UNIVERSITÀ DI PADOVA FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

PARISTORAGE
GESTIONE DATI IN LOCALE

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

Correlatore: **Egr. Ing. Michele Bonazza**

Laureando: **Nicola Danese**

Anno Accademico 2011-2012

Indice

Sommario

PariPari è un progetto tramite il quale si vuole realizzare una rete peer-to-peer priva di server, multi-funzionale, in grado, grazie al contributo di tutti gli utenti, di fornire molti servizi già oggi disponibili su Internet e di essere aggiornata con nuove funzionalità, qualora ce ne fosse l'esigenza.

Per raggiungere tale obiettivo è stato costituito un gruppo di ricerca formato e coordinato da laureandi.

Questo elaborato tratta del lavoro da me svolto all'interno del plugin LocalStorage, ovvero il modulo che si occupa di gestire i file del disco locale. Il mio ruolo è stato di Tester e di Maintainer. Sono partito da un plugin già fatto e funzionante scrivendo dapprima nuovi test per apprenderne il funzionamento e nel contempo aggiornando il codice per mantenerlo in linea col resto del progetto.

Il plugin presentava una prima soluzione implementativa complessa e poco performante che è stata snellita e rifinita migliorandone le performance per quanto riguarda l'uso di memoria e la velocità di risposta.

In questa tesi si procederà quindi a descrivere il funzionamento di PariPari e in particolar modo di LocalStorage. Successivamente si tratterà l'utilizzo di questo plugin da parte degli altri moduli di PariPari concludendo con alcune note sul testing.

Introduzione

Figura 1: Il logo di PariPari

La diffusione negli ultimi anni delle connessioni ad alta velocità a costo fisso mensile, quindi indipendente dal numero di ore di effettivo collegamento e dalla quantità di traffico generato, ha favorito la comparsa di numerosi programmi che basano i loro servizi sulla rete Internet; tra questi i più noti sono sicuramente quelli di audio e video conferenza, messaggistica istantanea e *file sharing*. Ogni programma offre un servizio specifico, e programmi diversi che offrono lo stesso tipo di servizio comunicano spesso con protocolli differenti rendendo incompatibili i vari programmi tra loro. Questo porta un utente ad avere bisogno di svariati programmi aperti per avere più servizi contemporaneamente. Per l'utente ciò significa dover reperire, installare e mantenere aggiornati, nonché imparare ad utilizzare tutti questi distinti programmi. Indubbiamente un'attività onerosa anche per il fatto che richie-

de una certa conoscenza e può esporre l'utente a rischi di frodi e quant'altro. Sicuramente un programma che comprenda tutte le funzioni di cui si ha bisogno sarebbe più semplice, più comodo e più utile. Questo è ciò che PariPari si propone di fare. L'idea alla base del progetto PariPari è quella di realizzare un programma che permetta, in un unico prodotto, di poter usufruire di molti programmi che, cooperando tra loro, possano fornire un servizio migliore e più semplice di quello di tanti singoli prodotti in esecuzione parallela, il tutto in un ambiente uniforme ed integrato. Questo garantisce anche la giusta spartizione delle risorse in maniera automatica, visto che l'utente potrebbe non avere le conoscenze necessarie per impostare i parametri di banda e consumo di *CPU* delle varie applicazioni; pertanto potrebbe ritrovarsi con molti servizi disponibili ma con un'esperienza pessima per quanto riguarda la fluidità di esecuzione dei comandi. PariPari vuole inoltre fornire un ambiente di sviluppo in cui chiunque voglia realizzare delle nuove funzionalità possa farlo facilmente sfruttando liberamente tutti i moduli esistenti, andando così ad estendere le funzionalità del programma stesso. PariPari non vuole essere una rete chiusa al mondo esterno, ma vuole proporsi come una rete integrata in modo trasparente con le reti esistenti. Agli occhi di un computer esterno alla sua rete, PariPari apparirà come un'unica macchina in grado di fornire molte delle funzionalità che solitamente vengono fornite, gratuitamente o a pagamento, da server commerciali. Per funzionare PariPari sfrutta l'architettura *P2P (peer-to-peer)*[1], cioè una rete di computer che non possiede nodi gerarchizzati come client o server fissi, ma un numero di nodi equivalenti (in inglese *peer*) che fungono sia da client che da server verso altri nodi della rete. I partecipanti alla rete P2P possono differire, anche di molto, nella potenza di calcolo, nella ampiezza di banda e nella quantità di dati memorizzabili. All'espandersi della rete corrisponderà un aumento del numero di richieste, ma al tempo stesso aumenteranno anche le risorse a disposizione di tutti gli utenti.

