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, 22 febbraio 2016

Indice

Figura 1: Heimdall guardiano del *Bifröst* e del *regno di Asgard*

1 Introduzione

1.1 Perché Heimdall?

Heimdall è un 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.

1.2 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

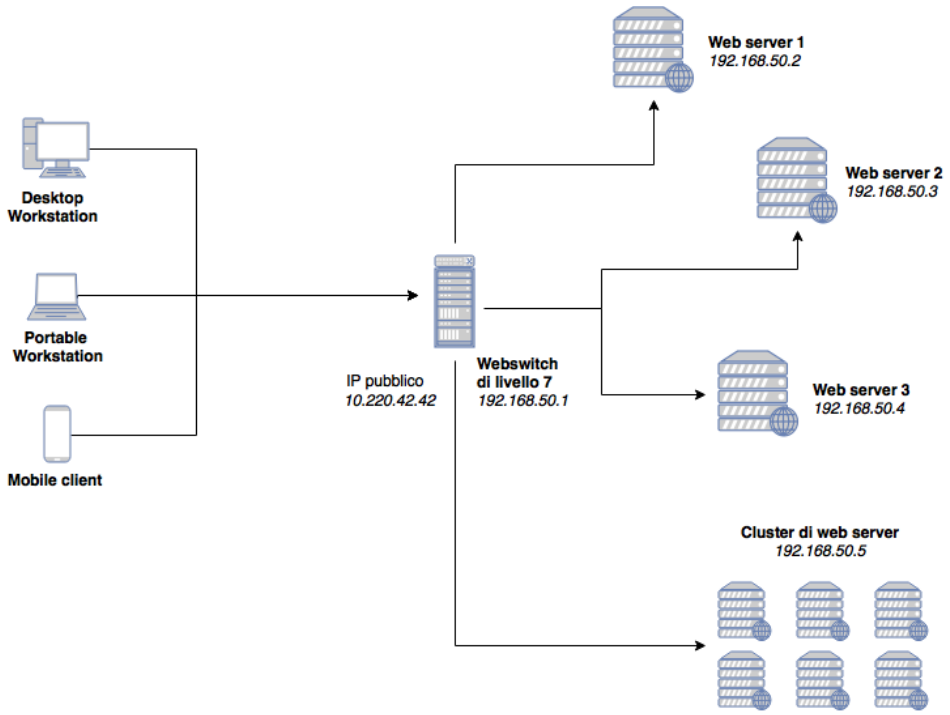

Figura 2: Esempio di uno *switch di livello 7 (OSI)*

gestire l'inoltro della richiesta ed il reinoltro della risposta del web server. Tutto questo in maniera totalmente trasparente al client.

1.3 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[1] in ascolto sulla porta 80;
- Ognuna delle macchine monta il modulo ApacheStatus (3.2.1) come monitor di carico.

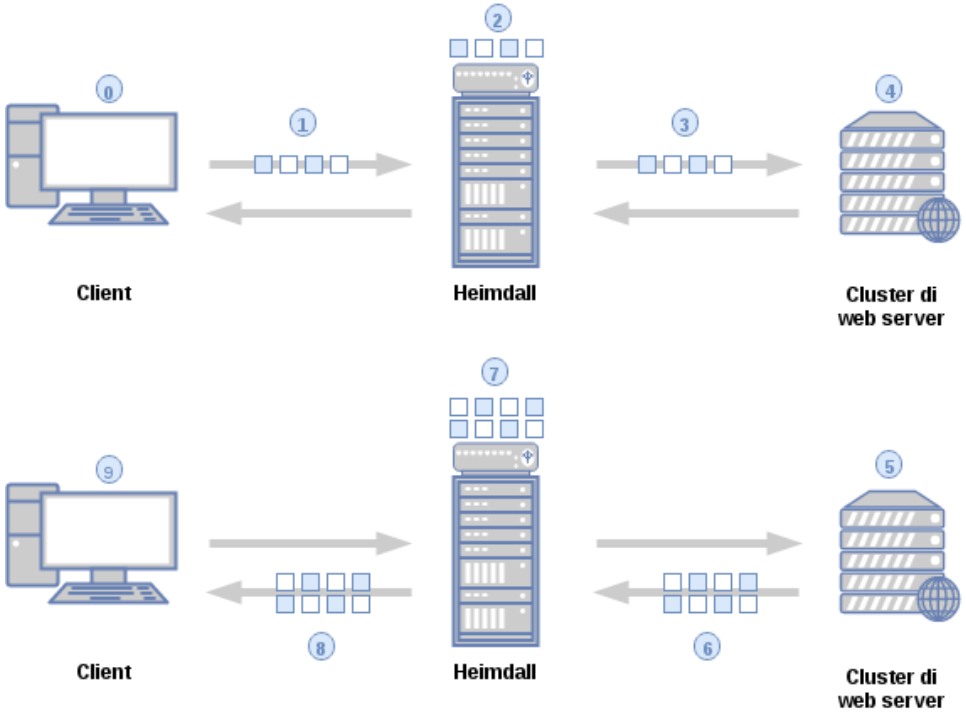

Figura 3: Overview sul funzionamento di Heimdall

2 Architettura

2.1 Overview

Nella figura 3 è possibile vedere uno schema di funzionamento di Heimdall (o più in generale di un Web Switch).
Un client effettua una richiesta verso Heimdall (punto **0** - **1**), che, una volta ricevuta, la inoltra verso un Web Server presente nel cluster (punto **2** - **3**). Una volta pronta la risposta (punto **4** - **5**) il server risponderà ad Heimdall (punto **6** - **7**), che, a sua volta, la inoltrerà al client (punto **8** - **9**).

2.2 Il processo principale

Il cuore di Heimdall è codificato nel *main.c* da cui inizia l'esecuzione del processo principale.
Il processo, una volta avviato, inizializzerà i componenti principali, tra questi abbiamo:

- Config (paragrafo 2.3);
- Log (paragrafo 2.4);
- Scheduler (paragrafo 2.7);
- Pool Manager (paragrafo 2.6).

Possiamo dividere questi quattro componenti in due 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) svolgono alcuni compiti dedicati che altrimenti dovrebbero essere eseguiti dal processo principale.

2.2.1 I processi figli e la memoria condivisa

Fatte le prime inizializzazioni, il processo principale si predispone ad eseguire il suo compito. 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 per fornire l'**IPC** (*Inter-Process Communication*). Per motivi che saranno più chiari in seguito, chiameremo da adesso in poi i processi figli *Worker*.

Evidenziamo qui i tratti principali della funzione:

```
1 ConfigPtr config = get_config();
2
3 int n_prefork = 0;
4 ThrowablePtr throwable = str_to_int(config->pre_fork, &n_prefork);
5 if (throwable->is_an_error(throwable)) {
6   get_log()->t(throwable);
7 }
```

Dal Config viene recuperato il numero di processi figli da creare e, 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 quattro array che ora vedremo nel dettaglio:

- ***worker_array***: Array in cui vengono memorizzati i *pid_t* dei Worker.
- ***worker_busy***: 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 in attesa di nuove connessioni.
- ***worker_counter***: Array che tiene traccia di quante connessioni sono state assegnate al Worker. Tale parametro sarà chiarito nel paragrafo del Pool Manager e servirà alla schedulazione dei Worker.
- ***worker_server*** : Struttura di passaggio utilizzata dal thread Scheduler, che ricordiamo essere uno dei componenti inizializzati all'avvio di Heimdall, la cui funzionalità sarà descritta in maniera più adeguata nel relativo paragrafo.

La seguente porzione di codice mette in evidenza le righe che si occupano della creazione, della mappatura e dell'inizializzazione della memoria condivisa.

```
1  int total_size = 0;
2  total_size += sizeof(THPSharedMem);
3  total_size += sizeof(pid_t)*n_prefork;  // Array worker
4  total_size += sizeof(int)*n_prefork;    // Array busy
5  total_size += sizeof(int)*n_prefork;    // Array counter
6  total_size += sizeof(Server)*n_prefork; // Array server
7
8  // Initializes shared memory
9  void *start_mem = init_shm(WRK_SHM_PATH, total_size, WRK_SEM_PATH);
10 if (start_mem == NULL)
11   exit(EXIT_FAILURE);
```

```
12
13 // Mapping shared memory segment
14 THPSharedMemPtr worker_pool = start_mem;
15 worker_pool->worker_array   = start_mem+sizeof(THPSharedMem);
16 worker_pool->worker_busy    = start_mem+sizeof(THPSharedMem)+sizeof(
       pid_t)*n_prefork;
17 worker_pool->worker_counter = start_mem+sizeof(THPSharedMem)+sizeof(
       pid_t)*n_prefork+sizeof(int)*n_prefork;
18 worker_pool->worker_server  = start_mem+sizeof(THPSharedMem)+sizeof(
       pid_t)*n_prefork+sizeof(int)*n_prefork+sizeof(int)*n_prefork;
19
20 int i = 0;
21 for (i = 0; i < n_prefork; ++i) {
22   worker_pool->worker_counter[i]  = 0;
23   worker_pool->worker_array[i]    = 0;
24   worker_pool->worker_busy[i]     = 0;
25 }
```

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*.

Siamo finalmente giunti alla creazione dei processi figli, i cosidetti Worker, ancora una volta evidenziamo qui la porzione di codice interessata:

```
1  int children;
2  for (children = 0; children < n_prefork; ++children) {
3    pid_t child_pid;
4    errno = 0;
5
6    child_pid = fork();
7    if (child_pid == -1) {
8      get_log()->t(get_throwable()->create(STATUS_ERROR,
           get_error_by_errno(errno), "do_prefork"));
9    }
10   // Child
11   if (child_pid == 0) {
12     // see worker.c
13     start_worker();
14     break;
15   } else {
```

```
16      if (sem_wait(sem) == -1) {
17        return get_throwable()->create(STATUS_ERROR, "sem_wait", "
              do_prefork");
18      }
19      // Scan array and set worker to first position available
20      int i, flag = 0;
21      for (i = 0; i < n_prefork; ++i) {
22        if (worker_pool->worker_array[i] == 0) {
23          worker_pool->worker_array[i] = child_pid;
24          flag = 1;
25          break;
26        }
27      }
28
29      if (sem_post(sem) == -1) {
30        return get_throwable()->create(STATUS_ERROR, "sem_post", "
              do_prefork");
31      }
32
33      if (flag == 0) {
34        return get_throwable()->create(STATUS_ERROR, "Cannot␣add␣
              worker_pid␣to␣array", "do_prefork");
35      }
36    }
37  }
```

Nel codice sopraesposto, viene eseguito un ciclo *n_prefork* volte, dove *n_prefork* è il numero di processi che si vuole creare. Il Worker, una volta creato, si avvia eseguendo la funzione *start_worker()*; il processo padre invece si occupa di accedere in memoria condivisa per aggiornare le strutture dati che abbiamo spiegato sopra.

