PARIWEB: IMPLEMENTAZIONE WEB SERVER DISTRIBUITO

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Dott. Vincenzo Cappelleri
LAUREANDO: Andrea Tomassetti

A.A. 2012-2013

UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA

PariWeb: Implementazione web server distribuito

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Dott. Vincenzo Cappelleri

LAUREANDO: *Andrea Tomassetti*

A.A. 2012-2013

"Se vuoi fare un passo avanti, devi perdere l'equilibrio per un attimo."
Massimo Gramellini

"A tutti coloro che mi hanno sempre sostenuto."

Indice

Abstract **1**

Introduzione **2**

1 PariPari **5**

1.1 Il progetto e la rete 5

1.2 La struttura 6

1.3 I plugin 7

2 Internet e il WorldWideWeb **11**

2.1 Storia di internet 11

2.2 HTTP 12

2.3 Messaggi HTTP 15

3 PariWeb: il web hosting secondo PariPari **21**

3.1 Configurazione e personalizzazione 21

3.2 I comandi 22

3.3 Modalità locale 23

3.4 Modalità distribuita 24

4 Implementazione **27**

4.1 Lo *Unit Testing* 27

4.2 Parsing delle richieste HTTP 29

4.3 *Code analysis* 34

5 Conclusioni **39**

Bibliografia **40**

Sommario

Un sistema distribuito è una rete, una collezione di computer indipendenti tra di loro, che appare agli utenti come un'unica entità, un sistema singolo e coerente. Leslie Lamport, informatico statunitense nato agli inizi degli anni '40 e ideatore del sistema di scrittura LaTex, esprime tale caratteristica attraverso una nota frase

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable
>
> (L. Lamport)

è chiaro, quindi, come l'utilizzatore di un tale sistema non si renda conto della complessità e della struttura della rete che lo compongono. In un sistema distribuito non esiste, e non viene quasi mai definito, un sistema di *clock* globale. Ciò significa che le diverse entità che costituiscono tale sistema non sono sincronizzate tra loro mediante un'unica unità centrale di riferimento, ma la sincronizzazione avviene generalmente attraverso lo scambio di messaggi.

I sistemi P2P(Peer-to-Peer), che costituiscono un caso particolare di sistema distribuito, hanno riscontrato, soprattutto in questi ultimi anni, una sempre maggiore attenzione da parte di aziende e consumatori. Ci si sta sempre di più spostando da una concezione dei sistemi *centralizzata* ad una *distribuita*. I sistemi distribuiti, infatti, presentano evidenti e consistenti benefici per quanto riguarda scalabilità, robustezza e tolleranza ai guasti rispetto ai sistemi centralizzati. Mentre questi ultimi basano il loro funzionamento su un'unica unità centrale, i sistemi distribuiti continuano a funzionare anche se un loro componente dovesse fallire improvvisamente.

PariWeb, plugin del progetto PariPari, si pone l'obiettivo di offrire un servizio di web server distribuito garantendo un elevato grado di efficienza, sicurezza, affidabilità e scalabilità.

In questo elaborato illustrerò il lavoro da me svolto all'interno del gruppo PariWeb, in particolare presenterò le modifiche apportate alle funzioni e alla logica di *parsing* delle richieste HTTP.

Introduzione

PariPari è un progetto che si pone come obiettivo principale quello di realizzare una rete peer to peer completamente serverless, in grado di fornire all'interno di un'unica piattaforma tutti quei servizi attualmente disponibili tramite Internet. [1] Affidabilità e scalabilità sono gli obiettivi principali del progetto.

L'obiettivo principale di PariPari è, quindi, quello di realizzare una singola piattaforma in grado di fornire all'utente finale buona parte dei servizi fruibili attraverso Internet in un'unica, comoda soluzione. Attraverso un unico strumento, PariPari, l'utente potrà avere a disposizione diverse applicazioni e *tool*, facilmente installabili e aggiornabili attraverso un'unica interfaccia, ovviando così all'onerosa e dispendiosa ricerca, installazione e configurazione di strumenti, spesso incompatibili tra loro, che svolgono una sola funzione. PariPari è quindi un vero e proprio ambiente di lavoro che offre diversi e diversificati strumenti tutti pienamente compatibili tra di loro.

In questa configurazione, all'interno di questo progetto, nasce PariWeb: un web server, ovvero un'applicazione che permette di rendere fruibile, attraverso la rete internet, alcune risorse che siano esse immagini, testo, video o altri tipi di documenti. PariWeb è un modulo di PariPari e, in quanto modulo, l'utente può decidere a piacimento se installarlo o meno e ne ha il pieno controllo in fase di configurazione. Inoltre, facendo parte di PariPari, il web server ha accesso alla rete P2P[1] creata da tale software: qui risiede la forza e la caratteristica di PariWeb. Esso infatti può funzionare sia in modalità locale che in modalità distribuita.

Aspetti fondamentali di questa tesi saranno, perciò, la ristrutturazione, rivisitazione e riorganizzazione del codice esistente, in modo da garantire le migliori prestazioni e la migliore scalabilità. Per raggiungere tali obiettivi sono state

[1] Peer to peer

riscritte alcune classi, rivedendone la logica e la struttura.

Dopo questa breve introduzione di carattere generale, nei prossimi capitoli verrà presentato il contesto del lavoro oggetto di questa tesi, in particolare verrà analizzato il progetto PariPari ed il lavoro svolto per renderlo il più performante possibile.

Capitolo 1

PariPari

Figura 1.1: Il logo di PariPari

1.1 Il progetto e la rete

PariPari è un progetto che si sta sviluppando da alcuni anni all'interno dell'università di Padova. L'obiettivo finale che si pone tale progetto, è di creare una rete serverless (attualmente basata su una variante di Kademlia), che sia in grado di garantire l'anonimato dei suoi nodi, che implementi un sistema di crediti più intelligente di reti come ED2K e che sia multifunzionale fornendo i più comuni servizi disponibili su Internet (IRC[1], IM[2], VoIP[3], DBMS [4], Web Server e DNS [5])[1]. Essendo una variante di Kademlia, un protocollo di rete peer-to-peer di ta-

[1]Internet Relay Chat
[2]Instant Messaging
[3]Voice Over Internet Protocol
[4]DataBase Management System
[5]Domain Name System

belle di hashing distribuite sui nodi, PariPari sfrutta appieno e sposa la logica e la potenza *serverless* del protocollo Kademlia. Si può dire infatti che PariPari mira ad essere una rete peer-to-peer pura, multifunzionale e caratterizzata dalla totale assenza di servers centrali. All'interno di tale rete, tutte le entità sono uguali tra di loro: non dovrebbe esistere alcun tipo di distinzione tra i client e i server. Ogni nodo della rete, ogni peer, infatti, si comporta sia da client che da server. Essa adotta un insieme complesso di protocolli basato su Distributed Hash Table (DHT) che consente di mappare la rete assegnando ogni risorsa a un particolare peer. Questa soluzione garantisce il reperimento delle risorse all'interno della rete in modo molto efficiente.

Il progetto è stato completamente scritto in Java, un linguaggio di programmazione *object oriented* e multi piattaforma grazie al quale PariPari può essere *compilato* una volta ed eseguito indipendentemente dal sistema operativo. Questa caratteristica è resa possibile dal modo in cui il linguaggio Java viene *compilato*. I compilatori Java, infatti, restituiscono in output un file contenente bytecode indipendente dalla piattaforma sul quale dovrà essere eseguito. Tali file di output, chiamati *class files*, dovranno essere interpretati da una JVM (Java Virtual Machine) che si adopererà per eseguirli. Per questo motivo, il linguaggio Java, è detto anche *semi-interpretato*.

1.2 La struttura

L'intero progetto è stato sviluppato attraverso una struttura modulare. Tale modularità è ottenuta tramite l'utilizzo di plugin, ovvero un programma non autonomo che sviluppa funzionalità utili o aggiuntive per il programma principale con il quale interagisce. Questa architettura a plugin garantisce al progetto multifunzionalità e ne favorisce gli sviluppi futuri aumentandone il grado di espandibilità e personalizzazione. Ogni nodo di PariPari deve contribuire alla realizzazione di tutti i servizi offerti dalla rete, per cui la sua struttura interna riveste un ruolo fondamentale. Ogni client è costituito da i plugin e da un modulo centrale, denominato Core, che funge da intermediario e collante tra i vari componenti. I plugin, che realizzano le singole funzionalità messe a disposizione da PariPari, sfruttano le risorse della macchina su cui è in esecuzione il client e interagiscono tra loro attraverso il Core. Grazie a questa particolare architettura e ad un siste-

ma di API[6], la scrittura di un nuovo plugin risulta semplificata, in quanto non richiede la conoscenza della struttura interna di tutti gli altri moduli ma solo del modo con cui interfacciarsi.