[1] `http://it.wikipedia.org/wiki/P2P`

Capitolo 1

Sviluppo software

1.1 Linguaggio di Programmazione

L'architettura P2P su cui si basa il progetto PariPari ha bisogno di appoggiarsi ad un vasto bacino di utenze; di conseguenza si è imposta la necessità di fornire un prodotto che fosse indipendente dalla macchina e dal sistema operativo dell'utente finale. Per questo il codice di PariPari è scritto in Java[1] affinché sia utilizzabile con qualsiasi sistema operativo in grado di eseguire la Macchina virtuale Java[2] senza la necessità di ricompilare e ottimizzare i file sorgente per ogni piattaforma esistente; inoltre, Java è il linguaggio che viene insegnato agli studenti dell'area di Ingegneria dell'Informazione. Per rendere l'installazione e l'esecuzione di PariPari un'operazione quanto più semplice possibile si è deciso di utilizzare la tecnologia Java Web Start[3]. Questo software è studiato per offrire una facile attivazione delle applicazioni mediante un singolo clic, garantendo che venga sempre eseguita l'ultima versione dell'applicazione eliminando le complicate procedure d'installazione e di aggiornamento. PariPari è software libero[4], rilasciato sotto licenza GNU GPL, quindi vuole coinvolgere attivamente quante più menti possibili per realizzare qualcosa che sia utile e ben fatto. Per ora la realizzazione di PariPari sta coinvolgendo alcuni studenti di Ingegneria di Padova, in un futuro potrà

[1] http://www.java.com/it/about/

[2] http://it.wikipedia.org/wiki/Macchina_virtuale_Java

[3] http://www.java.com/it/download/faq/java_webstart.xml

[4] http://it.wikipedia.org/wiki/Software_libero

coinvolgere centinaia di persone in tutto il mondo. Ciò implica che chiunque può avere libero accesso al codice sorgente per modificarlo o per aggiungere nuove funzionalità e anche in questo caso Java si rivela essere una scelta adeguata fornendo la creazione automatizzata di documentazione in un formato standard.

1.2 Extreme Programming

PariPari è un progetto molto vasto in cui collaborano più di cinquanta persone contemporaneamente, e per brevi periodi. Visto il contesto dinamico in cui è inserito necessita di linee guida solide e facilmente assimilabili. La scelta è caduta su *Extreme Programming*[5], una metodologia agile[6] e un approccio all'ingegneria del software[7] formulato da Kent Beck, Ward Cunningham e Ron Jeffries. L'Extreme Programming prevede di adottare una serie di accorgimenti mirati a migliorare la qualità della programmazione e del lavoro di gruppo. Non è stato possibile seguire tutti i dettami previsti dalla metodologia, che prevede di rivolgersi ad un gruppo di persone che lavorino un consistente numero di ore in un ambiente comune, ma si è cercato per quanto possibile di attenersi a quanto consigliato. E' stato quindi deciso di:

- Dividere il lavoro in gruppi pur mantenendo aperto il dialogo in modo tale che le esperienze di un gruppo possano essere utili anche per altri. Alcune problematiche si ripresentano facilmente in progetti modulari come questo, avere il codice di chi ha già trovato una soluzione rende il lavoro più facile e veloce, lasciando tempo ad eventuali miglioramenti.

- Effettuare riunioni collettive programmate in cui ogni gruppo espone agli altri i propri progressi e i problemi incontrati.

- Utilizzare un preciso standard di scrittura del codice, preferendo un design semplice e utilizzando un approccio del tipo "semplice è meglio" alla progettazione software.

[5] `http://it.wikipedia.org/wiki/Extreme_Programming`
[6] `http://it.wikipedia.org/wiki/Metodologia_agile`
[7] `http://it.wikipedia.org/wiki/Ingegneria_del_software`

- Integrare continuamente i cambiamenti al codice in modo da evitare ritardi nel ciclo del progetto causati da problemi d'integrazione.
- Progettare le interfacce delle classi e dei metodi e, prima di scrivere il codice, scrivere il test. Successivamente alla scrittura del codice va testata ogni singola riga.
- Piccole Release: la consegna del software avviene tramite frequenti rilasci di funzionalità che creano del valore concreto.