Si noti, 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 nella memoria del figlio e, grazie a questo il figlio si ritroverà i componenti come *Config*, *Log* e *Memoria Condivisa* già inizializzati e referenziati.

2.2.2 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 connessioni.

```
1  int port = 0;
2  throwable = str_to_int(config->server_main_port, &port);
3  if (throwable->is_an_error(throwable)) {
4    log->t(throwable);
5  }
6
7  // Creates a new server
8  int sockfd;
9  throwable = create_server_socket(TCP, port, &sockfd);
10 if (throwable->is_an_error(throwable)) {
11   log->t(throwable);
12   exit(EXIT_FAILURE);
13 }
14
15 log->i(TAG_MAIN, "Created␣new␣server␣that␣is␣listening␣on␣port␣%d",
       port);
16
17 // Starts listening for the clients
18 int backlog = 0;
19 throwable = str_to_int(config->backlog, &backlog);
20 if (throwable->is_an_error(throwable)) {
21   log->t(throwable);
22 }
23
24 throwable = listen_to(sockfd, backlog);
25 if (throwable->is_an_error(throwable)) {
26   log->t(throwable);
27   exit(EXIT_FAILURE);
28 }
29
30 log->i(TAG_MAIN, "Ready␣to␣accept␣incoming␣connections...");
31
32 int count = 0;
33
34 // Starts to listen incoming connections
```

```
35  while(TRUE) {
36    // Accepts new connection
37    int new_sockfd;
38    throwable = accept_connection(sockfd, &new_sockfd);
39    count++;
40    if (throwable->is_an_error(throwable)) {
41      log->t(throwable);
42      exit(EXIT_FAILURE);
43    }
44
45    while(TRUE) {
46      throwable = th_pool->add_fd_to_array(&new_sockfd);
47      if (throwable->is_an_error(throwable)) {
48        throwable->destroy(throwable);
49      } else {
50        break;
51      }
52    }
53
54    log->i(TAG_MAIN, "New connection accepted on socket number %d - 
          total %d", new_sockfd, count);
55  }
```

Questo è il vero cuore di Heimdall. Si noti che, 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 connessioni in un loop infinito. Quando un nuovo client si collega la funzione *accept_connection()* ritorna e il processo principale prosegue la sua strada. Si noti che lo sviluppo della funzione *main()* è stato pensato per poter eseguire il minor numero di linee di codice, 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 relativo al Pool Manager), il processo principale termina il primo ciclo del loop infinito e torna in ascolto di nuove connessioni. Come ultima cosa si sottolinea il fatto che l'esecuzione della chiamata *fork()* viene eseguita prima dell'avvio del "server in ascolto", questo per evitare di dover attendere i lunghi tempi di fork.

2.3 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 Web Switch alle proprie esigenze.

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

2.3.1 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 *state-less*, con il valore 1 si sceglie di usare l'algoritmo *state-aware*.
- *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 un 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 nella sezione dedicata (2.4).
- *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*: il massimo tempo di attesa del thread di controllo (*watchdog*), dopo il quale il worker viene ritenuto bloccato oppure si è verificata un'interruzione della connessione.

- *killer_time*: il tempo che indica la durata di un ciclo di controllo che il thread incaricato di verificare la corretta esecuzione del worker esegue.
- *update_time*: in riferimento al thread di scheduling, definisce il tempo di aggiornamento dello stato dei server del cluster, nel caso di utilizzo dell'algoritmo state-aware.
- *server_config*: specifica il percorso del file di configurazione dove vengono 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 di richiesta 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:

```
1 # STATELESSRR 0 STATEFUL 1
2 algorithm_selection 1
3
4 # The number of active processes to handle requests
5 # If prefork is 0, the system create anyway 1 child process
6 pre_fork 10
7
8 # 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

...
```

2.3.2 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
```

2.3.3 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.

```
#define CONFIGFILE "../code/config/heimdall_config.conf"
```

- Una struttura dati di tipo *ConfigPtr*

```c
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;
```

Si noti che 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.

```c
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

```

```
16    char *subset = malloc(sizeof(char) * total);
17    if (subset == NULL) {
18        fprintf(stderr, "Error␣in␣_get␣config_parser.␣\n");
19        return NULL;
20    }
21
22    int j;
23    for(j = 0; j < total - 1; ++j, ++from) {
24        subset[j] = array[from];
25    }
26
27    subset[total-1] = '\0';
28
29    return subset;
30  }
31
32  int init_config(const char *path, int config_handler(char *key, char *
        value, void *p_config), void *ptr_config) {
33
34    singleton_config = ptr_config;
35
36    // open file
37    FILE *config_file = fopen(path, "r");
38    if (config_file == NULL) {
39      fprintf(stderr, "Error␣while␣trying␣to␣open␣config␣file.␣\n");
40      return -1;
41    }
42
43    // while each line
44    char string[MAX_LENGTH];
45
46    while(fgets(string, MAX_LENGTH, config_file)) {
47      // skip, line comment or empty line
48      if (string[0] == '#' || string[0] == '\n') {
49        continue;
50      }
51
52      char *key = _get(string, 0, ESCAPE_CHARACTER);
53      if (key == NULL) {
54        fprintf(stderr, "Error␣in␣init_config␣config_parser.␣\n");
55        return -1;
```

```
56      }
57
58      char *value = _get(string, strlen(key)+1, '\n');
59      if (value == NULL) {
60        fprintf(stderr, "Error in init_config config_parser. \n");
61        return -1;
62      }
63
64      if(config_handler(key, value, ptr_config) == -1) {
65        fprintf(stderr, "Error config parser , no key '%s' found in
              Config. \n", key);
66        return -1;
67      }
68
69      free(value);
70      free(key);
71    }
72
73    fclose(config_file);
74
75    return 0;
76  }
```

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.

```
1   int config_handler(char *key, char *value, void *p_config) {
2
3     Config* config = (Config *) p_config;
4
5     if (strcmp(key, "algorithm_selection") == 0) {
6       if (asprintf(&config->algorithm_selection, "%s", value) == -1) {
7         return -1;
8       }
9     } else if (strcmp(key, "pre_fork") == 0) {
10      if (asprintf(&config->pre_fork, "%s", value) == -1) {
11        return -1;
12      }
13
14      ...
15
```

```
16    } else {
17      return -1;  /* unknown key, error */
18    }
19    return 0;
20  }
```

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:

```
1  typedef struct server_config {
2    char **servers_names;
3    char **servers_ip;
4    int total_server;
5  } 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:

```
1  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.

2.4 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 si noti 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()*:

```
1  Log *new_log() {
2    ConfigPtr config = get_config();
3
4    Log *log = malloc(sizeof(Log));
5    if (log == NULL) {
6      fprintf(stderr, "Memory␣allocation␣error␣in␣new_log!");
7      exit(EXIT_FAILURE);
8    }
9
10   // retrieving log file pointer or allocating it
11   req_log = fopen(config->log_file_req, "a+");
12   if (req_log == NULL) {
13     fprintf(stderr, "Error␣in␣log␣file␣opening!\n");
14   }
15
16   // retrieving log file pointer or allocating it
17   resp_log = fopen(config->log_file_resp, "a+");
18   if (resp_log == NULL) {
19     fprintf(stderr, "Error␣in␣log␣file␣opening!\n");
20   }
21
22   // Set "methods"
23   log->d = d;
24   log->i = i;
25   log->e = e;
26   log->r = r;
27   log->t = t;
28
29   return log;
30 }
31
32 Log *get_log() {
```

```
33    if (singleton_log == NULL) {
34      singleton_log = new_log();
35    }
36
37    // return singleton
38    return singleton_log;
39  }
```

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:

```
1  typedef struct log {
2    int (*d)(const char* tag, const char *format, ...);
3    int (*i)(const char* tag, const char *format, ...);
4    int (*e)(const char* tag, const char *format, ...);
5    int (*r)(int type, void *arg, char *host, int pid);
6    void (*t)(ThrowablePtr throwable);
7  } Log, *LogPtr;
```

Questo sarà lo stesso approccio applicato a tutti i componenti sviluppati di Heimdall, che incontreremo più avanti nella trattazione.

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

```
1  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:

```
1  static int i(const char* tag, const char *format, ...) {
2    ConfigPtr config = get_config();
3
4    int byte_read = 0;
5
6    int level = 0;
7    ThrowablePtr throwable = str_to_int(config->log_level, &level);
8    if (throwable->is_an_error(throwable)) {
9      t(throwable);
10   }
11
12   int print_enable = 0;
13   throwable = str_to_int(config->print_enable, &print_enable);
14   if (throwable->is_an_error(throwable)) {
15     t(throwable);
16   }
17
18   if (INFO_LEVEL >= level && print_enable == 1) {
19
20     char *formatted_str;
21
22     va_list arg;
23     va_start (arg, format);
24     byte_read = vasprintf(&formatted_str, format, arg);
25     va_end (arg);
26
27     char *output;
28     byte_read = asprintf(&output, "%s: I/%s: %s", timestamp(), tag,
           formatted_str);
29
30     free(formatted_str);
31
32     printf("%s \n", output);
33     fflush(stdout);
34     free(output);
35   }
36
37   return byte_read;
38 }
```

Si osserva che 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 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, con una simile implementazione, la funzione *r()* (response) è quella utilizzata per scrivere sui file di logging delle richieste e delle risposte. Invece la funzione *t()* (throwable) non è altro che un'estensione della funzione *e()*, questa viene utilizzata per stampare a schermo gli oggetti Throwable, dei quali parleremo nel paragrafo successivo.

2.5 Gestione degli errori

Per gestire facilmente gli errori di Heimdall è stata sviluppata la "classe" Throwable. Questo oggetto rappresenta il valore di ritorno di ogni funzione del sistema, uniformando i valori di ritorno delle funzioni.

```
1  typedef struct throwable {
2    int status;
3    char *message;
4    char *stack_trace;
5
6    struct throwable* (*create)(int status, char *msg, char *stack_trace
         );
7    struct throwable* (*thrown)(struct throwable* self, char *
         stack_trace);
8    int (*is_an_error)(struct throwable* self);
9
10   void (*destroy)(struct throwable *self);
11 } 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.
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.

Riportiamo qui un esempio di stampa:

```
1 Mon Feb 22 07:35:16 2016: E/Throwable:
2 Status: -1
3 Message: Operation not permitted
4 Stack Trace:
5  bind->unix_passive_socket->unix_listen
```

Avendo reso omogenee il valore di ritorno di tutte le 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 3.2.1), abbiamo:

```
1 if (strcmp(text, "idleworkers") == 0) {
2   throwable = str_to_int(text_data, &(self->idle_workers));
3   if (throwable->is_an_error(throwable)) {
4     return throwable->thrown(throwable, "get_data.idle_workers");
5   }
6 }
7
8 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, tramite la funzione *thrown_throwable()*, specificando da dove lo si sta effettuando (in questo caso, nella funzione *get_data_idle_workers*).
Siccome l'oggetto Throwable viene allocato dinamicamente (tramite la funzione *create()*) 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.

