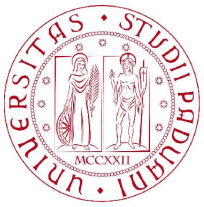
UNIVERSITÀ DI PADOVA  FACOLTÀ DI INGEGNERIA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

PariWEB

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

Correlatori: Ing. Paolo Bertasi, Ing. Michele Bonazza

Laureando: Stefano Bettineschi

A.A. 2010 - 2011

Padova, 19 Aprile 2011

"Alla mia famiglia
che mi è sempre stata vicina"

Sommario

PariPari è un progetto, o meglio una sfida, 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, mantenendoli usufruibili anche da computer esterni alla rete[1].

Per portare avanti quest'obiettivo, è stato costituito un gruppo di ricerca e sviluppo formato esclusivamente da laureandi. Il presente documento si colloca all'interno dell'universo di PariPari, con l'obiettivo di illustrare l'attività di studio, progettazione e realizzazione di un modulo della rete in grado di fornire all'utente finale un servizio completo di web hosting.

Nella parte iniziale (Capitolo 1) verrà presentato il contesto all'interno del quale si colloca PariWEB, in particolare analizzando la struttura e le caratteristiche del progetto PariPari e fornendo una panoramica sul mondo del Web. Successivamente (Capitolo 2) saranno illustrati più in dettaglio gli obiettivi, le specifiche e le caratteristiche richieste dal progetto in questione, quindi (Capitolo 3) saranno discusse le varie soluzioni prese in considerazione e le scelte progettuali adottate per raggiungere gli obiettivi sopracitati. Nel capitolo successivo (Capitolo 4) sarà invece presentata la realizzazione del servizio, con particolare attenzione all'implementazione delle scelte definite nella fase di progettazione, per poi illustrare le performance ottenute e descrivere brevemente come un utente può configurare ed utilizzare il modulo realizzato (Capitolo 5).
Infine (Capitolo 6) viene dato spazio ad alcune riflessioni sul lavoro svolto, valutando i risultati ottenuti e presentando alcuni possibili sviluppi futuri.

[1] `http://indy.dei.unipd.it/mediawiki/index.php/PariPari_en`

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Il grandissimo sviluppo delle tecnologie web che si è registrato negli ultimi anni ha portato alla realizzazione di numerose applicazioni diversificate e molte volte incompatibili tra loro. Questa grande varietà costringe l'utente finale a cercare, scaricare, installare, configurare, imparare a gestire e utilizzare svariati programmi, ognuno dei quali fornisce un'unica funzionalità, o è utilizzato solo per alcuni dei servizi che offre, essendo presenti sul mercato soluzioni migliori o semplicemente più diffuse. L'obiettivo principale, e l'idea innovativa, alla base di PariPari è di realizzare un'unica piattaforma in grado di fornire all'utente finale tutti i servizi ora disponibili per mezzo di Internet.

Uno dei servizi che PariPari si propone di offrire è quello di web hosting, cioè fornire agli utenti la possibilità di caricare delle risorse su di un server, denominato web server, rendendole accessibili tramite Internet. L'obiettivo finale del lavoro oggetto di questo documento è quello di realizzare un web server distribuito in grado di fornire un servizio completo e configurabile secondo le esigenze degli utenti, oltre che una struttura capace di sfruttare al meglio le caratteristiche di un ambiente P2P come quello della rete PariPari. Aspetti fondamentali di questo modulo saranno, perciò, la progettazione orientata alla distribuzione, alle performance e all'integrazione con il resto del progetto e l'implementazione di una struttura robusta, sicura e stabile, ma anche facilmente espandibile e configurabile.

Dopo questa breve introduzione di carattere generale, nelle prossime sezioni di questo capitolo verrà presentato il contesto del lavoro oggetto di questa tesi, in particolare verrà analizzato il progetto PariPari (sezione 1.1) ed il mondo del Web all'interno del quale PariWEB si colloca (sezione 1.2).

1.1 Il progetto PariPari

Come accennato nell'introduzione, questo lavoro si colloca all'interno del progetto PariPari. Nella prima parte di questa sezione saranno descritte le reti peer to peer e le principali caratteristiche che ne hanno consentita una così ampia diffusione, mentre nella seconda parte analizzeremo nello specifico la rete PariPari ed i suoi elementi caratterizzanti.

1.1.1 Reti peer to peer

Innanzitutto PariPari è una rete peer to peer (P2P in seguito), cioè una rete in cui non si distinguono i client dai server, ma all'interno della quale tutte le entità sono uguali (peer) e si comportano sia da client che da server nei confronti degli altri nodi della rete. Queste reti sono nate principalmente per la condivisione di brani musicali, ma in generale permettono di condividere qualsiasi tipo di risorsa, quali contenuti, spazio di memorizzazione, ampiezza di banda e potenza di calcolo.
Le prime reti definite P2P, come Napster[1], sono dette ibride in quanto non costituite da nodi tutti uguali, ma anche da server (o supernodi) cui i client accedono per conoscere la posizione della risorsa cercata. Solo in un secondo momento, per l'acquisizione vera e propria del file, vi è un collegamento tra due entità peer (client). Le realizzazioni più recenti di questo tipo di reti permettono una classificazione tra P2P non strutturate e P2P strutturate. Le prime, come Gnutella[2], realizzano i collegamenti tra peer in modo del tutto arbitrario con notevoli vantaggi nell'implementazione del protocollo, ma con altrettanti svantaggi in termini di prestazioni, in quanto la ricerca di risorse all'interno della rete avviene per inondazione (il nodo che sta effettuando una ricerca manda la richiesta ai propri vicini, i quali a loro volta la mandano ai propri vicini e così via finché la risorsa non viene trovata). Le reti strutturate, invece, adottano un protocollo più complesso basato su Distributed Hash Table[3] (DHT) che consente di mappare la rete assegnando ogni risorsa a un particolare peer. Questa soluzione, notevolmente più complessa, garantisce però il reperimento delle risorse all'interno della rete in modo molto più efficiente. PariPari, per com'è stata concepita, è un esempio di rete P2P strutturata e completamente serverless.

Le reti peer to peer hanno avuto una così ampia e rapida diffusione perché presentano delle caratteristiche molto interessanti che saranno brevemente analizzate nei prossimi paragrafi.

Fault tolerance

Il fatto che il funzionamento della rete non dipenda da un ristretto numero di macchine (server o supernodi), ma dall'insieme dei nodi, rende la struttura molto robusta e completamente indipendente dai singoli elementi. D'altra parte, siccome un nodo può abbandonare o aggiungersi alla rete in un qualsiasi momento, la

[1] `http://en.wikipedia.org/wiki/Napster`
[2] `http://en.wikipedia.org/wiki/Gnutella`
[3] `http://en.wikipedia.org/wiki/Distributed_hash_table`

struttura deve essere molto flessibile ed elastica, nonché adottare meccanismi per mantenere la consistenza dei dati presenti nella rete.

Più risorse

Ogni nodo che si aggiunge alla rete contribuisce a incrementare le risorse a disposizione di tutti gli altri elementi, sia in termini di contenuti, che per quanto riguarda spazio di archiviazione, ampiezza di banda e potenza di calcolo. Con l'aumentare del numero di nodi che compongono la rete, cresce la qualità dei servizi a disposizione di ogni singolo utente. Si parla in questo caso di Network Effect[4], un fenomeno tipico delle reti P2P che può costituire sia uno svantaggio, in quanto una rete appena nata avrà difficoltà a crescere, che un vantaggio, in quanto una struttura con molti nodi fornirà un servizio migliore attirando sempre nuovi utenti.

Scalabilità

I nodi all'interno della rete sono tutti uguali e in grado di fornire i medesimi servizi. Per questo motivo il lavoro, soprattutto nel caso di reti strutturate come PariPari, è diviso equamente tra tutti i nodi della rete ognuno dei quali è responsabile di un certo numero di risorse. Dato che, come introdotto nel paragrafo precedente, il servizio offerto da reti P2P è direttamente proporzionale al numero di nodi che compongono la rete stessa, si ottiene un'ottima scalabilità del sistema.

1.1.2 Caratteristiche di PariPari

Come introdotto all'inizio di questo documento, l'obiettivo di PariPari è creare una rete peer-to-peer strutturata, multipiattaforma e anonima, che fornisca sia i servizi tipici di Internet (mail, web hosting, DNS) che quelli tipici delle reti P2P (file sharing). L'insieme di questi servizi vuole essere fornito per mezzo di un'unica soluzione software, mantenendoli comunque usufruibili anche ai computer esterni alla rete, i quali percepiranno la rete PariPari come un'unica macchina virtuale.
Oltre alle proprietà tipiche delle reti peer to peer elencate in precedenza, PariPari presenta alcuni aspetti caratteristici, che saranno illustrati brevemente nei prossimi paragrafi.

Rete strutturata e completamente serverless

Una struttura di questo tipo permette agli utenti di non dover rimanere connessi alla rete in modo permanente e allo stesso tempo garantisce una maggiore robustezza e scalabilità della rete. Per realizzare ciò, PariPari adotta una variante del protocollo Kademlia[5] (basato su DHT) che consente di indicizzare e ricercare all'interno della rete sia risorse che host in modo efficiente e completamente decentralizzato.

[4] `http://en.wikipedia.org/wiki/Network_effect`
[5] `http://en.wikipedia.org/wiki/Kademlia`

Figura 1.1: Logo ufficiale del progetto PariPari.

Anonimato

PariPari prevede un sistema di anonimato che garantisce la privacy dell'utente, eliminando la possibilità che possano essere rintracciati gli indirizzi IP della fonte e del ricevente durante lo scambio di risorse tra due nodi della rete. Questa funzionalità è stata realizzata consentendo, a ogni utente che desideri proteggere la propria identità, la creazione di una catena di nodi tra sé e il suo interlocutore, lungo la quale i dati sono scambiati con un meccanismo di onion routing[6] che garantisce la segretezza e la sicurezza della comunicazione.

Crediti

Il ruolo che riveste il modulo dei crediti all'interno di una rete P2P è molto importante, in quanto regola i rapporti di collaborazione tra utenti e permette di quantificare i servizi e le risorse messi a disposizione o richiesti da ogni elemento della rete. Il concetto alla base di un qualsiasi modulo di crediti è il seguente: un nodo che fornisce un determinato servizio guadagna dei crediti, un nodo per poter utilizzare un servizio o per acquisire una risorsa deve pagare un certo numero di crediti. In molte reti P2P esistenti è presente un meccanismo di assegnazione dei crediti che ha lo scopo di incentivare gli utenti a rimanere nella rete più a lungo e di condividere il maggior numero di risorse possibili.
In PariPari viene utilizzato un modulo per la gestione intelligente dei crediti, basato sulla **transitività** di questi ultimi. Il vantaggio di un approccio simile risulta chiaro con un esempio. Assumiamo che Bob, Alice, Charlie siano tre utenti della rete. Alice e Bob spesso concludono affari tra loro così come Alice e Charlie, mentre Bob e Charlie supponiamo non abbiano mai avuto nessun contatto diretto. Immaginiamo ora che Charlie sia interessato ad acquisire una risorsa di Bob. Purtroppo, Charlie non è in credito con Bob e non ha modo per aggiudicarsi tale risorsa. Alice, però, è fortemente indebitata con Charlie. Charlie procederà quindi ad acquisire la risorsa grazie alla mediazione di Alice, che risulta essere in debito con Charlie e in credito con Bob. La transitività consente di realizzare un'economia più libera all'interno di PariPari, che a sua volta si traduce in maggiori risorse accessibili per gli utenti.

[6] `http://en.wikipedia.org/wiki/Onion_routing`

Il modulo crediti prevede, inoltre, che nel caso in cui un plug-in non abbia sufficienti crediti per una richiesta, questa venga direttamente respinta dal core, fino all'eventuale sospensione del plug-in stesso. Questo accorgimento impedisce che eventuali plug-in maligni possano realizzare attacchi DoS, inondando il core con richieste non soddisfabili al fine di interrompere il funzionamento di altri plug-in.

Architettura a plug-in

PariPari è stato concepito e realizzato con un'architettura a plug-in[7], atta a garantire la multifunzionalità e a favorire gli sviluppi futuri aumentando il grado di espandibilità del sistema. 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 un modulo centrale, denominato `Core`[8], il cui ruolo è d'intermediario e collante tra i vari plug-in. I plug-in, invece, realizzano le singole funzionalità messe a disposizione da PariPari sfruttando le risorse della macchina su cui gira il client e interagendo tra loro attraverso il Core. Grazie a questa particolare architettura, la scrittura di un nuovo plug-in risulta molto semplice, in quanto non richiede la conoscenza della struttura interna di tutti gli altri moduli ma solo del modo con cui interfacciarsi. Ogni modulo è visto come una black box[9], di cui è necessario conoscere solamente input e output per poter usufruire dei servizi offerti.
I plug-in di PariPari si dividono in due categorie principali: quelli della cerchia interna (Connectivity, Crediti, LocalStorage e DHT) necessari per la gestione delle 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.

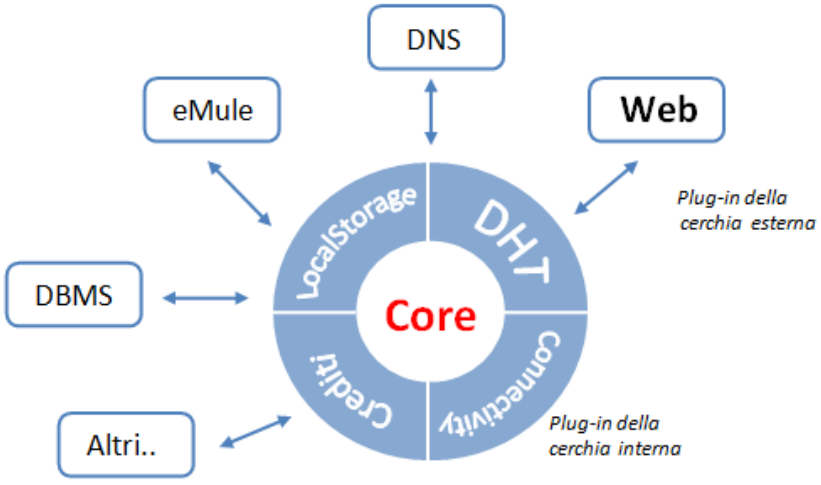


Figura 1.2: Architettura a plug-in di PariPari.

[7]Insieme di classi Java, che forniscono uno specifico servizio integrandosi con un altro programma per ampliarne le funzionalità.
[8]`http://indy.dei.unipd.it/mediawiki/index.php/Core_en`
[9]`http://en.wikipedia.org/wiki/Black_box`

Java

Affrontiamo ora l'argomento linguaggio di programmazione. PariPari è realizzato in Java, in quanto garantisce la portabilità del programma su tutte le principali piattaforme, senza rendere necessaria la ricompilazione del codice sorgente. Inoltre, Java mette a disposizione la tecnologia Java Web Start che consente di integrare l'applicazione con il browser, gestendo in modo automatico e trasparente il download e l'esecuzione, l'installazione e l'aggiornamento del software stesso.
Java, d'altra parte, presenta anche degli aspetti negativi, legati soprattutto alle performance. Essendo un linguaggio interpretato, le istruzioni prima di essere eseguite dalla macchina vengono processate dalla Java Virtual Machine, dando luogo a prestazioni inferiori a quelle ottenibili utilizzando un linguaggio compilato come il C. Questo aspetto rappresenta il prezzo da pagare per ottenere un'elevata portabilità dell'applicazione.

Open Source

Prima di passare alla sezione successiva, è importante sottolineare che, data la natura open source aperta al pubblico e agli sviluppatori, il progetto PariPari verrà rilasciato sotto licenza GPL[10]. Inoltre, ogni sviluppatore potrà realizzare il proprio plug-in utilizzando i servizi offerti dei moduli esistenti, senza la necessità di conoscere la loro struttura interna, ma semplicemente utilizzando le API (Application Programming Interface) messe a disposizione da ciascuno.

1.2 Il World Wide Web

Il World Wide Web(WWW) più spesso abbreviato in Web, è un servizio di Internet che consente di navigare ed usufruire di un insieme vastissimo di contenuti multimediali e di ulteriori servizi, accessibili a tutti o ad una parte selezionata degli utenti di Internet.

Il World Wide Web (d'ora in poi Web) ha avuto origine come un progetto dell'European Particle Physic Laboratory del CERN di Ginevra alla fine degli anni '80, per fare in modo che i suoi moltissimi ricercatori, distribuiti in tutto il mondo, potessero accedere a documenti condivisi (note personali, report, figure, progetti, disegni, ecc.) usando un semplice sistema ipertestuale. Collegando i documenti tra loro, divenne facile integrare i documenti provenienti da diversi progetti in un nuovo documento senza la necessità di modifiche centralizzate molto complesse. L'unica cosa necessaria era costruire un documento che fornisse i collegamenti ad altri documenti rilevanti. Il Web crebbe gradatamente includendo anche siti diversi da quelli per i fisici nucleari, ma la sua popolarità aumentò improvvisamente quando divennero disponibili delle interfacce grafiche (Mosaic[11]) per presentare ed accedere ai documenti in modo semplice ed immediato. Un documento veniva prelevato da un server, trasferito al client e visualizzato sullo schermo, rendendo trasparente all'utente finale il fatto che il contenuto richiesto fosse mantenuto in

[10] `http://it.wikipedia.org/wiki/GNU_General_Public_License`
[11] `http://en.wikipedia.org/wiki/Mosaic_browser`

remoto o in locale. Dal 1994, gli sviluppi del Web sono stati promossi dal World Wide Web Consortium, una collaborazione tra il CERN e il M.I.T, responsabile della standardizzazione dei protocolli, del miglioramento dell'interoperabilità e delle funzionalità del Web.

1.2.1 Architettura client-server

L'architettura di un sistema Web, pur essendosi evoluta nel corso degli anni, può essere esemplificata come un'organizzazione di tipo **client-server**, in cui vi è un processo server (**web server**) che ospita delle risorse ed N client che vi possono accedere. Un client interagisce con un Web server per mezzo di un'applicazione chiamata **browser** che consente all'utente di inserire il riferimento alla risorsa di interesse, per poi prelevarla e visualizzarla. Perché un client possa fare riferimento ad una risorsa di un web server, è necessario che questa sia identificabile in modo univoco. Il modo adottato nel Web è quello di utilizzare un URL (Uniform Resource Locator[12]), cioè una sequenza strutturata di caratteri che indica il nome DNS del server associato, il percorso e il nome del contenuto richiesto.

Figura 1.3: Richiesta di una risorsa ad un web server

La comunicazione tra queste due entità (web server e browser del processo client) avviene in modo standardizzato tramite il protocollo HTTP, che verrà analizzato nella prossima sezione.

[12] `http://en.wikipedia.org/wiki/URL`

1.2.2 Il protocollo HTTP

Il Web, come presentato nella sezione precedente, può essere immaginato come un insieme di client e server che collaborano tra loro per reperire i contenuti richiesti parlando il medesimo linguaggio: il protocollo HTTP. Il protocollo HTTP (Hyper-Text Transfer Protocol) è un protocollo di livello applicativo per sistemi distribuiti e collaborativi, che standardizza le modalità per le richieste e la trasmissione delle risorse Web attraverso Internet.
Fin dal 1990 HTTP è stato utilizzato per il Web, prima con una versione molto semplice (HTTP 0.9) che consentiva esclusivamente il trasferimento di dati attraverso Internet, in seguito con una versione più articolata (HTTP 1.0[13]) in grado di supportare i messaggi MIME (Multipurpose Internet Mail Extensions) [14], ed infine con l'attuale versione HTTP 1.1[15]. Quest'ultima versione introduce alcune funzionalità e migliora le prestazioni del protocollo in seguito all'introduzione, nel mondo Web, di meccanismi quali virtual hosting, caching e proxies gerarchici.
Nei prossimi paragrafi verrà analizzato in dettaglio il protocollo HTTP 1.1, in quanto rappresenta la versione attuale e più utilizzata in ambito web, nonché il protocollo di riferimento utilizzato nell'ambito del progetto PariWEB per la realizzazione del web server.

1.2.2.1 HTTP 1.1

HTTP 1.1 è un protocollo request/response di tipo testuale basato sullo scambio di messaggi, la cui forma generica può essere così descritta:

```
Start-line CRLF
*[Message-header CRLF]
CRLF
[Message-body CRLF]
```

dove `CRLF` simboleggia l'accoppiata *carriage-return* e *line-feed*. La prima riga (`Start-line`) indica se si tratta di un messaggio di richiesta o di un messaggio di risposta, mentre il successivo insieme di righe (identificato con `Message-header`) specifica una raccolta di parametri e opzioni che qualificano la richiesta o la risposta. Vi possono essere zero o più di queste righe, terminate da una riga vuota che indica il termine di quella che viene indicato come l'header delle richiesta HTTP. Infine, dopo la riga vuota possiamo trovare il contenuto del messaggio (`Message-body`), che solitamente è vuoto nel caso di messaggi di richiesta. Nei prossimi paragrafi analizzeremo nello specifico la struttura dei messaggi di richiesta e di risposta previsti dal protocollo.