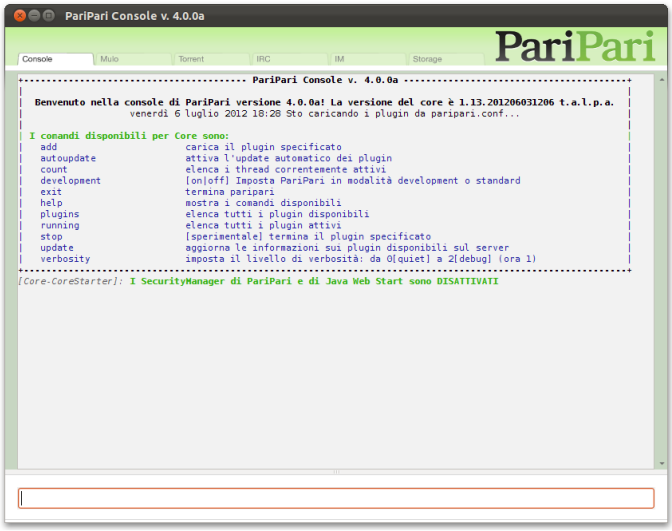


Figura 1.2: La Console di PariPari

L'avvio della applicazione si basa sul caricamento del Core (fig. 1.2), che rappresenta il vero nucleo e programma principale del nostro client. Questo mette a disposizione dell'utente una console testuale implementata all'interno di un'interfaccia grafica intuitiva. Lo stile che contraddistingue visivamente PariPari è minimale, pulito, non invasivo. Ciò è indispensabile per presentare in maniera chiara la moltitudine delle funzionalità offerte.[5] In questo modo, per utilizzare il servizio di nostro interesse, basterà semplicemente caricare il plugin che lo implementa, digitando il comando

```
add <nome plugin>
```

1.3 I plugin

I plugin di PariPari si dividono in due categorie principali (fig. 1.3): quelli della cerchia interna (Connectivity, LocalStorage e DHT) necessari per la gestione delle

[6] Application Programming Interface

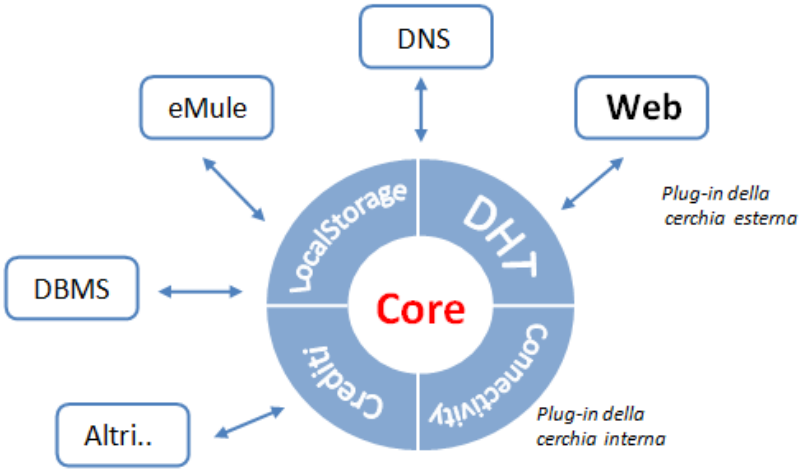


Figura 1.3: La cerchia dei plugin

risorse e per la realizzazione della rete, e quelli della cerchia esterna (BitTorrent, DistributedStorage, DNS, Database, eMule, FileSharing, WebServer, IRC, VOIP) che realizzano le singole funzionalità offerte da PariPari. La distinzione tra le due cerchie presenta una natura di tipo pratico: mentre i plugin facenti parte della cerchia interna vengono distribuiti insieme al Core e sono essenziali per il corretto funzionamento del software, gli altri, appartenenti alla cosiddetta cerchia esterna, sono installabili a discrezione dell'utente che, in base alle sue necessità, potrà decidere di aggiungerli per migliorare la sua esperienza di utilizzo di PariPari. Si è voluto rendere il procedimento di aggiunta dei plugin il più semplice possibile: quando l'utente eseguirà il comando per caricare ed avviare il plugin desiderato, il Core recupererà la versione più recente di tale componente aggiuntivo, nel caso in cui esso non esista nel computer locale, o lo aggiornerà nel caso la versione locale risulti essere precedente rispetto a quella presente sul server di PariPari.

Segue una breve descrizione delle caratteristiche dei plugin fondamentali, o base, appartenenti alla cerchia interna:

Connectivity

ha il compito di centralizzare tutto quello che riguarda la connettività, come ad esempio la creazione dei socket, cioè di quegli oggetti che consentono ad un generico plug-in di interfacciarsi con la rete. C'è l'intenzione di espandere ulteriormente questo modulo in modo tale che possa offrire servizi di

anonimato, multicast, anycast e impedire il passaggio di pacchetti per certe aree geografiche[1].

LocalStorage

è il componente che restituisce accesso in lettura o scrittura e puntatori ai file. Qualsiasi plugin che voglia accedere ad un determinato file, deve inviare una richiesta al Core che la girerà al componente di LocalStorage che verificherà se la richiesta può essere soddisfatta e risponderà di conseguenza. Mantenere questa sorta di centralità nelle richieste di accesso ai file costituisce un grado di sicurezza aggiuntivo: si impedisce così ai plugin di eliminare o modificare file che non gli competono.

DHT

questo plugin è il cuore della rete di PariPari: esso la crea e la mantiene attiva. Sempre attivo, dal caricamento della Console fino alla chiusura di PariPari, esso gestisce l'interconnessione delle diverse istanze di PariPari presenti su altri nodi della rete. Mette inoltre a disposizione diverse API che permettono agli altri plugin di interagire con questa rete.

Capitolo 2

Internet e il WorldWideWeb

Internet è un sistema globale di reti di computer interconnesse tra loro che utilizzano il protocollo Internet, ossia il cosiddetto IP[1], per servire diversi miliardi di utenti in tutto il mondo. È una rete di reti che consiste di milioni di reti private, pubbliche, accademiche, del business e reti di governo. Internet trasporta una vasta gamma di risorse e servizi di informazione, come ad esempio pagine con link ipertestuali, l'infrastruttura di supporto per i servizi di messaggistica e-mail e reti peer-to-peer.

2.1 Storia di internet

Riassumiamo brevemente quali sono state le tappe fondamentali che hanno segnato la nascita di Internet così come la conosciamo noi oggi:

1967 : Lawrence Roberts cominciò a lavorare ad un progetto in grado di interconnettere host *eterogenei* attraverso degli speciali nodi denominati Interface Message Processor (IMP). Il progetto venne finanziato dall' Advanced Research Projects Agency (ARPA), una sezione del Dipartimento della Difesa degli Stati Uniti d'America: siamo alla nascita di ARPAnet

1970-1980 : si assistette alla proliferazione e sviluppo di differenti reti di computer non interconnesse ed eterogenee

1972 : Vint Cerf e Bob Kahn avanzarono l'idea, che risultò vincente, di usare speciali dispositivi creati appositamente per interconnettere reti di natura

[1]Internet Protocol

diversa. Siamo alla nascita dei primi *Gateway*. I due studiosi, durante la realizzazione dei primi dispositivi, si trovarono però di fronte a diversi problemi legati alla diversità delle reti che si voleva interconnettere. I Gateways dovevano infatti risolvere alcune diversità presenti tra le reti come la differenza nella dimensione dei pacchetti, diversi tipi di interfacce e di rate di trasmissione. Essi dovevano essere progettati come dei veri e propri traduttori universali tra reti eterogenee. Si decise perciò di creare un protocollo di trasmissione sicuro ed affidabile, battezzato Transmission Control Protocol, per permettere di gestire le connessioni attraverso questo mix complicato di reti diversificate. L'idea alla base di questo nuovo protocollo era di delegare l'oneroso fardello del controllo e recovery degli errori dagli IMP, all'host di destinazione. Sarà proprio questa la chiave del successo e della scalabilità che permetterà ad Internet di diffondersi in maniera capillare.