I dettagli delle implementazioni delle suddette funzioni si trova all'interno dei file *throwable.c* e *throwable.h*.

2.6 Pool manager

Il *Pool Manager* è un thread del processo principale creato all'avvio del programma. Il compito 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:

```
1 // Initializes Thread Pool
2 ThreadPoolPtr th_pool = get_thread_pool();
3 if (th_pool == NULL) {
4   exit(EXIT_FAILURE);
5 }
```

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

```
1  ThreadPoolPtr get_thread_pool() {
2    if (singleton_thdpool == NULL) {
3      singleton_thdpool = init_thread_pool();
4    }
5
6    // return singleton
7    return singleton_thdpool;
8  }
9
10 static ThreadPoolPtr init_thread_pool() {
11
12   get_log()->i(TAG_THREAD_POOL, "Thread␣pool␣start.");
13
14   pthread_t t1;
15   int born;
16
```

```
17    born = pthread_create(&t1, NULL, init_pool, NULL);
18    if (born != 0) {
19      get_log()->e(TAG_THREAD_POOL, "Error in pthread_create");
20      return NULL;
21    }
22
23    ThreadPoolPtr th_pool = malloc(sizeof(ThreadPool));
24    if (th_pool == NULL) {
25      get_log()->e(TAG_THREAD_POOL, "Memory allocation error in
            init_thread_pool!");
26      return NULL;
27    }
28
29    th_pool->thread_identifier   = t1;
30    th_pool->add_fd_to_array    = add_fd_to_array;
31    th_pool->print_fd_array     = print_fd_array;
32
33    return th_pool;
34  }
```

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 *ThreadPool*.

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

```
1   static void *init_pool(void *arg) {
2     // detach itself
3     pthread_detach(pthread_self());
4
5     ConfigPtr config = get_config();
6
7     max_fd = 0;
8     ThrowablePtr throwable = str_to_int(config->max_fd, &max_fd);
9     if (throwable->is_an_error(throwable)) {
10      get_log()->t(throwable);
11      exit(EXIT_FAILURE);
12    }
```

```
13
14    // Init static array and save pointer to global variable
15    fd_array = malloc(sizeof(int) * max_fd);
16    if (fd_array == NULL) {
17      get_log()->e(TAG_THREAD_POOL, "Memory allocation error in
          init_pool!");
18      exit(EXIT_FAILURE);
19    }
20
21    int i;
22    for (i = 0; i < max_fd; ++i) {
23     fd_array[i] = 0;
24    }
25
26    // enter in thread pool loop
27    thread_pool_loop();
28
29    // Never reached
30    return arg;
31  }
```

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 2.3 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.

```
1  for (;;) {
2    int fd = 0;
3    throwable = get_fd(&fd);
4    if (throwable->is_an_error(throwable)) {
5      //get_log()->i(TAG_THREAD_POOL, "No fd to serve.");
```

```
6       throwable->destroy(throwable);
7       continue;
8     }
9     ...
10  }
```

All'interno della funzione *thread_pool_loop()* è presente 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:

```
1   static ThrowablePtr get_fd(int *fd_ptr) {
2     int s = 0;
3     s = pthread_mutex_lock(&mtx_wait_request);
4     if (s != 0) {
5       get_log()->e(TAG_THREAD_POOL, "Error in pthread_mutex_lock");
6     }
7     // Scan array and get the first fd != 0
8     int i, flag = 0;
9     for (i = 0; i < max_fd; ++i) {
10
11      if (fd_array[i] != 0) {
12        *fd_ptr = fd_array[i];
13        fd_array[i] = 0;
14        flag = 1;
15        break;
16      }
17    }
18
19    s = pthread_mutex_unlock(&mtx_wait_request);
20    if (s != 0) {
21      get_log()->e(TAG_THREAD_POOL, "Error in pthread_mutex_unlock");
22    }
23    if (flag == 0) {
24      return get_throwable()->create(STATUS_ERROR, "Cannot get fd", "
            get_fd");
25    } else {
26      return get_throwable()->create(STATUS_OK, NULL, "get_fd()");;
27    }
28  }
```

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:

```
1  int i, min = -1, position = -1;
2  for (i = 0; i < n_prefork; ++i) {
3    if (worker_pool->worker_busy[i] == 0) {
4      if (min == -1) {
5        min = worker_pool->worker_counter[i];
6        position = i;
7      }
8
9      if (worker_pool->worker_counter[i] <= min) {
10       min = worker_pool->worker_counter[i];
11       position = i;
12     }
13   }
14 }
15 // Error no worker available
16 if(position == -1) {
17
18   get_log()->i(TAG_THREAD_POOL, "No Worker available, wait for space."
         );
19
20   if(sem_post(sem) == -1){
```

```
21      get_log()->e(TAG_THREAD_POOL, "Error in sem_wait - thread_pool_loop");
22      exit(EXIT_FAILURE);
23    }
24    continue;
25
26  } else {
27    worker_pool->worker_busy[position] = 1;
28    worker_pid = worker_pool->worker_array[position];
29    worker_pool->worker_counter[position] = worker_pool->worker_counter[position] + 1;
30
31    get_log()->i(TAG_THREAD_POOL, "Get Worker %ld", (long)worker_pid);
32
33    // Retrieving server from scheduler
34    ServerPtr server = get_scheduler()->get_server(get_scheduler()->rrobin);
35    // Storing server in shared memory
36    worker_pool->worker_server[position] = *server;
37
38    if (sem_post(sem) == -1) {
39      get_log()->e(TAG_THREAD_POOL, "Error in sem_wait - thread_pool_loop");
40      exit(EXIT_FAILURE);
41    }
42    break;
43  }
```

All'interno della variabile *position* viene salvato l'indice del worker selezionato, se questo è -1 vuol dire che tutti i worker sono occupati. 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 quindi il semaforo, visto che abbiamo terminato l'utilizzo della memoria condivisa, e ci avviamo

ad eseguire le ultime linee di codice della funzione:

```
1  while (TRUE) {
2    throwable = send_fd(fd, worker_pid);
3    if (throwable->is_an_error(throwable)) {
4      get_log()->e(TAG_THREAD_POOL, "Failed␣attempt␣%d␣to␣send␣file␣
           descriptor␣to␣%ld", attempt, (long) worker_pid);
5      throwable->destroy(throwable);
6      attempt++;
7    } else {
8      break;
9    }
10 }
11
12 // close fd from main side
13 throwable = close_connection(fd);
14 if (throwable->is_an_error(throwable)) {
15   get_log()->t(throwable);
16 }
```

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_controllller.c* e *message_controller.h*. Vengono utilizzate le socket perché non è possibile scambiare il file descriptor senza scambiare anche i permessi di lettura di 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.

2.7 Scheduler

Lo scheduler è un componente fondamentale di un sistema informatico: si occupa di stabilire un ordinamento temporale per 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 al paragrafo 3).

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 nei file *scheduler.c* e *scheduler.h*, ne vediamo un estratto.

```
1 typedef struct scheduler_args {
2   RRobinPtr     rrobin;                       // Round Robin struct
3   ServerPoolPtr server_pool;                  // Server Pool struct
4
5   ServerPtr (*get_server)(RRobinPtr rrobin);  // to retrieve a server
6 } 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:

```
1 typedef struct server_node {
2   char *host_address;       // machine canonical name
3   char *host_ip;            // machine ip address
4   int  status;              // machine status
5   int  weight;              // machine weight
6
7   struct server_node *next; // next server_node
```

```
8 } ServerNode, *ServerNodePtr;
```

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 lo recupererà tramite memoria condivisa (2.2.1).

Lo scheduler è recuperabile tramite la chiamata *get_scheduler()*.

```
1 \* inside thread_pool.c ... *\
2 // Retrieving server from scheduler
3 ServerPtr server = get_scheduler()->get_server(get_scheduler()->rrobin
      );
4 // Storing server in shared memory
5 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* (non disponibile) e non *READY* (operativo) è nella circostanza in cui ogni server del cluster risulta non disponibile per cui il Worker (che analizzeremo in 2.8) 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. Lo vedremo nel dettaglio al paragrafo 3.

2.8 Worker

Il *Worker* 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 Worker (si veda 2.3). Ciò è utile per evitare di rallentare l'esecuzione a causa del ritardo dovuto al tempo di creazione.

Descriviamo ora il lavoro del worker, il codice dei seguenti paragrafi è presente nel file *worker.c* e *worker.h*.

2.8.1 Gestione delle richieste

L'applicazione soddisfa le specifiche *HTTP 1.1*[3], gestendo **connessioni persistenti**[4] e supportando il **pipelining**[5] 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.
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."*[5]

La coda è molto semplice e implementa le seguenti operazioni:

```
1 void (*enqueue)(struct request_queue *self, RequestNodePtr node);
2 struct request_node*(*dequeue)(struct request_queue *self);
```

```
3  int (*is_empty)(struct request_queue *self);
4  struct request_node*(*get_front)(struct request_queue *self);
5  int (*get_size)(struct request_queue *self);
6  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.