[13] [2]
[14] `http://en.wikipedia.org/wiki/MIME`
[15] [1]

Messaggi di Richiesta

Un messaggio di richiesta HTTP 1.1 si presenta nella seguente forma:

```
Request-line
*((General-header | Request-header | Entity-header) CRLF)
CRLF
[Message-body]
```

La prima riga di un messaggio HTTP 1.1 è così strutturata:

```
Request_line = Method SP Request-URI SP HTTP-Version CRLF
```

in cui `HTTP-Version` rappresenta la **versione** del protocollo utilizzata, nella forma "HTTP/" 1*DIGIT "." 1*DIGIT. Il campo `Method` contenuto nella prima riga della richiesta indica, invece, uno degli **otto metodi** definiti dal protocollo:

- **GET**: richiede il recupero della risorsa identificata dal campo `Request-URI`;
- **HEAD**: richiede una risposta identica a quella che corrisponderebbe ad una richiesta GET, ma senza il corpo del messaggio. E' utile per recuperare le metainformazioni (headers), senza dover trasportare l' intero contenuto;
- **OPTIONS**: richiede quali metodi http il server supporta per la risorsa specificata. Può anche essere usato per richiedere le funzionalità del web server inserendo il carattere "*" al posto del `Request-URI`;
- **POST**: fornisce al server dei dati, includendoli nel `Message-body` della richiesta. Può comportare la creazione di una nuova risorsa o il suo aggiornamento;
- **PUT**: carica sul server la risorsa trasmessa all'interno del `Message-body`;
- **DELETE**: cancella dal server la risorsa specificata nel campo `Request-URI`;
- **TRACE**: effettua l'eco del messaggio di richiesta. Può essere usato per vedere i cambiamenti o le aggiunte fatte nella richiesta dai server intermedi;
- **CONNECT**: converte la connessione richiesta in un tunnel tcp/ip trasparente, normalmente per facilitare connessioni SSL criptate (https) attraverso proxy.

Questi metodi possono essere classificati in due categorie: metodi sicuri e metodi di non sicuri. I metodi HEAD, GET, OPTIONS e TRACE sono definiti **sicuri**, in quanto la loro funzione è volta solo al recupero di informazioni e quindi non modificano lo stato del server. Diversamente, i metodi POST, PUT e DELETE comportano azioni che possono cambiare lo stato del server, perciò devono essere implementati con maggiore attenzione.

La **Uniform Resource Identifier**, `Request-URI`, identifica la risorsa richiesta, e può essere nella seguente forma:

```
Request-URI = "*" | absoluteURI | abs-path | authority
```

Le quattro opzioni dipendono dalla natura della richiesta. L'asterisco indica che la richiesta non fa riferimento ad una particolare risorsa ma al server intero, e può essere utilizzato solo con metodi che non si applicano necessariamente ad una risorsa (per esempio il metodo OPTIONS). La forma con l'`absoluteURI`, che specifica sia il dominio che il percorso del file, è necessaria quando la richiesta è effettuata da un proxy, mentre la forma con `authority` è utilizzata solo con il metodo CONNECT . La forma più comune per la `Request-URI` è quella che specifica l'`abs-path`, cioè il path assoluto che individua la risorsa del web server a cui fa riferimento la richiesta. In questo caso il dominio, che indica la locazione della risorsa nella rete, deve essere trasmesso nel campo dell'header denominato `Host`.

Di seguito alla `Request-line` è possibile aggiungere uno o più campi **header** per fornire al server informazioni aggiuntive. La struttura di un campo dell'header è la seguente:

```
Field-name ":" [ field-value ]
```

I campi header consentiti si dividono in `General-header`, `Request-header` e `Entity-header`. I `General-header` possono essere utilizzati sia per messaggi di richiesta che per messaggi di risposta e specificano particolari comportamenti del protocollo che vengono applicati relativamente al solo messaggio trasmesso. Alcuni tra i `General-header` più comuni sono:

- `Cache-Control`: specifica le direttive da utilizzare per il meccanismo di cache;
- `Connection`: specifica il tipo di connessione preferita dal client;
- `Date`: indica la data e l'ora a cui è stato inviato il messaggio.

I `Request-header` consentono di fornire al server informazioni aggiuntive sulla richiesta e sul client stesso. Alcuni esempi di Request-header sono:

- `Accept`: specifica i Content-Types accettabili;
- `Authorization`: contiene le credenziali necessarie al processo di autenticazione;
- `Accept-Encoding`: elenco delle codifiche accettate dal mittente del messaggio;
- `Accept-Language`: elenco delle lingue accettate per la risposta;
- `Host`: specifica il nome di dominio a cui fa riferimento la risorsa;
- `User-Agent`: descrizione dello user-agent (browser) che ha prodotto la richiesta;

Gli `Entity-header`, invece, definiscono metainformazioni relative al corpo del messaggio o, se questo non è presente, riguardo la risorsa identificata dalla richiesta. Gli header di questo tipo più comuni sono:

- `Allow`: elenco delle azioni consentite per la risorsa specificata nella richiesta;
- `Content-Encoding`: indica il tipo di codifica utilizzate per i dati della risposta;
- `Content-Language`: indica la lingua del contenuto della risposta;
- `Content-Length`: indica la lunghezza del corpo del messaggio;
- `Content-Location`: specifica la posizione dei dati contenuti nel messaggio;
- `Content-Type`: il tipo di MIME del contenuto del messaggio;

- `Expires`: indica la data dopo la quale il contenuto è da considerarsi obsoleto;
- `Last-Modified`: indica la data di ultima modifica dei dati della risposta.

La forma tipica di una richiesta HTTP/1.1 sarà quindi:

```
GET / HTTP/1.1
Host: www.paripari.it\r\n
Connection: keep-alive\r\n
Cache-Control: max-age=0\r\n
User-Agent: Mozilla/5.0 (Windows;U;Windows NT 6.1;en-US) Chrome/10.0.648.127\r\n
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain\r\n
Accept-Encoding: gzip,deflate,sdch\r\n
Accept-Language: it-IT,it;q=0.8,en-US;q=0.6,en;q=0.4\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3\r\n
If-Modified-Since: Fri, 15 May 2009 08:56:30 GMT\r\n
\r\n
```

Messaggi di Risposta

Una volta ricevuto ed interpretato un messaggio HTTP di richiesta, un web server produce un messaggio di risposta così strutturato:

```
Status-line
*((General-header|Request-header|Entity-header) CRLF
CRLF
[Message-Body]
```

La prima riga del messaggio di risposta consiste nella versione del protocollo, seguita da uno `Status-Code` e dal relativo messaggio testuale.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Lo `Status-Code` è un codice di tre cifre che descrive il risultato del tentativo di interpretare ed eseguire la richiesta ricevuta da parte del server. I codici si dividono in cinque classi, che si distinguono tramite la prima cifra dello `Status-Code`:

Code	Categoria	Descrizione
1xx	**Informational**	La richiesta è stata ricevuta correttamente e il server la sta eseguendo.
2xx	**Success**	La richiesta è stata ricevuta, interpretata, accettata ed eseguita correttamente del server.
3xx	**Redirection**	Perché la richiesta venga completata, il client deve eseguire delle azioni aggiuntive.
4xx	**Client Error**	La richiesta contiene errori di sintassi o non può essere soddisfatta per motivi imputabili al client.
5xx	**Server Error**	Il server non è riuscito ad eseguire una richiesta apparentemente valida.

Tabella 1.1: Classificazione dei messaggi di risposta Http 1.1.

Nella tabella 1.2 inserita a fine di questo capitolo, invece, è possibile consultare l'elenco completo degli `Status-Code` previsti dal protocollo HTTP/1.1, con una breve descrizione del loro significato.

Sempre in analogia con i messaggi di richiesta, quelli di risposta possono contenere una o più righe nel `Message-Header`, per fornire al client ulteriori informazioni. Infine, nel caso più comune in seguito a richieste del tipo GET, il messaggio di risposta conterrà al proprio interno la pagina richiesta codificata in modo opportuno. Un possibile messaggio di risposta alla richiesta riportata nella sezione precedente potrebbe essere il seguente:

```
HTTP/1.1 200 OK\r\n
Server: PariPariWebServer\r\n
Last-Modified: Mon, 14 Feb 2011 06:43:40 GMT\r\n
Date: Thu, 10 Mar 2011 18:20:22 GMT\r\n
Expires: Fri, 11 Mar 2011 18:20:23 GMT\r\n
Vary: Accept-Encoding\r\n
Cache-Control: max-age=86400\r\n
Content-Type: text/html\r\n
Content-Length: 5692\r\n
Keep-Alive: timeout=15, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html\r\n
\r\n
... Pagina richiesta
\r\n
```

Novità introdotte

Come accennato all'inizio di questo capitolo, il protocollo Http 1.1 è stato sviluppato al fine di migliorare le prestazioni della versione 1.0 [2] ed introdurre alcune funzionalità molto utili. Di seguito verranno brevemente presentate le novità più significative introdotte in questa versione del protocollo.

- **Identificazione Hostname**
  Ogni richiesta Http 1.1 deve identificare l'hostname della richiesta, cioè il nome DNS del server che possiede il contenuto della richiesta. A differenza della versione 1.0, il protocollo prevede che l'hostname sia passato al server sotto forma di URI all'interno della `Request_line`, 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 virtuali**, cioè permette che più domini siano ospitati dallo stesso server, e quindi puntino allo stesso indirizzo IP. A differenza delle versioni precedenti del protocollo che non prevedevano l'invio dell'hostname all'interno della richiesta, con la versione 1.1 il server è in grado di gestire correttamente le richieste in ingresso anche in caso di *multidomain*.
- **Connessioni Persistenti**
  Al giorno d'oggi, la maggior parte delle pagine web sono costituite da più elementi come immagini, suoni e video. Con le precedenti versioni del protocollo queste pagine venivano caricate molto lentamente, in quanto ogni singolo elemento della pagina veniva richiesto separatamente al server, instaurando una

nuova connessione. In questo modo, per caricare completamente una pagina web, il browser doveva ripetere in modo sequenziale le seguenti operazioni: connessione al server, richiesta del documento, attesa della risposta ed infine disconnessione.
Per evitare questo effetto, Http 1.1 definisce le **connessioni persistenti**, in modo che i diversi componenti di una pagina vengano richiesti attraverso un'unica connessione. Http 1.1 assume per default che le connessioni siano persistenti e, a meno che il browser che effettua la richiesta non espliciti il contrario all'interno dell'header della richiesta, il server assume di poter ricevere richieste multiple mediante un'unica connessione, mantenendo il collegamento attivo per un certo periodo anche dopo aver eseguito la prima richiesta pervenuta.
Questo accorgimento si traduce in un notevole aumento delle performance del protocollo (in quanto non vengono sprecate banda e tempo per le continue fasi di connessione e disconnessione dal server), a scapito di un lieve aumento della complessità del web server.

– **Chunked Transfers**
Normalmente, quando viene generata una risposta con una specifica risorsa, il server conosce la sua lunghezza ed è in grado di settare in modo corretto il campo `Content-Length` dell'header; nel caso di contenuti creati dinamicamente, però, questo compito non è altrettanto banale, sia per le possibili grosse dimensioni dei contenuti, che per il tempo necessario alla valutazione della lunghezza di una risorsa che viene creata dinamicamente. Questo aspetto rappresenta un problema quando vengono utilizzate le connessioni persistenti, dato che per poter essere utilizzate in modo corretto esse prevedono che il server conosca la lunghezza del contenuto del messaggio (campo `Content-Length` dell'header).
Per risolvere questo aspetto, la versione 1.1 propone una soluzione, denominata *chunked-encoding method*, che prevede la possibilità per il server di inviare la risposta in piccole porzioni (o chunk) di cui è in grado di calcolare la lunghezza. In questo modo i web server possono inviare contenuti dinamici di dimensioni considerevoli o prodotti troppo lentamente senza la necessità di disabilitare le connessioni persistenti.

– **Ulteriori caratteristiche**
Oltre a quelle riportate nelle sezioni precedenti, la versione 1.1 ha introdotto altre utili novità tra cui, per esempio, la possibilità di richiedere ad un web server solo parte di un contenuto, specificando il range di byte di interesse attraverso il campo Range dell'header. Http 1.1 inoltre, fornisce nuove funzionalità che permettono di formulare richieste condizionali in modo da ridurre il traffico nella rete sfruttando al meglio i **proxy** ed i meccanismi di **cache**.

Tabella 1.2: Status-Code Http 1.1

Code	Status Name	Descrizione
1xx	**Informational**	
100	Continue	Il client può continuare con la sua richiesta.
101	Switching Protocol	Il server ha ricevuto una richiesta per il cambiamento del protocollo per la connessione.
2xx	**Success**	
200	OK	La richiesta è stata accolta, il server risponde con i dati richiesti.
201	Created	La richiesta è stata eseguita ed ha creato una nuova risorsa all'URI contenuto nella risposta.
202	Accepted	La richiesta è stata accettata ma non ancora processata.
203	Non-Authoritative	L'insieme delle informazioni rimandate sono una copia fatta in locale o da terzi.
204	No Content	Il server ha effettuato la richiesta ma non si necessita l'invio dell'entity-body.
205	Reset Content	Il browser dovrebbe resettare il contenuto del form che ha causato l'invio della richiesta.
206	Partial Content	Il server ha effettuato un GET parziale della risorsa in risposta ad un Range header.
3xx	**Redirection**	
300	Multiple Choices	L'URI richiesto corrisponde a più documenti.
301	Moved Permanently	La risorsa richiesta è stata assegnata definitivamente ad un nuovo URI.
302	Moved Temporarily	La risorsa richiesta è stata assegnata temporaneamente ad un nuovo URI.
303	See Other	La risorsa richiesta si trova in un altro URI specificato nel Location header.
304	Not Modified	In risposta ad una richiesta GET condizionale, ma la risorsa non è stata ancora modificata.
305	Use Proxy	La risorsa richiesta deve passare per un proxy il cui l'URI è dato nel Location field.
306	Switch Proxy	Questo codice è inutilizzato.
307	Temporary Redirect	La risorsa richiesta si trova temporaneamente all'URL specificato nel Location header.
4xx	**Client Error**	
401	Authorization Required	Indica che la richiesta era sprovvista dell'autorizzazione richiesta.
402	Payment Required	Questo codice è riservato ad usi futuri.
403	Forbidden	Il server capisce la richiesta ma si rifiuta di compierla.
404	Not Found	Il server non ha trovato nulla che corrisponda all'URI richiesto.

Continua nella pagina successiva

Continua dalla pagina precedente

Code	Status Name	Descrizione
405	Method Not Allowed	Il metodo specificato nella request line non è disponibile per l'URI richiesto.
406	Not Acceptable	La risorsa richiesta genera una risposta incompatibile con gli headers della richiesta.
407	Proxy Authentication Required	Il client deve autenticarsi con il proxy, usando l'header Proxy-Authenticate.
408	Request Timed Out	Il client non ha fornito una richiesta nel tempo massimo di attesa del server.
409	Conflict	La richiesta non può essere completata causa conflitto con il corrente stato della risorsa.
410	Gone	La risorsa non è più disponibile e il server non conosce indirizzi su cui ridirezionare.
411	Length Required	Il server non accetta la richiesta in quanto il client non ha definito il Content-Length.
412	Precondition Failed	Una o più condizioni specificate negli headers è risultata falsa durante il test del server.
413	Request Entity Too Large	La richiesta è più grande rispetto a quello che il server può processare.
414	Request URI Too Long	L'URI richiesto è troppo lungo per essere interpretato dal server.
415	Unsupported Media Type	Il corpo della richiesta è in un formato non supportato.
416	Requested Range Not Satisfiable	La porzione di file richiesta non è valida.
417	Expectation Failed	Il server non può soddisfare i requisiti contenuti nel campo Expect della richiesta.
5xx	**Server Error**	
500	Internal Server Error	Il server è in una situazione inaspettata e non può rispondere alle richieste.
501	Not Implemented	Il server non è implementato per rispondere correttamente alla richiesta effettuata.
502	Bad Gateway	Il server ha ricevuto una risposta non valida dal gateway/proxya a cui è collegato.
503	Service Unavailable	Il server non può rispondere causa temporaneo overload.
504	Gateway Timeout	Il server, collegato ad un gateway, non riceve una risposta nel tempo di attesa massimo.
505	HTTP Version Not Supported	Il server non supporta la versione del protocollo HTTP utilizzato.

Capitolo 2

Obiettivi del progetto

Il termine web server è utilizzato per indicare un processo in esecuzione su un computer responsabile di accettare richieste http provenienti dai client e di servirle con risposte http contenenti le risorse richieste (solitamente documenti html ed oggetti collegati).
L'obiettivo finale dell'attività di tesi qui documentata è quello di realizzare un plug-in che realizzi un web server distribuito in grado di sfruttare le potenzialità della rete PariPari per fornire un servizio di web hosting completo.
Nei prossimi paragrafi verranno brevemente descritti gli obiettivi ritenuti fondamentali per ottenere il risultato finale prefissato, individuati all'inizio dell'attività di tesi.

2.1 Http 1.1 Compliant

Il web server deve essere in grado di gestire correttamente le richieste in ingresso per tutti i metodi previsti da Http 1.1 e di intraprendere i comportamenti e le azioni adeguate secondo quanto specificato nell'RFC 2616 [1], in particolare per quanto riguarda la generazione di messaggi di risposta con i relativi status code. In questo modo il servizio potrà essere utilizzato con tutti i principali browser presenti in rete.

2.2 Stabilità

La stabilità rappresenta un aspetto fondamentale, in quanto, perché il servizio possa essere utilizzato, il plug-in deve essere dotato di una struttura robusta che garantisca all'utente la disponibilità e la persistenza dei contenuti ospitati, nonché una certa tolleranza ad errori e comportamenti anomali.

2.3 Sicurezza

Per un web server la sicurezza è cruciale e comprende molti aspetti differenti, per cui è necessario definire una politica di sicurezza (security policy) che descrive esattamente quali azioni possono essere intraprese all'interno del sistema e quali sono

quello proibite. La politica adottata in PariWEB sfrutta i meccanismi di sicurezza forniti da Http 1.1 per la protezione dei contenuti trasmessi all'interno dei messaggi Http e per l'autenticazione degli utenti (verificare l'identità di un utente, valutando che esso sia chi dice di essere), unitamente a meccanismi appositamente realizzati per il **controllo degli accessi** (verificare quali sono le operazioni consentite ad un certo utente) e per far fronte ad attacchi di tipo **denial of service** (DoS[1]).

2.4 Performance

Fin dalle fasi iniziali del progetto è necessario tenere presente che uno degli obiettivi di PariWEB è quello di avere delle buone prestazioni. Gli indicatori di performance utilizzati in questo caso sono il tempo medio necessario a processare una richiesta (in differenti condizioni di carico) e l'utilizzo di memoria e di CPU da parte del plug-in.

2.5 Configurabilità

Questo aspetto risulta molto importante in quanto, agli occhi dell'utente finale, avere a disposizione un servizio completamente configurabile rappresenta un notevole punto di forza. Nel caso di web hosting, la configurabilità si traduce nella possibilità di definire utenti autorizzati e gruppi di utenti per ciascun dominio ospitato sul server, nonché l'opportunità di specificare diverse politiche di accesso per ogni singola risorsa.

[1] `http://en.wikipedia.org/wiki/Denial-of-service_attack`

Capitolo 3

Progettazione

La progettazione è quel processo che, prima della fase di realizzazione e sviluppo del codice, permette di definire un modello ed una struttura in grado di rispettare requisiti e obiettivi individuati.

Nel caso oggetto del presente documento, la fase di progettazione del plug-in è influenzata in modo determinante, oltre che dagli obiettivi presentati nel capitolo precedente, anche dalla natura distribuita dell'ambiente in cui il servizio di web server va inserito. Un utente, per esempio, potrebbe volere realizzare un web server locale, cioè esclusivamente sulla propria macchina senza essere necessariamente connesso ad Internet, o, allo stesso modo, voler fornire un servizio di web server pubblico e distribuito con caratteristiche sostanzialmente diverse dal primo (si pensi per esempio alla diversa disponibilità di servizio richiesta). Ovviamente, seppure il servizio in questione sia lo stesso, per ottenere risultati soddisfacenti, in termini di raggiungimento degli obiettivi, le due situazioni comportano scelte progettuali diverse. Per ovviare a questo, il plug-in Webserver prevede due diverse modalità di funzionamento, una denominata **"modalità locale"** ed una chiamata **"modalità distribuita"**.

Nei prossimi paragrafi verranno presentati alcuni aspetti affrontati nella fase di progettazione del plug-in, in particolare verrà descritta la struttura interna utilizzata per il web server (sezione 3.1) e le scelte adottate per le due diverse modalità di utilizzo consentite da Webserver (sezione 3.2) e (sezione 3.3).

3.1 Processing delle richieste

La struttura interna del plug-in Webserver deve presentare alcune caratteristiche di fondamentale importanza per un progetto con le proprietà e le dimensioni di PariPari. Tra queste vi è sicuramente la **modularità**, che permette di estendere e modificare i singoli moduli che costituiscono il server senza influenzare gli altri componenti e che consente di testare il codice in modo più semplice e veloce.
Per questi motivi l'architettura adottata per il processo di esecuzione delle richieste in ingresso al server è composta da moduli indipendenti, ognuno responsabile di una singola funzionalità.

La soluzione adottata per il plug-in Webserver prevede che le richieste in ingresso vengono processate secondo lo schema riportato in figura 3.1.

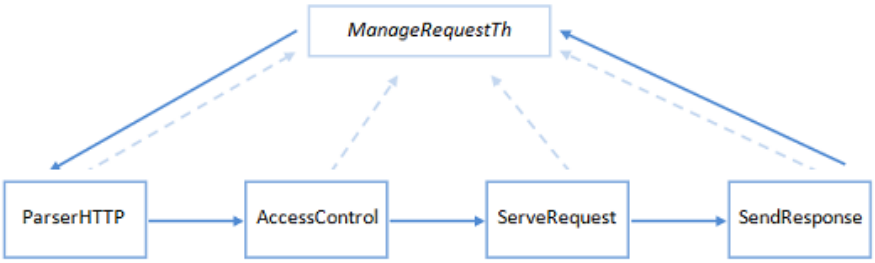


Figura 3.1: Architettura modulare per il processing delle richieste HTTP.

Per ogni richiesta pervenuta al server 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 verificando che sia in forma corretta, il modulo `AccessControl` verifica che l'operazione sia consentita, il modulo `ServeRequest` esegue la richiesta generando la risposta, ed infine il modulo `SendResponse` si occupa dell'invio della risposta al client. Nei prossimi paragrafi verranno analizzati nel dettaglio gli aspetti di progettazione relativi ai moduli qui introdotti.