1977 : Vennero interconnesse alcune delle reti principali esistenti all'epoca. Come, ad esempio: ARPANET, ALOHANET, Packet Radio network. Esse daranno origine ad una rete più grande denominata ARPA Internet, la prima rete formata dall'interconnessione di reti di natura diversa. Inoltre il protocollo TCP venne diviso in TCP ed IP.

1981 : I sistemi operativi UNIX, i più diffusi all'epoca, implementarono nel loro ambiente i protocolli TCP/IP

1983 : ARPANET (con ormai più di 200 nodi) sostituì il suo protocollo privato, NCP[2], con TCP/IP

1983 : Nacquero i primi sistemi DNS[3] che permettono di tradurre i nomi di una rete, host, nel rispettivo indirizzo IP e viceversa.

2.2 HTTP

Uno dei servizi che viene maggiormente utilizzato dagli utenti che si collegano alla rete Internet è il WorldWideWeb (spesso abbreviato come WWW o più semplicemente come web). Esso è un servizio di Internet che permette di navigare

[2]Network Control Program
[3]Domain Name System

ed usufruire di un insieme vastissimo di contenuti (multimediali e non) collegati tra loro attraverso link[2]. L' *HTTP*[4] è utilizzato come principale sistema per la trasmissione di informazioni attraverso la rete internet e quindi attraverso il *web*. L'HTTP funziona attraverso un meccanismo di richiesta/risposta (client/server): il client esegue una richiesta e il server restituisce la risposta. Nell'uso comune, il client corrisponde al browser ed il server ad un computer remoto che restituisce l'informazione, solitamente una pagina web, richiesta. Il protocollo HTTP utilizza come protocollo di trasmissione il TCP; un server web, infatti, altro non è se non un'applicazione, solitamente installata su macchine con hardware dedicato, che rimane in ascolto sulla porta 80 in attesa di una richiesta da parte del client. L'HTTP si è evoluto negli anni:

1988-89 : Viene realizzata Tim Berners-Lee la prima versione del protocollo HTTP. Essa viene denominata HTTP/0.9 e venne utilizzata esclusivamente all'interno del CERN di Ginevra per la condivisione delle informazioni tra la comunità dei fisici delle alte energie.

1991 : bisogna aspettare qualche anno perché l'HTTP giunga ad una versione più matura ed adatta ad un utilizzo su larga scala. É lo stesso Berners-Lee che migliora il protocollo portandolo alla versione 1.0: nasce così agli inizi degli anni '90 l'HTTP/1.0[3] le cui caratteristiche sono documentate nella RFC[5] 1945. Il WWW conobbe un successo crescente e divennero evidenti alcuni limiti della versione 1.0 del protocollo.

1997-99 : è per questo che, alcuni anni più tardi, esso viene aggiornato alla versione 1.1, quella tutt'ora utilizzata e maggiormente diffusa. L'HTTP/1.1[4] risolve alcuni problemi del suo predecessore: ora, per esempio, il client deve identifcare l'hostname della richiesta, cioè il nome DNS del server. A differenza della versione 1.0, il protocollo prevede che l'hostname venga inviato al server sotto forma di URI all'interno della prima riga della richiesta HTTP, oppure all'interno di uno specifico campo dell'header denominato Host. Questa modifica, apparentemente banale, aumenta di molto le potenzialità del protocollo, in quanto consente l'implementazione di host

[4]HyperText Transfer Protocol
[5]Request for Comments

virtuali, cioè permette che più domini siano ospitati dallo stesso server, e quindi puntino allo stesso indirizzo IP.

HTTP differisce da altri protocolli come FTP[6], per il fatto che le connessioni vengono generalmente chiuse una volta che una particolare richiesta (o una serie di richieste correlate) è stata soddisfatta. HTTP è un protocollo *stateless*, ovvero non mantiene informazioni sulle precedenti richieste del client. Questo comportamento rende il protocollo HTTP ideale per il World Wide Web, in cui le pagine molto spesso contengono dei collegamenti (link) a pagine ospitate da altri server diminuendo così il numero di connessioni attive limitandole a quelle effettivamente necessarie con aumento quindi di efficienza (minor carico e occupazione) sia sul client che sul server.

Le pagine web, le informazioni e le risorse che il client e il server si scambiano, sono rappresentate attraverso il linguaggio HTML[7]. Esso è un linguaggio di *markup* ovvero un insieme di regole globalmente condivise attraverso le quali vengono formattati testi, immagini e altri oggetti. Le pagine web, infatti, altro non sono che documenti HTML che possono contenere e referenziare diversi tipi di contenuti, come ad esempio: testo, link ad altre pagine, immagini, video ed altri contenuti multimediali. Queste risorse, questi oggetti, vengono identificati univocamente da una URL[8], ovvero un indirizzo univoco a livello mondiale che definisce il percorso per raggiungere la risorsa desiderata. Un esempio di URL è il seguente

```
http://www.nomeserver.it/percorso/nomerisorsa.html
```

Analizziamolo un pezzo alla volta:

http : in questo modo si sta comunicando al programma che dovrà inviare la richiesta http (solitamente un browser) che ci si vuole collegare al server attraverso il protocollo HTTP e quindi sulla porta 80

www.nomeserver.it : esso rappresenta il nome di dominio, ossia la rappresentazione *testuale* dell'indirizzo IP del server al quale ci si vuole connettere. Sarà compito del browser convertire tale nome in un indirizzo IP valido contattando un server DNS

[6] File Transfer Protocol
[7] HyperText Markup Language
[8] Uniform Resource Locator

percorso/nomerisorsa.html : una volta collegati al server il browser richiederà tale risorsa e mostrerà a video la risposta del server

2.3 Messaggi HTTP

Come si è discusso precedentemente, la richiesta di risorse dal client al server avviene attraverso lo scambio di messaggi HTTP. Pertanto ci si concentrerà ora sullo studio di questi messaggi. Ogni riga di un messaggio HTTP finisce con il carattere `CRLF`. Questo carattere è, in realtà, l'unione di due: `CR` sta per *carriage return* letteralmente *ritorno del carrello*, mentre `LF` è l'acronimo di *line feed*. L'unione di questi due caratteri indica perciò che il simbolo successivo dovrà essere rappresentato a capo e su una nuova riga. Il carattere `CRLF` è spesso rappresentato anche tramite la notazione `\n\r`. Inoltre ogni messaggio HTTP, che sia esso di richiesta o di risposta, deve terminare con una riga vuota.

Messaggi di richiesta

Viene riportata di seguito un esempio di richiesta HTTP

```
GET /percorso/nomerisorsa.html HTTP/1.1\r\n
Host: www.nomeserver.it\r\n
Connection: Keep-Alive\r\n
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.2; Linux)\r\n
Accept: text/html, image/jpeg, image/png, text/*, image/*, */*\r\n
Accept-Encoding: x-gzip, x-deflate, gzip, deflate, identity\r\n
Accept-Charset: iso-8859-1, utf-8;q=0.5, *;q=0.5 \r\n
Accept-Language: it\r\n
\r\n
```

La prima riga della richiesta HTTP comincia con uno degli 8 comandi riportati in tabella 2.1. Tale prima riga si chiama *Request Line* e comunica al server a che risorsa locale vogliamo accedere, l'operazione che si vuole eseguire su tale risorsa (visionarla, modificarla) e che versione del protocollo HTTP si sta usando. In questo caso il protocollo è `HTTP/1.1` e, infatti, nella richiesta è incluso anche il nome dell'host a cui tale istanza deve essere notificata. La risorsa alla quale si vuole accedere deve essere specificata nella *request line*, immediatamente dopo

il comando che definisce come accedervi. La risorsa richiesta viene identificata tramite *request-URI*[9].

Gli altri parametri sono tutti facoltativi, a discrezione del software utilizzato per comporre tale richiesta. Essi assumono tutti la stessa forma, ossia:

```
nome-parametro: valore
```

L'intero set di parametri costituisce l'*header* del messaggio HTTP. Tali parametri definiscono le modalità con la quale dovrà essere portata a termine la transazione HTTP.