```
1  typedef struct request_node {
2      pthread_t thread_id;
3      HTTPRequestPtr request;
4      HTTPResponsePtr response;
5      time_t request_timeout;
6      struct request_node *previous;
7      struct request_node *next;
8      ChunkPtr chunk;
9      int *worker_status;
10     pthread_mutex_t mutex;
11     pthread_cond_t condition;
12 } RequestNode, *RequestNodePtr;
```

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.

2.8.2 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 2.5). L'implementazione delle chiamate è stata resa minimale, in modo tale da concentrarsi solo sulla logica dell'applicazione e non sulla sua effettiva implementazione. Per questo motivo le funzioni per creare una nuova connessione di tipo server e di tipo client sono definite come funzioni all'interno dei file *connection.c* e *connection.h*. Nel dettaglio per il server:

```
1  ThrowablePtr create_server_socket(const int type, const int port, int
       *sockfd) {
2
3    ThrowablePtr throwable = create_socket(type, sockfd);
4    if (throwable->is_an_error(throwable)) {
5      return throwable->thrown(throwable, "create_server_socket");
6    }
7
8    struct sockaddr_in addr;
9
10   memset((void *) &addr, 0, sizeof(addr));  // Set all memory to 0
11   addr.sin_family = AF_INET;                // Set IPV4 family
12   addr.sin_addr.s_addr = htonl(INADDR_ANY); // Waiting a connection on
         all server's IP addresses
13   addr.sin_port = htons(port);              // Waiting a connection on
         PORT
14
15   if (bind(*sockfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
16     return get_throwable()->create(STATUS_ERROR, get_error_by_errno(
           errno), "create_server_socket");
17   }
18
19   return get_throwable()->create(STATUS_OK, NULL, "
       create_server_socket");
```

```
20 }
```

Mentre per il client:

```
1  ThrowablePtr create_client_socket(const int type, const char *ip,
       const int port, int *sockfd) {
2
3    ThrowablePtr throwable = create_socket(type, sockfd);
4    if (throwable->is_an_error(throwable)) {
5      return throwable->thrown(throwable, "create_client_socket");
6    }
7
8    struct sockaddr_in addr;
9
10   memset((void *) &addr, 0, sizeof(addr));  // Set all memory to 0
11   addr.sin_family = AF_INET;                // Set IPV4 family
12   addr.sin_port = (in_port_t) htons((uint16_t) port); // Set server
         connection on specified PORT
13
14   if (inet_pton(AF_INET, ip, &addr.sin_addr) == -1) {
15     return get_throwable()->create(STATUS_ERROR, get_error_by_errno(
           errno), "create_client_socket");
16   }
17
18   if (connect(*sockfd, (struct sockaddr *) &addr, sizeof(addr)) == -1)
         {
19     return get_throwable()->create(STATUS_ERROR, get_error_by_errno(
           errno), "create_client_socket");
20   }
21
22   return get_throwable()->create(STATUS_OK, NULL, "
         create_client_socket");
23 }
```

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 (definite in apposite funzioni nello stesso file), senza stravolgere, per quanto possibile, la logica del sistema.

Richieste HTTP

Alla base della capacità di Heimdall 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 protocollo[3] per poter *parsare* il pacchetto HTTP una volta che questo è correttamente giunto (completo) al web switch. È stata quindi definita la struct seguente.

```
1  typedef struct http_request {
2    char *status;           // whether the request can be handled
3    char *req_type;         // type of the request
4    char *req_protocol;     // accepted only HTTP/1.1
5    char *resp_code;
6    char *resp_msg;
7    char *req_resource;     // resource locator
8    char *req_accept;       // accepting content info
9    char *req_from;         // client and request generic info
10   char *req_host;
11   char *req_content_type; // content type info
12   int   req_content_len;
13   char *req_upgrade;      // no protocol upgrade are allowed
14
15   char *header;
16
17   ThrowablePtr (*get_header)(struct http_request *self, char *req_line
         );
18   ThrowablePtr (*get_request)(struct http_request *self, char *
         req_line, int len);
19   ThrowablePtr (*read_headers)(struct http_request *self, char *string
         , int type);
20   ThrowablePtr (*make_simple_request)(struct http_request *self, char
         **result);
21   ThrowablePtr (*set_simple_request)(struct http_request *self, char *
         request_type, char *request_resource, char *request_protocol,
         char *host);
22   void (*destroy)(struct http_request *self);
23 } 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 la richiesta 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 a fine riga e doppio rispettivamente come separatore fra headers e corpo, 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, nonché la lunghezza della risorsa contenuta nel pacchetto, necessari per la corretta lettura di un completo pacchetto di risposta. Per cui è bastato utilizzare le funzionalità già implementate ed "estendere" la struttura precedente come segue.

```
1 typedef struct http_response {
2   struct http_request *response;
3   int http_response_type;
4   char *http_response_body;  // the body of the message
5
6   ThrowablePtr (*get_http_response)(struct http_response *self, char *
        buffer);
7   ThrowablePtr (*get_response_head)(struct http_response *self, char *
        head);
```

```
8    ThrowablePtr (*get_response_body)(struct http_response *self, char *
         body);
9    void (*destroy)(struct http_response *self);
10 } HTTPResponse, *HTTPResponsePtr;
```

L'unica differenza con l'implementazione già descritta è il mantenimento, oltre che di un campo che specifica trattarsi di una risposta, in un'area di memoria dell'originale contenuto del pacchetto. Tale contenuto verrà reinoltrato al client che ha generato la richiesta.

2.8.3 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 2.3), creando, per ognuna di esse un thread di richiesta che, dialogando con il server schedulato, si occuperà di gestirla. Il thread di lettura sarà bloccato costantemente in attesa di nuove richieste, aggiornando la variabile adibita alla verifica dello stato di esecuzione del worker (si veda il paragrafo 2.8.6), alla ricezione di ogni nuova richiesta.

```
1  // Gets queue
2  RequestQueuePtr queue = worker->requests_queue;
3  ConfigPtr config = get_config();
4  ...
5  while (TRUE) {
6    // Waits
7    while (max_thr_request > max_thread_pchild) {
8      if (pthread_cond_wait(&cond_thr_request, &mtx_thr_request) != 0) {
9          // Handling error
10     }
11   }
12   // Updates timer
13   worker->watchdog->timestamp_worker = time(NULL);
14
15   // Creates the node
16   RequestNodePtr node = init_request_node();
17   ...
18   // Enques the new node
```

```
19    queue->enqueue(queue, node);
20
21    // Receives request
22    ThrowablePtr throwable = receive_http_request(worker->sockfd, node->
          request);
23    if (throwable->is_an_error(throwable)) {
24          // Handling error
25          ...
26    }
27
28    // Creates the request thread
29    int request_creation = pthread_create(&(node->thread_id), NULL,
          request_work, (void *) node);
30    if (request_creation != 0) {
31      // Handling error
32      ...
33    }
34    ...
35  }
```

2.8.4 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:

```
1
2  // retrieving remote host from the shared memory
3  if(sem_wait(sem) == -1){
4    // Handling error
5    ...
6  }
7
8  // Get server from shared memory
9  remote = &(worker_pool->worker_server[i]);
10
11 if(sem_post(sem) == -1){
12   // Handling error
13   ...
14 }
```

```
15
16 // checking for remote host status
17 if (remote->status == SERVER_STATUS_BROKEN) {
18   // Server status error
19   // Handling error
20   ...
21 }
```

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:

```
1  // Creates a new client
2  int sockfd;
3  throwable = create_client_socket(TCP, host, 80, &sockfd);
4  if (throwable->is_an_error(throwable)) {
5    // Handling error
6    ...
7  }
8
9  // Logging - request
10 HTTPRequestPtr request = node->request;
11
12 // Sends request
13 throwable = send_http_request(sockfd, node->request);
14 if (throwable->is_an_error(throwable)) {
15   // Handling error
16   ...
17 }
18
19 // Receives header into http_response
20 throwable = receive_http_response_header(sockfd, node->response);
21 if (throwable->is_an_error(throwable)) {
22   // Handling error
23   ...
24 }
25
26 // Logging - response
27 HTTPResponsePtr response = node->response;
28
29 // Sends signal to condition
```

```
30  if (pthread_cond_signal(&node->condition) != 0) {
31    // Handling error
32    ...
33  }
34
35  // Releases mutex
36  if (pthread_mutex_unlock(&node->mutex) != 0) {
37    // Handling error
38    ...
39  }
40
41  // Receives the response in chunks
42  throwable = receive_http_chunks(sockfd, node->response, node->chunk);
43  if (throwable->is_an_error(throwable)) {
44    // Handling error
45    ...
46  }
```

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

2.8.5 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 costantemente 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*.

```
1  // Gets queue
2  RequestQueuePtr queue = worker->requests_queue;
3
4  while(TRUE) {
5    // Gets node
6    RequestNodePtr node = queue->get_front(queue);
7
8    if (node != NULL) {
9      // Gets mutex
10     if (pthread_mutex_lock(&node->mutex) != 0) {
11      // Handling error
```

```
12        ...
13      }
14
15      while (node->response->response->header == NULL) {
16        if (pthread_cond_wait(&node->condition, &node->mutex) != 0) {
17          // Handling error
18          ...
19        }
20      }
21
22      // Sends the response header
23      ThrowablePtr throwable = send_http_response_header(worker->sockfd,
            node->response);
24      if (throwable->is_an_error(throwable)) {
25          // Handling error
26          ...
27      }
28
29      // Releases mutex
30      if (pthread_mutex_unlock(&node->mutex) != 0) {
31          // Handling error
32          ...
33      }
34
35      // Gets the chunk
36      ChunkPtr chunk = node->chunk;
37
38      // Sends the response chunks
39      throwable = send_http_chunks(worker->sockfd, chunk, node->response
            ->response->req_content_len);
40      if (throwable->is_an_error(throwable)) {
41          // Handling error
42          ...
43      }
44
45      // Dequeues the request and it destroys that
46      node = queue->dequeue(queue);
47      node->destroy(node);
48    }
49  }
```