3.1.1 ParserHTTP

`ParserHTTP` è il primo modulo attivato dal thread incaricato di processare la richiesta, e si occupa di farne il parsing, cioè di leggere l'input proveniente dal client determinandone la struttura e la correttezza. Il parser deve 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, in quanto le performance di questo componente del server influenzano direttamente quelle dell'intero web server. Inoltre, una volta determinata la struttura delle richiesta Http, essa deve essere salvata in modo da rendere efficiente l'accesso da parte degli altri moduli incaricati di eseguire la richiesta.

La scelta adottata consiste nella lettura dell'header della richiesta dallo stream di input "byte per byte" una riga alla volta. Prima di procedere con il parsing della successiva, la riga letta viene processata al fine di evitare la lettura dell'intera richiesta nel caso di una struttura non corretta. Il processing è differente a seconda che si tratti della `Start-Line` o delle righe `Message-Header`, ma in entrambi i casi individua i valori dei campi che compongono una richiesta Http e popola un'hashtable con coppie <*NomeCampo*, *ValoreCampo*>. L'utilizzo di questa struttura consente di accedere al valore desiderato in tempo costante, semplicemente utilizzando il nome del campo desiderato. Terminata la lettura dei `Message-Header`, se presente, viene letto il contenuto del `Message-Body`, anche in questo caso "byte per byte" ma per una lunghezza massima pari al valore dell'header `Content-Length`, e inserito nella hashtable con chiave "*AdditionalData*".

Questa tabella è il risultato del processo di parsing e viene restituita al thread responsabile esclusivamente se la richiesta si presenta in formato corretto. Se nella fase di processing si verifica un errore o se la richiesta non è correttamente strutturata, il modulo ParserHTTP genera un'eccezione con cui comunica lo Status-Code che il `ManageRequestTh` chiamante dovrà comunicare al client.

3.1.2 AccessControl

Il modulo `AccessControl` ha il compito di determinare se la richiesta ricevuta dal server può essere eseguita sulla base delle politiche di accesso definite dall'amministratore del dominio. Questo componente estrae dalla tabella restituita dal modulo di parsing il metodo Http e la risorsa richiesta, e restituisce ALLOW nel caso in cui l'operazione è conforme alle regole definite, DENY altrimenti.
L'obiettivo che si prefigge il plug-in Webserver è quello di implementare un modulo di AccessControl che consenta di definire delle politiche di accesso complete e adatte all'ambiente web.

Innanzitutto, ai fini della progettazione del modulo, è necessario definire i concetti di autorizzazione e di operazione.

```
Operazione = Azione + [User]
Autorizzazione = ALLOW/DENY + Operazione
```

Un'**operazione** è costituita da un'`Azione` ed opzionalmente da un `User`. Le azioni previste dal plug-in riguardano la possibilità di rendere visibile il contenuto (GET) e di consentire o meno l'esecuzione dei metodi POST , PUT e DELETE . Se l'utente non viene specificato nella definizione dell'operazione l'autorizzazione consente o nega indistintamente l'azione associata, altrimenti è possibile definire regole specifiche per un User, sia esso un singolo utente oppure un gruppo di utenti definito per il dominio a cui appartiene la risorsa. Un'**autorizzazione**, invece, fa direttamente riferimento ad una risorsa del web server, e definisce se una specifica operazione è consentita (ALLOW) oppure è vietata (DENY).

Una volta definiti questi concetti è necessario adottare un modello secondo cui realizzare il modulo AccessControl (vedi [18]). La soluzione adottata prevede un modello di accesso **Unix-like**, secondo il quale ogni utente, al momento di una richiesta, può appartenere ad un solo gruppo e ogni risorsa ha un utente proprietario (generalmente l'utente che ha caricato la risorsa sul server) ed un gruppo (solitamente il gruppo a cui appartiene il suo proprietario). Il modello prevede che le autorizzazioni possono essere specificate per tre differenti categorie:

- per il proprietario del contenuto,
- per il gruppo a cui appartiene la risorsa,
- per "il resto del mondo", cioè tutti gli utenti che non rientrano nelle prime due categorie.

Il plug-in Webserver associa ad ognuna di queste categorie due insiemi, quello delle regole ALLOW e quello delle regole DENY . Questo approccio consiste nella combinazione della *closed policy*, che consiste nello specificare le operazioni consentite, con la *open policy*, che si basa sulla definizione delle operazioni vietate.
L'uso di permessi positivi e negativi può sembrare un'inutile complicazione, ma risulta particolarmente adatto per definire eccezioni e casi particolari, che nell'ambito di un web server sono abbastanza comuni. Per esempio si potrebbe volere consentire l'accesso ad una risorsa (metodo GET) a tutti i membri di un gruppo composto da cento utenti, ad eccezione di uno specifico membro X. Utilizzando un'approccio di tipo closed policy, sarebbe necessario definire un'autorizzazione per ogni membro del gruppo ad eccezione di X, invece combinando le due tecniche è possibile specificare la regola definendo un'autorizzazione positiva per il gruppo ed una negativa per l'utente X.
L'introduzione di un approccio di questo tipo comporta due tipi di problematiche: l'*incompletezza*, cioè cosa fare nel caso in cui non siano specificate autorizzazioni, e l'*inconsistenza*, cioè cosa fare se vi sono state definite sia autorizzazioni positive che negative per una certa operazione.
Per risolvere il problema dell'incompletezza, il plug-in Webserver utilizza la seguente tecnica: se non è stata definita la politica d'acceso per una risorsa viene utilizzata, se esiste, quella definita per la cartella che la contiene, altrimenti quella impostata per il relativo dominio (che deve essere definita obbligatoriamente); se la politica è stata definita ma non vi sono regole riguardo l'operazione il modulo restituisce l'autorizzazione positiva o negativa a secondo del valore di default impostato.
Il problema dell'inconsistenza, invece, è stato risolto con due azioni: la prima prevede un controllo di consistenza al momento della definizione delle politiche di accesso per verificare che la regola definita e la sua negazione non siano già presenti, la seconda consiste nel considerare come valida l'autorizzazione più specifica. In particolare quest'ultima convenzione implica che le regole che definiscono un utente prevalgono sulle regole che definiscono esclusivamente un'azione.

Il modello di AccessControl progettato, inoltre, consente all'utente di definire per ogni risorsa la precedenza secondo la quale devono essere valutate le autorizzazioni:

- se l'ordine specificato è "deny, allow" le regole negative vengono valutate per prime e l'operazione è consentita per default nel caso in cui non vi sia corrispondenza tra i divieti o vi siano autorizzazioni positive,

- se l'ordine è "allow, deny" le regole positive vengono controllate per prime e l'operazione è negata per default se non è stata riscontrata nessuna corrispondenza con autorizzazioni positive o se vi sono divieti che non consentono di autorizzare l'azione.

Un diverso ordine utilizzato nella fase di controllo dei permessi può incidere notevolmente sulle prestazioni del modulo AccessControl a seconda del numero e di come sono definite le autorizzazioni.

3.1.3 ServeRequest e SendResponse

`ServeRequest` è il modulo del web server che ha il compito di eseguire le richieste ricevute, producendo il messaggio Http di risposta. Questo componente viene attivato dal thread esclusivamente se il modulo AccessControl restituisce ALLOW, cioè se l'operazione è conforme alla politica di accesso definita per la risorsa coinvolta. ServeRequest riceve l'hashtable che descrive la richiesta prodotta dal modulo Parser-HTTP, individua la risorsa coinvolta, esegue il metodo Http richiesto e compila una seconda hashtable che contiene i campi di cui si compone il messaggio di risposta. Se la richiesta non può essere portata a termine a causa di un errore, il modulo genera un'eccezione con cui comunica al thread responsabile lo Status-Code da utilizzare per la risposta al client.

`SendResponse`, invece, è il componente che si prende carico di inviare la risposta al client che ha effettuato la richiesta, sia nel caso in cui il processing sia andato a buon fine, sia nel caso in cui sia stata generata un'eccezione e il messaggio consista nello Status-Code e nel dettaglio dell'errore.
Questo modulo invia la risposta estraendo un campo alla volta dalla tabella prodotta dal componente ServeRequest.

3.2 Modalità Locale

Nel caso in cui l'utente di PariPari voglia realizzare un web server costituito da un unico nodo, il plug-in viene avviato in modalità locale che consente di attivare il servizio di hosting sulla porta specificata nel file di configurazione del plug-in stesso. Il servizio è quindi disponibile specificando l'IP pubblico del nodo, se il nodo è collegato in rete, oppure l'indirizzo IP di loop-back (http://127.0.0.1) se non è disponibile un collegamento Internet.

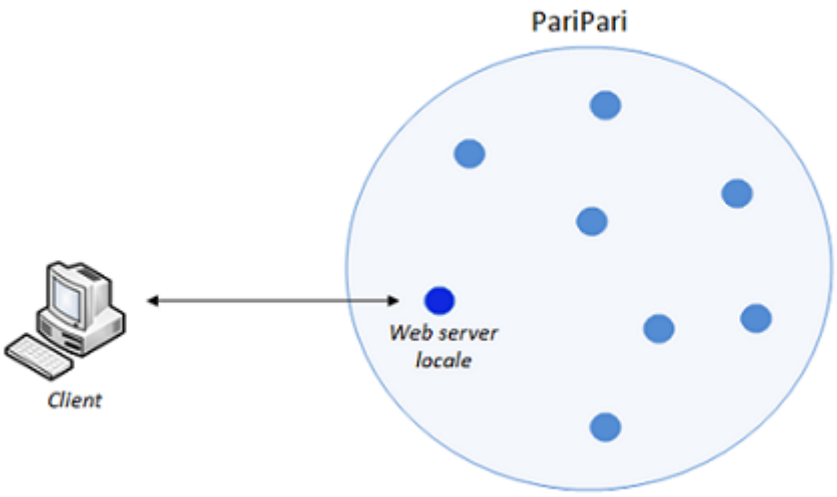


Figura 3.2: Web server locale all'interno di PariPari.

Questa modalità di utilizzo rappresenta, se vogliamo, un *caso particolare della modalità distribuita*, e con questa visione è stata affrontata la fase di progettazione. Le principali motivazioni di questo approccio si possono riassumere nei due aspetti seguenti:

- essendo il caso locale molto meno complesso di quello distribuito, la possibilità di considerarlo in modo separato permette di ottimizzare l'uso delle risorse (in particolare non sprecare memoria o attivare inutilmente thread utilizzati per la modalità distribuita) e di conseguenza migliorare le prestazioni, quando il plug-in viene attivato in questa modalità,
- ma allo stesso tempo, il fatto di considerarlo un caso particolare della modalità distribuita vincola l'utilizzo delle stesse strutture dati, permettendo di passare in qualsiasi momento dalla modalità locale a quella distribuita e viceversa.

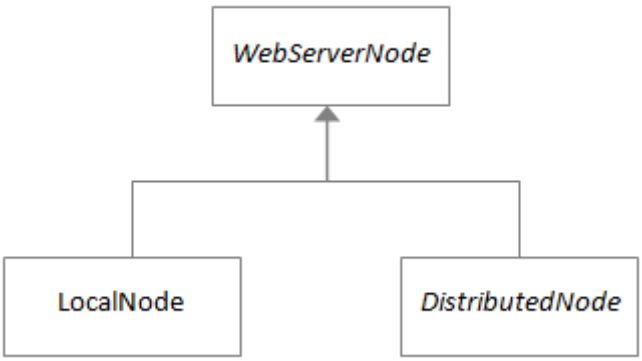


Figura 3.3: Modalità locale e modalità distribuita.

La soluzione adottata è quella riportata in figura 3.3, per cui *WebServerNode* racchiude al suo interno i comportamenti di un'istanza di Webserver comuni ad entrambe le modalità, mentre *LocalNode* e *DistributedNode* includono quegli aspetti caratteristici della modalità di funzionamento.

3.3 Modalità Distribuita

La modalità distribuita rappresenta il funzionamento tipico del plug-in, mettendo a disposizione dell'utente finale un servizio di web hosting che coinvolge più nodi della rete PariPari. 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. I vantaggi di un approccio di questo tipo sono:

- **Persistenza**
  il server così composto non dipende dalla macchina da cui l'utente ha deciso di attivare il server, ma continua ad esistere e fornire il proprio servizio all'interno della rete coinvolgendo in modo dinamico altri nodi.

– **Tolleranza ai guasti**
il server distribuito è in grado di evolvere in modo corretto nonostante il guasto di una o più macchine che lo compongono, e la sua evoluzione avviene in modo trasparente agli occhi dei client.

– **Risorse dinamiche**
in qualsiasi momento è possibile modificare le caratteristiche del server, in termini di spazio di archiviazione o banda disponibile, adattandolo alle esigenze del client o in funzione del carico in ingresso a cui è sottoposto.

Per poter ottenere questi vantaggi caratteristici delle soluzioni distribuite, in fase di progettazione è necessario tenere in considerazione, oltre agli obiettivi elencati nella sezione precedente, ulteriori aspetti quali la **scalabilità** delle soluzioni adottate e la **trasparenza alla distribuzione**. Nei prossimi paragrafi verranno analizzate le scelte ritenute più significative relative alla progettazione della modalità distribuita del plug-in.

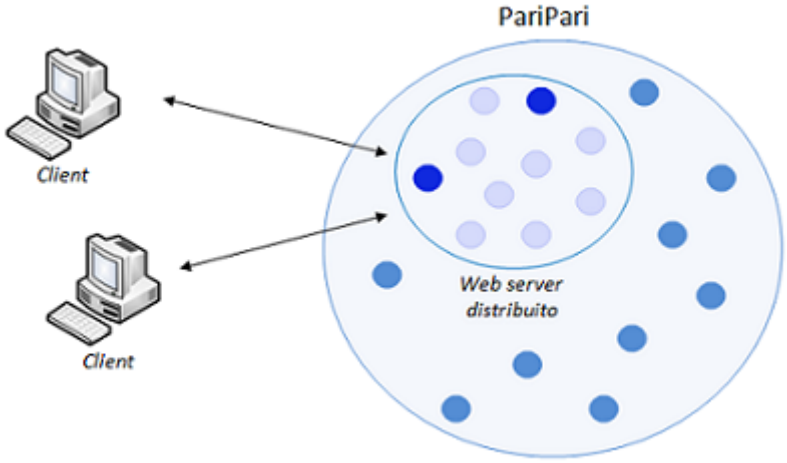


Figura 3.4: Un'istanza di web server distribuito all'interno di PariPari. I nodi chiari del server distribuito sono i nodi Relay, mentre quelli scuri sono nodi Host.

3.3.1 Ridondanza

Una delle soluzioni adottate nella modalità distribuita del plug-in è quella della **replicazione** dei contenuti, al fine di ottenere una ridondanza dei dati utile sia per gli aspetti di tolleranza ai guasti che per una prima forma di bilanciamento del carico.
Il server risulterà costituito da un numero prefissato di nodi "server" uguali, con i medesimi contenuti e le stesse strutture dati, in grado di soddisfare in modo indipendente tutte le richieste Http sicure. Questi nodi, che d'ora in poi chiameremo HostNode, implementeranno un algoritmo di elezione distribuito in modo che vi sia sempre un nodo leader (CoordinatorNode) responsabile del soddisfacimento delle richieste http non sicure (metodi POST , PUT e DELETE) e della sincronizzazione degli altri componenti del server. Nel caso in cui una richiesta non sicura venga

rivolta ad un nodo diverso dal CoordinatorNode, esso semplicemente redirige la richiesta all'HostNode leader in modo trasparente, sfruttando lo status code Temporary redirect di Http 1.1.
Questa soluzione molto semplice permette di garantire una buona tolleranza ai guasti e, sfruttando la possibilità di registrare un dominio con associati più indirizzi IP, fornisce una prima tecnica per il bilanciamento del carico.

3.3.2 Evoluzione dinamica

Per raggiungere l'obiettivo di fornire un servizio di web hosting con le caratteristiche descritte all'inizio di questa sezione è necessario individuare una soluzione da adottare per consentire al server di evolvere in modo dinamico, adattandosi alle esigenze della rete e del carico in ingresso, mantenendo raggiungibili i domini ospitati.
Quando viene creato un nuovo dominio, il suo nome deve essere registrato all'interno dei server per la risoluzione dei nomi (Domain Name Server) associando il nome ad uno o più indirizzi IP necessari per raggiungerlo. Se il server evolve coinvolgendo nuovi nodi, al fine di mantenere raggiungibili i domini, gli indirizzi IP associati devono essere aggiornati [6]. Il problema deriva dal fatto che questa operazione non avviene in modo istantaneo, ma richiede un arco di tempo per la propagazione delle modifiche che può essere anche di alcune ore.

La soluzione utilizzata in PariWEB combina il meccanismo di aggiornamento dei record DNS descritto, con una tecnica ad hoc che sfrutta le funzionalità dal protocollo Http 1.1 e l'aggregazione, all'interno del server distribuito, di molti più nodi rispetto a quelli necessari per fornire le risorse definite dall'utente finale. Le istanze che compongono il server vengono classificate in due categorie: HostNode e RelayNode.

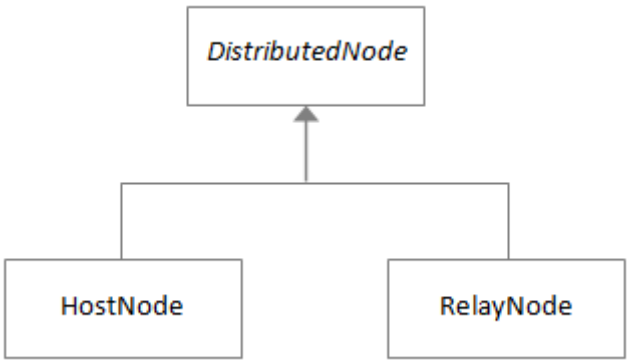


Figura 3.5: Distinzione tra HostNode e RelayNode.

Gli **HostNode** sono in numero limitato rispetto al totale dei nodi che compongono il server, e costituiscono dei web server a tutti gli effetti, ospitando i contenuti dei domini di cui il server è responsabile e tutte le strutture dati necessarie a fornire il servizio.

I **RelayNode**, invece, sono in numero molto maggiore rispetto ai precedenti, ma vengono unicamente utilizzati per reindirizzare verso i nodi Host le richieste che ricevono. Questi nodi non ospitano nessun contenuto e non possiedono nessuna struttura dati, ad eccezione di una lista aggiornata degli HostNode.
Una corretta relazione tra il numero di nodi Host e di nodi Relay potrebbe essere quella di 10 RelayNode per ogni HostNode, mentre il numero di HostNode viene determinato sulla base dei requisiti di spazio di archiviazione, banda e disponibilità di servizio che l'utente definisce al momento della creazione del web server.

Fornendo, come IP associati al dominio, non un solo indirizzo ma gli indirizzi di tutti i nodi aggregati (di entrambi i tipi), il DNS che risolve il nome indirizza la richiesta ad uno nodi del web server (a seconda della politica adottata dal server DNS nel caso di record con più indirizzi IP la richiesta potrebbe essere inoltrata utilizzando la tecnica round robin oppure in modo random). Se il nodo interpellato è un nodo Host, esso riceve, processa e risponde direttamente, mentre se si tratta di un nodo Relay, viene effettuato un redirect della richiesta verso uno degli HostNode. Il redirect è completamente trasparente all'utente, in quanto viene realizzato con lo status code **Temporary Redirect** (codice 307) di Http 1.1; quando il browser riceve un codice 307 in seguito ad una richiesta Http, semplicemente re-invia la richiesta all'indirizzo contenuto all'interno della risposta.

Questa tecnica permette di mantenere raggiungibile il server nell'arco di tempo che intercorre tra due refresh dei record DNS, a condizione che i nodi selezionati inizialmente per formare il server non siano scelti in modo casuale ma secondo particolari criteri atti a identificare, tra quelli disponibili, i nodi più affidabili e con la probabilità maggiore di rimanere attivi in un certo arco di tempo (funzionalità offerta dal modulo DHT della rete PariPari). Infatti, utilizzando una lista di IP adeguatamente lunga, molti degli IP continueranno a far riferimento a nodi del server (Host o Relay) anche durante il processo di evoluzione e la tecnica di refresh potrà essere utilizzata solamente per aggiornare la lista quando il disallineamento sarà elevato.
La soluzione adottata, inoltre, permette di spostare i contenuti del server da un nodo ad un altro (per esempio in seguito ad un guasto o per sfruttare determinate caratteristiche del nodo stesso) senza interrompere il servizio, dato che il redirect da parte dei nodi Relay è eseguito sempre verso uno dei nodi o il nodo che detiene le risorse.

Altri vantaggi derivanti da questa tecnica mista sono individuabili nel fatto che:

- consente di adottare politiche differenti di bilanciamento del carico del server, semplicemente modificando la politica di redirect dei nodi Relay, che di default seleziona casualmente un nodo tra i nodi Host.
- rispetto ad altre soluzioni ad hoc considerate, consente il funzionamento del web server, così come descritto, non solo con il DNS di PariPari ma con qualsiasi DNS.

3.3.3 Bilanciamento del carico

La struttura del plug-in, così come è stata descritta fin qui, consente di bilanciare il carico in funzione di come il DNS reindirizza le richieste ai diversi indirizzi IP associati ad ogni singolo dominio, ed in base al criterio adottato dai nodi Relay per il redirect verso gli HostNode. Questi meccanismi, però, potrebbero non bastare per far fronte a situazioni in cui si verifica un incremento notevole del traffico in ingresso del web server, per esempio nel caso di attacchi DoS.

La soluzione sviluppata sfrutta la presenza dei RelayNode all'interno del server, prevedendo la possibilità che ognuno di questi nodi possa diventare un HostNode in grado di soddisfare direttamente le richieste ricevute. Ogni HostNode monitora il proprio traffico in ingresso, quando questo rimane al di sopra di una certa soglia di guardia (HIGH_TRAFFIC) per un periodo maggiore a quello definito tra i parametri del server viene avviata la seguente procedura:

1. l'HostNode in questione sceglie casualmente uno dei RelayNode che compongono il server e lo contatta,
2. invia al RelayNode selezionati le strutture e i contenuti necessari,
3. il RelayNode, se ha ricevuto correttamente i dati inviati, attiva tutte le strutture necessarie a fornire il servizio e comunica a tutti i nodi del server la sua transizione a HostNode.

