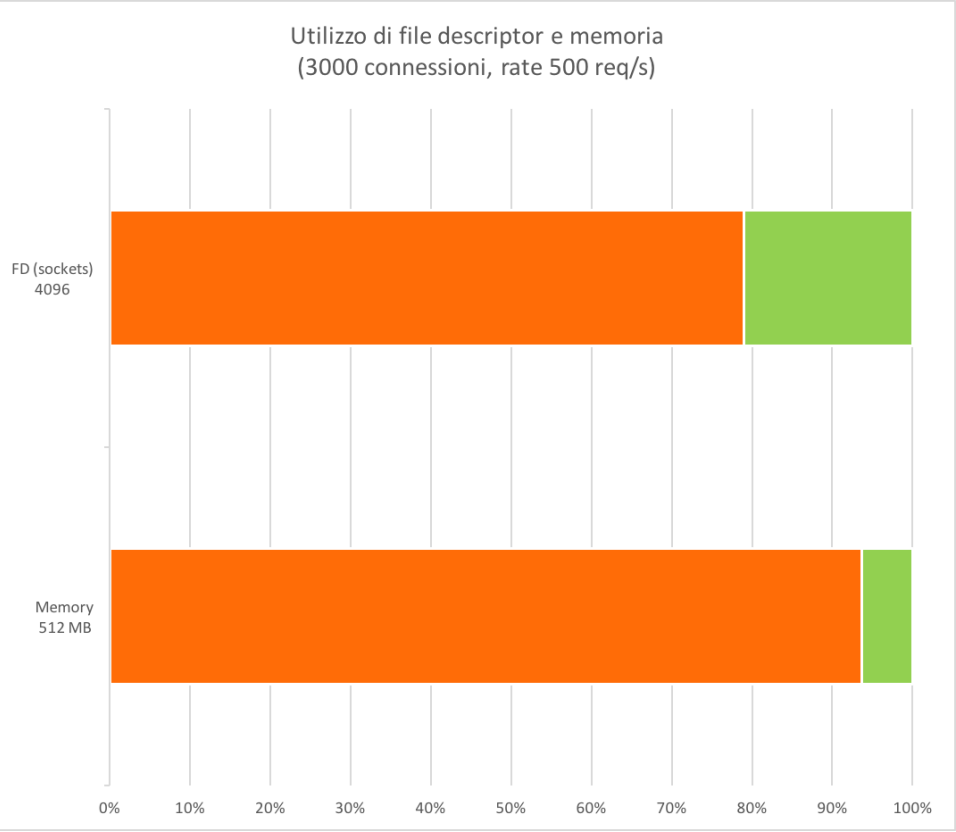

Figura 12: Visualizzazione dell'occupazione di memoria e fd durante uno stress test.

5.5 Conclusioni

6 Future implementazioni

In questo capitolo vogliamo parlare di alcune ulteriori proposte che ci sono venute in mente per poter migliorare Heimdall. Si tratta quindi di sviluppi futuri che abbiamo elaborato nel corso dello sviluppo del progetto e a seguito di alcune limitazioni che abbiamo identificato nei componenti da noi sviluppati.

6.1 Analisi della richiesta

Come abbiamo già visto nei paragrafi dedicati al *parsing* delle richieste e delle risposte HTTP, è stato possibile, in fase progettuale, scegliere su quali *header* basare il funzionamento elementare del web switch. Abbiamo visto anche come i file di configurazione garantiscono veramente molta flessibilità al sistemista in sede di installazione.
Non è difficile immaginare che, in una futura implementazione, l'applicazione non possa basarsi su file di configurazione più dettagliati e quindi possibilità di utilizzare tutti i parametri a disposizione per l'analisi di richiesta e risposta. Per cui si aprono scenari in cui si possono, ad esempio, configurare server di *cache di immagini* in particolari posizioni del cluster, oppure utilizzare un algoritmo di schedulazione *state-aware* che si basi anche sulla grandezza dei file richiesti e che quindi, a monte, eviti il carico di macchine sensibili.