2.8.6 Thread di watchdog

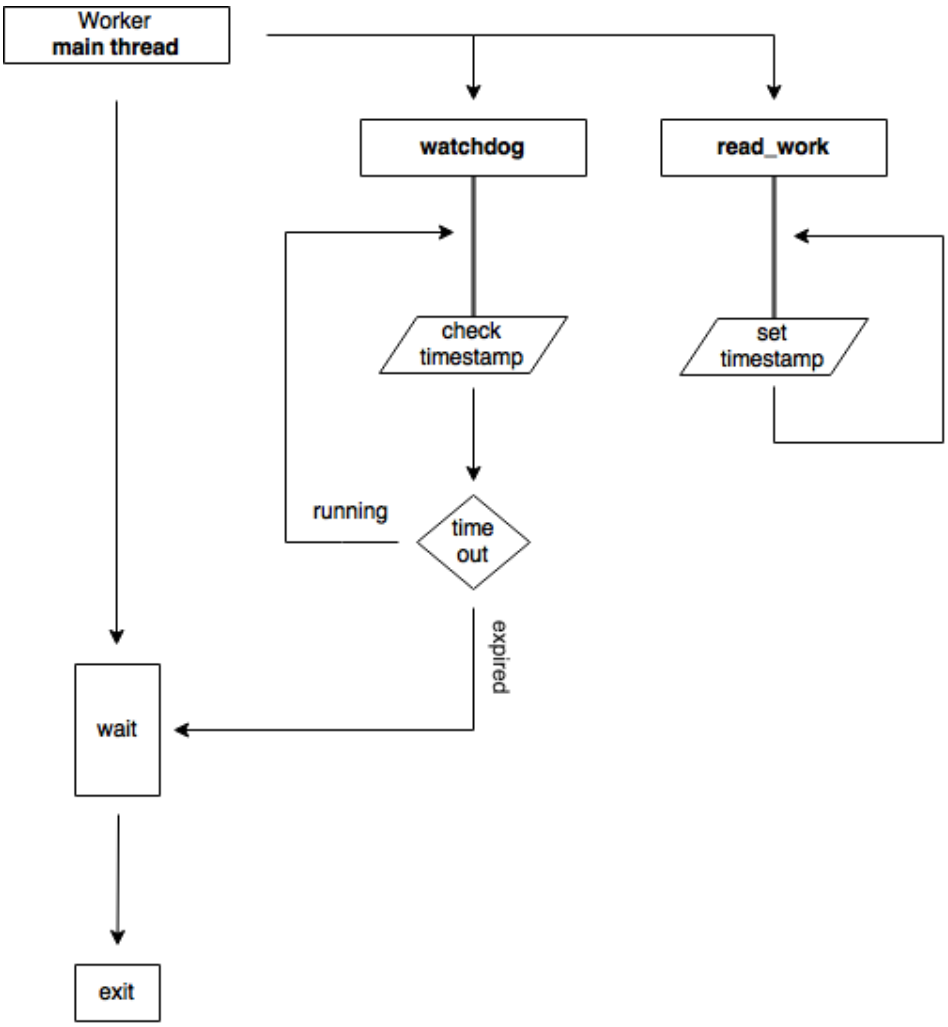

Figura 4: Schema della procedura di controllo sul tempo di esecuzione

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 vada a creare un collo di bottiglia che via via porti al collasso del programma. Per far questo esso viene eseguito in modalità *detached* dal thread principale del worker restando però legato ad alcune variabili del processo:

- *worker_await_cond*: condizione thread POSIX su cui è in attesa il

thread principale del worker;

- *worker_await_flag*: variabile legata alla condition di cui sopra, che viene utilizzata per ciclare in attesa dello sblocco della condizione;
- *timestamp_worker*: valore temporale settato dal thread di lettura della richiesta (o risposta) in arrivo per confermare la lettura in corso di un pacchetto.

Come appare da una prima analisi della struct del watchdog.

```
1 typedef struct watchdog_thread {
2   ...
3   pthread_cond_t *worker_await_cond;
4   int *worker_await_flag;
5   time_t killer_time;       // to schedule the watchdog wakeup
6   time_t timeout_worker;    // to abort a thread run
7   time_t timestamp_worker;  // timestamp last worker operation
8 } Watchdog, *WatchdogPtr;
```

Dove il *killer_time* ed il *timeout_worker* sono definiti dal file di configurazione (2.3), 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 figura 4. 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 sui tempi di esecuzione viene iterato 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 il pericoloso stato di attesa che porterebbe ad uno stallo del programma.

3 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 definito in 2.3.2 .

Nella nostra implementazione il 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. Le implementazioni di seguito descritte si trovano nei file *round_robin.c*, *scheduler.c*, *circular.c* e *server_pool.c*.

3.1 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 RR e che contiene i puntatori alle funzioni *wrapper* che garantiscono il suo corretto funzionamento.

```
1 typedef struct round_robin_struct {
2   CircularPtr circular;
3
4   ThrowablePtr (*weight)(CircularPtr circular, Server *servers, int
        server_num);
5   ThrowablePtr (*reset)(RRobinPtr rrobin, ServerPoolPtr pool, int
        server_num);
6   Server *(*get_server)(CircularPtr circular);
7 }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 funzione a cui si farà riferimento è quella per il recupero del server correntemente selezionato.

L'algoritmo funziona utilizzando un *buffer circolare*, come possiamo vedere in figura 5: questo permette di iterare la selezione su una lista di elementi precedentemente caricata.

```
1  typedef struct circular_buffer {
2    Server *buffer;
3    int buffer_position;
4    int buffer_len;
5
6    Server *head;
7    Server *tail;
8
9    pthread_mutex_t mutex;
10
11   ThrowablePtr (*allocate_buffer)(CircularPtr *circular, Server **
         servers, int len);
12   ThrowablePtr (*acquire)(struct circular_buffer *circular);
13   ThrowablePtr (*release)(struct circular_buffer *circular);
14   void (*progress)(struct circular_buffer *circular);
15   void (*destroy_buffer)(struct circular_buffer *circular);
16 } Circular, *CircularPtr;
```

Possiamo osservare che, oltre alle funzioni e le variabili necessarie a garantire l'accesso atomico all'area di memoria che contiene il buffer, condizione necessaria che vedremo nel caso *state-aware* (per evitare la concorrenza con il thread che si occupa dell'update dello stato), sono mantenuti:

- *buffer*: un puntatore all'array di server;

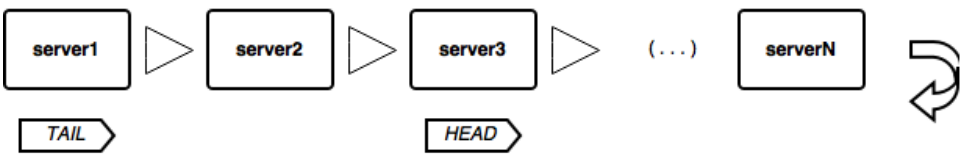

Figura 5: Schema di funzionamento del buffer circolare

- *buffer_position*: la posizione attuale del puntatore di testa;
- *buffer_len*: la lunghezza del buffer, necessaria anche per le operazioni di aggiornamento dei puntatori;
- *head, tail*: 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. È necessario quindi specificare tre passi per il corretto funzionamento.

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:

```
1 /* inside allocate_buffer ... */
2 // allocating the buffer
3 circular->buffer = *servers;
4 circular->buffer_len = len;
5 // setting params
6 circular->head = circular->buffer;
7 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 selezionato 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:

```
1 void progress(CircularPtr circular) {
2   // recomputing tail, head and buffer position
3   circular->tail = circular->head;
4   circular->buffer_position = (circular->buffer_position + 1) %
        circular->buffer_len;
5   circular->head = circular->buffer + circular->buffer_position;
6 }
```

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.

```
1 /* inside get_server ... */
2 // allocating server ready struct
3 ServerPtr server_ready = malloc(sizeof(Server));
4 ...
5 // stepping the circular buffer
6 circular->progress(circular),
7 // retrieving server from tail
8 *server_ready = *(circular->tail);
9 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.

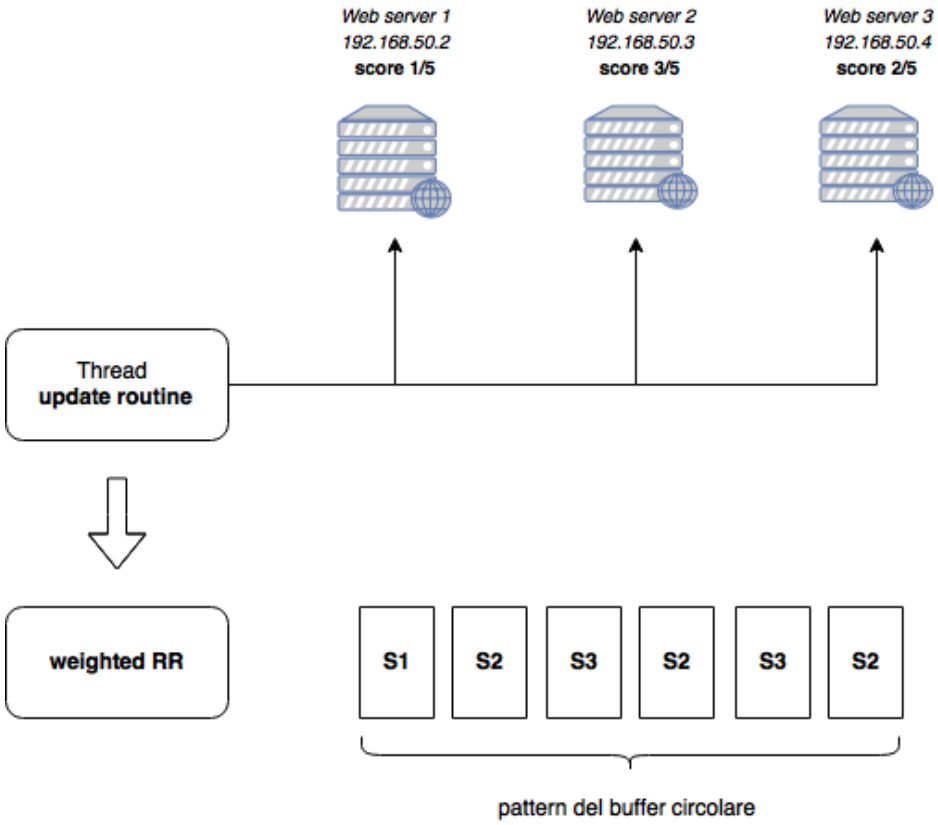

Figura 6: Schema della procedura di aggiornamento dello stato dei server

3.2 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, questo è il modulo *ApacheStatus* di cui si parlerà più avanti in 3.2.1). 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:

```
1 typedef struct server {
2     char *address;
3     char *ip;
4     int  weight;
5     int  status;
6 } Server, *ServerPtr;
```

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 stateless 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 6.

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 2.7). 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 si ottiene è uno *score* che viene settato nel nodo contenuto nella *pool dei server* che viene definito dalla relazione matematica che è così esplicitata:

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

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 arrivo. 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 inattivo (*BROKEN*), fino al prossimo aggiornamento.

RR pesato: dai nodi della pool dei server aggiornati con il loro peso viene costruito, secondo lo schema in figura 6, un pattern dei server, 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. Dunque una macchina comparirà massimo W volte in caso di basso carico di lavoro ed al minimo w volte in condizioni di forte stress. I due parametri sono, in questa implementazione, di default $w = 1$ e $W = 5$, stabiliti in seguito alla fase di test di cui parleremo dopo. Alla prima iterazione tutte le macchine sono settate con peso minimo (pari a w).

In conclusione, con questa opzione abilitata, si ha la possibilità di ridistribuire equamente il lavoro, permettendo a Heimdall di adattare la distribuzione del carico a seconda dello stato delle macchine, 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. 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 sviluppatori, e possa essere oggetto di un *tuning* più approfondito, è stata intrapresa perseguendo il principio per cui *simplicity favours regularity*.

3.2.1 Modulo Apache Status

Il modulo Apache Status (mod_status)[7] è un modulo che fornisce informazioni sull'attività e le prestazioni del server su 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 web server. 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 readable*, versione utilizzata in questa applicazione).
I dettagli che fornisce sono:

- Il numero di worker che servono le richieste;
- Il numero totale di worker in pausa;

- Lo stato di ciascun worker: il numero di richieste che il worker ha eseguito e il numero totale di bytes serviti;
- 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.

Quindi, tramite questo modulo, Heimdall è in grado di verificare lo stato di ciascuna macchina, senza la necessità di installare alcun componente aggiuntivo o di valutare empiricamente il carico del server tramite, per esempio, il tempo di risposta di quest'ultimo. Infatti Heimdall è in grado di recuperare direttamente informazioni riguardanti l'effettiva occupazione dei Worker in esecuzione sulla macchina remota.

3.2.2 Performance della politica state aware

È 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.