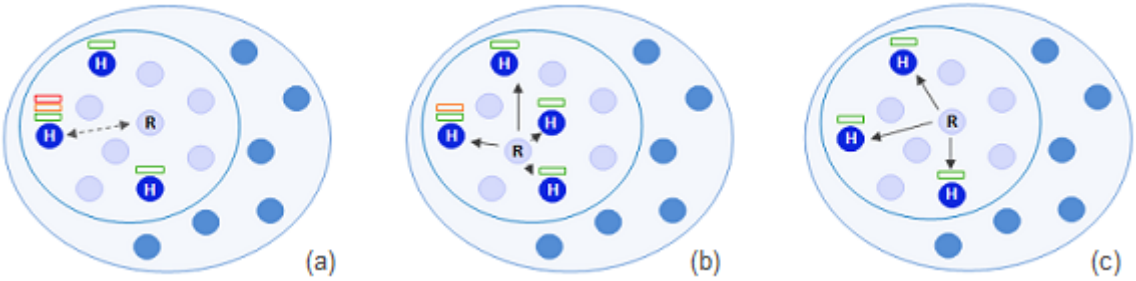


Figura 3.6: (a) scelta del nodo Relay da trasformare in HostNode. (b) suddivisione del traffico tra tutti gli HostNode. (c) ritorno alla situazione iniziale.
A fianco ad ogni HostNode è riportato un indicatore del traffico in ingresso al nodo stesso.

Questo processo, riassunto della figura 3.6(a), permette di aumentare il numero di nodi in grado di rispondere direttamente alle richieste Http 1.1 ricevute, suddividendo il carico in ingresso su più HostNode come riportato 3.6(b) (viene suddiviso sia il traffico diretto che quello derivante dal redirect dei RelayNode ancora presenti). La soluzione prevede anche il procedimento inverso, per cui, quando il traffico in ingresso al nuovo nodo Host rimane al di sotto di una seconda soglia (LOW_TRAFFIC) per un tempo maggiore a quello definito nel file di configurazione, il nodo torna ad essere un RelayNode, liberando lo spazio occupato da strutture e contenuti e comunicando la transizione ai restanti nodi. Quest'ultima fase è riportata nella figura 3.6(c).

Riassumendo, all'aumentare delle richieste in ingresso, il server si estende, aumentando il numero di HostNode che gli permettono di smaltire il traffico, per poi tornare alla dimensione adatta ad una situazione di traffico standard.

3.3.4 Comunicazione tra i nodi

Le scelte adottate per la modalità distribuita del server comportano l'utilizzo di un protocollo di comunicazione che definisca il modo di interagire tra i diversi nodi aggregati, ponendo come obiettivo quello di **minimizzare il traffico sulla rete**. Nella fase di progettazione, perciò, è stato previsto che i singoli nodi che compongono il server distribuito possano essere contattati ad una specifica porta, denominata **InternalPort**, impostabile dall'utente tramite file di configurazione. Inoltre, in questa fase, sono state delineate le procedure basate su scambio di messaggi che consentono di poter realizzare le soluzioni descritte nelle sezioni precedenti.

Il primo aspetto affrontato riguarda il meccanismo da utilizzare per fare in modo che i nodi del server si conoscano l'uno con l'altro. In particolare, è necessario che i nodi Host conoscano tutti i componenti dell'istanza di Webserver, in quanto possono decidere di estendere il server su nodi Relay, mentre i RelayNode devono sapere l'insieme degli HostNode, per poter bilanciare il carico con la tecnica del redirect, ma non necessariamente conoscersi tra loro.
La soluzione utilizzata prevede che l'HostNode che avvia l'istanza del server (e successivamente il nodo leader tra i nodi Host), invii periodicamente un messaggio (denominato I_AM_LEADER) a tutti i nodi aggregati con una doppia finalità: far conoscere il suo ruolo di leader e fornire, ai nodi appena aggregati al server, la lista dei nodi Host che compongono il server. Inoltre, il protocollo definito per il plug-in prevede che non appena un nodo viene aggregato al server come nodo Relay esso invii un messaggio broadcast (denominato HELLO_MSG) con il quale si fa conoscere da tutti i nodi ed in particolare dagli HostNode.

La volontà di poter adattare le dimensioni del server in funzione del traffico in ingresso implica l'utilizzo di comunicazioni interne per gestire la transizione di un RelayNode in un HostNode, e viceversa. Si è scelto di utilizzare due procedure molto simili tra loro, costituite da due fasi:

- nella prima viene utilizzato un messaggio che, inviato al nodo selezionato sulla porta dedicata alle comunicazioni interne del server, avvia la procedura che porta alla transizione di stato del nodo,

- nella seconda fase, una comunicazione broadcast informa tutti i nodi che compongono il server che la transizione è stata completata correttamente.

In particolare, per quanto riguarda il passaggio da RelayNode a HostNode, il primo messaggio spedito al nodo selezionato lo avvisa di rimanere in ascolto sull'InternalPort per ricevere tutte le strutture necessarie a fornire il servizio, inviategli dal nodo che ha avviato la transizione.

L'ultimo aspetto affrontato per la definizione del protocollo di comunicazione riguarda i meccanismi di elezione e di sincronizzazione necessari quando il server è composto da più HostNode. Queste procedure prevedono l'utilizzo di specifici messaggi, in particolare l'aggiornamento degli altri nodi da parte del CoordinatorNode avviene tramite una comunicazione multicast che comunica le azioni necessarie per allineare lo stato a quello del leader.

3.3.5 Libreria DiESeL

Una volta determinate le politiche da adottare per garantire tolleranza ai guasti, evoluzione dinamica e bilanciamento del carico, è importante determinare il modo e i criteri secondo cui verranno selezionati e aggregati i nodi del web server.

In questo caso la scelta adottata prevede l'utilizzo della libreria **DiESeL**[1] (DIstributed Extensive SErver Layer) che permette di gestire la distribuzione di applicazioni di tipo server sulla rete PariPari, indipendentemente dalle funzionalità particolari del programma stesso. La libreria realizza un modulo che va a posizionarsi idealmente tra il livello applicativo (in questo caso il web server) e la rete PariPari, garantendo dei requisiti minimi di scalabilità, trasparenza e stabilità del server distribuito. Inoltre, DiESeL permette di specificare delle feature che descrivono il server e che vengono utilizzate per aggregare i nodi. Nel caso del plug-in Webserver, i parametri utilizzati di default per l'aggregazione sono lo spazio di archiviazione che l'utente vuole mettere a disposizione sul server e la banda disponibile, oltre ad una terza feature che influenzerà il numero totale dei nodi da aggregare (inteso come somma di nodi Host e nodi Relay).

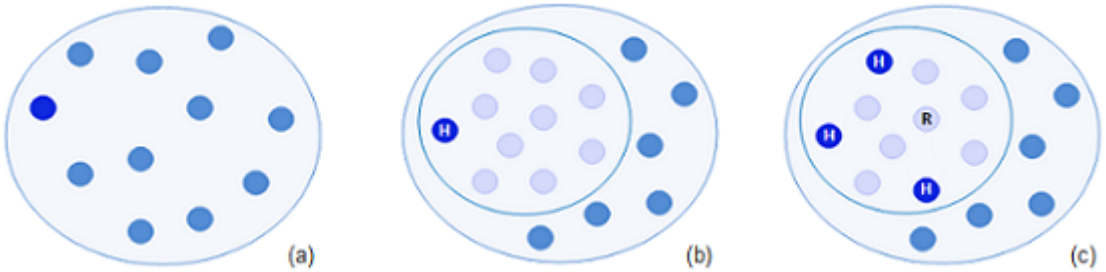


Figura 3.7: (a) avvio del plug-in. (b) aggregazione dei nodi del server gestita dall'istanza attiva di DiESeL dell'HostNode. (c) transizione di un numero adeguato di RelayNode in HostNode fino ad ottenere le caratteristiche desiderate dall'utente.

Un'istanza della libreria DiESeL può essere passiva o attiva: nel primo caso il nodo si rende disponibile ad essere unito ad un server distribuito già esistente come RelayNode, nel secondo caso DiESeL attiva un procedura che si prende carico di selezionare i nodi di PariPari da aggregare al nuovo server distribuito (figura 3.7)(b).

[1] [15] e `http://indy.dei.unipd.it/mediawiki/index.php/DiESeL_(distributore)`

3.3.6 Strutture dati

Le strutture dati utilizzate in Webserver sono state oggetto di numerose riflessioni nella fase di progettazione, in quanto influenzano in modo determinante sia le performance che la stabilità del plug-in stesso. In particolare è importante che la soluzione adottata sia particolarmente efficiente nella fase di accesso ai dati, in quanto nel mondo Web l'operazione più utilizzata è il GET , mentre inserimento, modifica e cancellazione di risorse sono significativamente meno frequenti. Un ulteriore requisito, come accennato nella sezione riguardante la versione locale, riguarda la necessità di utilizzare le stesse strutture dati, sia per il caso locale che per la modalità distribuita, in modo da rendere sempre possibile la transizione da una versione all'altra.

La soluzione scelta per PariWEB, nell'ambito di un singolo dominio, è stata quella di utilizzare come strutture dati delle **hashtable** (nello specifico vengono utilizzate tre tabelle per la gestione: dei contenuti, dei gruppi e degli utenti autorizzati), che consentono di memorizzare coppie chiave-valore. In questo modo, conoscendo l'identificatore della risorsa (per esempio il percorso del file seguito dal nome del contenuto), è possibile accedere in **tempo costante** alle informazioni di interesse o alla risorsa stessa. Le hashtable utilizzate dal plug-in sono state estese in modo tale che venga eseguito automaticamente il salvataggio su disco dopo l'esecuzione di ogni operazione che ne modifica lo stato (insert, delete o update), al fine di garantire la consistenza dei dati. Siccome il web server fornisce un servizio multidomain, cioè permette di gestire più domini, all'avvio del plug-in vengono inizializzate tre **hashtable a due livelli**, che associano al nome del dominio la relativa hashtable.
Questa soluzione è molto flessibile, anche nella prospettiva di partizionare i domini gestiti dal server tra i diversi nodi Host invece che duplicarli su ognuno, in quanto l'inizializzazione di questa struttura individua i domini gestiti dal nodo e carica solamente i dati contenuti nelle singole hashtable relativi ad essi.

Capitolo 4

Realizzazione

In questo capitolo sarà descritta nel dettaglio l'implementazione del plug-in Webserver sulla base delle scelte progettuali discusse nel precedente capitolo.
Nelle prime due sezioni saranno presentate la classe che standardizza le comunicazioni interne al server e le classi di supporto realizzate, in modo che la sezione che analizza in dettaglio la struttura del plug-in risulti più chiara e comprensibile.

4.1 InternalWebServerMessage

Un ruolo centrale nella struttura del plug-in è svolto da **InternalWebServerMessage**, la classe che definisce la struttura dei messaggi scambiati dai nodi del web server in modalità distribuita. Questa classe consente di standardizzare le comunicazioni interne al server, semplificando il protocollo di comunicazione senza limitarne le funzionalità. La struttura di un oggetto `InternalWebServerMessage`, riportata in figura 4.1, comprende i seguenti campi:

- `msgID`: un intero utilizzato come identificatore del messaggio;
- `timeRef`: rappresenta il riferimento temporale del server;
- `msgType`: variabile che identifica il tipo di messaggio;
- `senderAddress`: identifica il mittente del messaggio tramite indirizzo IP e porta su cui è attivo il servizio;
- `receiverAddress`: identifica il destinatario del messaggio tramite indirizzo IP e porta su cui è attivo il servizio;
- `message`: questo campo contiene il corpo della comunicazione. Per ragioni di efficienza, al fine di poter inviare più oggetti all'interno di un unico messaggio e quindi di un'unico header, il campo è un array di Object.

Il protocollo prevede che alcuni di questi messaggi siano utilizzati principalmente per comunicazioni broadcast, mentre altri vengano utilizzati specificatamente in modalità unicast o multicast. In particolare per realizzare **messaggi broadcast** il plug-in sfrutta il servizio messo a disposizione dalla libreria DiESeL

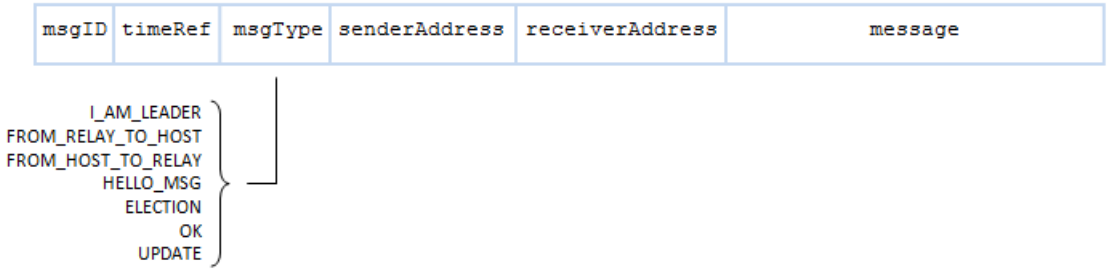


Figura 4.1: Struttura dei messaggi utilizzati per comunicare tra i diversi nodi di un server distribuito.

che consente di contattare tutti i nodi aggregati al server, mentre per le **comunicazioni unicast** (o multicast) viene realizzata una comunicazione punto-punto sull'InternalPort del nodo che si vuole contattare. Di seguito vengono brevemente descritti i tipi di messaggi previsti dal protocollo.

`I_AM_LEADER`

messaggio utilizzato esclusivamente in broadcast dal nodo Host coordinatore per informare periodicamente tutti i nodi che compongono il server che il leader è attivo. Il corpo del messaggio contiene le informazioni necessarie per contattare il nodo coordinatore (indirizzo IP, porta del web server e porta dedicata alle comunicazioni interne) e la lista aggiornata degli HostNode che costituiscono il server.

`HELLO_MSG`

messaggio broadcast inviato da un nodo Relay non appena questo viene unito ad web server, al fine di fornire ai nodi già aggregati le informazioni necessarie a identificarlo e contattarlo.

`FROM_RELAY_TO_HOST`

messaggio utilizzato per il processo che determina la transizione di un nodo Relay in un nodo Host. Inizialmente il messaggio viene inviato unicamente al nodo Relay selezionato, mentre, quando la modifica è completa il nodo in questione invia un secondo messaggio FROM_RELAY_TO_HOST a tutti i nodi del server con cui notifica il proprio cambiamento di stato. In questo modo, quando un nodo riceve un messaggio di questo tipo in broadcast, è in grado di aggiornare la composizione del server, spostando il nodo mittente dalla lista dei RelayNode alla lista degli HostNode.

`FROM_HOST_TO_RELAY`

anche in questo caso, come per la tipologia precedente, il messaggio viene utilizzato per avviare la procedura che trasforma un nodo Host in un nodo Relay e, successivamente, in broadcast per notificare la transizione a tutti i nodi che compongono il web server.

`UPDATE`

questo messaggio viene utilizzato dall'HostNode coordinatore per aggiornare gli altri nodi Host che compongono il server, in seguito a operazioni che hanno modificato le strutture. In questo caso il corpo del messaggio consiste in un array di parametri, i quali definiscono le azioni che i nodi devono intraprendere per allinearsi allo stato del nodo leader.

`ELECTION e OK`

messaggi utilizzati per implementare l'algoritmo di elezione tra i nodi Host del server al fine di determinare il leader. Il loro utilizzo verrà spiegato in modo dettagliato nella sezione riguardante il meccanismo di elezione.

4.2 Classi di supporto

In questa sezione verranno brevemente presentate quelle classi sviluppate all'interno del progetto PariWEB a supporto di quelle che realizzano la struttura del web server e forniscono il servizio di web hosting.

La prima classe che prendiamo in considerazione è **WebServer**, che inizializza le strutture interne del plug-in necessarie a interfacciarsi con il Core e gli altri moduli di PariPari. Per quanto riguarda il suo ruolo all'interno del plug-in, essa ha il compito, all'avvio di Webserver, di interfacciarsi con l'utente per determinare le caratteristiche del servizio di hosting in termini di: distribuzione, spazio di archiviazione e banda disponibile. In base alle scelte specificate, la classe WebServer attiva il web server inizializzando uno tra LocalNode, HostNode o RelayNode.

La classe **WebServerNodeDescriptor**, invece, è stata sviluppata per descrivere i nodi che compongono un web server distribuito. Un'istanza di questa classe descrive un singolo nodo del server memorizzando: l'indirizzo su cui è attivo il servizio nella forma "indirizzoIP:portaWebServer", il numero della porta per le comunicazioni interne e quello utilizzato dalla libreria DiESeL per la distribuzione dell'applicazione, lo stato del nodo (Host o Relay) e, nel caso di un HostNode, l'identificatore del nodo. `WebServerNodeDescriptor` al momento dell'inizializzazione controlla i dati forniti, verificando la validità dei numeri di porta e dell'indirizzo IP del nodo e mette a disposizione un metodo che consente di verificare l'ugualgianza di due nodi. Quest'ultima funzione, in particolare, confronta la coppia <indirizzo IP:DiESeL Port> per riuscire a distinguere nodi diversi anche in presenza di NAT[1].

WebServerNodeList è una classe molto importante all'interno del plug-in, in quanto realizza una struttura che descrive la composizione del server distribuito, mantenendo l'insieme dei nodi Host e dei nodi Relay. Ogni istanza della classe `WebServerDistrNode` (sia `WebServerHostNode` che `WebServerRelayNode`) ha, tra i propri campi, un'istanza di questa classe che utilizza per mantenere aggiornata la descrizione del web server in seguito all'evoluzione dello stesso. `WebServerNodeList`

[1] `http://en.wikipedia.org/wiki/Network_address_translation`

mette a disposizione dei nodi alcuni metodi per gestire l'evoluzione del server, tra cui:

- `getRandomHostNode()`: sceglie casualmente uno dei nodi Host che compongono il server e ne restituisce il descrittore;
- `getRandomRelayNode()`: sceglie casualmente uno dei nodi Relay che compongono il server e ne restituisce il descrittore;
- `addHostNode(int ID, String addr, String portDescr, String info, boolean isCord)`: aggiunge alla lista degli HostNode il nodo descritto dai parametri passati al metodo in seguito alla ricezione di un messaggio broadcast FROM_RELAY_TO_HOST;
- `addRelayNode(String nodeAddr, String portDescr, String info)`: aggiunge alla lista dei RelayNode il nodo descritto dai parametri passati al metodo in seguito alla ricezione di un messaggio broadcast HELLO_MSG o FROM_HOST_TO_RELAY;
- `removeNode(String nodeAddr, int dieselPort)`: consente di rimuovere il nodo descritto dalla lista in cui è presente;
- `fromHostToRelay(String hostNodeAddres, String portDescr)`: metodo utilizzato dai nodi Host per mantenere aggiornata la composizione del server in seguito alla ricezione in broadcast del messaggio FROM_HOST_TO_RELAY che notifica l'avvenuta transizione del nodo mittente;
- `fromRelayToHost(int id, String relayNodeAddress, String portDescr)`: metodo utilizzato dai nodi Host per mantenere aggiornata la composizione del server in seguito alla ricezione in broadcast del messaggio FROM_RELAY_TO_HOST che notifica la transizione del nodo mittente da RelayNode a HostNode.

ManageList e **ManageDistributedList** sono le classi che realizzano le strutture generiche (non specificano i tipi di dati che contengono) utilizzate dal server per gestire contenuti e dati necessari a fornire il servizio di web hosting. Come deciso in fase di progettazione, entrambe le classi sono state implementate in modo che le tabelle vengano aggiornate su disco dopo ogni operazione che ne modifica lo stato. `ManageList` utilizza un'hashtable per memorizzare dati che riguardano l'intero plug-in, mentre `ManageDistributedList` realizza un'hashtable a due livelli per gestire le informazioni relative ai singoli domini, le quali vengono caricate in quest'unica struttura al momento dell'avvio di Webserver.
Le classi **ManageUsers**, **ManageGroups** e **WebServerACL** sono le specializzazioni della classe `ManageDistributedList`, ed in particolare `ManageUsers` è utilizzata per la gestione degli utenti autorizzati, `ManageGroups` consente di gestire i gruppi di utenti, mentre `WebServerACL` associa ad ogni risorsa del server un descrittore della politica di accesso.

4.3 Struttura del server

La struttura logica del plug-in che realizza uno dei nodi che compongono il web server è illustrata in figura 4.2. Ogni riquadro rappresenta una classe, mentre le frecce indicano le relazioni logiche tra le classi stesse. Le classi con il nome in corsivo sono classi astratte, mentre le frecce tratteggiate rappresentano la possibilità che la classe da cui hanno origine attivi il thread collegato.