6.2 Webserver performante

6.3 Worker come thread

Abbiamo visto nel paragrafo dedicato che il Worker nel contesto attuale è un processo figlio del processo principale. Questa è stata una scelta progettuale dettata da alcune caratteristiche che ritenevamo opportune per il WebSwitch, ad esempio l'isolamento della connessione HTTP. Infatti nella nostra architettura ogni Worker lavora in un contesto indipendente dagli altri quindi un eventuale problema con una connessione non danneggia le altre. L'utilizzo però di processi ci ha portato a dover gestire alcuni piccoli problemi, infatti ci siamo resi conto degli "elevati" tempi di fork() e anche dei tempi che occorrono ad eseguire un cambio di contesto. Inoltre ci siamo

dovuti equipaggiare di memoria condivisa per poter scambiare dati con i Worker e adottare un meccanismo di messaggi per l'invio di file descriptor. Pensiamo quindi che utilizzare un approccio a Thread ci permetta di essere più performanti in quanto si sa che la creazione e la schedulazione di un thread è più veloce rispetto a quella di un processo, inoltre come sappiamo i thread condividono lo stesso spazio di memoria e questo evita i problemi di memoria condivisa e invio file descriptor.

Annotazioni

[1] Leslie Lamport, *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.

[2] Apache HTTP Server Project, `https://httpd.apache.org/`

[3] Throwable (Java Platform SE 8), `https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html`

[4] RFC: 2616: Hypertext Transfer Protocol - HTTP/1.1, `https://tools.ietf.org/html/rfc2616`

[5] RFC: 2616: Persistent Connections, `https://tools.ietf.org/html/rfc2616#section-8.1`

[6] RFC: 2616: Pipeling, `https://tools.ietf.org/html/rfc2616#section-8.1.2.27`

[7] RFC: 793: Connection Establishment and Clearing, `https://tools.ietf.org/html/rfc793#section-2.7`

[8] Apache Module: mod_status, `http://httpd.apache.org/docs/2.2/mod/mod_status.html`

[9] GDB: The GNU Project Debugger, `https://www.gnu.org/software/gdb/`

[10] htop: an interactive process viewer for Unix, `http://hisham.hm/htop/`

[11] Valgrind, `http://valgrind.org/`

A Manuale per l'uso

B Vagrant

Durante lo sviluppo ci siamo imbattuti in alcune problematiche legate alla portabilità del codice che stavamo scrivendo, errori di inclusione di file header, funzioni con comportamenti anomali, macro differenti e problemi nella compilazione. Questo perché lo sviluppo procedeva su macchine con sistemi operativi differenti, nello specifico Mac OSX e Debian. Da qui la necessità di avere un ambiente unificato per l'esecuzione del codice. La soluzione al problema era di facile intuizione, creare una macchina virtuale su VirtualBox e distribuirla su tutti i computer utilizzati per lo sviluppo, purtroppo però mettere in piedi questa soluzione può rivelarsi un'operazione tediosa, installazione del sistema operativo, configurazione dei programmi per lo sviluppo e condivisione di una VM che pesa diversi MB.

Vagrant è uno strumento per la creazione di ambienti di sviluppo completo. Fondamentalmente si tratta di un'applicativo scritto in Ruby che sfruttando le API messe a disposizione da VirtualBox è in grado di manipolare la gestione delle macchine virtuali al suo interno. Il tutto semplicemente compilando una "ricetta" chiamata Vagrantfile. Il **Vagrantfile** è un file all'interno del quale si inseriscono tutte le specifiche riguardo la VM che vogliamo preparare, impostando il sistema operativo, ulteriori programmi da installare, cartelle condivise, configurazioni di rete e quant'altro. Una volta preparato il vagrantfile questo può essere condiviso tra tutti gli sviluppatori, quindi senza dover condividere l'intera VM basterà solo questo file per poter avere tutte le macchine virtuali allo stesso stato. Ogni volta che uno sviluppatore avrà necessità di modificare il comportamento della VM basterà modificare il Vagrantfile e condividerlo con gli altri. Ultima caratteristica è che vagrant è pensato per lasciare allo sviluppatore la scelta dell'IDE che preferisce creando un ambiente completamente trasparente per lo sviluppo del software.