```
1  [Wed Feb 10 11:31:44 2016] - 192.168.1.3 to: 192.168.1.4 - worker: 3032 - GET / HTTP/1.1
2  [Wed Feb 10 11:31:44 2016] - 192.168.1.3 to: 192.168.1.5 - worker: 3031 - GET / HTTP/1.1
3  [Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.6 - worker: 3040 - GET / HTTP/1.1
4  [Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.5 - worker: 3039 - GET / HTTP/1.1
5  [Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.6 - worker: 3038 - GET / HTTP/1.1
6  [Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.5 - worker: 3037 - GET / HTTP/1.1
7  [Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.6 - worker: 3036 - GET / HTTP/1.1
8  [Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.5 - worker: 3035 - GET / HTTP/1.1
9  [Wed Feb 10 11:31:45 2016] - 192.168.1.3 to: 192.168.1.6 - worker: 3034 - GET / HTTP/1.1
```

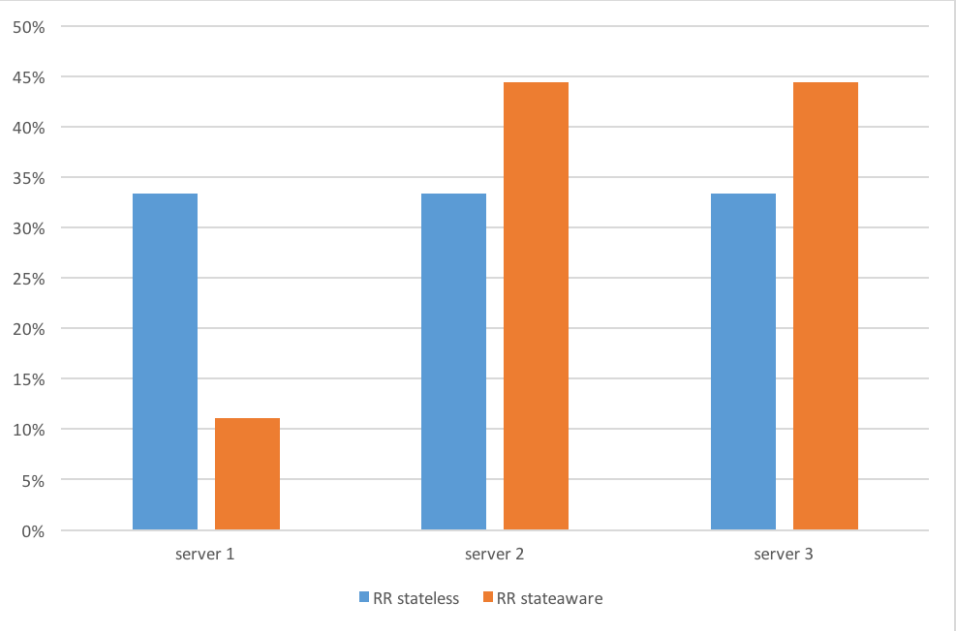

Figura 7: Ripartizione del carico di lavoro, *confronto fra politica state-less e state-aware*

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.
È possibile accertarlo sia dall'estratto del file di log riportato che dalla figura 7 che ne quantifica i risultati.

4 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 14.04 - 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. Considereremo 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* (E.3) di inviare 10 richieste per ogni connessione instaurata. Quindi ogni worker avrà una coda di 10 richieste da smaltire.

4.1 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 venivano condotti i *benchmark* sia via browser che sia attraverso il tool da linea di comando *httperf*.

Mantenendo *frequenza di richieste per secondo* costante, osserviamo due situazioni diverse, riscontrate con l'aumentare di tale frequenza. In figura 8 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 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 9. 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. Andremo ora a vedere più nel dettaglio, nello *stress test*, questa peculiarità del nostro programma.

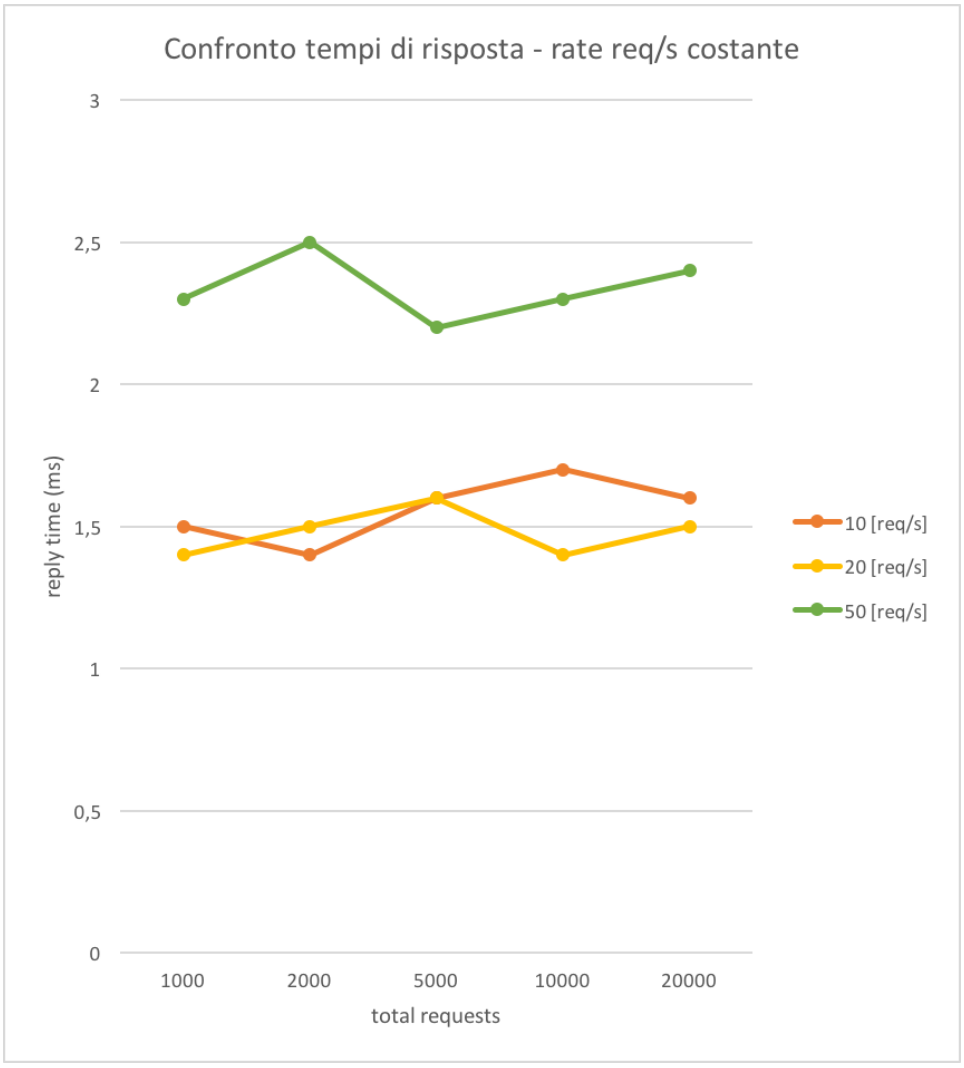

Figura 8: Confronto fra i tempi di risposta con diversi rate all'aumentare delle richieste. Si osserva lo stabilizzarsi intorno ad un livello medio.

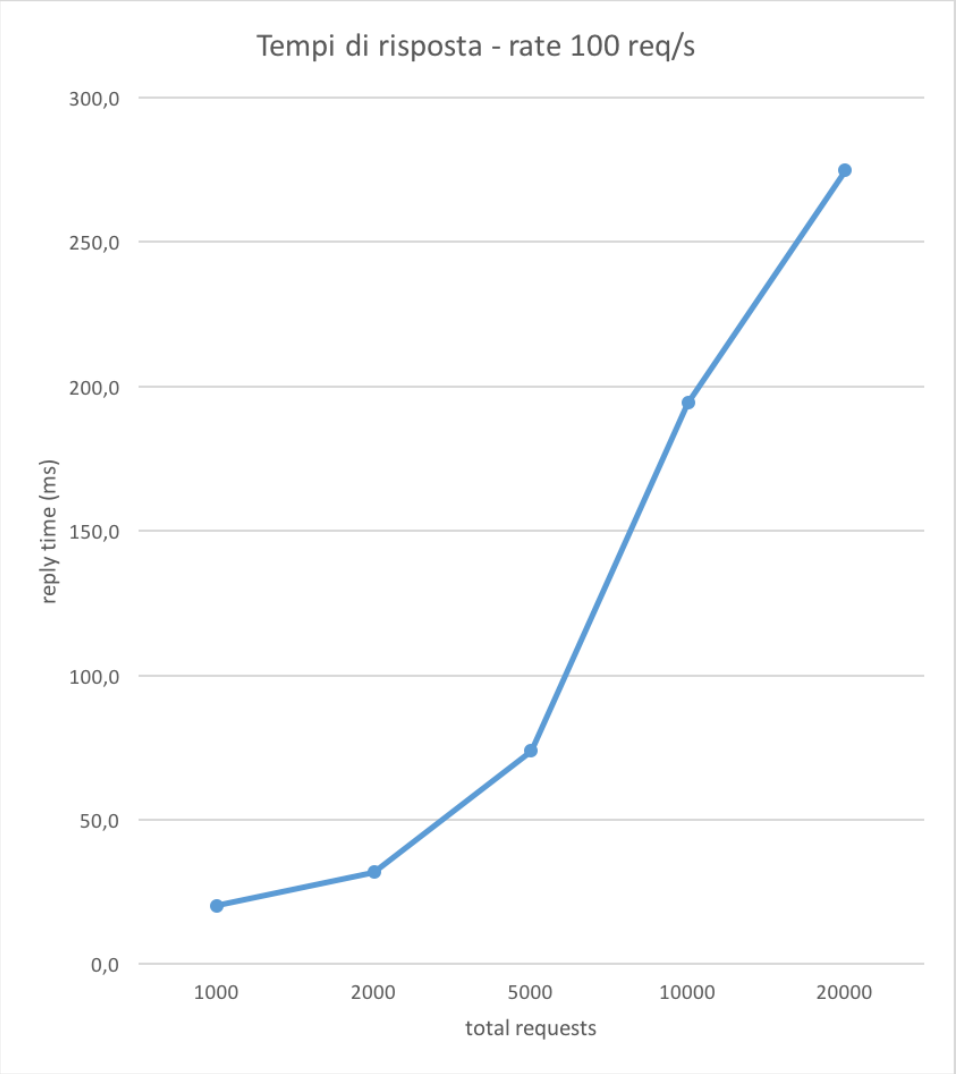

Figura 9: Crescita dei tempi di risposta all'aumentare del numero di richiesta, si osserva come il lavoro dei worker non è sufficiente è compensare la frequenza delle richieste.

4.2 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 e 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 10, è il crescere dei tempi di risposta (grafico arancione) 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. È 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.

Sempre dallo stesso grafico, analogamente, 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 risposta (grafico blu) 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.

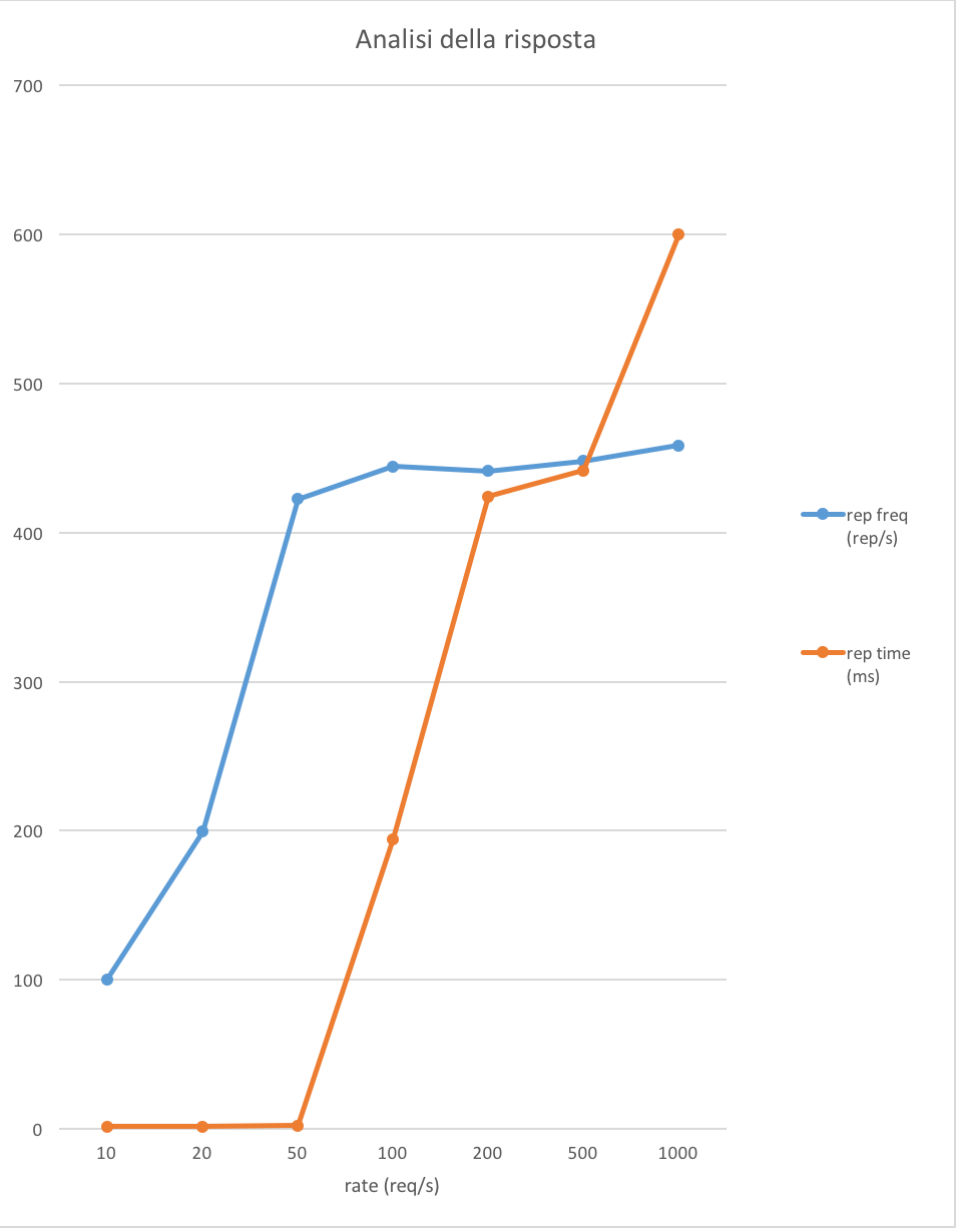

Figura 10: Analisi delle performance confrontando i tempi di risposta e la frequenza con cui la risposta viene fornita.

4.3 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 con il grafico di figura 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.

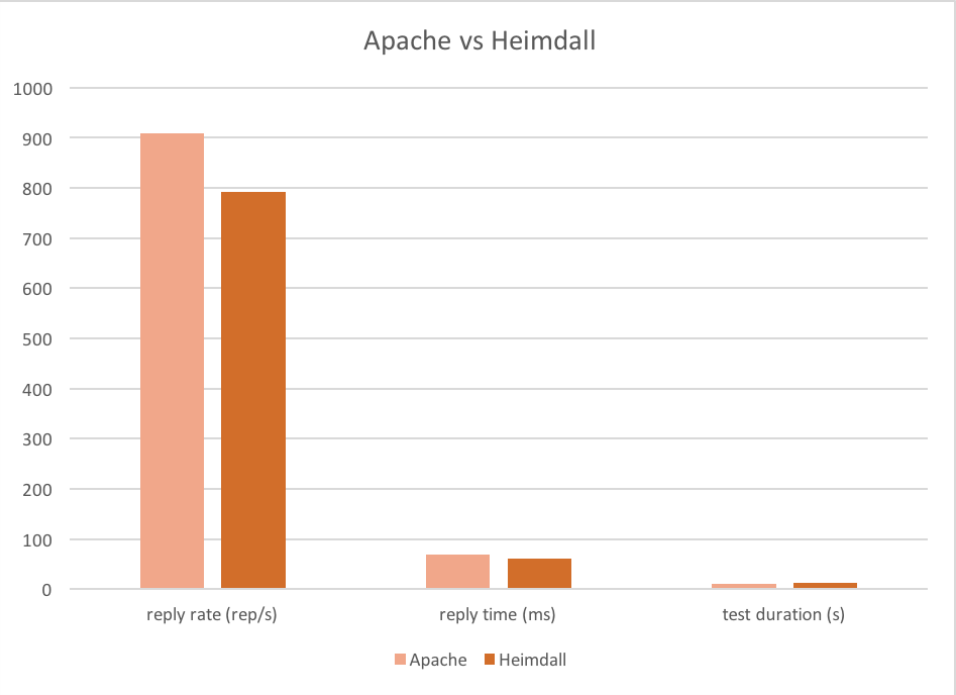

Figura 11: Confronto dei tempi di risposta di un web server Apache con il web switch Heimdall (10000 richieste per una frequenza di 100 req/s).

4.4 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 ulte-

riormente 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.

È 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.

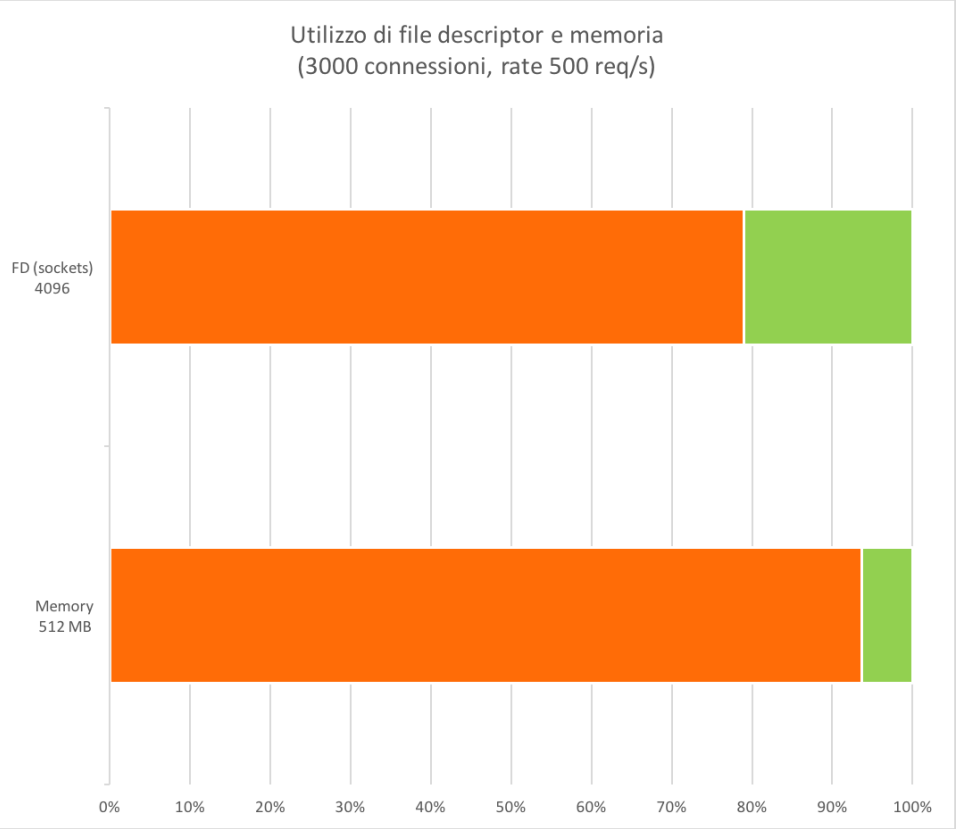

Figura 12: Visualizzazione dell'occupazione di memoria e fd durante uno stress test.

4.5 Conclusioni

Heimdall è un Web Switch di livello 7 concorrente, state-less, state-aware e dalle buone performance. È altamente configurabile e permette di essere adattato ad un gran numero di scenari applicativi. Dai vari test eseguiti è risultato che le performance di Heimdall sono sufficienti a garantire trasparenza all'utente nella comunicazione verso le macchine del cluster. Siamo soddisfatti dell'attuale implementazione, ma nella sezione future implementazioni (5) abbiamo descritto alcune modifiche che siamo fiduciosi possano migliorare ancora di più le performance di Heimdall.