Messaggi di risposta

I messaggi di risposta vengono generati dal server a fronte di una richiesta pervenutagli da un client. Essi hanno fondamentalmente la stessa struttura dei messaggi di richiesta: ogni riga, infatti, termina con un carattere `CRLF`, la prima riga viene chiamata *response line* mentre le righe successive costituiscono l'header della risposta. I messaggi di risposta includono, spesso, un cosiddetto *document body*, ovvero un corpo della risposta. Esso è separato dalla sezione contenente gli header attraverso una riga vuota. Nel corpo della risposta, generalmente, il server include la risorsa richiesta dal client. La lunghezza di tale corpo, ovvero della risorsa, il tipo e la modalità di *encoding*, sono tutti parametri presenti all'interno dell'header del messaggio di risposta. Il protocollo HTTP permette, all'interno del *document body*, l'invio di dati binari; perciò non c'è bisogno di utilizzare codifiche particolari per inviare, per esempio, immagini, video o altri oggetti non testuali.

Come si è accennato sopra, la prima riga del messaggio di risposta viene chiamata *response line*. A differenza della sua controparte nel messaggio di richiesta, la *request line* contiene, come primo carattere, la versione del protocollo HTTP utilizzata dal server per fornire la risposta al client. Separato da un carattere di spazio, si trova invece il risultato dell'operazione, sotto forma di codice, che il server ha eseguito a fronte della richiesta. Tale codice è chiamato *status code*, una tabella riassuntiva di codici di risposta può essere trovata alla fine del capitolo (tabella 2.3).

[9] Uniform Resource Identifier

Comando	Descrizione
GET	Questo comando serve a recuperare la risorsa specificata nel campo request-URI. L'unico effetto lato server di questo comando è, dunque, quello di rispondere al client con un messaggio contenente la risorsa richiesta senza modificarla
HEAD	Identico al comando GET ma la risposta non conterrà la risorsa specificata ma solo gli header. Molto comodo per leggere i metadati contenuti negli headers della risposta, senza dover scaricare l'intera risorsa
POST	Fornisce al server dei dati, includendoli nel corpo del messaggio della richiesta. Può comportare la creazione di una nuova risorsa o il suo aggiornamento. Utilizzato spesso per l'invio di parametri.
PUT	Utilizzato per trasferire sul server una risorsa. L'oggetto che verrà caricato deve essere specificato nel corpo del messaggio della richiesta. Se tale risorsa esiste già, il server deve modificare la risorsa preesistente; nel caso in cui la risorsa non esista, il server deve crearla
DELETE	Il server eliminerà, se esiste, la risorsa specificata nella request URI.
TRACE	Utilizzato per controllare come viene modificato il messaggio di risposta dai server intermedi. Tale comando permette al client di vedere come viene ricevuta la sua richiesta dal server finale e utilizzare tale risposta a fini di diagnostica.
OPTIONS	Con questo comando il client richiede al server le operazioni supportate dalla risorsa specificata nella request URI. In questo modo il server risponderà senza restituire l'intera risorsa e/o effettuare alcuna modifica su di essa.
CONNECT	Utilizzato da proxy o quando c'è la necessità di inizializzare un tunnell SSL. Converte la richiesta HTTP in una connessione TCP/IP trasparente.

Tabella 2.1: Comandi HTTP

Viene riportato, di seguito, un esempio di risposta ad una richiesta HTTP

```
HTTP/1.1 200 OK\r\n
Date: Thu, 24 Oct 2013 13:42:19 GMT\r\n
Server: Apache/2.2.9 (Debian) PHP/5.2.6\r\n
Last-Modified: Fri, 07 Aug 2009 11:43:33 GMT\r\n
ETag: "177446-40b-4708bbeefe340"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 1035\r\n
Vary: Accept-Encoding\r\n
Content-Type: text/html\r\n
\r\n
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
[.......]
\r\n
```

Codice di risposta	Descrizione
`2xx`	Rappresenta la conferma che l'operazione richiesta è andata a buon fine. Il più frequente è il `200`.
`3xx`	Questa famiglia di codici indica che la risorsa specificata nella request URI è stata spostata. Tra i più diffusi e frequenti: `302`, significa che la risorsa è stata spostata permanentemente e la si può trovare nella nuova posizione specificata dall'header `Location`.
`4xx`	Mentre i codici di stato precedente erano riferiti ad errori o successi lato server, questa famiglia di codice rappresenta errori nel messaggio di richiesta. Come ad esempio: `400`, ovvero sintassi della richiesta errata; `403`, codice restituito quando non si possiedono i diritti sufficienti per accedere alla risorsa specificata.
`5xx`	Rappresenta la famiglia di errori propri del server che deve inviare la risposta. Ad esempio il codice `500` indica un errore interno del server, dovuto presumibilmente al sovraccarico di richieste o ad un intervento di manutenzione in corso.

Tabella 2.2: Codici di risposta HTTP

Capitolo 3

PariWeb: il web hosting secondo PariPari

Uno dei plugin che possono essere caricati all'interno dell'ambiente PariPari è PariWeb. Questo plugin mette a disposizione dell'utente un servizio di web server altamente personalizzabile, scalabile ed efficiente.

3.1 Configurazione e personalizzazione

Attraverso un file di configurazione l'utente può specificare diverse opzioni ed impostazioni per adattare alle proprie esigenze il servizio di hosting. Tra le diverse opzioni modificabili:

host Rappresenta l'indirizzo IP al quale deve rispondere il webserver

port L'utente può specificare una porta, diversa dalla porta 80 predefinita per il protocollo http, sulla quale il servizio debba rimanere in ascolto

maxReq Anche il numero massimo di richieste gestibili contemporaneamente dal web server può essere specificato

distribuited Accetta valori booleani (`true` o `false`) ed indica se avviare il web server in modalità distribuita o locale.

isProxy Il webserver può perfezionare anche l'attività di semplice proxy, cioè prendere in carico le richieste e inoltrarle, per conto del client, al server interessato.

Dopo aver caricato PariWeb, digitando nella console di PariPari il comando

```
add webserver
```

il plugin rimarrà in ascolto all'indirizzo `host:port` in attesa di richieste HTTP. Per ogni richiesta pervenuta viene attivato un thread, denominato `ManageRequestTh`, che svolge un compito di controllo e supervisione durante la fase di processing, chiamando in sequenza i singoli componenti. Il modulo ParserHTTP legge la richiesta in ingresso e ne fa il parsing verificandone la correttezza, il modulo ServeRequest esegue la richiesta generando la risposta, ed infine il modulo SendResponse si occupa dell'invio della risposta al client.

3.2 I comandi

Come tutti i plugin di PariPari anche il webserver espone dei comandi che possono essere richiamati tramite la Console. Questi comandi servono all'utente per ottenere informazioni dettagliate sul servizio di hosting, per aggiungere e modificare o eliminare account, gruppi di domini e per fermare un'esecuzione del webserver. Viene riportato di seguito un elenco con i principali comandi disponibili e relativa descrizione.

`help`
: Visualizza a monitor la lista dei comandi disponibili e una loro breve descrizione.

`lsprop`
: Stampa a video il valore a cui sono state inizializzate le variabili definite dall'utente nel file di configurazione.

`lsfs`
: Mostra tutti i file del plugin.

`stop`
: Termina l'esecuzione del plugin all'interno di PariPari. Il webserver non sarà più attivo e non risponderà più alle richieste. Tutti i threads vengono terminati e viene liberata la memoria.

`stat`

Visualizza le statistiche del webserver da quando è stato fatto partire. Come, ad esempio, quante richieste sono state soddisfatte e quante con esito negativo o positivo, l'indice di carico e il tempo di risposta medio.

`lsdom`

Restituisce una lista contenente tutti i domini configurati per il server corrente

`adddom <domain> <folder> <adminPwd>`

Aggiunge un dominio alla lista dei domini configurati. Tale comando accetta tre parametri: `<domain>` rappresenta il nome del nuovo dominio da creare, `<folder>` la cartella in cui risiederanno i file relativi a tale nuovo dominio e `<adminPwd>` la password dell'amministratore di tale dominio, necessaria per apportare modifiche al dominio appena creato.

