Project Heimdall

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

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

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

Roma, 8 febbraio 2016

Indice

1 Example section

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

Heimdall, custode del Bifröst

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

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

2 Introduzione

2.1 Perchè Heimdall?

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

2.2 Web switch di livello 7

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

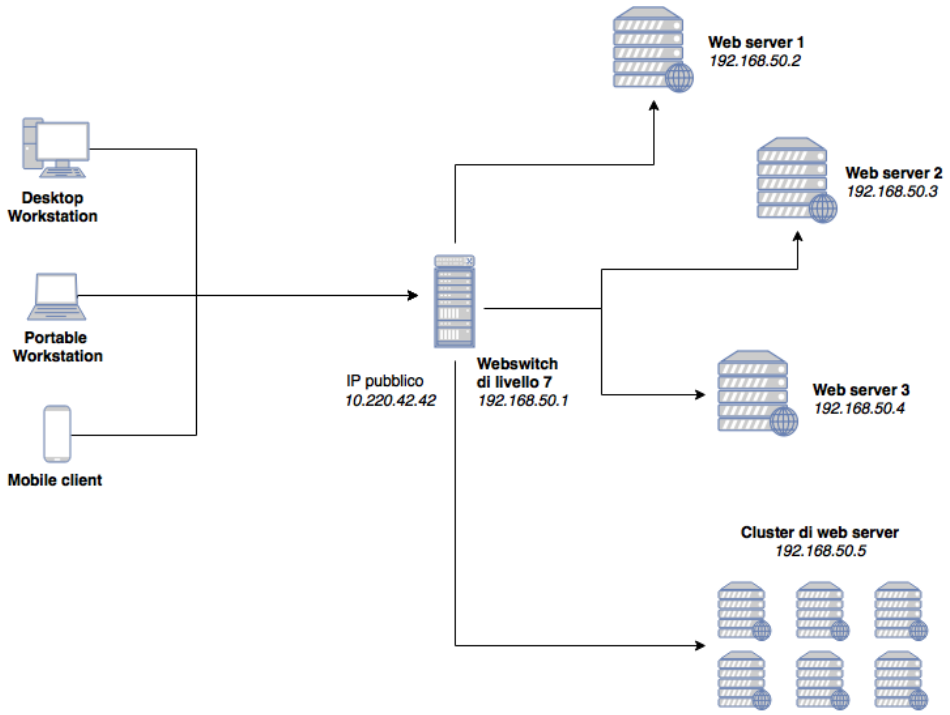

Figura 2: Esempio di uno *switch di livello 7 (OSI)*

di inoltro generata con una determinata **politica di scheduling** e quindi gestire l'inoltro della richiesta ed il reinoltro della risposta del webserver. tutto questo in maniera totalmente trasparente al client, qualsiasi sia la macchina che ha effettivamente risposto, che sia un web server oppure un cluster di macchine associate ad un ulteriore switch.

2.3 Assunzioni progettuali sul cluster

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

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

3 Architettura

3.1 Server in ascolto

3.1.1 File di configurazione

3.1.2 Logging

3.1.3 Gestione degli errori

3.2 Pool manager

3.3 Scheduler

Lo scheduler è un componente fondamentale di un sistema informatica: si occupa di stabilire un ordinamento temporale l'esecuzione di un set di richieste di accesso ad una risorsa. Nel caso di un web switch di livello 7, lo scheduler va a garantire che ognuna delle richieste in arrivo possa essere inoltrata immediatamente alla prima macchina disponibile, secondo una politica di scheduling che sia *state-less*, quindi che non consideri l'attuale carico di lavoro delle macchine del cluster, oppure *state-aware*, che monitori costantemente tale carico e modifichi di conseguenza l'assegnazione delle richieste (verrà spiegato nel dettaglio come lavorano e quando sono disponibili tali politiche in *5*).

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

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

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

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

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

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

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

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

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

Viene sempre selezionato un server che sia disponibile, quindi viene sempre effettuato un controllo sullo *status* dello stesso server, nel caso in cui sia abilitato il controllo sullo stato della macchina: l'unico caso in cui questi risulta *BROKEN* e non *READY* è nella circostanza in cui ogni server del cluster risulta non disponibile per cui il worker (che analizzeremo in *3.4*) non avvierà nessuna connessione di inoltro della richiesta.

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

3.4 Worker

3.4.1 Gestione delle richieste

Coda delle richieste

Chunk di dati

3.4.2 Gestione delle connessioni

Connessione

Richieste HTTP

Risposte HTTP

3.4.3 Thread di lettura

3.4.4 Thread di scrittura

3.4.5 Thread di richiesta

3.4.6 Thread di watchdog

4 Ulteriori proposte

5 Politiche di scheduling

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

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

5.1 State-less: implementazione con Round-Robin

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

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

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

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

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

L'algoritmo funziona utilizzando un **buffer circolare** come possiamo vedere in *figura 3*: questo permette di iterare la selezione su una lista di elementi precedentemente caricata. Possiamo osservare che, oltre alle funzioni e le variabili necessarie a garantire l'accesso atomico all'area di memoria che contiene il buffer, necessario come vedremo nel caso *state-aware* per evitare la concorrenza con il thread che si occupa dell'update dello stato, sono mantenuti:

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

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

```
/*
 * ------------------------------------------------------------------------
```