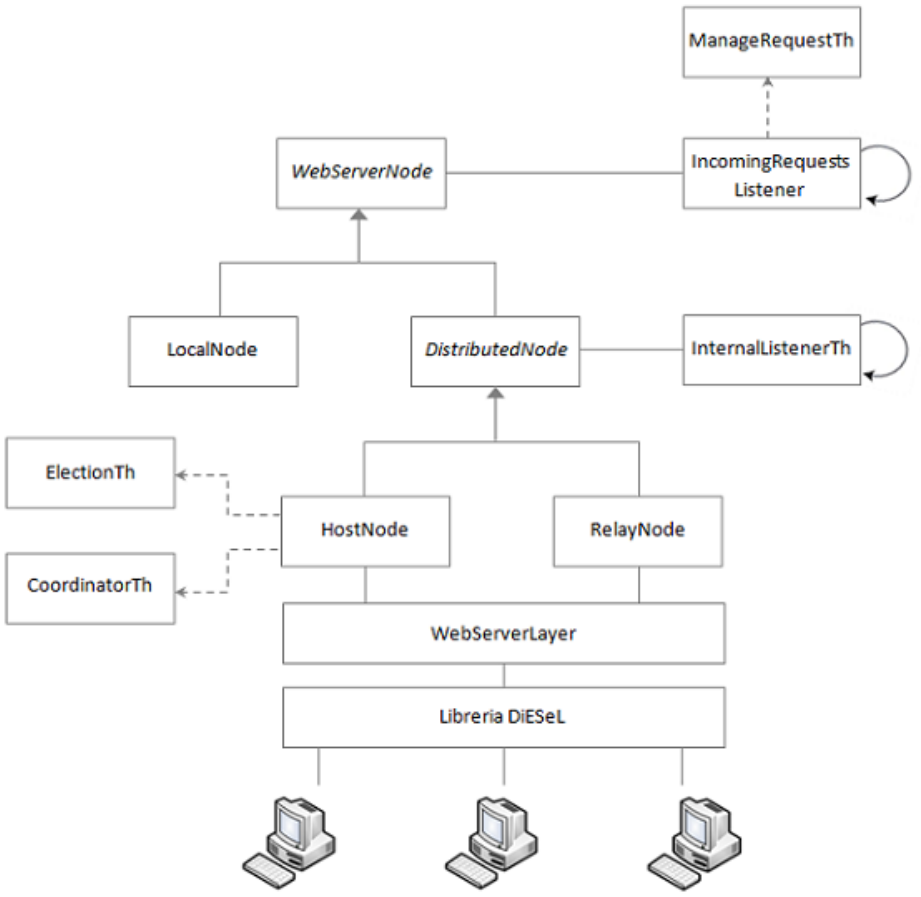


Figura 4.2: Struttura del plug-in Webserver.

Il plug-in è caratterizzato da una struttura gerarchica che facilita sia la comprensione del suo funzionamento, sia eventuali interventi di estensione delle funzionalità. In questo modo, infatti, i metodi e le strutture del plug-in si trovano a livelli differenti della gerarchia a seconda del ruolo che ricoprono: se essi sono indipendenti dalla modalità di esecuzione di Webserver (locale o distribuita) si troveranno all'interno della classe `WebServerNode`, mentre se sono legati ad aspetti di distribuzione del server ma sono indipendenti dal tipo di nodo (RelayNode o HostNode) si troveranno all'interno della classe `WebServerDistrNode`.

4.3.1 WebServerNode, IncomingRequestsListener e ManageRequestTh

La classe astratta **WebServerNode** rappresenta un nodo generico di un web server, implementando i comportamenti che non dipendono dalla modalità di esecuzione del plug-in. All'interno di questa classe, per esempio, sono definite le strutture dati necessarie a fornire il servizio di hosting, ed in particolare:

- una hashtable `domains` che associa ad ogni nome di dominio ospitato sul server la cartella corrispondente in cui sono salvati i contenuti;
- una hashtable a due livelli `accessControl` che, per ogni dominio, associa il percorso di ogni risorsa ad un descrittore delle politiche di accesso alla stessa;
- una hashtable a due livelli `groups` che, per ogni dominio, associa al nome di un gruppo la lista degli utenti che vi appartengono;
- una hashtable a due livelli `authUsers` che, per ogni dominio, conserva nome utente e password di particolari utenti definiti dall'amministratore del dominio stesso.

Il suo metodo di inizializzazione crea la struttura su disco del plug-in (una cartella "conf" che contiene i file di configurazione del plug-in e dei domini che il nodo ospita, e una cartella "data" dove vengono salvati i contenuti ospitati dal server) ed inizializza le strutture dati sopra elencate. `WebServerNode`, inoltre, contiene un metodo astratto, `initWebServerNode()`, che deve essere implementato dalle sottoclassi per inizializzare quelle strutture specifiche della modalità di funzionamento.

Come si può vedere dalla figura 4.2, a `WebServerNode` è associato un thread **IncomingRequestsListener** il cui compito è quello di rimanere in attesa delle richieste Http in arrivo sulla porta designata dall'utente per offrire il servizio di web hosting. Questo thread si avvia nel momento in cui viene istanziata una delle classi che estendono `WebServerNode` (che essendo una classe astratta non può essere istanziata), cioè quando viene chiamato il costruttore di uno tra LocalNode, HostNode o RelayNode.

Listing 4.1: Attivazione di IncomingRequestsListener da parte di WebServerNode.java

```
277 ...
278 public void startManageIncomingReq(){
279     if(incomingPariPariThread != null){
280         incomingPariPariThread.kill();
281     }
282
283     // Request to the plug-in Connectivity a LimitedServerSocket
284     LimitedServerSocketAPI sock;
285     sock = getLimitedServerSocket(new InetSocketAddress(webServerPort), bandwidth);
286
287     // Starts thread IncomingRequestsListener
288     incomingReqTh = new IncomingRequestsListener(this, sock);
289     incomingPariPariThread = new PariPariThread(incomingReqTh,"[IncomingReqListener]");
290     incomingPariPariThread.start();
291 }
292 ...
```

`IncomingRequestsListener` a sua volta attiva un thread **ManageRequestTh** per ogni richiesta pervenuta, responsabile del processing e della generazione della risposta. Come si può vedere dal codice 4.2, `ManageRequestTh` chiama in sequenza i moduli che processano la richiesta secondo lo schema riportato nella figura 3.1 del precedente capitolo.
Particolarmente significativo è il comportamento del thread nel caso in cui il plug-in sia in modalità distribuita ed il nodo coinvolto sia un RelayNode (riga 294). In questa situazione la richiesta viene processata dal `ParserHTTP` esclusivamente fino alla lettura del campo Host dell'header, successivamente viene determinato l'url a cui la risorsa è disponibile scegliendo casualmente uno degli HostNode che compongono il server, ed infine viene creato il messaggio di risposta *Temporary Redirect* che verrà inviato al client dal modulo `SendResponse`.

Listing 4.2: Corpo del thread ManageRequestTh.java

```
277 ...
278 boolean connected = true;
279 ILog log = webServer.getLogger();
280 String webName = "[WebServer-" + thName + "]: ";
281 IWebServerNode wsNode = webServer.getWebServerNode();
282
283 try{
284     while(connected){
285         try{
286             setFile(null);
287             IRequestParser parser;
288             String request, res, newUrl = null;
289             tableRes = new TableResponse();
290             tabReq = new Hashtable<String, String>();
291             IWebServerDistrNode distrNode = ((IWebServerDistrNode)wsNode);
292
293             try {
294                 if(WebServer.distributed&&distrNode.getNodeStatus()==NodeStatus.RELAY_NODE){
295                     //Parser Module
296                     //create table containing data on the request through a parsing
297                     parser = new ParserHTTP_Module(input, log, thName, REDIRECTION);
298                     tabReq = parser.parse();
299
300                     // This node is a Relay_Node, then redirect request to Host_Node
301                     res = "/"+tabReq.get("Host")+tabReq.get("Path")+tabReq.get("File-Name");
302                     newUrl = "http://"+distrNode.getRandomRealNode().getNodeAddress()+res;
303                     tableRes.setTemporaryRedirection(newUrl);
304                     WebServer.redirectedRequest++;
305                 } else{
306                     //Parser Module
307                     //create table containing data on the request through a parsing
308                     parser = new ParserHTTP_Module(input,log,thName,!REDIRECTION);
309                     tabReq = parser.parse();
310                     request = parser.getRequest();
311
312                     // extract all resource info as domain, folder, path, etc.
313                     WebResourceInfo resInfo = new WebResourceInfo(webServer,tabReq);
314
315                     //Access Control Module
316                     //check if the request could be served
317                     AccessControl_Module ACmanager;
318                     ACmanager = new AccessControl_Module(webServer,tabReq,resInfo,log,thName);
319                     auth = ACmanager.checkIfAuthorized();
320
321                     //Serve Module
322                     //create response to the request, compiling tableRes
323                     if(auth == Authorization.ALLOW){
324                         ServeRequest_Module serveModule;
325                       serveModule = new ServeRequest_Module(this,resInfo,request);
```

```
326                         tableRes = serveModule.serve(tabReq);
327                       } else {
328                         tableRes = new TableResponse(HttpErrorCode.NotAuthorized);
329                       }
330                   }
331               } catch(HttpException e){
332                   //Processing error, than create a response with correct Status-Code
333                 tableRes = new TableResponse(e.getErrorCode());
334               }
335
336               //Response Module
337               if(SendResponse_Module.send(sock,tableRes.getTable(),file)){
338                 WebServer.correctlyDoneRequest++;
339               } else {
340                   //Impossible to send HTTP Response
341                 WebServer.errorRequest++;
342                 connected = false;
343               }
344
345               String connection = tabReq.get("Connection");
346               if(connection != null && connection.equals("close")){
347                 // Not persistent connection
348                 connected = false;
349                 input.close();
350                 sock.close();
351               }
352           } catch(SocketTimeoutException se){
353             // Socket timeout, so i close the connection!
354             connected = false;
355             input.close();
356             sock.close();
357           }
358       }
359   } catch (IOException e) {
360     // Impossible to close the socket (IOException)!!
361     WebServer.errorRequest++;
362   }
363
364   // Warn the WebServer that ended management request
365   wsNode.finishManageRequest(this, tableRes.getHash());
366   ...
```

Un aspetto importante del codice riportato, riguarda il meccanismo con cui il server gestisce le connessioni persistenti introdotte dalla versione Http 1.1. La tecnica adottata prevede che venga impostato un timeout (il cui valore può essere settato tramite il file di configurazione del plug-in) sul socket da cui viene letta la richiesta, in modo che un'operazione di lettura rimanga in attesa di eventuali ulteriori dati al massimo per questo tempo. Se il timeout scade viene sollevata un'eccezione di tipo `SocketTimeoutException` che causa la chiusura della connessione con il client. Questa soluzione consente di processare più richieste consecutive inviate lungo la medesima connessione, limitando allo stesso tempo l'attesa del server, che chiude il socket quando il canale rimane inattivo per un tempo sufficientemente lungo.

4.3.2 WebServerLocalNode, WebServerDistrNode e InternalListenerTh

WebServerLocalNode è una delle classi instaziabili all'interno di questa struttura gerarchica, ed in particolare è quella che realizza la modalità locale del plug-in. `WebServerLocalNode` richiama semplicemente il costruttore della classe padre

`WebServerNode`, inizializzando in modo corretto indirizzo e porta su cui rimanere in ascolto per fornire il servizio. Se il nodo è connesso alla rete Internet il suo indirizzo sarà il suo IP pubblico, altrimenti verrà inizializzato con l'indirizzo di *loopback* 127.0.0.1.

La classe astratta **WebServerDistrNode** estende `WebServerNode` con le strutture necessarie per gestire la distribuzione del server e tutti gli aspetti comuni sia ad un nodo Host che ad uno Relay. L'inserimento del nodo all'interno di un contesto distribuito, comporta che esso possa esser contattato dagli altri nodi aggregati, per cui sono necessarie delle procedure per l'invio e la ricezione di messaggi.
All'interno di `WebServerDistrNode` viene gestito questo aspetto, definendo i metodi per realizzare comunicazioni unicast e comunicazioni broadcast, oltre che le procedure che consentono la transizione di un nodo Relay in un nodo Host e viceversa, e le informazioni riguardanti lo stato della libreria DiESeL e dell'algoritmo di elezione. Tra i metodi di `WebServerDistrNode`, inoltre, è presente `receivedMessage(IInternalWebServerMessage msg)`, un metodo astratto che le classi `WebServerHostNode` e `WebServerRelayNode` implementano per definire le diverse azioni da intraprendere alla ricezione di un messaggio broadcast.

Ad ogni esemplare di `WebServerDistrNode` è associato un thread, denominato **InternalListenerTh**, che rimane in ascolto sull'*InternalPort* del nodo per ricevere le comunicazioni interne al web server distribuito ed esegue le procedure che la ricezione dei messaggi comportano. All'interno di questa classe, in particolare, vengono gestiti i messaggi relativi all'algoritmo di elezione e all'aggiornamento degli HostNode da parte del nodo leader, oltre che le comunicazione che attivano la transizione di stato di un nodo.
Il listato riportato di seguito riporta il metodo `fromRelayToHost()` della classe `WebServerDistrNode`, il quale realizza il passaggio del nodo da RelayNode ad HostNode. Questo metodo viene chiamato dall'`InternalListenerTh` del nodo, dopo aver ricevuto il messaggio FROM_RELAY_TO_HOST e tutte le strutture necessarie per attivare il servizio.

Listing 4.3: Metodo fromRelayToHost della classe WebServerDistrNode.java

```
260 ...
261 public boolean fromRelayToHost(IWebServerRelayNode fNode){
262     if(nodeStatus == NodeStatus.RELAY_NODE){
263
264         // Initialize node as a new host node
265         IWebServerNode wsNode = new WebServerHostNode(fNode);
266         webServer.setWebServerNode(wsNode);
267
268         // Update node list (remove node from relay list and add it into host list)
269         int nodeId = ((IWebServerHostNode)wsNode).getMyID();
270         nodeList.fromRelayToHost(nodeId,fNode.getMyAddress(),fNode.getPortDescriptor());
271
272         // Set listener node as a host node
273         listenerTh.setNode((IWebServerDistrNode)wsNode);
274
275         // Set WebServerLayer node as a host node
276         webServerLayer.setWebServerNode((IWebServerDistrNode)wsNode);
277
278         // Set IncomingRequestsListener node as a host node
279         if(incomingRequestTh != null){
```

```
280             incomingRequestTh.setWebServerNode(wsNode);
281         }
282
283         // Send FROM_RELAY_TO_HOST message to other nodes
284         IWebServerHostNode wshNode = (IWebServerHostNode)wsNode;
285         Object[] text = new Object[]{fNode.getPortDescriptor()+":"+wshNode.getMyID()};
286         sendBroadcastMsg(text, MessageType.FROM_RELAY_TO_HOST);
287         return true;
288     }
289     return false;
290 }
291 ...
```

Dopo aver re-istanziato il nodo come `WebServerHostNode` (riga 265) ed aver aggiornato i thread attivi, il nodo invia in broadcast, come prevede il protocollo definito in fase di progettazione, un messaggio di tipo FROM_RELAY_TO_HOST che notifica l'avvenuta transizione a tutti i nodi che compongono il web server.

4.3.3 WebServerRelayNode, WebServerHostNode e WebServerLayer

`WebServerHostNode` e `WebServerRelayNode` sono le altre due classi che possono essere istanziate all'interno della struttura di figura 4.2. Esse risultano più complesse rispetto alla classe `WebServerLocalNode`, in quanto devono implementare quegli aspetti legati alla distribuzione e alla struttura interna del web server che variano a seconda del ruolo rivestito dal nodo. Entrambe le classi implementano il metodo astratto `initWebServerNode()` di `WebServerNode`, che permette di inizializzare le strutture specifiche legate al tipo di nodo, e il metodo `receivedMessage(...)` di `WebServerDistrNode`, che consente di definire le diverse azioni da intraprendere alla ricezione di un messaggio broadcast.
Sia i nodi Host che i nodi Relay utilizzano la classe **WebServerLayer** che consente di avviare un'istanza della libreria DiESeL per distribuire il server. Vi sono però delle differenze sul modo in cui la libreria viene utilizzata:

- se l'utente ha deciso di realizzare un nuovo server, all'avvio del plug-in viene inizializzato un HostNode che a sua volta avvia un'istanza attiva di DiESeL con il compito di aggregare il numero adeguato di nodi per realizzare il server distribuito;
- invece, se l'utente ha avviato Webserver con l'intento di offrire una parte della propria banda e del proprio spazio su disco per il servizio di hosting, viene inizializzato un RelayNode che a sua volta avvia un'istanza passiva di DiESeL, la quale unisce il nodo ad un web server già esistente.

Quando viene chiamato il costruttore di **WebServerRelayNode**, questo inizializza un oggetto `WebServerLayer` che si occupa di avviare un'istanza di DiESeL passiva e, nel momento in cui il nodo viene aggregato ad un server, di attivare le strutture necessarie per rimanere in ascolto sulla porta che fornisce il servizio.
Un RelayNode, essendo semplicemente un ripetitore, non possiede ne la struttura del plug-in su disco ne nessuna delle strutture definite in precedenza per la gestione

del servizio.

WebServerHostNode, invece, inizializza tutte le strutture di `WebServerNode` e `WebServerDistrNode`, richiamando i costruttori delle super-classi, e attiva la classe `WebServerLayer` che provvede ad avviare un'istanza attiva di DiESeL. Questa classe realizza i metodi che consentono di aggiungere un nodo Host al server o di rimuoverne uno, oltre alle procedure che permettono all'HostNode leader di sincronizzare gli altri nodi Host. Nel listato seguente viene riportato il codice con il quale un nodo Host, sottoposto a condizioni di carico che non è in grado di gestire, è in grado di duplicarsi.

Listing 4.4: Metodo addHostNodeToServer() della classe HostNode.java

```
171 ...
172 public boolean addHostNodeToServer(){
173
174     boolean res = false;
175     LimitedSocketAPI intSock;
176     InternalWebServerMessage msg;
177     IWebServerNodeDescriptor fNode;
178
179     // Select a fake node that will be changed into real node
180     fNode = nodeList.getRandomRelayNode();
181
182     if(fNode != null){
183         int intPort = fNode.getInternalPort();
184         String nodeAdr = fNode.getNodeAddress();
185         String host = nodeAdr.substring(0, nodeAdr.indexOf(":"));
186
187       // Connect this node to the selected relay node
188       intSock = getInternalSocket(host, intPort);
189
190         try {
191           if(intSock != null){
192                 ObjectOutputStream output = new ObjectOutputStream(intSock.getOutputStream());
193
194                 // Send a FROM_FAKE_TO_REAL message to the selected relay node
195                 MessageType type = MessageType.FROM_RELAY_TO_HOST;
196                 long time = ((Distributore)webServerLayer.distr).getTimeRef();
197                 Object[] msgText = new Object[]{nodeList};
198                 msg = new InternalWebServerMessage(myAddress,nodeAdr,msgText,time,type);
199                 output.writeObject(msg);
200                 output.flush();
201
202                 // Send all data structures and files to the selected node
203                 if(sendDataStructures(output)){
204                 res = sendHostedFiles(output);
205                 }
206
207                 intSock.close();
208                 return res;
209           }
210       } catch (IOException e) {
211         Console.printImportant(webName + "Error requesting ObjectOutputStream");
212       }
213     } else{
214       Console.printImportant(webName + "No fake node found!!");
215     }
216     return false;
217 }
218 ...
```

La classe `WebServerHostNode`, come si può vedere dalla figura 4.2, ha la possibilità di attivare due diversi thread: `ElectionTh`, che realizza l'algoritmo di elezione tra i nodi Host, e `CoordinationTh`, che consente all'HostNode leader di inviare periodicamente un messaggio di tipo I_AM_LEADER . Queste due classi verranno descritte nella prossima sezione.

4.3.4 ElectionTh e CoordinationTh

Nelle precedenti sezioni, è stata illustrata la necessità di utilizzare un algoritmo di elezione per coordinare gli HostNode, in modo da delegare al solo nodo leader le operazioni che modificano lo stato del web server.
La classe **ElectionTh** implementa a questo scopo il **Bully algorithm**, una semplice procedura distribuita che consente di eleggere un coordinatore sulla base di un ID che identifica i singoli nodi. All'interno di PariWEB, gli HostNode utilizzano come identificatore un intero scelto casualmente in un range di un milione (in modo da minimizzare la possibilità che due nodi Host abbiamo lo stesso identificatore) al momento dell'attivazione del nodo stesso. L'ID viene comunicato agli altri nodi che compongono il server all'interno del messaggio I_AM_LEADER per quanto riguarda il leader, e all'interno del messaggio broadcast FROM_RELAY_TO_HOST quando avviene la transizione da un nodo Relay ad uno Host.

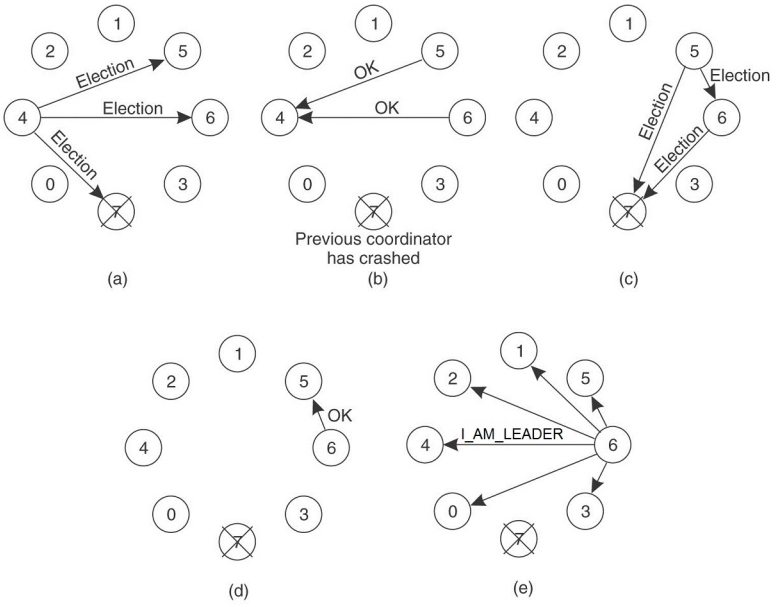


Figura 4.3: Un esempio del funzionamento dell'algoritmo di elezione Bully tra 8 nodi.

L'algoritmo Bully viene avviato quando uno dei nodi determina che l'attuale leader non è più attivo, oppure quando viene ricevuto un messaggio ELECTION da uno dei nodi con identificativo minore.
Supponendo che il nodo Host H si trovi in una di queste due condizioni, esso avvia la procedura di elezione, che evolve nel seguente modo:

1. H invia un messaggio di tipo ELECTION a tutti gli HostNode inseriti nella sua `WebServerNodeList` che hanno identificativo maggiore;

2. se non riceve risposta, significa che H è il nodo attivo con ID maggiore, per cui la procedura termina e H vince l'elezione;

3. se uno dei nodi con identificativo maggiore risponde con un messaggio di tipo OK , questo prende il controllo dell'elezione partendo dal punto 1. e l'HostNode H termina il proprio lavoro.