3.3 Modalità locale

L'utente che avvia il plugin webserver ha la facoltà di decidere se esso debba avviarsi in modalità locale o distribuita. Nel caso in cui venga deciso di attivare la modalità locale, ovvero nel caso in cui si voglia realizzare un web server costituito da un unico nodo, il plugin viene avviato in modo di attivare il servizio di hosting sulla porta specificata nelle opzioni di configurazione del plugin stesso. Tale servizio di hosting si attiverà pertanto sull'indirizzo IP pubblico o, nel caso il computer ospitante non sia connesso alla rete, verrà utilizzato l'indirizzo IP di loop-back (http://127.0.0.1). Questa modalità di utilizzo rappresenta, se vogliamo, un caso particolare della modalità distribuita. In questo caso, infatti, il funzionamento del webserver è molto più lineare: non esiste alcuna rete ma è un unico nodo, un'unica entità che risponde e prende in carico tutte le richieste che gli vengono inviate. Se da una parte ciò si traduce in una maggiore facilità di progettazione e realizzazione, dall'altra invece risente di pesanti limiti legati alla scalabilità e all'efficienza. Se il server, a causa di qualche guasto, dovesse arrestarsi improvvisamente nessuna richiesta potrebbe venire soddisfatta e le risorse che esso ospita rimarrebbero irraggiungibili fino alla soluzione di tali problemi.

In Figura 3.1 viene schematizzato un funzionamento di questo tipo: si noti come non sussiste alcun link o collegamento tra i nodi che formano la stessa rete.

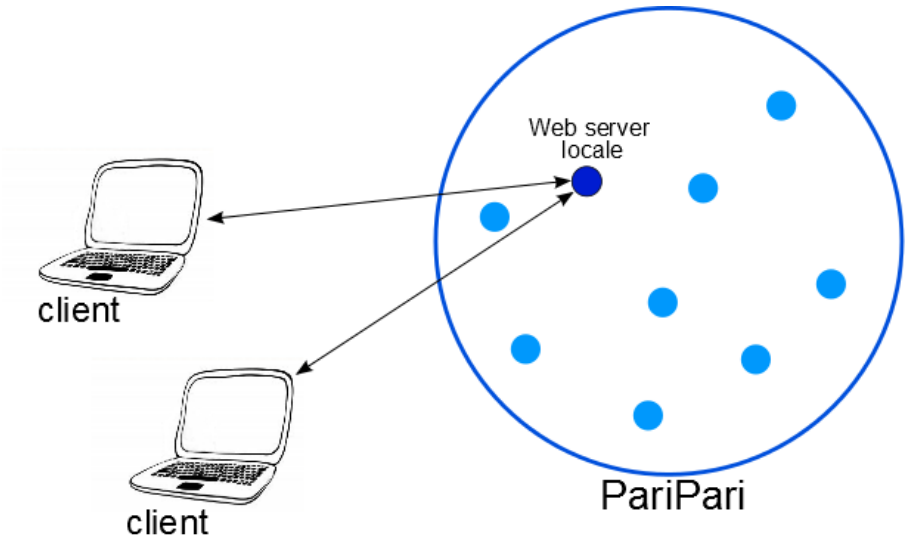


Figura 3.1: Schema funzionamento Webserver in modalità locale

3.4 Modalità distribuita

In modalità distribuita, invece, il plugin sfrutta appieno la rete peer-to-peer messa a disposizione da PariPari. Infatti il servizio finale non si baserà più solamente su unico nodo ma si appoggerà ad una sottostruttura di nodi interconnessi. L'obiettivo è quello di ottenere un server dinamico, in grado di adattarsi al carico di richieste in ingresso e di garantire la persistenza del web server indipendentemente dal fatto che l'utente che lo ha creato sia connesso alla rete. Infatti le risorse offerte da tale webserver vengono replicate intelligentemente sui diversi nodi facenti parte della rete PariPari e su cui è stato abilitato il plugin webserver. I vantaggi di un approccio di questo tipo sono la persistenza dei dati, la tolleranza ai guasti, il bilanciamento del carico e la maggior robustezza. Come si può infatti vedere in figura 3.2, quando uno o più client invieranno una richiesta al webserver, tale richiesta verrà presa in carico non da un singolo nodo ma da una moltitudine di nodi interconnessi tra loro. É facile notare come i client, gli utilizzatori finali del servizio, non si rendano conto di questo comportamento, né della complessità o della struttura della rete con la quale stanno interagendo. Tutta

questa complessità, però, si riflette in maggiore scalabilità, sicurezza, velocità di risposta, robustezza e tolleranza ai guasti. Infatti, grazie ad una replicazione intelligente ed ad una certa ridondanza nei dati, il webserver riuscirà a rispondere alle richieste del client anche se dovesse essersi interrotta la connessione tra alcuni nodi interni della rete distribuita.

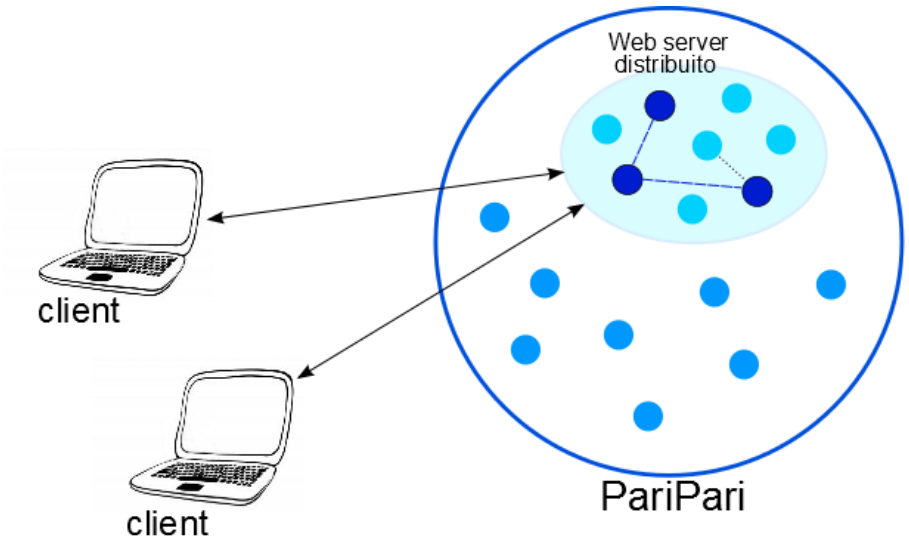


Figura 3.2: Schema funzionamento Webserver in modalità distribuita

Capitolo 4

Implementazione

In questo capitolo verrà illustrato il lavoro da me svolto ed i benefici che questo ha apportato all'interno del progetto PariPari, con particolare attenzione al plugin WebServer e al *parsing* delle richieste HTTP.

4.1 Lo *Unit Testing*

Quando ho iniziato a lavorare a PariWeb, esso presentava alcuni problemi o *bug* che ne compromettevano irrimediabilmente il normale utilizzo e funzionamento. Ho proceduto allora ad un'attenta analisi delle classi che compongono il progetto, cercando di capirne a fondo ed assimilarne al meglio il funzionamento e la logica di base. Una volta conclusa questa tediosa ma necessaria fase iniziale, sono passato alla creazione di classi di *testing* atte a isolare e individuare colli di bottiglia e problemi preesistenti. Tali classi di *testing* sono state realizzate mediante l'utilizzo di *JUnit*, ossia un *framework* di programmazione specifico per il *testing*, che mette a disposizione del programmatore un insieme di *tool* e API che permettono di verificare la correttezza dei metodi e delle funzioni, senza dover caricare ogni volta l'intero progetto.

La fase di *debug* e controllo della correttezza del codice generato, è fondamentale tanto quanto la scrittura dello stesso. Fino ad alcuni anni fa, si eseguiva tale controllo attraverso il *debugger*, strumento mediante il quale il programmatore può seguire, istruzione per istruzione, il flusso di codice. Se, mentre si sta eseguendo il codice, un'istruzione genera un'eccezione allora, tramite tale strumento, si può modificare al volo il codice incriminato sostituendolo e cercando

di risolvere tale errore. Oppure, si può cercare di procedere al *debug* del codice scrivendo istruzioni di test il cui risultato verrà visualizzato nella finestra di *debug*. Entrambi questi metodi, seppur efficaci qualitativamente, presentano il grande svantaggio di dover richiedere la costante presenza di un programmatore, o comunque di una figura specializzata e competente che giudichi correttezza e coerenza dei risultati ottenuti. Inoltre, questi metodi risultano ampiamente inefficaci quando il progetto che si vuole testare è vasto e presenta una moltitudine di *thread* che lavorano parallelamente. Lo *unit testing*, invece, si pone come obbiettivo quello di riuscire a creare test i cui risultati non hanno bisogno di essere interpretati: la loro correttezza viene giudicata direttamente dalla macchina. Oltretutto, *JUnit*, come molti altri *framework* di *testing*, permette l'esecuzione sequenziale di test diversi.