5 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.

5.1 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.

5.2 Chiamate non bloccanti

L'attuale implementazione si basa sulle chiamate di tipo bloccanti delle API di Berkley. Una ulteriore proposta potrebbe essere quella di ripensare l'intero sistema sfruttando la modalità non bloccante nella gestione dell'I/O.
Attualmente ogni Worker gestisce una e una sola connessione. Tale sistema limita Heimdall poiché gestisce un numero definito di connessioni, pari al numero di Worker massimo configurato. L'utilizzo delle chiamate non bloccanti potrebbe permetterci quindi di aumentare il throughoutput del sistema permettendo ai Worker di servire più di una connessione contemporaneamente.

5.3 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 web switch, ad esempio l'isolamento delle 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 portati a dover gestire alcuni piccoli problemi: 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 è noto che la creazione e la schedulazione di un thread è più veloce rispetto a quella di un processo. Ulteriore vantaggio è dato dal fatto che i thread condividono lo stesso spazio di memoria e questo evita i problemi di memoria condivisa e invio dei file descriptor.

Annotazioni

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

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

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

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

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

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

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

[8] CMake http://cmake.org

[9] Vagrant by HashiCorp http://www.vagrantup.com

[10] Oracle VM VirtualBox http://www.virtualbox.com

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

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

[13] Valgrind, http://valgrind.org/

[14] Httperf by HP Labs http://www.labs.hpe.com/research/linux/httperf

[15] Postman Chrome Extension http://www.getpostman.com

[16] Apple Safari http://www.apple.com/safari

[17] Google Chrome Web Browser http://www.google.com/chrome

[18] Mozilla Firefox Web Browser http://www.mozilla.org/it/firefox

A Manuale per l'uso

L'installazione e l'utilizzo del web switch Heimdall sono progettati per essere il più immediati possibile. Vediamo nel dettaglio quale è la procedura per installare e compilare per la prima volta l'intero progetto.

Nota: nel caso in cui si voglia procedere direttamente all'esecuzione del programma, il cui file binario è allegato a questa relazione, è necessario procedere dalla A.4 per modificare alcuni file di configurazione e creare alcune cartelle necessarie all'esecuzione.

A.1 Download

Il progetto può essere recuperato dalla repository pubblica ospitata su GitHub all'indirizzo:
`https://github.com/HeimdallProject`
La repository scaricata conterrà:

- codice sorgente;
- script di installazione e compilazione;
- file necessari alla compilazione del progetto.

A.2 Dipendenze

È incluso il *binary* del progetto all'interno del file allegato alla relazione, tuttavia in caso di download del progetto dalla repository su GitHub, per procedere alla compilazione che vedremo al punto successivo, è necessario installare *CMake*.
CMake[8] è una famiglia di tool **open-source** e **cross-platform**, disegnati per poter controllare il processo di compilazione attraverso una piattaforma indipendente sia dall'architettura che dall'ambiente di compilazione prescelto sulla propria macchina. Tale obiettivo viene raggiunto tramite un file di configurazione che possiamo trovare nella cartella del progetto sotto il nome *CMakeLists.txt*. Nel nostro caso è stato utilizzato per poter far fronte, utilizzando un tool professionale, al grande numero di dipendenze e poter gestire

sia la ricerca di **package e librerie** particolari sia l'utilizzo di particolari **direttive di precompilazione**, il tutto mantenendo un file di configurazione compatto e facilmente leggibile.

La versione di minima di CMake necessaria è la 3.2 e può essere scaricata sia dal sito sopra indicato che, procedendo da linea di comando. Vediamo brevemente come fare, tenendo conto che, per l'installazione di software, è necessario possedere i privilegi dell'utente *root*.