Alla fine di questa procedura, tutti i nodi termineranno l'algoritmo, tranne uno che sarà il nuovo nodo leader. A questo punto esso attiverà il **CoordinatorTh**, responsabile di inviare periodicamente messaggi broadcast di tipo I_AM_LEADER per notificare la propria presenza.
Come si può vedere dal listato 4.5 per determinare quando il leader non è più attivo, ogni HostNode imposta un timeout alla ricezione di ogni messaggio di tipo I_AM_LEADER . Se il messaggio successivo viene ricevuto in tempo il timer viene resettato e riavviato, altrimenti il timer scade determinando l'avvio dell'algoritmo di elezione.

Listing 4.5: Utilizzo del timer per l'algoritmo di elezione

```
286 ...
287 case I_AM_LEADER:
288
289     if(!IAmCoordinator()){
290         if(currentTaskLeaderExpire != null){
291             // Reset previous timer
292             currentTaskLeaderExpire.cancel();
293         }
294
295         // Set and run a new timer
296         currentTaskLeaderExpire = new LeaderExpiration(this);
297         new Timer(true).schedule(currentTaskLeaderExpire, COORD_TIMEOUT_ms);
298     }
299 break;
300 ...
```

Di seguito si riporta la porzione di codice che implementa l'algoritmo di elezione sopra descritto, e il metodo `checkRecMessage()` utilizzato per controllare le risposte ricevute dal nodo una volta avviata l'elezione oppure per verificare se il thread è stato attivato in seguito alla ricezione di un messaggio di tipo ELECTION .

Listing 4.6: Corpo della classe ElectionTh.java

```
45 ...
46 // Check received message
47 checkRecMessage();
48
49 // Send a message to all real nodes with id bigger
50 int myNodeID = hNode.getMyID();
51 String text;
52 String adr;
53 int internalPort;
54
55 for(int i=0; i<hostNodeList.size();i++) {
56   IWebServerNodeDescriptor currentNode = hostNodeList.get(i);
57   int currentNodeID = currentNode.getNodeID();
```

```
58
59    if (( myNodeID < currentNodeID ) && !currentNode.isCoordinator()) {
60      adr = currentNode.getNodeAddress();
61      internalPort = currentNode.getInternalPort();
62      text = myNodeID + "/" + internalPort;
63      hNode.sendUnicastMsg(adr, internalPort, text, MessageType.ELECTION);
64    }
65  }
66
67  try{
68    // With this operation no resource are available
69    semaphore.acquire();
70
71    if (!semaphore.tryAcquire(5000, TimeUnit.MILLISECONDS)){
72      // No message has been received, than this node is the leader
73      hNode.setCoordinator(true);
74      hNode.startCoordThread(hNode);
75      hNode.sendBroadcastMsg(Integer.toString(myNodeID), MessageType.I_AM_LEADER);
76      stop();
77    }
78    else{
79      // One or more messages have been received
80      checkRecMessage();
81    }
82  } catch(InterruptedException e) {
83    e.printStackTrace();
84  }
```

Listing 4.7: Metodo checkRecMessage() della classe ElectionTh.java

```
89  ...
90  /**
91   * Method that checks all messages received by this node until the election thread was
92   * running (waiting possibly response) or the message that has been caused the election
93   * thread activation.
94   */
95  private void checkRecMessage(){
96    IInternalWebServerMessage msg;
97    int myNodeID = hNode.getMyID();
98    messageReceived = hNode.getMessageQueue();
99
100   if(messageReceived != null && !messageReceived.isEmpty()){
101     String adr;
102     String text;
103     int internalPort;
104
105     while(!messageReceived.isEmpty()){
106       msg = messageReceived.removeFirst();
107
108       if(msg.getMessageType() == MessageType.ELECTION){
109         text = (String) msg.getMessage();
110         int nodeID = new Integer(text.substring(0,text.indexOf("/")));
111         internalPort = new Integer(text.substring(text.indexOf("/") + 1));
112
113         // Check node id
114         if(myNodeID >= nodeID){
115           // Send OK message
116           adr = msg.getSenderAddress();
117           hNode.sendUnicastMsg(adr,internalPort,Integer.toString(myNodeID),MessageType.OK);
118         } else{
119           // Stop election procedure, because there is a host node with id >
120           stop();
121         }
122       } else if(msg.getMessageType() == MessageType.OK){
123         // Stop election procedure, because there is a host node with id >
124         this.stop();
125       }
126     }
127
```

```
128         // If i'm here ==> response only with OK message ==> start election procedure
129         ((IWebServerHostNode)hNode).startElectionAlg(hNode, null);
130       }
131   }
132   ...
```

Capitolo 5

Funzionamento e Prestazioni

In questo capitolo sarà descritto il funzionamento dell'attuale versione di Webserver (sezione 5.1), sottolineando in particolare come l'utente finale può configurare il servizio ed interagire con il plug-in attraverso la console, e verranno analizzate le prestazioni ottenute (sezione 5.2).

5.1 Funzionamento

Webserver mette a disposizione dell'utente di PariPari un servizio di web hosting completamente configurabile e semplice da utilizzare. In questa sezione verrà descritto come configurare il servizio alla prima attivazione(5.1.1) e come utilizzare il plug-in attraverso la console di PariPari (5.1.29.

5.1.1 File di configurazione

Webserver è configurabile tramite il file `webServer.conf` posizionato all'interno della cartella "conf" del plug-in. I campi settabili dall'utente sono:

`port: integer`

Consente di specificare la porta su cui il web server deve rimanere in ascolto delle richieste in ingresso. La porta di default è la 10080.

`bandwidth: integer`

Consente di specificare la massima banda che il web server è autorizzato ad utilizzare. La banda impostata di default è di 1Mbps.

`maxReq: integer`

Consente di definire il massimo numero di richieste contemporanee che il server è in grado di gestire. Il valore di default è 100.

`portDistr: integer`

Consente di specificare la porta che DiESeL utilizza per mantenere connesso il nodo alle altre istanze che compongono il server. DiESeL utilizza due porte consecutive, per cui se il numero di porta specificato è 7262, DiESeL userà le porte 7262 e 7263. Il valore di default per questo campo è 7262.

`distributed: boolean`

Consente di specificare la modalità di default con cui attivare il web server. Se il valore inserito è `true` il server verrà attivato in modalità distribuita, altrimenti in modalità locale. Di default il campo distributed è impostato a true.

`featureFileName: string`

Consente di specificare il nome del file xml che definisce le caratteristiche del nodo del web server.

`serverFeatureFileName: string`

Consente di specificare il nome del file xml che definisce le caratteristiche dell'intero web server che l'utente vuole realizzare all'interno di PariPari.

`usersFileName, domainsFileName, groupsFileName: string`

Consentono di definire i nomi dei file all'interno dei quali vengono salvate le strutture per, rispettivamente, gli utenti autorizzati, i domini ospitati e i gruppi.

`ACLFileName, propertiesFileName: string`

Consentono di definire il nome del file all'interno del quale viene salvata la struttura dati per la gestione delle politiche di accesso e il nome del file che contiene le proprietà del plug-in.

`minLoadTreshold, maxLoadTreshold: integer`

Questi due parametri consentono di definire le soglie di traffico (in termini di richieste contemporanee) che il nodo utilizza per determinare quando avviare le procedure `fromHostToRelay` e `fromRelayToHost`. Al di sotto della soglia minima il traffico è impostato a LOW, mentre al di sopra della soglia massima risulta essere HIGH. I valori di default sono: 20 richieste per la soglia minima, 100 per quella massima.

`sendingPeriod_ms: integer`

Consente di definire il periodo con cui il thread `CoordinationTh` del nodo leader invia i messaggi I_AM_LEADER. Il valore utilizzato di default è pari a 10 secondi.

`coordTimeout_ms: integer`

Consente di impostare il periodo massimo che può intercorrere tra la ricezione di due messaggi dell'HostNode leader, senza che venga attivata la procedura di elezione. Il valore utilizzato di default è pari a 15 secondi.

`persistentConnectionTimeout_ms: integer`

Consente di impostare il valore massimo di attesa su una connessione in ingresso da parte del server, utilizzato per la realizzazione delle connessioni persistenti. il valore utilizzato di default è pari ad 200 millisecondi.

`maxTimeHighTraffic_ms: integer`

Consente di impostare la durata dell'intervallo di tempo in cui il traffico in ingresso al nodo risulta HIGH necessario ad attivare la procedura di replicazione che porta alla transizione di un nodo Relay in un HostNode. Il valore utilizzato di default è pari a 5 secondi.

`maxTimeLowTraffic_ms: integer`

Consente di impostare la durata dell'intervallo di tempo in cui il traffico in ingresso ad un nodo che è diventato Host per il bilanciamento del carico risulta LOW necessario ad attivare la procedura che lo ri-trasforma in RelayNode. Il valore utilizzato di default è pari ad un minuto.

Nel caso il plug-in venga utilizzato in modalità distribuita è necessario definire un file xml che descriva le caratteristiche del nodo, e, se l'obiettivo è quello di attivare un nuovo web server, un secondo file xml che definisca le caratteristiche complessive che l'utente desidera per il server stesso. Entrambi i file vengono utilizzati dalla libreria DiESeL per determinare quanti e quali nodi aggregare per comporre il server. I file, che di default sono denominati `featureConfig.xml` e `serverFeatureConfig.xml`, devono presentare la seguente struttura:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<featureDefinition>
  <feature>
    <name>FeatureName</name>
    <value>FeatureValue</value>
  </feature>
...
</featureDefinition>
```

Se l'utente non definisce questi due file, il plug-in li crea in modo automatico, utilizzando come caratteristiche per descrivere sia il nodo che l'eventuale nuovo web server lo spazio di archiviazione su disco e la banda disponibile.

5.1.2 Webserver Console

Il plug-in Webserver mette a disposizione dell'utente finale una serie di comandi con cui gestire il web server attivato all'interno di PariPari. L'insieme delle funzionalità accessibili tramite la console di PariPari sono riportate nella figura 5.1. Di seguito verranno brevemente descritti i singoli comandi.

`help`

consente all'utente di visualizzare la lista dei comandi disponibili per il plug-in Webserver, come in figura 5.1.

`lsprop`

visualizza l'elenco delle proprietà attualmente impostate per il plug-in Webserver.

```
| Possible commands for WebServer are:
| help                                                | print this message
| info                                                | print node information
| lsprop                                              | list of the current properties values
| lsauth                                              | list all users of the authorized users list
| lsfs                                                | print the files owned by the plugin (fileset)
| lsdom                                               | list of domains configured for this server
| stop                                                | stop webserver plugin
| adddom <domain> <folder> <adminPwd>                 | add a domain
| addfile <domain> <path> [fileGroup]                 | add a file to the webserver. Use absolute filepath
| rmdom <domain> <adminName> <adminPwd>               | remove a domain
| rmfile <domain> <file> <ownerName> <ownerPwd>       | remove the file from this domain
| adduser <domain> <user> <pass>                      | add an user into the authorized users list
| rmuser <domain> <user> <pass>                       | remove an user from the authorized users list
| wizard <domain> <fold> <resPath> <adminPwd>         | create new domain with all files in resFolderPath
| addgroup <domain> <name> <groupPwd>                 | add an user's group at the domain specified
| remgroup <domain> <name> <groupPwd>                 | rem the user's group at the domain specified
| addusergroup <domain> <group> <uName>               | add an user at the group specified
| remusergroup  <domain> <group> <uName>              | rem the user specified from the group
| setfileaccess  <domain> <file> <resGroup> <access>| set the access policy of this file
| remfileaccess  <domain> <file>                      | rem access policy of this file
| stat                                                | print webserver statistics
```

Figura 5.1: Elenco dei comandi messi a disposizione dal plug-in Webserver.

`info`

visualizza le informazioni relative al nodo web server, ed in particolare l'indirizzo web al quale può essere contattato (inidirizzoIP:porta), le porte utilizzate per la libreria DiESeL e per le comunicazioni interne, lo stato del nodo (LocalNode, HostNode o RelayNode), l'indicazione se esso è il coordinatore e l'elenco dei nodi che costituiscono il server.

`stat`

visualizza le statistiche relative al nodo del web server a partire dall'avvio del plug-in, specificando il numero di richieste Http ricevute, rifiutate, reindirizzate, che hanno generato un errore e che sono state eseguite correttamente.

`lsdom`

visualizza l'elenco dei domini attualmente ospitati dal nodo, mettendo in evidenza il nome di dominio e la cartella di riferimento all'interno della directory "data" in cui sono salvati i relativi contenuti.

`lsfs`

visualizza l'elenco delle risorse ospitate dal server, classificate per dominio.

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

permette di aggiungere un dominio al server, specificando il nome della cartella in cui salvare i relativi contenuti all'interno della directory "data" del plug-in e la password da utilizzare per il gruppo degli amministratori del dominio. Questa operazione comporta, inoltre, la creazione di una cartella con il nome del nuovo dominio all'interno della directory "conf", nella quale vengono inizializzate e salvate le strutture dati relative alle politiche di accesso e alla gestione dei gruppi e degli utenti autorizzati.

`rmdom <domain> <adminName> <adminPwd>`

rimuove il dominio e tutte le risorse collegate ad esso dal web server. Questa operazione è consentita solo agli utenti del gruppo *AdminGroup* del dominio, i quali possono identificarsi in due differenti modi: o fornendo il nome del gruppo e la relativa password, oppure utilizzando le proprie credenziali personali se essi sono inseriti nel gruppo *AdminGroup*.

`addfile <domain> <path> [fileGroup]`

permette di aggiungere la risorsa identificata dal campo `<path>` al dominio specificato. Opzionalmente è possibile definire il gruppo a cui appartiene la risorsa in modo che le venga applicata la stessa politica di accesso.

`rmfile <domain> <path> <ownerName> <ownerPwd>`

permette di rimuovere la risorsa identificata dal campo `<path>` dal dominio specificato. Questa operazione è sempre consentita al proprietario del contenuto oppure agli utenti del gruppo *AdminGroup*, mentre per gli altri utenti viene consultata la politica di accesso definita.

`wizard <domain> <fold> <resPath> <adminPwd>`

procedura che consente di aggiungere un dominio al server, caricando direttamente tutte le risorse contenute nella cartella specificata dal percorso `<resPath>`.

`adduser <domain> <user> <pass>`

consente di aggiungere un utente ad un dominio, in modo che possano essere definite delle politiche di accesso differenziate a seconda dell'utente che esegue l'operazione.

`rmuser <domain> <user> <pass>`

rimuove l'utente individuato dai campi `<user>` e `<pass>` dagli utenti del dominio.

`addgroup <domain> <name> <groupPwd>`

consente di aggiungere il gruppo di utenti identificato da nome e password specificati al dominio. Questo gruppo viene automaticamente inserito all'interno della struttura dati che identifica gli utenti del dominio per cui è possibile specificare i criteri di accesso, in modo che possano essere definite delle politiche di gruppo.

`remgroup <domain> <name> <groupPwd>`

consente di rimuovere il gruppo identificato da nome e password dal dominio specificato.

`addusergroup <domain> <group> <uName>`

consente di aggiungere un utente precedentemente definito tra gli utenti del dominio al gruppo identificato dal campo `<group>`.

`remusergroup <domain> <group> <uName>`

rimuove un utente precedentemente definito tra gli utenti del dominio dal gruppo identificato dal campo `<group>`.

`setfileaccess <domain> <file> <resGroup> <access>`

consente di definire la politica di accesso per la risorsa identificata dal percorso `<file>`. In particolare è possibile modificare il gruppo a cui appartiene la risorsa (`<resGroup>`) e definire un nuova autorizzazione nella forma `ALLOW/DENY + azione + [utente]`.

`remfileaccess <domain> <file>`

consente di rimuovere la politica di accesso, precedentemente definita, per la risorsa identificata dal percorso `<file>`.

`stop`

interrompe l'esecuzione del plug-in Webserver sul nodo.

5.1.3 Esempio di utilizzo

In questa sezione verrà riportato un esempio di utilizzo del plug-in da parte di un utente che vuole realizzare un web server per ospitare più domini.

Al primo avvio di Webserver, se non sono stati già creati, il plug-in salva all'interno della propria cartella "conf" dei modelli dei file di configurazione, utilizzando dei valori di default che possono essere modificati dall'utente. In seguito viene creato il dominio di default "localhost" all'interno del quale il plug-in cercherà i contenuti nel caso in cui la richiesta pervenuta al server specifichi un nome di dominio inesistente o non valido. Quest'ultima operazione comporta la creazione su disco della cartella "htdocs" nella folder "data" per il salvataggio dei contenuti, e la cartella "localhost" in "conf" all'interno della quale vengono salvate le strutture dati relative alle politiche d'accesso e alla gestione dei gruppi e degli utenti autorizzati.

A questo punto supponiamo che l'utente voglia *salvare un dominio* (denominato www.domain1.it) sul server appena creato. L'operazione

```
adddom www.domain1.it domainFolder1 adminPwd
```

crea all'interno della sezione "data" la cartella "domainFolder1", che ospiterà le risorse relative al nuovo dominio, e all'interno della cartella "conf" la cartella "domain1.it", che conterrà le strutture dati per la gestione del dominio. L'aggiunta di un dominio al server comporta, automaticamente, la definizione del gruppo *AdminGroup* e l'inserimento in esso dell'utente che ha eseguito l'operazione che viene riconosciuto come amministratore del dominio. Questo gruppo viene inserito, inoltre, tra gli utenti autorizzati del dominio, associando al nome la password specificata.

Per *aggiungere una risorsa al dominio* l'utente può inviare una richiesta di tipo PUT al server oppure utilizzare il seguente comando

```
addfile www.domain1.it C:\PariPari\Resources\Index.html adminGroup
```

che importa la risorsa individuata dal path all'interno della cartella "domainFolder1", definendo allo stesso tempo il gruppo a cui il contenuto aggiunto appartiene e l'utente proprietario.

Se le risorse del dominio sono molte, il procedimento appena descritto risulta complesso e poco pratico. Per questo motivo il plug-in Webserver mette a disposizione dell'utente un secondo comando che consente di definire un nuovo dominio e caricare sul server tutte le relative risorse specificando semplicemente il percorso della cartella che le contiene.