Le classi di *unit testing*, hanno una struttura molto semplice, riportata nel Listato 4.1 e analizzata di seguito:

`@Before`

Tramite questo comando si indica al *framework* di programmazione che la *routine* che segue l'indicatore `@Before` deve essere eseguita prima di ogni test. In questo modo, in tale funzione, si possono inizializzare variabili comuni con il beneficio di non dover riscrivere tale codice per ogni test della classe.

`@After`

Come per l'etichetta `@Before`, solo che, in questo caso, la funzione che segue verrà richiamata ogni qualvolta che un test verrà terminato.

`@Test`

Mentre le prime due istruzioni sono di *setup* e servono solamente per configurare l'ambiente e le variabili di *testing*, l'indicatore `@Test` identifica tutte le funzioni atte a svolgere il lavoro di test. Esse devono essere scritte cercando di stressare il più possibile il metodo che si vuole testare, spaziando dai casi base a casi più anomali, lavorando sugli argomenti presi in carico da tale funzione. La *routine* di test deve terminare con l'interpretazione del risultato ottenuto. Tale interpretazione viene effettuata attraverso delle funzioni di `assert`. Chi scrive la funzione di test specifica a *JUnit* cosa si aspetta che debba essere restituito dal metodo che sta testando. Se il valore

ritornato è coerente con quello aspettato allora il test sarà superato. Se, per esempio, si vuole testare la funzione `compareToOne`, che ritorna `true` se il parametro passato è `1`, allora nella funzione di test del caso base potremmo scrivere `assertTrue(compareToOne(1));`. Il metodo `assertTrue` segnalerà il test come correttamente passato se l'espressione contenuta nelle parentesi valuterà a `true`.

```
1  @Before
2  public void setUp(){
3
4  }
5
6  @After
7  public void tearDown(){
8
9  }
10
11 @Test
12 public void test1() {
13
14 }
15
16 @Test
17 public void test2() {
18
19 }
```

Listato 4.1: Struttura di una classe JUnit

4.2 Parsing delle richieste HTTP

Attraverso i test JUnit si è palesata l'inefficienza e l'incorrettezza della classe `ParserHTTP_Module`, essa infatti commetteva alcuni errori durante la fase del *parsing*: molti *header* non venivano correttamente riconosciuti e processati. Inol-

tre, tale classe introduceva un bug che si propagava all'intero progetto e rendeva il webserver inutilizzabile.

ParserHTTP è la prima classe attivata dal thread incaricato di processare la richiesta pervenuta dal client. Essa si occupa di farne il *parsing*, cioè di leggere l'input proveniente dal client determinandone la struttura e la correttezza. Inoltre, la fase di *parsing*, ha come scopo anche quello di suddividere le varie parti della richiesta HTTP e renderle fruibili alle classi successive. Per rendere il più semplice possibile alle altre classi attingere alle informazioni estratte dalla richiesta HTTP, si è deciso di inserire ogni *header*, costituito dalla coppia `nome:valore`, in una *hashtable*. In questo modo tutti gli *header* risultano indicizzati e facilmente iterabili e ricercabili utilizzando i metodi propri della *hashtable*. É fondamentale che gli *header* vengano inseriti in una struttura di dati che ne renda immediata la ricerca: essi infatti dovranno essere utilizzati da molteplici classi, come `SendResponse_Module` e `ServeRequest_Module` che necessitano di costruire il messaggio di risposta a partire dagli *header* presenti nella richiesta.

Il *parser* deve, inoltre, essere in grado di processare sia richieste HTTP 1.1 che richieste HTTP 1.0, offrendo buone prestazioni in termini di tempo necessario al *parsing*. Le performance di questo componente, infatti, influenzano direttamente quelle dell'intero web server: un *parser* lento corrisponde, per esempio, ad un lento tempo di risposta. La logica e il processo adottati per la lettura e il *parsing* dello stream di input sono stati progettati per ottenere il massimo grado di efficienza, riutilizzabilità e leggibilità del codice. La lettura infatti avviene byte per byte, una riga alla volta. Prima di procedere con il *parsing* della successiva, la riga letta viene processata e ne viene confermata la correttezza, al fine di evitare la lettura dell'intera richiesta nel caso di una struttura non corretta.

Sfruttando la modularità delle richieste HTTP, il codice è ottimizzato per processare in maniera diversa tre differenti tipi di righe:

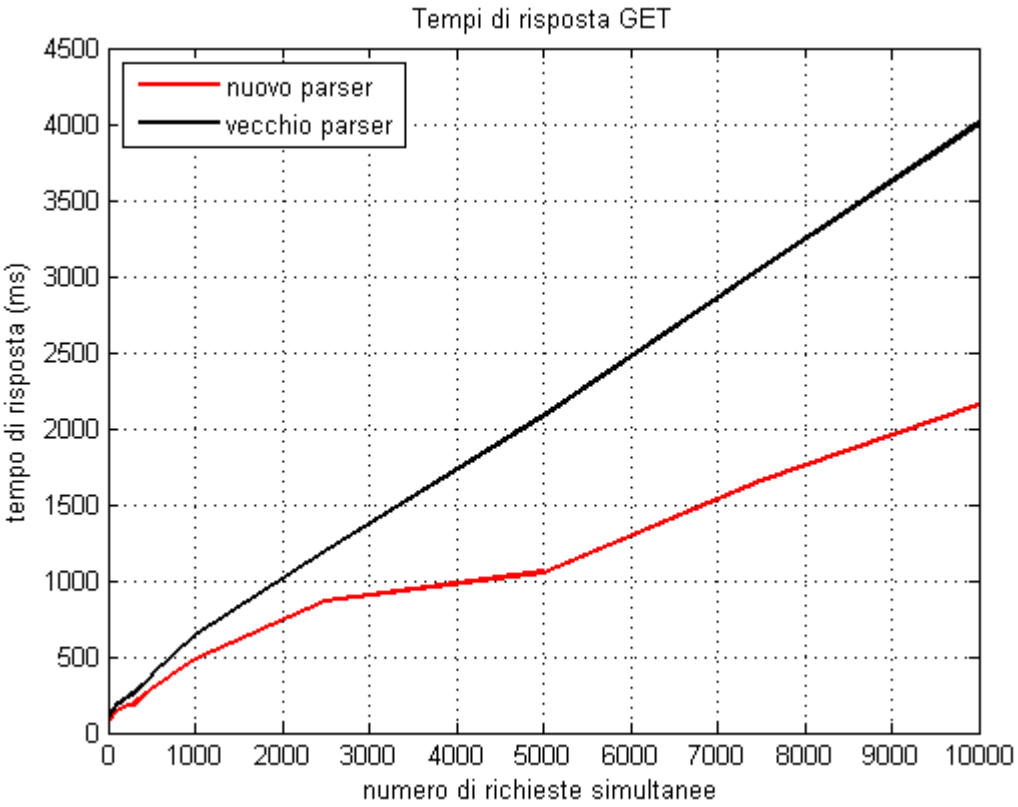
- La prima riga. Essa contiene una serie di informazioni, come metodo, URI della risorsa e versione HTTP della richiesta, che devono essere processate e controllate separatamente. Tale operazione è riportata nel Listato 4.2.