Capitolo 2

Le componenti di PariPari

La rete PariPari è stata progettata per fornire numerose tipologie di servizi; secondo la filosofia del peer-to-peer questi servizi vengono erogati dagli stessi nodi che compongono la rete. Ogni servizio viene gestito da appositi moduli chiamati plugin. In senso lato un plugin è un insieme di componenti software che aggiunge delle funzionalità specifiche ad un'applicazione software più grande. Per esempio, i plugin sono comunemente utilizzati nei browser web per riprodurre video, fare la scansione di virus, o visualizzare tipi di file particolari, come i PDF. Un esempio noto di plugin è Adobe Flash Player o QuickTime. In ogni singolo nodo i plugin installati vengono avviati e coordinati da un programma centrale chiamato Core.

2.1 Architettura a plugin

L'architettura a plugin consente di poter creare nuovi moduli molto facilmente dato che, per sfruttare le funzionalità già offerte da un modulo, non è necessario conoscerne la struttura interna, ma basta sapere come interfacciarsi con esso. Per questo motivo ogni plugin mette a disposizione degli *API (Application Programming Interface)*.[1] Tutto questo permette non solo di poter aggiungere alla rete nuovi servizi, qualora ce ne sia la necessità, ma dà anche la possibilità all'utente finale di decidere quali plugin installare e attivare sulla propria macchina. Come si può vedere dalla figura 2.1, i plugin sono di-

[1] `http://it.wikipedia.org/wiki/Application_programming_interface`

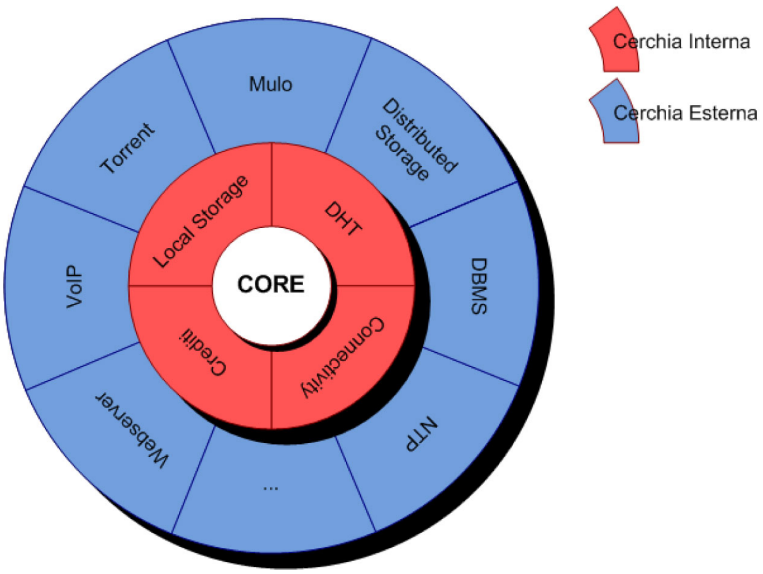

Figura 2.1: Schema architettura software di PariPari

visi in due categorie o cerchie. Della cerchia interna fanno parte i moduli con permessi speciali per utilizzare una specifica risorsa hardware della macchina, per questo sono chiamati anche gestori di risorse. Della cerchia esterna fanno parte tutti gli altri moduli che basano il loro funzionamento sui plugin della cerchia interna. Il motivo della separazione è intrinseco nella natura a plugin: mentre da un lato caricando il plugin relativo si aggiunge una funzionalità al sistema, dall'altro aumenta la richiesta di risorse condivise contese tra più pretendenti. I gestori di risorse si occuperanno di regolamentare le attività dei plugin in modo da mantenere una elevata esperienza utente, fornendo dei buoni compromessi. Ad esempio un plugin B che voglia instaurare un servizio DBMS necessiterà di accedere alle informazioni su disco in modo prioritario per mantenere una velocità di risposta elevata. La presenza simultanea di un altro plugin A che fa un uso pesante del disco -come per esempio un programma di file sharing- potrebbe rendere non soddisfabile la richiesta del plugin B, a meno che A sia disposto ad aspettare. Entra allora in gioco il gestore di risorse specifico che fornisce gli strumenti utili per dare maggior priorità al plugin B e mettere in coda A, che verrà servita non appena B finisce la sua attività ad alta priorità.