```
wizard www.domain2.org domainFolder2 C:\PariPari\ResDom2\ adminPwd
```

Il comando `wizard` si preoccupa di definire il nuovo dominio, eseguendo le medesime operazioni descritte per il comando `adddom`, e di caricare sul server tutti i contenuti, impostando come owner delle singole risorse l'utente che ha eseguito l'operazione e come gruppo di appartenenza il gruppo *AdminGroup*.

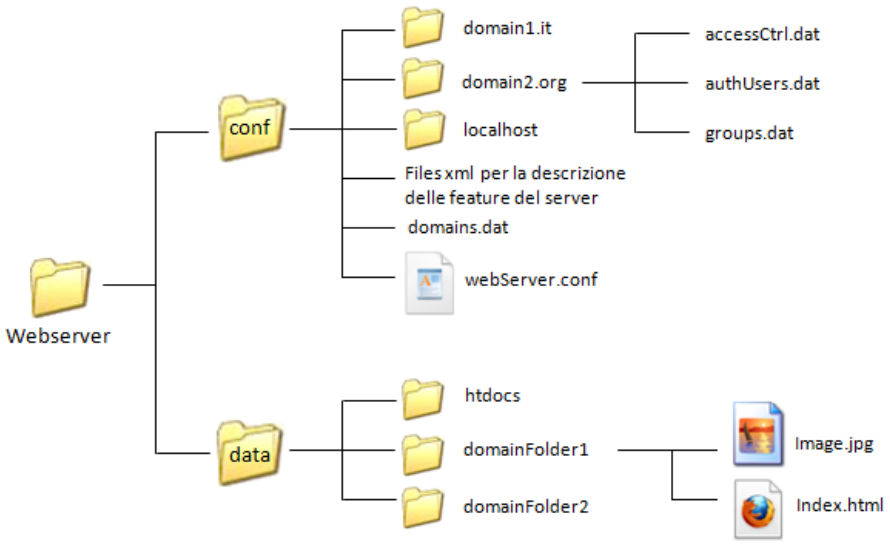


Figura 5.2: Esempio della struttura su disco di Webserver.

La figura 5.2 mostra la struttura su disco che si ottiene eseguendo i comandi riportati in precedenza.

5.2 Prestazioni

In questa sezione del documento viene riportata un'analisi delle prestazioni ottenute per il plug-in, valutando in particolare: il tempo di risposta del server, il numero di thread attivati e l'utilizzo di memoria e di CPU da parte di Webserver. I test effettuati riguardano un singolo nodo all'interno di un web server distribuito e sono stati eseguiti realizzando delle simulazioni in Java ed utilizzando la versione di prova della suite **Webserver stress tool 7**[1], un programma che consente di effettuare stress test per i web server.
I risultati riportati nei prossimi paragrafi fanno riferimento ai test effettuati per valutare, innanzitutto, le prestazioni di un nodo del server in corrispondenza di una singola richiesta e, successivamente, le performance all'aumentare del traffico in ingresso. Lo scenario utilizzato per questa seconda situazione prevede test della durata di 10 minuti, all'interno dei quali il numero di client connessi al web server cresce di una unità ogni 50 secondi, fino ad ottenere 10 client contemporanei (limite imposto dalla versione prova del programma). Ogni client attivo effettua le richieste in modo random, lasciando passare tra due richieste successive al massimo 5 secondi.

5.2.1 Tempo di risposta

Il tempo di risposta, cioè il tempo che intercorre tra l'invio della richiesta di una risorsa tramite browser e la visualizzazione della stessa a video, rappresenta l'indice di prestazione più importante per un web server in quanto un processing lento viene percepito dall'utente finale come un malfunzionamento del servizio.

Il **tempo di risposta medio** relativo ad una **singola richiesta** ottenuto per il plug-in Webserver è molto buono e varia tra i **60 ms** e i **170 ms** a seconda del numero di header inseriti. Ovviamente, nel caso in cui il nodo interpellato sia un RelayNode, il tempo di risposta medio percepito dall'utente finale risulta essere circa il doppio di quello riportato, in quanto comprende il redirect trasparente della richiesta verso un HostNode.

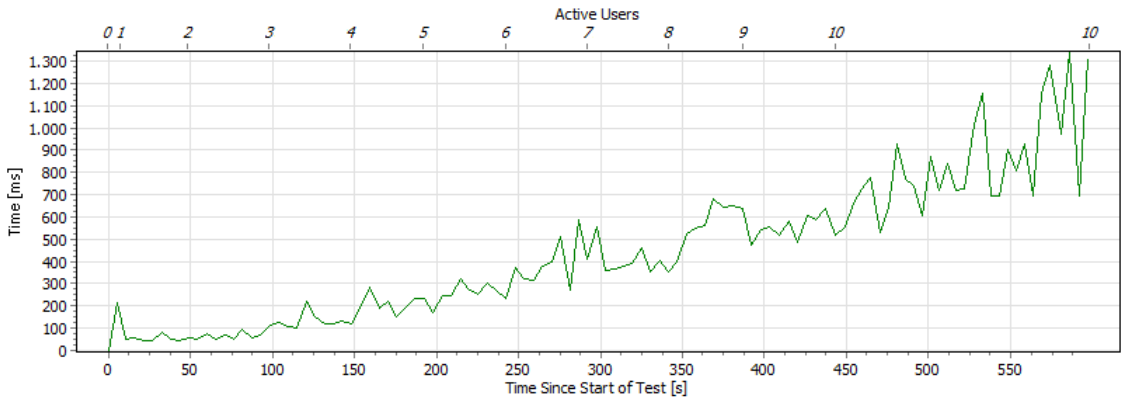


Figura 5.3: Tempo medio di risposta di PariWEB in funzione del carico in ingresso.

[1] [10]

Per quanto riguarda il **test** sopra descritto, le prestazioni inizialmente ottenute sono quelle riportate in figura 5.3. Il grafico evidenzia un notevole aumento del tempo di risposta al crescere dei clienti connessi e di conseguenza del numero di richieste in ingresso, fino a raggiungere picchi di 1,3 secondi per il processing di una richiesta. Questi tempi non sono soddisfacenti, soprattutto considerando il limitato numero di client contemporanei che la suite consente di utilizzare, per cui si è resa necessaria una fase di ottimizzazione atta a ridurre il tempo di risposta medio del server nel caso di richieste simultanee.

La fase di ottimizzazione ha permesso di ottenere le prestazioni riportate in figura 5.4 affiancate ai risultati ottenuti per due dei web server più diffusi: **Apache Server 2.2**[2] e **Internet Information Services 7.0 (IIS7)**[3].

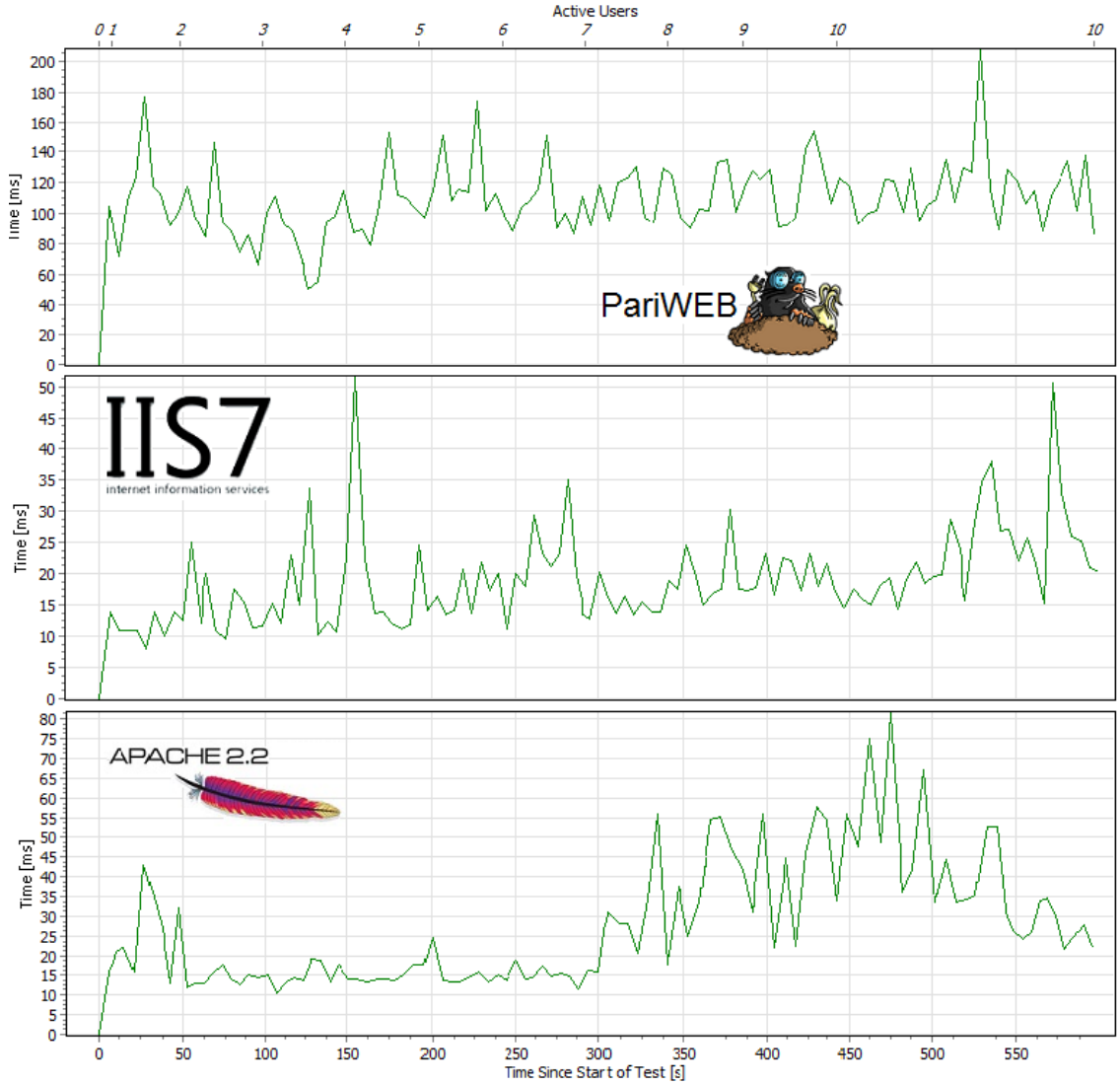


Figura 5.4: Andamento del tempo medio di risposta in funzione del carico in ingresso per i web server PariWEB, Apache 2.2 e IIS7.

[2] [8]
[3] [9]

I grafici mostrano l'andamento del tempo di risposta dei web server riportando l'istante in cui avviene un aumento del numero di client che generano richieste (nella parte superiore dell'immagine) e il tempo trascorso dall'inizio del test (nella parte inferiore dell'immagine). Innanzittutto è importante sottolineare il notevole miglioramento ottenuto per PariWEB con il processo di ottimizzazione, consentendo al web server sviluppato di raggiungere un **tempo medio di risposta** pari a **120 ms** praticamente indipendentemente dal numero di richieste contemporanee ricevute. Si nota che all'aumentare dei client connessi il server continua a comportarsi in modo soddisfacente, presentando alcuni picchi che rimangono al di sotto dei 200 ms.
Il confronto con le prestazioni ottenute dai web server Apache 2.2 e IIS7, i quali presentano tempi di risposta inferiori rispettivamente a 80 ms e a 50 ms, mette in evidenza che le performance del plug-in Webserver sono leggermente inferiori ma allo stesso tempo molto buone se si considera che:

- i server Apache 2.2 e IIS7 sono le soluzioni commerciali più diffuse e performanti attualmente presenti nel mondo del Web;
- l'accesso e la lettura delle risorse web salvate su disco, da parte di PariWEB, deve passare attraverso il plug-in LocalStorage, e questo passaggio aggiuntivo rappresenta il 60% del tempo totale necessario al processing di una richiesta. Se si riuscisse ad ottimizzare anche questa fase dell'esecuzione, le prestazioni del plug-in WebServer potrebbero essere migliorate ulteriormente.

5.2.2 Utilizzo della CPU

Il plug-in, una volta attivato, comporta un consumo di CPU inferiore all'1% in situazione idle, mentre l'utilizzo medio in corrispondenza del processing di una singola richiesta Http è pari a 3.5%. Il caso peggiore si ottiene quando il nodo riceve più richieste contemporaneamente: in questo caso il consumo aumenta fino a raggiungere picchi intorno del 30%, per poi tornare ad un valore inferiore all'1% non appena tutte le richieste sono state processate.

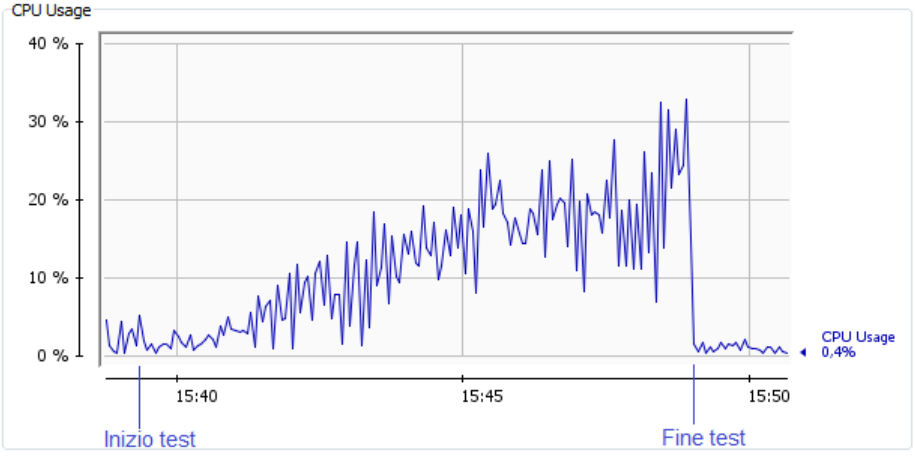


Figura 5.5: Utilizzo della CPU da parte del plug-in Webserver.

L'immagine 5.5 descrive l'utilizzo di CPU da parte del plug-in durante l'esecuzione del test descritto. Il consumo di CPU cresce all'aumentare del numero di client connessi e di conseguenza del numero di richieste pervenute, ma quando il test termina il server torna immediatamente in una situazione idle con un consumo inferiore all'1%.

5.2.3 Utilizzo di memoria

Dal punto di vista dell'utilizzo di memoria il plug-in Webserver presenta delle ottime prestazioni, con un consumo medio inferiore ai 10MB.

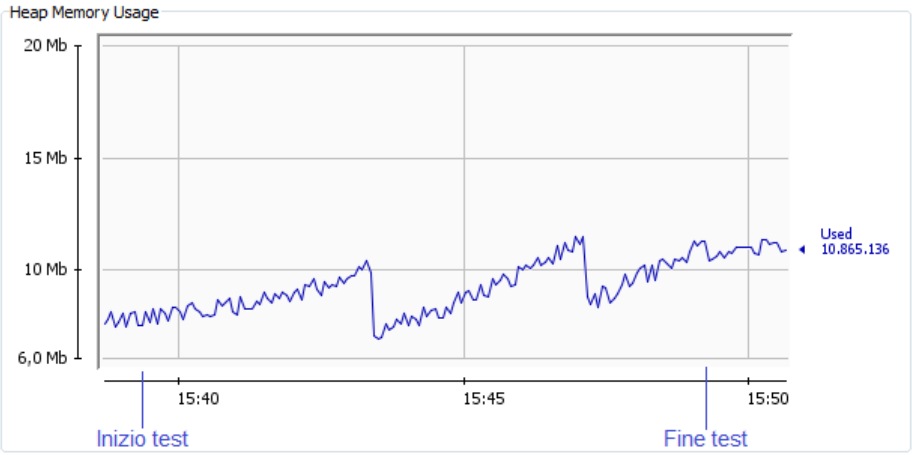


Figura 5.6: Utilizzo della memoria da parte del plug-in Webserver.

La figura 5.6, relativa al test descritto in precedenza, mette in evidenza come al crescere del numero di richieste processate dal server il consumo di memoria cresca, ma allo stesso tempo permette di valutare che gli oggetti istanziati vengono correttamente dereferenziati in quanto l'intervento del Garbage Collector di Java riporta l'utilizzo di memoria a valori molto vicini a quelli iniziali.

5.2.4 Threads

Uno degli aspetti considerati per valutare le prestazioni di Webserver, è rappresentato dal numero di thread in esecuzione quando il plug-in è attivo. Per quanto riguarda la modalità locale, l'unico thread è quello responsabile di rimanere in ascolto delle richieste in ingresso al nodo, per cui non vi sono problemi di spreco di risorse. Nel caso distribuito, invece risultano attivi:

- `IncomingRequestsListener`: rimane in attesa delle richieste Http in ingresso al nodo;
- `InternalListenerTh`: rimane in attesa di eventuali messaggi interni al server inviati al nodo;
- `WebServerLayer`: gestisce l'aspetto di distribuzione mediante la libreria DiESeL.

Inoltre, se il nodo in questione è l'HostNode leader, anche `CoordinationTh` è attivo per segnalare in modo periodico a tutti i nodi che compongono il server la sua presenza. Oltre a questi thread sempre attivi, il plug-in avvia:

– un'istanza di `ManageRequestTh` per ogni richiesta Http ricevuta,

– un'istanza di `ElectionTh` ogni volta che è necessaria un'elezione.

Analizzando il numero di thread relativi al plug-in Webserver è stato possibile constatare che gli unici thread sempre attivi sono effettivamente quelli elencati sopra, e che, anche nel caso di anomalie o errori nell'esecuzione delle richieste pervenute, le istanze di `ManageRequestTh` e `ElectionTh` vengono terminate correttamente senza determinare uno spreco delle risorse del server.

Capitolo 6

Conclusioni e sviluppi futuri

In questo capitolo verrà dato spazio ad alcune riflessioni personali riguardanti il lavoro svolto e, successivamente, verrà posta l'attenzione su alcuni possibili sviluppi futuri individuati per il plug-in Webserver.

6.1 Conclusioni

PariWEB si è rivelato un progetto molto interessante e gratificante, in quanto permette di confrontarsi e misurarsi con tutti gli aspetti, e le relative difficoltà, legati alla realizzazione di un software di questa importanza e complessità: dallo studio e dall'analisi del contesto in cui inserire il progetto all'individuazione dei requisiti, dalla progettazione della struttura del plug-in e dei singoli componenti alla loro realizzazione, dal testing del codice prodotto alla simulazione degli aspetti legati alla distribuzione, dall'analisi delle performance all'ottimizzazione.
Lavorare all'interno di PariPari, inoltre, consente di acquisire un'importante esperienza nella gestione di un progetto e di un team di sviluppatori, nonché nell'utilizzo di particolari strumenti per la sincronizzazione, per il testing e per lo sviluppo.

Il presente documento ha descritto l'intero processo che ha portato alla realizzazione di un modulo per la rete PariPari in grado di fornire all'utente finale un servizio di web hosting completo, configurabile e performante. Il lavoro realizzato ha portato ad una prima versione del plug-in, completa di tutti quegli aspetti ritenuti fondamentali all'inizio del progetto. I risultati ottenuti sono soddisfacenti, sia in termini di prestazioni, che per quanto riguarda il conseguimento degli obiettivi posti in fase di analisi.

Lo sviluppo di Webserver proseguirà con l'integrazione dell'attuale modulo all'interno di una release di PariPari, senza per questo interrompere il processo di sviluppo e mantenimento del plug-in.

6.2 Sviluppi futuri

In questa sezione verranno evidenziati e analizzati dei possibili sviluppi, alcuni dei quali già anticipati durante la descrizione delle diverse fasi del progetto, da realizzare per una successiva versione del plug-in Webserver. In particolare verrà posta l'attenzione su alcuni aspetti di Webserver che possono essere modificati e migliorati e su alcune idee che potrebbero essere implementate in futuro per potenziare il servizio offerto.

Il primo aspetto che affrontiamo riguarda l'utilizzo del plug-in **DistributedStorage** per il salvataggio dei contenuti del web server. Attualmente le risorse ospitate sono effettivamente salvate sul disco dei nodi Host che compongono il server, ma in futuro il plug-in Webserver sfrutterà al massimo la rete PariPari, salvando i contenuti in modo distribuito all'interno della rete. La modifica riguarderà il modulo `ServeRequest` responsabile di generare la risposta Http: esso dovrà recuperare dalla rete la risorsa richiesta, ricostruirla (in quanto i file vengono salvati in rete in piccoli blocchi), e poi inserirla nel messaggio di risposta. Questa variazione è immediata, in quanto il plug-in è stato sviluppato tenendo in considerazione questa futura estensione. In termini di tempo di risposta, questa tecnica introduce un ritardo dovuto alla latenza necessaria a recuperare e ricostruire il contenuto. Per questo motivo sarà necessario realizzare un **meccanismo di cache** che consenta di mantenere in locale i contenuti più richiesti.

Il secondo aspetto che affrontiamo riguarda la scelta di replicare semplicemente le strutture e i contenuti (finchè non verrà utilizzato il plug-in `DistributedStorage`) del server ottenendo HostNode tutti uguali. Questa soluzione potrebbe essere modificata suddividendo i domini gestiti dal web server tra i diversi nodi di tipo Host e realizzando una redirezione interna di tipo **content-aware**, cioè basata sulla risorsa richiesta. Una tecnica di questo tipo ha numerosi vantaggi. Per esempio, se le richieste che riguardano uno stesso contenuto vengono inoltrate sempre allo stesso nodo, questo può essere in grado di mettere nella cache tale risorsa riducendo il tempo di risposta. Inoltre, con un approccio di questo tipo, si può ridurre notevolmente lo spreco di memoria dovuto alla tecnica di semplice replicazione, suddividendo i contenuti tra i nodi del server utilizzando un grado di ridondanza inferiore (se gli HostNode sono N, invece di avere N copie di una risorsa è possibile avere un numero di copie inferiore).

Ulteriori sviluppi futuri riguardano la ricerca di una tecnica che consenta a tutti gli HostNode di eseguire i metodi Http non sicuri senza dare origine a conflitti, e la realizzazione di un servizio di proxy web. Per quanto riguarda il primo aspetto la soluzione attuale, che prevede che queste operazioni siano consentite al solo nodo coordinatore, rappresenta un collo di bottiglia per il server distribuito, anche se non troppo rilevante se si considera l'utilizzo di richieste PUT, POST e DELETE rispetto richieste a GET.
Il secondo aspetto riguarda la possibilità di fornire all'utente la scelta se utilizzare il proprio nodo per realizzare un web server oppure un proxy. Un **proxy web** è un programma che si interpone tra un client e un server, inoltrando le richieste e

le risposte dall'uno all'altro, e viene utilizzato per svariati motivi tra cui: fornire un servizio di caching, realizzare politiche di monitoraggio e controllo del traffico, garantire un maggiore livello di sicurezza e privacy. Questa estensione del servizio offerto dal plug-in Webserver è in sviluppo e, attualmente realizza un semplice proxy che fa da intermediario tra i client e i server senza però fornire nessuno dei servizi aggiuntivi sopracitati.

L'ultimo aspetto citato in questa sezione, rappresenta un'idea legata alla capacità del server distribuito, di adattare le sue dimensione alle diverse situazioni di carico. La capacità del server di individuare delle **periodicità** nelle situazioni di alto carico per potersi adattare in modo preventivo potrebbe rappresentare un notevole plus per il servizio offerto dal web server di PariPari.

Appendice A

Classi e metodi

A.1 AccessControl_Module

Classe che realizza il modulo responsabile, all'interno del Webserver, di verificare che la richiesta giunta sia conforme alle politiche di accesso definite per la risorsa coinvolta. Attraverso il costruttore viene fornita la hashtable che descrive il messaggio Http in ingresso al server, da cui `AccessControl_Module` ricava il metodo, la risorsa e l'utente coinvolti.

- `checkIfAuthorized()`: metodo che verifica se la richiesta fornita al modulo tramite costruttore può essere eseguita sulla risorsa coinvolta. Se l'operazione è autorizzata il metodo restituisce ALLOW, altrimenti ritorna DENY.

A.2 CoordinationTh

Classe che realizza il thread utilizzato dal nodo leader per comunicare periodicamente che è attivo e per fornire la lista aggiornata di HostNode che compongono il server.

A.3 DateRFC1123

Classe che permette di ottenere la data nel formato richiesto dal protocollo HTTP.

- `getDateRFC1123()`: restituisce la data secondo lo standard RFC1123.

A.4 ElectionTh

Classe che realizza l'algoritmo di elezione utilizzato dagli HostNode per determinare il nodo coordinatore. Il thread viene attivato alla ricezione di un messaggio di tipo ELECTION o quando il nodo si accorge che l'attuale leader non è più attivo.

A.5 IncomingRequestsListener

Classe che rimane in attesa delle richieste in ingresso al server. Per ogni richiesta ricevuta, controlla la disponibilità di risorse, attiva un'istanza di `ManageRequestTh` per il suo soddisfacimento ed aggiorna la `TableThread` del server.

A.6 InfoThread

Record utilizzato all'interno di `TableThread` per memorizzare il nome, il puntatore del thread `ManageRequestTh` in esecuzione, ed il momento, espresso in millisecondi, in cui è stato avviato.

- `getDate()`, `getName()`, `getThread()`: restituisce rispettivamente momento in millisecondi in cui il thread è stato lanciato, nome del thread e puntatore a quest'ultimo.

A.7 InternalListenerTh

Questa classe viene utilizzata dalle istanze di `WebServerDistrNode` (sia `WebServerHostNode` che `WebServerRelayNode`) per la ricezione di eventuali messaggi interni ad un web server distribuito; in base alla tipologia di comunicazione ricevuta la classe esegue le azioni necessarie, per esempio una procedura di aggiornamento di una struttura dati o l'attivazione dell'algoritmo di elezione.

A.8 InternalWebServerMessage

Questa classe definisce la struttura dei messaggi utilizzati dai nodi del server per comunicare tra loro quando il plug-in Webserver viene utilizzato in modalità distribuita. In particolare la classe prevede che, oltre al corpo del messaggio, una comunicazione interna contenga: un msgID, l'indicazione del tipo di comunicazione e dell'istante in cui è stato inviato, un mittente ed un destinatario.

A.9 ManageDistributedList

Classe che definisce una struttura dati generica, basata su una hashtable, per gestire informazioni che si riferiscono ai singoli domini e che su disco sono salvati in percorsi differenti. La struttura, a partire dall'elenco dei domini gestiti dal server fornita tramite il costruttore, è in grado di caricare i dati dalle diverse cartelle, e salvare su disco solo i dati dei domini che hanno subito modifiche.

- `loadFromDisk()`: metodo che carica i dati dalle cartelle dei singoli domini, popolando la struttura utilizzando come chiave il nome del dominio in questione;

- `writeOnDisk(String path, V value)`: metodo che scrive su disco, ed in particolare nella posizione indicata dal path passato come parametro, il valore V;

- `writeOnDisk(IManageList<String, String> domains)`: metodo utilizzato per inizializzare, all'interno di ogni cartella della lista di domini passati come parametro, le strutture dati utilizzate dalla classe;

- `addElem(K key, V value)`: metodo che consente di aggiungere un elemento, creando le strutture necessarie solo all'interno della cartella del dominio coinvolto. Il metodo ritorna true se l'operazione di scrittura su disco è avvenuta correttamente, false altrimenti;

- `remElem(K key)`: metodo che consente di rimuovere dalla struttura l'elemento individuato dalla chiave passata come parametro. Il metodo elimina anche la corrispondente struttura all'interno della cartella del dominio coinvolto. Anche in questo caso il metodo ritorna true se la cancellazione da disco è avvenuta correttamente, false altrimenti;

- `getElem(K key)`; metodo che restituisce il valore della coppia individuata dalla chiave passata come parametro.

A.10 ManageGroups

Questa classe rappresenta una specializzazione di `ManageDistributedList` per la gestione dei gruppi di utenti dei singoli domini e del web server. In particolare `ManageGroups` realizza una hashtable a due livelli, utilizzando come chiave il nome di dominio e come valore una hashtable che associa la lista di utenti al nome del relativo gruppo di appartenenza.