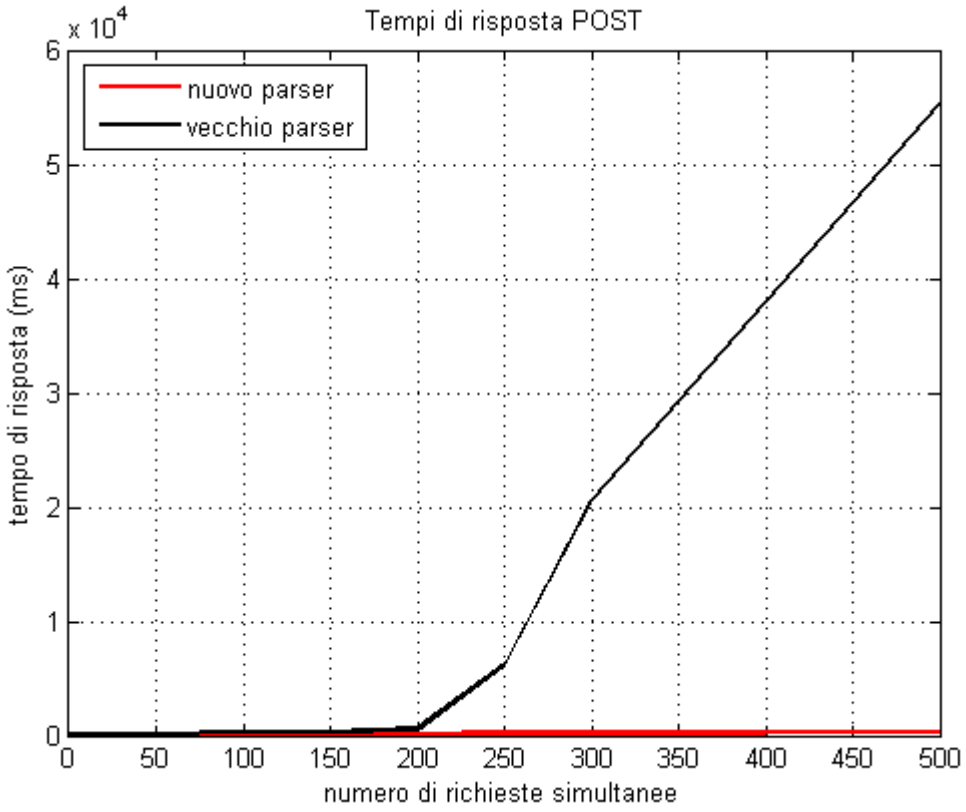
- Gli *header*. Dopo la prima riga, e fino alla prima vuota, sono contenuti tutti gli *header* della richiesta. Avendo tutti la stessa forma `<nome header>=<valore>\r\n` essi possono essere processati, riga per riga, dalla stessa funzione. Si veda, a tal proposito, il codice riportato nel listato 4.3.

- Dati aggiuntivi. Tali dati aggiuntivi sono presenti solo se esiste un *header* chiamato `Content-Length` con valore diverso da `0`. In questo caso, dati addizionali vengono inseriti dal client a partire dalla riga successiva a quella vuota dopo gli *header* ed anch'essi terminano con una riga vuota, o meglio contenente il solo carattere `\r\n`. Il codice che svolge tale funzione è riportato nel listato 4.4.

Attraverso lo studio della regolarità e modularità dei messaggi di richiesta HTTP, si è giunti a riconoscere una proprietà di sottostruttura che ha permesso di scrivere codice altamente performante e di immediata comprensione. Grazie a tale proprietà, inoltre, il problema del *parsing* è stato suddiviso in sottoproblemi di taglia più piccola con notevoli benefici computazionali, che verranno discussi e trattati nel prosieguo. Prima della riscrittura completa, la classe si presentava come un caotico susseguirsi di cicli `while` annidati e sviluppati mediante nomi di variabili monosillabiche, il tutto corredato dalla totale assenza di commenti esplicativi. La riscrittura del *parser* http è risultata necessaria e non più rimandabile quando ci si è accorti, tramite il *testing*, che alcuni campi *header* non venivano correttamente riconosciuti ed inseriti nell'hashtable. Inoltre non erano state implementate alcune direttive riguardanti la versione del protocollo HTTP 1.1, non veniva, per esempio, controllato che venisse effettivamente specificato l'host a cui la richiesta era indirizzata. In questo modo, dato che il webserver prevede, supporta ed è in grado di gestire diversi domini su un unico nodo, richieste prive del nome del dominio a cui sono indirizzate potrebbero venire processate in maniera errata, restituendo, ad esempio, una risorsa contenuta in un altro dominio o generare un errore irreversibile all'interno del plugin.

Mediante l'utilizzo di JUnitPerf[6], un componente aggiuntivo che si affianca a JUnit, si sono stimati i tempi di risposta differenti delle due classi di *parsing*. JUnitPerf, infatti, mette a disposizione diversi tool per testare non solo la correttezza ma anche l'efficienza, le performance e la scalabilità di un dato metodo e di una data classe. Eseguendo diversi test di carico, simulando diversi scenari in cui un numero sempre crescente di utenti faceva pervenire richieste al webserver, si sono raccolti dati sui tempi di risposta delle due classi. Le misurazioni hanno riguardato i tempi di *parsing* delle richieste di tipo `GET` e `POST`; i risultati sono portati in figura 4.2. Come si può notare osservando i grafici, che rappresentano l'aumento del tempo di risposta a fronte di un aumento di richieste, con pochi


Tempi di risposta GET
nuovo parser
vecchio parser
tempo di risposta (ms)
numero di richieste simultanee
0
500
1000
1500
2000
2500
3000
3500
4000
4500
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000


Tempi di risposta POST
x 10^4
nuovo parser
vecchio parser
tempo di risposta (ms)
numero di richieste simultanee
0
1
2
3
4
5
6
0
50
100
150
200
250
300
350
400
450
500

utenti simultanei le due classi presentano un andamento pressoché uguale. Più cresce il carico di lavoro, però, più la discrepanza tra i due andamenti diventa significativa. Soprattutto nelle richieste di `POST` (fig. 4.2), in cui ci sono dati aggiuntivi da leggere oltre agli *header*, il divario tra le prestazioni delle due classi risulta consistente.

```
1 if (firstLine != null && !firstLine.isEmpty()){
2     parseMethod(firstLine);
3     parseURI(firstLine);
4     parseHTTPVersion(firstLine);
5     ...
6 }
```

Listato 4.2: Parsing della prima riga della richiesta HTTP

```
1  intrec = input.read();
2  // read headers line by line
3  while((intrec != -1) && ((c = (char)intrec) != '\n')){
4      headerLine = headerLine + c;
5      intrec = input.read();
6  }
7  if(headerLine.equals("\r")){  //equals to "\r" and not "\r
       \n" because the while exit before read "\n"
8      headerLine = "";
9      headers += WebServer.CRLF;
10 } else{
11     parseParameters(headerLine);
12     headers += headerLine + '\n';
13     if(hostField && redirection){
14         parsePathAndFileName();
15         return headers;
16     }
17 }
```

Listato 4.3: Parsing degli header

```
1 if(contentLength) {
```

```
2       try{
3           int dataLength = new Integer(request.get("Content-
        Length"));
4
5           while(dataLength > 0){
6               intrec = input.read();
7               c = (char)intrec;
8               additionalData = additionalData + c;
9               dataLength--;
10          }
11          request.put("AdditionalData", additionalData.trim
        ());
12          additionalData = additionalData + WebServer.CRLF;
13      }catch (IOException e) {
14          throw new HttpException(HttpErrorCode.
        LengthRequired);
15      }
16  }
```

Listato 4.4: Parsing di dati addizionali

4.3 *Code analysis*

Una volta risolti tutti i problemi di *parsing* ci si è accertati, mediante *unit test*, che anche per diversi tipi di richieste gli *header* venissero completamente riconosciuti ed inseriti nell'*hashtable*. Successivamente si è iniziato ad utilizzare il plugin webserver, si è abbozzato un sito web locale e lo si è visitato mediante browser. Ci si è subito accorti che, mentre le pagine `html` e i documenti di puro testo venivano mostrati e visualizzati correttamente, i file immagine e altri documenti non testuali invece non riuscivano ad essere correttamente interpretati dal browser. Essendo tale comportamento difficilmente riproducibile con le sole tecniche di *testing*, si è deciso di procedere ad un *debug* del codice delle classi `ParserHTTP_Module`, `SendResponse_Module` e `ServeRequest_Module`. Tale pratica, però, non ha messo in evidenza alcun tipo di errore o eccezione non

gestita. Si è allora analizzato, a fondo e con attenzione, il percorso logico seguito dalla risorsa: dalla sua richiesta fino ad essere servita al browser web. Tale percorso è stato schematizzato nella Figura 4.1: nella precedente versione del *parser*, la risorsa veniva letta completamente dalla classe `ParserHTTP_Module`, il suo contenuto veniva convertito in stringa e salvato nell'*hashtable*; infine la classe `ServeRequest_Module` recuperava il contenuto della risorsa, lo riconvertiva in *byte* e lo scriveva nello *stream* di output. Nell'ottica di rendere il codice più scalabile e performante possibile, ci si è posti alcune domande:

- La conversione da `byte` a `String`, e viceversa, viene effettuata senza specificare alcuna codifica. In questo modo, però, alcuni caratteri potrebbero non venire codificati e decodificati in maniera corretta provocando degli errori. Come si potrebbe risolvere tale problema?
- É accettabile incapsulare risorse di alcuni *megabyte* all'interno dell'*hashtable*. Ma cosa accadrebbe, invece, se la risorsa richiesta fosse dell'ordine dei *gigabyte*?