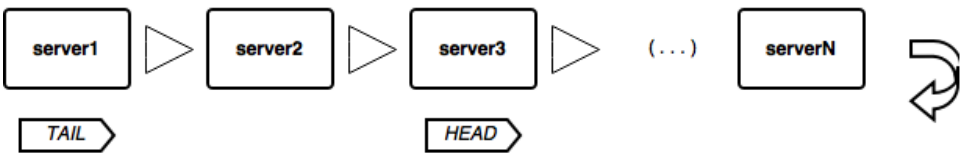

Figura 3: Schema di funzionamento del buffer circolare

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

    Server      *head;
    Server      *tail;

    pthread_mutex_t mutex;

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

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

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

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

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

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

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

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

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

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

    /* ... */

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

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

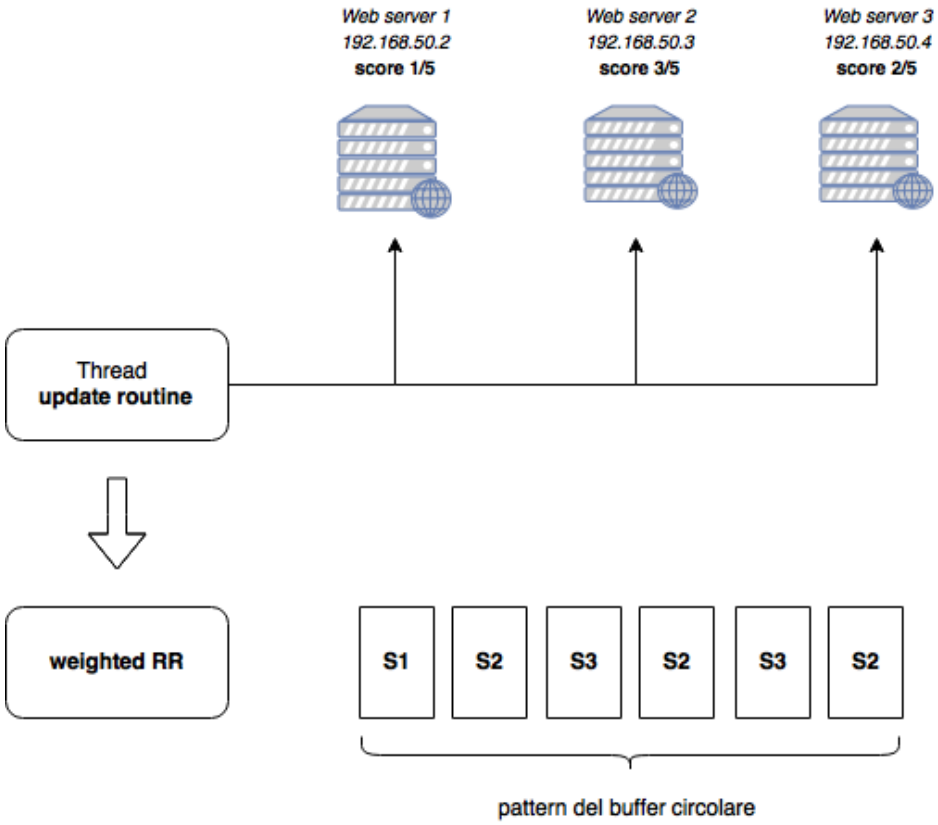

Figura 4: Schema della procedura di aggiornamento dello stato dei server

5.2 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, questi è i modulo *ApacheStatus* di cui si parlerà più avanti in *5.2.1*). 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

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

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

Il secondo caso è proprio quello qui descritto e corrisponde a lavorare utilizzando, oltre al thread principale che si occupa di accettare le connessioni in arrivo, un **thread predisposto alla sola verifica dello stato dei server**. Tale thread viene istanziato nel momento in cui viene inizializzato lo scheduler e vengono allocate le strutture dati alla base di RR.
Il lavoro di tale thread, che ora vedremo nel dettaglio, è quello deducibile da *figura 4*.

Routine di score all'interno di questa routine, che viene eseguita da un thread distaccato e che viene eseguita una volta ogni *UP_TIME* secondi, tempo di update in secondi definito dall'utente nei file di configurazione, viene richiamata più volte la funzione che si occupa di recuperare e parsare l'interrogazione del modulo *ApacheStatus* e recupare da questa i **worker in *idle state* ed i worker in *busy state***. A questo punto si va a modificare il nodo della pool dei server precedentemente allocata (di cui si è già parlato in *3.3*). Viene quindi eseguita la sequente routine.

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

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

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

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

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

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

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

RR pesato dai nodi della pool dei server aggiornati con il loro peso viene costruito, secondo lo schema in *4*, un pattern dei server secondo il loro peso, di modo da distribuire il carico secondo sempre un algoritmo RR, ma in cui per ogni sequenza il server viene selezionato un numero di volte pari al suo peso: comparirà massimo W in caso di basso carico di lavoro ed al minimo w volte in condizioni di forte stress. I due parametri sono, in questa implementazione, macro che possono essere modificate a seconda dei limiti

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

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

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

5.2.1 Modulo ApacheStatus

6 Logging

7 Performance

7.1 Test di carico

7.2 Comparazione con Apache

8 Future implementazioni

8.1 Analisi della richiesta

8.2 Webserver performante

Annotazioni

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

A Manuale per l'uso

B Vagrant

C Cluster virtuale

D Tool per i debug

D.1 GDB

D.2 Valgrind

E Tool per i test

E.1 PostMan

E.2 Telnet

E.3 HttPerf

E.4 Browser