Vagrant makes the "works on my machine" excuse a relic of the past.

C Cluster virtuale

D Tool per i debug

D.1 GDB

GNU debugger (chiamato semplicemente GDB)[9] è il potente debbugger del progetto GNU capace di analizzare numerosi linguaggi di programmazione tra cui C.
Il suo utilizzo è stato di fondamentale importanza per poter scovare errori difficilmente individuabili a causa dell'architettura multiprocesso di Heimdall.

D.2 htop

Htop[10] è un visualizzatore interattivo di processi per i sistemi Unix. Anche se non è un tool di debug ha contribuito al "debug" di Heimdall permettendoci di controllare lo stato dei processi, dei threads e della memoria in tempo reale.

D.3 Valgrind

Valgrind[11] è un potente strumento di debug che nello specifico scova memory leaks e allocazioni improprie.
Anche questo tool è stato di fondamentale importanza dato che ci ha permesso di scovare leak causati dall'errata gestione della memoria, in particolar modo dai *Throwable*.

E Tool per i test

In fase di sviluppo è necessario eseguire molteplici test prima di poter dare per *certificato* il corretto funzionamento di un componente del programma. Nel nostro caso non è stata tanto la quantità dei test l'elemento caratterizzante, quanto la necessità di compiere questi test in un set molto variegato di casi.

Andiamo ora a vedere nel dettaglio quali sono stati i *tool* che ci hanno permesso prima di verificare la corretta risposta del web switch all'arrivo di una richiesta secondo protocollo HTTP/1.1, quindi di aumentare il livello di complessità: dalla corretta ricezione di un file da un server remoto fino al funzionamento in condizioni di ideali utilizzo (via browser da parte di un client qualsiasi), passando per le condizioni di stress da carico nell'analisi delle performance.

E.1 Telnet

In questo caso ci si riferisce al programma da linea di comando che, implementando il protocollo di rete omonimo lato client, permette di instaurare una sessione di login verso un host remoto. Nel nostro caso ci ha permesso di eseguire il *debugging* della risposta ad un regolare, semplificato, pacchetto HTTP. E' possibile utilizzare un client telnet da qualsiasi sistema operativo, inoltre è già presente nella maggior parte delle distribuzioni Linux.

```
$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.1
Host: 127.0.0.1
```

E.2 PostMan

Questa è una applicazione presente nel *Chrome Web Store* e che gira come plug-in del famoso browser Google Chrome. E' concepita come tool per testare le API, in particolare permette di specificare una richiesta HTTP con focus su quelli che sono i parametri ed i metodi specifici del protocollo. Nel

nostro caso ha permesso di costruire un pacchetto base HTTP e verificare la corretta risposta del web switch al variare dei parametri. Potendo, inoltre, concentrarci su un singolo file da richiedere al server remoto, si è potuto sfruttare PostMan per analizzare i tempi di risposta, soprattutto in condizioni di file di grandi dimensioni.

E.3 HttPerf

E' un tool per misurare le performance di un server web, svilupatto dai laboratori della HP. E' stato usato per generare diversi set di carico di lavoro da sottoporre al web switch per analizzarne le performance. Questo strumento è particolarmente versatile e permette di specificare tutta una serie di parametri come il *numero di connessioni*, il *numero di richiesto*, il *rate* con cui effettuare tali connessioni, la *risorsa* da richiedere, nonchè la possibilità di aprire un numero necessario di porte TCP per verificare la risposta di un certo carico di lavoro durante una sessione.

```
$ httperf --server=localhost --uri=/index.html --port=8080
          --num-conns=3000 --num-calls=2 --rate=500
```

E.4 Browser

Questo è il tool principe per quanto riguarda il test "finale" del web switch: ci permette di verificare che l'esperienza utente nell'utilizzo di Heimdall nell'inoltrare le richieste ad un clsuter è confrontabile con una richiesta diretta ad una delle macchine del cluster stesso. Tuttavia in fase di sviluppo ci si è scontrati con le caratteristiche proprie di ciascun browser, per poterne valutare le performance.
Ci si è imbattuti innanzitutto nella funzionalità di *pipelining* disabilitata di default su **Firefox** ma che è possibile abilitare, oppure l'apertura di diverse connessioni con il server entro le quali vengono effettuate molteplici richieste come *Chrome* o *Safari* o, generalmente, un qualsiasi browser moderno.