```
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository ppa:george-edison55/cmake-3.x
$ sudo apt-get update
$ sudo apt-get install cmake
```

Altrimenti si può procedere alla compilazione di cmake stesso. Anche in questo caso è necessario disporre dei privilegi di *root* (cambiare il numero della versione per scaricare la versione desiderata).

```
$ sudo apt-get install build-essential
$ wget http://www.cmake.org/files/v3.2/cmake-3.2.2.tar.gz
$ tar xf cmake-3.2.2.tar.gz
$ cd cmake-3.2.2
$ ./configure
$ make
$ sudo make install
```

A.3 Compilazione

Una volta installato il programma utilizzato per la compilazione del progetto, si può procedere alla fase finale di preparazione per l'esecuzione del programma eseguendo lo script **install.sh** presente all'interno della cartella di installazione.

```
$ cd <installation folder>
$ sh install.sh
```

A questo punto si può procedere direttamente al passo di esecuzione illustrato in A.5.

In alternativa è possibile procedere alla sola compilazione del progetto,

con il suggerimento di creare una cartella separata in cui salvare i file relativi alla compilazione.

```
$ cd <installation folder>
$ mkdir build
$ cd build
$ cmake ..
$ make WebSwitch
```

A.4 Modifica file di configurazione

Nel caso si disponesse già di una copia dei file binary o si avesse compilato il programma indipendentemente dallo script illustrato al punto precedente, è possibile procedere alla configurazione delle cartelle necessarie all'esecuzione del programma eseguendo lo script **config.sh** fornito.

```
$ cd <installation folder>
$ sh config.sh
```

Per eseguire un test bisogna modificare la configurazione del cluster, specificando almeno una macchina all'interno del file *code/config/server_config.conf*, scrivendo indirizzo ip e nome del server.

Per ulteriori necessità e in particolare per il tuning del web switch, si può fare riferimento ai file presenti nell cartella *<installation folder>/code/config* e al paragrafo sui file di configurazione (2.3) .

A.5 Esecuzione

A questo punto si può supporre creata una cartella *build* dov'è presente il file binario del progetto. Supponendo di aver configurato il nostro web switch adeguatamente, possiamo procedere ad avviare l'applicazione da linea di comando.

```
$ cd build
$ ./WebSwitch
```

Nota che il web switch è di default configurato per essere in ascolto sulla porta 8080. Per metterlo in ascolto sulla porta 80 è necessario possedere i privilegi di root.

Buon lavoro!

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[9] 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

Nella fase di sviluppo ha avuto particolare importanza la possibilità di poter verificare sia la correttezza dell'architettura che la sua funzionalità. In particolare quest'ultima ha richiesto la simulazione di un effettivo scenario di utilizzo del web switch, riprodotto in scala e senza pretese di un'infrastruttura distribuita geograficamente. Da qui nasce la necessità di utilizzare un cluster virtuale.
Per renderlo operativo si è deciso di utilizzare tre macchine virtuali operanti da **VirtualBox**[10], programma che ospita anche Vagrant. A questo punto le quattro macchine risultati erano operative in un'unica **rete locale virtuale**.

Nel dettaglio è stata utilizzata l'opzione di **internal networking** disponibile nel pannello proposto da VirtualBox per la gestione delle macchine virtuali. La rete risultante è totalmente isolata, con indirizzi IP assegnati staticamente, definita comune per ognuna delle macchine, che permette di agire con sicurezza nella fase di test.

Nota: per rispettare le assunzioni progettuali, su ognuna delle macchina era installato Ubuntu Server 14.04, con web server Apache e con il modulo per la verifica dello status della macchina abilitato.

D Tool per i debug

D.1 GDB

GNU debugger (chiamato semplicemente GDB)[11] 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[12] è un visualizzatore interattivo di processi per i sistemi Unix. Pur non essendo un tool di debug, ha contribuito al "debug" di Heimdall, permettendo di controllare in tempo reale lo stato dei processi, dei threads e della memoria.

D.3 Valgrind

Valgrind[13] è un potente strumento di debug, che 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.

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. È possibile utilizzare un client telnet da qualsiasi sistema operativo, inoltre è già presente nella maggior parte delle distribuzioni Linux.

```
1 $ telnet localhost 8080
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 GET / HTTP/1.1
6 Host: 127.0.0.1
```

E.2 PostMan

Postman[15] è una applicazione presente nel *Chrome Web Store* che gira come plug-in del famoso browser Google Chrome. È 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

È un tool per misurare le performance di un server web, svilupatto dai laboratori della HP[14]. È 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 richieste, il rate con cui effettuare tali connessioni, la risorsa da richiedere, nonchè la possibilità di aprire un numero di porte TCP elevato per verificare la risposta di un certo carico di lavoro durante una sessione.

```
1 $ httperf --server=localhost --uri=/index.html --port=8080
2           --num-conns=3000 --num-calls=2 --rate=500 --hog
```

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 cluster 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**[18] ma che è possibile abilitare, oppure l'apertura di diverse connessioni con il server entro le quali vengono effettuate molteplici richieste come *Chrome*[17] o *Safari*[16] o, generalmente, un qualsiasi browser moderno.

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

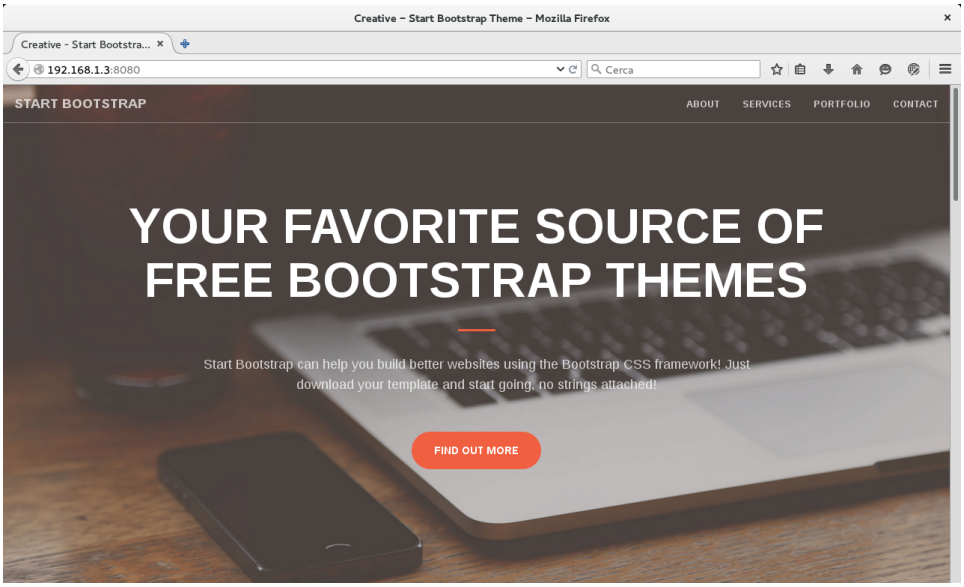

Figura 13: Schermata con la richiesta, soddisfatta con successo, del sito ospitato durante lo sviluppo su una delle macchine del cluster (browser utilizzato: Mozilla Firefox)

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

```
1  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
2  [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
3  [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
4  [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
5  [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
6  [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
7  [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
8  [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
9  [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
10 [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
11 [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
12 [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
13 [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
14 [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
```

Figura 14: Il team di sviluppo del Progetto Heimdall durante i test del web switch a pieno regime (*febbraio 2016, Laboratori di Tor Vergata*)