É subito risultato chiaro come le due domande fossero la risposta al malfunzionamento del webserver. Quando non diversamente specificato, infatti, Java utilizza la codifica UTF-8[1], cioè una codifica di un sottoinsieme dei caratteri Unicode[7]. Essendo tale codifica solo un sottoinsieme, non contiene al suo interno molti dei caratteri necessari per codificare e decodificare oggetti di tipo binario. La codifica UTF-8, infatti, è utilizzata principalmente per codificare testi e non oggetti che hanno una rappresentazione binaria come immagini, video e musica. Perciò, quando la risorsa richiesta veniva trasformata in stringa dopo essere stata letta, venivano perse delle informazioni che ne compromettevano il corretto recupero da parte del browser.

Per quanto riguarda la seconda domanda, invece, è facile immaginare come possa influenzare negativamente le prestazioni dell'intero progetto caricare in una *hashtable* quantità di dati dell'ordine dei *gigabyte*. Si è deciso, dunque, per rendere più lineare il percorso logico del processo di risposta e per rendere più performante il codice, di demandare la lettura della risorsa alla classe `SendResponse_Module`. Durante la lettura della richiesta, la nuova classe `ParserHTTP_Module`, costruisce il percorso locale completo della risorsa in oggetto e lo inserisce nell'*hashtable*.

[1] Unicode Transformation Format, 8 bit

Successivamente, la classe progettata per servire la risposta, dopo aver scritto sullo *output stream* l'intestazione e gli *header* della risposta, leggerà la risorsa dal percorso locale e la scriverà simultaneamente sullo *stream* di uscita. Nel Listato 4.5, viene riportato il codice che esegue la procedura appena decritta.

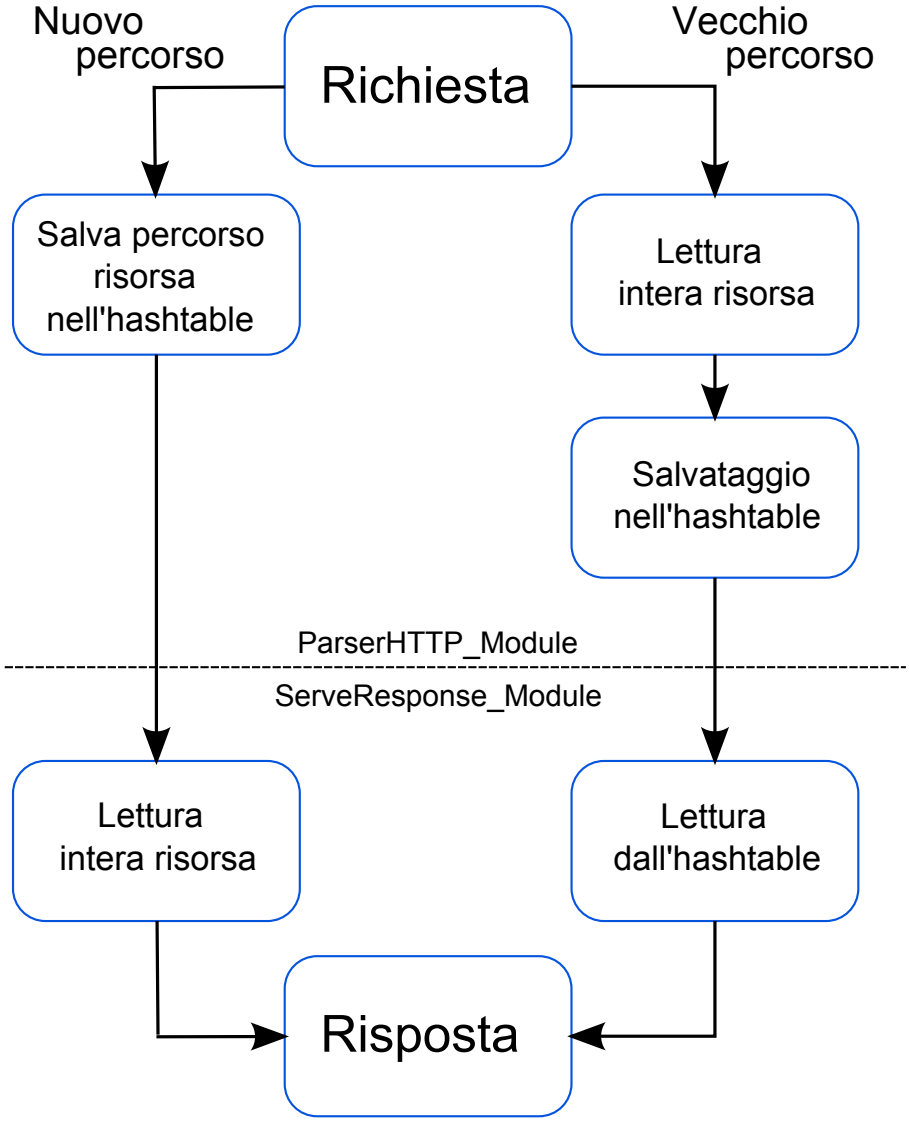


Figura 4.1: Diagramma del percorso logico effettuato dalla risorsa: dalla richiesta, fino alla sua risposta

```
1 paramLine = tabResp.get("resPath");
2 if (paramLine != null){
3     //Set up feature object
```

```
4       IFeatureValue[] feature = new IFeatureValue[1];
5       feature[0] = new FeatureValue("time",60000);
6       //Retrieve file reader from storage
7       FileInputStreamAPI reader = webServer.askTheCore(
        FileInputStreamAPI.class, new ConstructorParameters(
        feature, paramLine), true);
8       // Allocate buffer
9       byte[] bytes = new byte[1024];
10      // Read and send requested file
11      int ch = reader.read(bytes, 0, 1024);
12      while (ch != -1) {
13          output.write(bytes, 0, ch);
14          ch = reader.read(bytes, 0, 1024);
15      }
16  }
```

Listato 4.5: Nuovo metodo di costruzione della risposta HTTP

Capitolo 5

Conclusioni

In questo elaborato è stato presentato il lavoro svolto per rendere scalabile, performante e affidabile un web server distribuito scritto interamente in Java, basato sulla rete PariPari, di cui sfrutta la rete. Dopo brevi cenni storici, in cui si sono voluti ripercorrere i passi fondamentali che hanno portato alla creazione del *WorldWideWeb*, ci si è concentrati sull'architettura e la struttura del progetto PariPari e sull'ambiente in cui PariWEB affonda le sue radici. Successivamente sono state fornite le basi di HTTP, la differenza tra funzionamento locale e distribuito e l'architettura client-server. In questo modo, sono stati forniti gli strumenti necessari per capire e giustificare le scelte applicate per apportare scalabilità e velocità al web server.

In questo documento, si è inoltre discussa l'importanza fondamentale rappresentata dal *parser* all'interno del web server. Tale modulo, infatti, determina la differenza tra un web server efficiente ed uno lento: tutte le richieste dovranno essere processate mediante tale componente, che dovrà decodificarle e renderle fruibili alle altre classi. Le informazioni decodificate dal *parser* verranno utilizzate al fine di creare un corretto messaggio di risposta. Per questi motivi ci si è concentrati sulla riscrittura di tale classe, ponendosi come obbiettivo il miglioramento prestazionale e computazionale.

Bibliografia

[1] PariPari contributors, "PariWiki", `http://indy.dei.unipd.it/mediawiki/index.php/PariPari_en`.

[2] Wikipedia contributors, "WorldWideWeb", *Wikipedia, The Free Encyclopedia*, `http://it.wikipedia.org/wiki/Web`.

[3] Berners-Lee T., Fielding R., Frystyk H., "RFC 1945: Hypertext Transfer Protocol – HTTP/1.0", Maggio 1996, `http://tools.ietf.org/html/rfc1945`.

[4] Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T., "RFC 2616: Hypertext Transfer Protocol – HTTP/1.1", Giugno 1999, `http://tools.ietf.org/html/rfc2616`.

[5] Samory M., 2010, *PariGUI 2010*, Dipartimento di Ingegneria dell'Informazione, Università di Padova

[6] Clark M., "Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications", Luglio 2004, The Pragmatic Bookshelf

[7] Sharan K., "Harnessing Java 7: A Comprehensive Approach to Learning Java", 2012, Creatinina Independent Pub, pp. 75-80