E' possibile osservare tali peculiarità, durante la richiesta verso un sito mediamente complesso, come quello utilizzato durante i test, dal listato

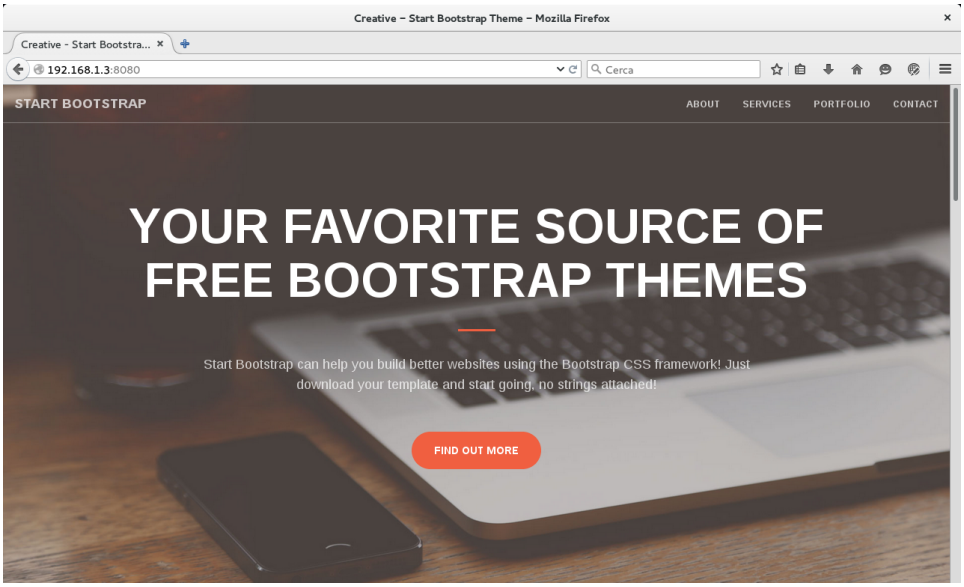

Figura 13: Schermata con la richiesta, soddisfatta con successo, del sito ospitato durante lo sviluppo su una delle macchine del cluster (browser utilizzato: Mozilla Firefox)

sottostante, preso dai *file di log durante una navigazione verso l'homepage con Chrome Web Browser*.

```
Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.5 - worker: 6882 - GET /font-awesome/css/font-awesome.min.css HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.6 - worker: 6881 - GET /css/animate.min.css HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.4 - worker: 6883 - GET /js/bootstrap.min.js HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.5 - worker: 6879 - GET /js/jquery.easing.min.js HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.5 - worker: 6882 - GET /js/jquery.fittext.js HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.4 - worker: 6880 - GET /js/wow.min.js HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.6 - worker: 6881 - GET /js/creative.js HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.6 - worker: 6878 - GET /img/portfolio/2.jpg HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.4 - worker: 6883 - GET /img/portfolio/4.jpg HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.5 - worker: 6879 - GET /img/portfolio/5.jpg HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.4 - worker: 6880 - GET /img/portfolio/6.jpg HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.4 - worker: 6883 - GET /img/header.jpg HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.4 - worker: 6883 - GET /font-awesome/fonts/fontawesome-webfont.woff?v=4.3.0 HTTP/1.1
[Wed Feb 10 12:10:58 2016] - 192.168.1.3:8080 to: 192.168.1.4 - worker: 6877 - GET /favicon.ico HTTP/1.1
```