A.11 ManageList

Classe che definisce una struttura dati generica, basata su una hashtable, per gestire informazioni che riguardano l'intero server, ed in particolare la lista dei domini ospitati. La struttura è stata realizzata in modo che in seguito a operazioni che ne modificano lo stato, venga salvata automaticamente su disco.

- `loadFromDisk()`: metodo che carica i dati salvati in precedenza all'interno della cartella "conf" del plug-in;

- `writeOnDisk()`: metodo che scrive su disco, all'interno della cartella "conf" del plug-in, la struttura dati;

- `addElem(K key, V value)`: metodo che consente di aggiungere un elemento alla struttura. Il metodo ritorna true se l'operazione di scrittura su disco è avvenuta correttamente, false altrimenti;

- `remElem(K key)`: metodo che consente di rimuovere dalla struttura l'elemento individuato dalla chiave passata come parametro. Anche in questo caso il metodo ritorna true se la cancellazione da disco è avvenuta correttamente, false altrimenti;

- `getElem(K key)`; metodo che restituisce il valore della coppia individuata dalla chiave passata come parametro.

A.12 ManageProperties

Classe utilizzata per la gestione delle proprietà del plug-in Webserver.

- `getAllProperties()`: restituisce una hashtable contenente tutte le proprietà (nome, valore) associate al plug-in chiamante;

- `getProperties(String[] param)`: restituisce una hashtable contenente solamente le proprietà indicate nell'array passato come parametro d'ingresso e associate al plug-in chiamante;

- `setProperties(Hashtable<String, String> arg)`: crea un file (nomeplugin_properties.conf), all'interno della cartella "conf" associata al plug-in chiamante, contenente le proprietà indicate nell'hash table passata come parametro in ingresso.

A.13 ManageRequestTh

Questa classe realizza un thread responsabile del processing delle richieste Http ricevute. In particolare `ManageRequestTh` svolge un ruolo di controllo e supervisione sul processo, chiamando in successione i moduli `ParserHTTP_Module`, `AccessControl_Module`, `ServerRequest_Module` e `SendResponse_Module` e gestendo i risultati e gli eventuali errori ottenuti.

A.14 ManageUsers

Questa classe rappresenta una specializzazione di `ManageDistributedList` per la gestione degli utenti autorizzati dei singoli domini e del web server. In particolare `ManageUsers` realizza una hashtable a due livelli, utilizzando come chiave il nome di dominio e come valore una hashtable che associa alla password codificata relativo nome utente.

A.15 ParserHTTP_Module

Classe che effettua il parsing di una richiesta Http, passata al costruttore tramite un `InputStream`, e ne salva i dati ottenuti all'interno di una hashtable. La richiesta deve essere conforme allo standard Http 1.0 o 1.1.

- parse: effettua il parsing di un messaggio di richiesta ed inserisce i dati ottenuti all'interno di un'hashtable. Questo metodo a sua volta chiama i seguenti metodi per processare le diverse parti della richiesta ricevuta: `parseMethod()`, `parseURI()`, `parseHTTPVersion()`, `parseParameters()`, `parsePathAndFileName()`, ;

- `convert()`: permette di convertire una stringa passata come parametro d'ingresso in una comprensibile al web server eliminando lo schema percentuale.

A.16 Permission

Classe che descrive la politica di accesso di una risorsa del web server, specificandone: il proprietario, il gruppo a cui appartiene, la precedenza "allow-deny" o "deny-allow" da utilizzare e le autorizzazioni positive e negative per i tre gruppi di utenti previsti dal modello (proprietario, gruppo della risorsa, resto del mondo).

- `defineAccessPolicy(Authorization[] auths, IAccessRule[] actions, String user)`: metodo che consente di definire una o più autorizzazioni per l'utente passato come parametro;

- `check(String user, Actions action)`: metodo che restituisce l'autorizzazione positiva o negativa per l'operazione costituita dall'azione e dall'utente passati come parametri;

A.17 SendResponse_Module

Classe che si occupa di inoltrare un messaggio di risposta Http, descritto da un oggetto `TableResponse`, al client che ha effettuato la richiesta.

A.18 ServeRequest_Module

Modulo utilizzato dalla classe `ManageRequestTh` per processare la richiesta assegnatagli dal plug-in Webserver. A seconda del valore del campo *Method* viene generata la risposta Http e, se necessario, recuperato il file.

- `serve(HttpMethodEnum method, Hashtable<String, String> tReq)`: metodo che si occupa di processare la richiesta pervenuta al server e descritta dalla hashtable passata come parametro e di creare la relativa risposta. A seconda del metodo Http richiesto (descritto dall'enum `HttpMethodEnum`) viene eseguito uno tra i seguenti metodi: `serveGET()`, `servePUT()`, `servePOST()`, `serveDELETE()`, `serveHEAD()`, `serveTRACE()` e `serveOPTIONS()`.

Il metodo Http CONNECT non è stato ancora implementato, per cui se il server riceve una richiesta di questo tipo, la classe genera una risposta "NotImplemented".

A.19 TableResponse

Questa classe permette di costruire un messaggio di risposta Http tra quelli previsti dal protocollo Http 1.1, popolando un'hashtable con coppie <*Nome Campo*, *Valore Campo*>. Alcuni dei metodi che mette a disposizione sono i seguenti:

- `getTable()`: restituisce l'hashtable contenente i campi che formeranno il messaggio di risposta.
- `setOK(String mime, String length, Date lastMod)`: imposta la risposta a "OK" ricevendo in ingresso il valore del campo `Content-Type`, la dimensione del messaggio di risposta e, se necessario, la data di ultima modifica della risorsa indicata dalla richiesta;
- `setTemporaryRedirection(String url)`: imposta la risposta a "Temporary Redirection" ricevendo in ingresso la url a cui è possibile trovare la risorsa richiesta;
- `setContinue()`: imposta la risposta a "Continue";
- `setBadRequest()`: imposta la risposta a "Bad Request";
- `setUnauthorized()`: imposta la risposta a "Unauthorized";
- `setFileNotFound()`: imposta la risposta a "File Not Found";
- `setInternalServerError()`: imposta la risposta ad "Internal Server Error";
- `setNotImplemented()`: imposta la risposta a "Not Implemented";

A.20 TableThread

Questa classe viene utilizzata per la gestione della tabella formata da `InfoThread` e contenente le informazioni riguardanti i `ManageRequestTh` in esecuzione per processare le richieste pervenute al nodo del web server.

- `isHere()`: restituisce true se un determinato InfoThread passato in ingresso si trova all'interno della tabella, altrimenti false;
- `put(IInfoThread elem)`: inserisce il record `InfoThread` passato come parametro in ingresso all'interno della tabella. Se l'operazione ha successo restituisce true, altrimenti false;
- `remove(String name)`: rimuove l'`InfoThread` associato al nome passato come parametro d'ingresso. Se l'operazione ha successo restituisce true, altrimenti false.

A.21 Utilities

Classe che fornisce alcuni metodi statici utili all'interno delle altre classi del progetto per svolgere delle attività comuni.

- `checkpath(String pp)`: metodo che verifica che il percorso fornito sia un percorso valido;
- `checkDomain(String domain)`: metodo utilizzato per eliminare dal nome di dominio inserito dall'utente, se presenti, i prefissi "http://", "www." o entrambi;
- `checkPathSeparator(String path)`: metodo che sostituisce i caratteri utilizzati come separatori all'interno del percorso in modo conforme al sistema operativo presente sul nodo. In questo modo, per esempio, il passaggio di informazioni contenenti percorsi di risorse da nodi Linux a nodi Windows non causa errori dovuti all'utilizzo di separatori differenti;
- `encodeB64(String toencode)` e `decodeB64(String dec)`: metodi che realizzano rispettivamente la codifica e la decodifica in Base64;
- `removeSlash(String str)`: metodo che rimuove eventuali slash presenti all'inizio e alla fine di un percorso;
- `validateIp(String ip)`: metodo che verifica se l'IP fornito è un indirizzo IP4 o IP6 valido.

A.22 WebServerACL

Questa classe rappresenta una specializzazione di `ManageDistributedList` per la gestione delle politiche d'accesso delle risorse dei singoli domini e del web server. In particolare `WebServerACL` realizza una hashtable a due livelli, utilizzando come chiave il nome di dominio e come valore una hashtable che associa ad ogni risorsa un oggetto `Permission` che ne descrive la politica di accesso definita.

A.23 WebServerConsole

Questa classe si occupa della gestione della console del plug-in Webserver. All'interno di `WebServerConsole` vengono definite le azioni necessarie per eseguire ogni comando messo a disposizione dell'utente.

A.24 WebResourceInfo

Questa classe viene utilizzata per individuare alcune informazioni relative ad una risorsa del server coinvolta da una richiesta Http. A partire dalla hashtable che descrive la richiesta ricevuta, questa classe estrae i seguenti dati riguardanti il contenuto: il nome, il percorso, il percorso assoluto, il dominio a cui fa riferimento e la cartella all'interno della quale si trova.

A.25 WebServer

Questa classe è usata dal Core per lanciare il thread principale del plug-in WebServer. Tramite alcune finestre di dialogo l'utente è in grado decidere se utilizzare il WebServer in modalità locale o in modalità distribuita, e in questo secondo caso scegliere se attivare un nuovo web server o semplicemente consentire che il proprio nodo venga aggregato ad un server già esistente.

- `checkManageProperties()`: metodo per la gestione del risultato derivante dalla classe `ManageProperties` che estrae le informazioni necessarie al funzionamento del plug-in Webserver impostate nel file di configurazione; nel caso in cui il file contenente le properties non esista ne crea uno con i valori di default;

- `createDirs()`: metodo che crea su disco la struttura del plug-in, in particolare le cartelle "conf" e "data".

A.26 WebServerDistrNode

Classe astratta che rappresenta un nodo di un web server distribuito, al cui interno sono presenti tutti i metodi e le strutture che riguardano gli aspetti di distribuzione comuni sia a nodi `WebServerHostNode` che a nodi `WebServerRelayNode`. `WebServerDistrNode` contiene, inoltre, il riferimento al thread `InternalListenerTh` responsabile della gestione dei messaggi interni ricevuti dal nodo.

- `IAmCoordinator()`: metodo che restituisce true se il nodo è il coordinatore degli HostNode, false altrimenti;

- `getNodeList()`: metodo che restituisce la lista dei nodi che compongono il server distribuito;

- `getNodeStatus()`: metodo che restituisce il tipo di nodo in questione, cioè se il nodo è un HostNode o un RelayNode;

- `getRandomRealNode()`: metodo che restituisce il descrittore di un HostNode estratto casualmente dalla `WebServerNodeList` del nodo;

- `sendBroadcastMsg(Object[] msgText, MessageType type)`: metodo che consente di inviare un messaggio in broadcast a tutti i nodi che compongono il server distribuito;

- `sendUnicastMsg(String recAdr, Integer port, Object[] msgText, MessageType type)`: metodo che consente di inviare un messaggio unicast sulla porta interna del nodo individuato dal campo `recAdr`;

- `fromHostToRelay(IWebServerHostNode hNode)`: metodo che realizza la transizione del nodo Host passato come parametro in RelayNode. Ritorna true se la transizione viene completata, false altrimenti;

- `fromRelayToHost(IWebServerRelayNode fNode)`: metodo che realizza la transizione del nodo Relay passato come parametro in HostNode. Ritorna true se la transizione viene completata, false altrimenti.

A.27 WebServerHostNode

Classe che rappresenta un nodo Host di un web server distribuito. `WebServerHostNode` contiene i riferimenti ai thread `CoordinationTh` ed `ElectionTh` quando questi sono attivi.

- `getMyID()`: metodo che ritorna l'ID dell'HostNode;
- `addHostNodeToServer()`: metodo che consente di attivare la procedura di duplicazione del nodo. Ritorna true se la procedura termina correttamente, false altrimenti;
- `removeHostNodeFromServer()`: metodo che consente di ridurre di uno il numero di HostNode di cui è composto il web server. Ritorna true se la procedura termina correttamente, false altrimenti;
- `syncHostNodes(String[] parameters)`: metodo che consente di attivare la procedura per l'aggiornamento degli HostNode che compongono il server. Ritorna true se l'aggiornamento termina correttamente, false altrimenti.
- `receivedMessage(IInternalWebServerMessage msg)`: metodo utilizzato per definire il comportamento del nodo in seguito alla ricezione di un messaggio in broadcast.

A.28 WebServerLayer

Questa classe implementa `IServerLayer` e viene utilizzata dal plug-in Webserver per creare il Distributore, cioè quello strato che si inserisce tra il livello applicativo e quello della rete PariPari per rendere trasparente la distribuzione.

A.29 WebServerLocalNode

Questa classe rappresenta il web server quando il plug-in viene attivato in modalità locale. Essa richiama semplicemente i metodi messi a disposizione dalla classe `WebServerNode` per fornire il servizio di web hosting in locale.

A.30 WebServerNode

Classe astratta che rappresenta un nodo del web server, realizzando al suo interno tutti i metodi indipendenti dalla modalità di utilizzo del plug-in Webserver.

- `initDataStruct()`: metodo utilizzato dal costruttore della classe per inizializzare tutte le strutture dati del nodo necessarie a fornire il servizio di web hosting;
- `initIncomingRequest()`: metodo per inizializzare le strutture necessarie a rimanere in ascolto sulla porta del nodo predefinito;
- `finishManageRequest()`: metodo utilizzato dalla classe `ManageRequestTh` per informare il server del soddisfacimento della richiesta. Aggiorna la `TableThread` eliminando il thread che ha terminato;
- `deleteDir(String dirPath)`: metodo che consente di eliminare la struttura su disco del plug-in Webserver. Questo metodo viene utilizzato in particolare quando si verifica la transizione di un HostNode in un RelayNode.
- `startIncomingRequestsListener()`: metodo che crea e avvia il thread `IncomingRequestsListener` responsabile di rimanere in attesa di richieste Http sulla porta del server specificata.

A.31 WebServerNodeDescriptor

Questa classe viene utilizzata all'interno del progetto per descrivere i nodi che compongono un web server distribuito. Un'istanza di questa classe contiene l'indirizzo su cui è disponibile il servizio di web hosting, il numero delle porte utilizzate per la libreria DiESeL e per le comunicazioni interne, il tipo di nodo e, se si tratta di un HostNode, l'ID relativo.

A.32 WebServerNodeList

Classe che fornisce una struttura utile a descrivere la composizione del server nel caso di utilizzo del plug-in in modalità distribuita. Tra i metodi che `WebServerNodeList` mette a disposizione dei nodi per la gestione dell'evoluzione del server citiamo:

- `getRandomHostNode()`: sceglie casualmente uno dei nodi Host che compongono il server e ne restituisce il descrittore;
- `getRandomRelayNode()`: sceglie casualmente uno dei nodi Relay che compongono il server e ne restituisce il descrittore;
- `addHostNode(int ID, String addr, String portDescr, String info, boolean isCoord)`: aggiunge alla lista degli HostNode il nodo descritto dai parametri passati al metodo in seguito alla ricezione di un messaggio broadcast FROM_RELAY_TO_HOST;
- `addRelayNode(String nodeAddr, String portDescr, String info)`: aggiunge alla lista dei RelayNode il nodo descritto dai parametri passati al metodo in seguito alla ricezione di un messaggio broadcast HELLO_MSG o FROM_HOST_TO_RELAY;

- `removeRelayNode(String nodeAddr, String portDescr)`: consente di rimuovere dalla lista dei nodi Relay il nodo descritto dai parametri passati al metodo;
- `removeHostNode(String nodeAddr, String portDescr)`: consente di rimuovere dalla lista dei nodi Host il nodo descritto dai parametri passati al metodo;
- `removeNode(String nodeAddr, int dieselPort)`: consente di rimuovere il nodo descritto dalla lista in cui è presente;
- `fromHostToRelay(String hostNodeAddres, String portDescr)`: metodo utilizzato dai nodi Host per mantenere aggiornata la composizione del web server in seguito alla ricezione in broadcast del messaggio FROM_HOST_TO_RELAY che notifica l'avvenuta transizione del nodo mittente;
- `fromRelayToHost(int id, String relayNodeAddress, String portDescr)`: metodo utilizzato dai nodi Host per mantenere aggiornata la composizione del web server in seguito alla ricezione in broadcast del messaggio FROM_RELAY_TO_HOST che notifica la transizione del nodo mittente da RelayNode a HostNode.

A.33 WebServerRelayNode

Questa classe rappresenta un nodo di tipo Relay in un web server distribuito. Utilizza i metodi messi a disposizione delle classi che estende, `WebServerDistrNode` e di conseguenza `WebServerNode`, e definisce i comportamenti del nodo in seguito alla ricezione di messaggi broadcast.

- `receivedMessage(IInternalWebServerMessage msg)`: metodo utilizzato per definire il comportamento del nodo in seguito alla ricezione di un messaggio in broadcast.

Bibliografia

[1] Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T., *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, 1999,
url: http://www.ietf.org/html/rfc2616.

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

[3] Fielding R., Gettys J., Mogul J., Frystyk H., *RFC 2145:Use and Interpretation of HTTP Version Numbers* , 1997,
url: http://tools.ietf.org/html/rfc2145.

[4] Berners-Lee T., Fielding R., Masinter L., *RFC 2396: Uniform Resource Identifiers (URI) – Generic Syntax* , 1998,
url: http://tools.ietf.org/html/rfc2396.

[5] Braden R., *RFC 1123: Requirements for Internet Hosts – Application and Support* , 1989,
url: http://http://tools.ietf.org/html/rfc1123.

[6] Vixie P., Thomson S., Rekhter Y., Bound C., *RFC 2136: Dynamic Updates in the Domain Name System*, 1997,
url: http://http://tools.ietf.org/html/rfc2136.

[7] Franks J., Hallam-Baker P., Hostetler J., Lawrence S., Leach P., Luotonen A., Stewart L., *RFC 2617: HTTP Authentication – Basic and Digest Access Authentication*, 1999,
url: http://http://tools.ietf.org/html/rfc2617.

[8] *Web Server Apache 2.2* ,
url: http://httpd.apache.org/docs/2.1.

[9] *Internet Information Services 7.0* ,
url: http://www.iis.net/.

[10] *Webserver stress tool 7*,
url: http://www.paessler.com/webstress.

[11] Peterson Larry L., Davie Bruce S., *Reti di calcolatori*, Apogeo, Milano, 2004.

[12] Pressman Roger S., *Principi di ingegneria del software*, McGraw-Hill, Milano, 2005.

[13] Tanenbaum Andrew S., Van Steen Maarten , *Distributed systems: principles and paradigms*, Pearson Education, 2007.

[14] Bertasi P., *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*, Università degli studi di Padova, 2005.

[15] Marcassa A., *PariPari: DiESeL*, Università degli studi di Padova, 2008.

[16] Ranieri I., *PariPari: DNS*, Università degli studi di Padova, 2009.

[17] Aline Baggio, Maarten van Steen, *Transparent Distributed Redirection of HTTP Requests*, NCA '03 Proceedings of the Second IEEE International Symposium on Network Computing and Applications, USA, 2003.

[18] Pierangela Samarati, Sabrina Capitani de Vimercati, *Access control: Policies, models, and mechanisms*, Springer Berlin/Heidelberg, 2001.