2.2 Gestione dei Crediti

L'architettura di PariPari è per definizione aperta: uno sviluppatore può in qualsiasi momento procedere all'implementazione di un nuovo modulo, facendo uso degli API messi a disposizione dagli altri plugin. Il rischio è però che un lavoro di programmazione poco accorto o malevolo possa creare degli sprechi nell'allocazione delle risorse. Un plugin potrebbe infatti chiedere al modulo Connectivity la creazione di numerosi socket di rete, ed utilizzarne effettivamente solo alcuni. I socket non usati sarebbero stati creati con spreco di cicli di CPU, di memoria e di banda di connessione. Per contrastare questo problema è stata introdotta una moneta virtuale interna chiamata "crediti". Come in una sorta di mercato, del "denaro" verrà guadagnato ogni volta che un proprio API viene utilizzato da altri plugin, mentre verrà speso ogni volta che si utilizzano API di altri plugin, cosicché ogni programmatore è incentivato ad un uso ragionevole delle risorse, rilasciandole nel momento in cui non ne abbia più bisogno. I prezzi sono specificati nel *descriptor.xml* (di cui parleremo in seguito) e, per ora, sono relativi ad un quanto temporale. Il Core, tramite il *Credit System*, si occupa della gestione di deposito e prelievo dei crediti nell'account dei plugin e, in caso un plugin non avesse credito sufficiente per pagare una transizione, la richiesta viene rifiutata e viene notificato al plugin richiedente la mancanza di crediti per procedere. Oltre al sistema economico intra-peer appena individuato, riguardante cioè gli scambi di servizi tra i vari moduli del programma (crediti interni), ne esiste un secondo di tipo inter-peer: lo scambio di risorse tra i peer della rete (crediti esterni). L'implementazione del sistema economico per lo scambio di servizi tra peer è basata sul concetto del baratto. Vengono infatti barattate promesse di futuri servizi, il compratore Bob paga Alice promettendo che fornirà una congrua quantità di una propria risorsa quando il venditore Alice gliela richiederà. Il sistema sarà altresì estremamente transitivo, in modo che un creditore Carl del fornitore Alice potrà richiedere di saldare il debito di Alice a Bob.

2.3 Comunicazione tra Plugin

Come detto in precedenza per comunicare tra loro ogni plugin mette a disposizione degli altri un set di API. Un API rappresenta un insieme di metodi. Se ad un plugin serve un certo API (ad esempio per scrivere caratteri su file necessiterà di `FileWriterAPI`), deve farne richiesta al Core che provvederà a veicolarla al giusto plugin che dichiara, all'interno del suo file *descriptor.xml*, di implementare tale API (nel nostro esempio il Core passerebbe la richiesta a LocalStorage). Un esempio di "descrittore" si può vedere nel codice riportato in figura 2.2 in cui si possono notare, dopo la riga di intestazione, le interfacce (API) dichiarate. Dentro questo file sono inoltre specificati gli API richiesti dal plugin per funzionare, cosicché il Core possa precaricare i plugin necessari a soddisfare le richieste. Una volta ottenuta l'istanza del API desiderato,

```
<?xml version="1.0"?>
<plugin name="storage" version="0.9.201104062043" class="paripari.storage.Storage" SVN="5000">
    <interfaces>
        <API>
            <abstract>paripari.API.FileAPI</abstract>
            <implementation>paripari.storage.implementation.StorageFile
            </implementation>
            <billing timeunit="60000"></billing>
        </API>
        <API>
            <abstract>paripari.API.PFileAPI</abstract>
            <implementation>paripari.storage.implementation.StorageFile
            </implementation>
            <billing timeunit="30000"></billing>
        </API>
        <API>
            <abstract>paripari.API.FileInputStreamAPI</abstract>
            <implementation>paripari.storage.implementation.QuotedFileInputStream
            </implementation>
            <billing timeunit="90000"></billing>
        </API>
        <API>
            <abstract>paripari.API.PFileInputStreamAPI</abstract>
            <implementation>paripari.storage.implementation.QuotedFileInputStream
            </implementation>
            <billing timeunit="60000"></billing>
        </API>
    </interfaces>
</plugin>
```

Figura 2.2: descriptor.xml

la comunicazione avviene direttamente fra i plugin e le chiamate ai metodi finiscono in una speciale coda del plugin servente. Per garantire che questo

sistema funzioni correttamente è necessario che nessun plugin possa operare al di fuori di esso. Perciò il Core monitorizza le chiamate di sistema tramite il *Security Manager* che provvede a lanciare una `SecurityException` nel caso un plugin effettui una chiamata ad una classe che non è di sua competenza. Nel caso specifico dell'accesso a file e stream su disco l'unico plugin autorizzato è LocalStorage. LocalStorage mette a disposizione 5 API (più 5 uguali ma con priorità maggiore) che corrispondono alle rispettive classi della Sun (Oracle). Ognuna di esse ha un costo e le classi prioritarie costano ovviamente di più. (Vedere figura 2.3)

Classi Java	LocalStorage API	Priority LocalStorage API	Implementate da:
File	FileAPI	PFileAPI	StorageFile
FileInputStream	FileInputStreamAPI	PFileInputStreamAPI	QuotedFileInputStream
FileOutputStream	FileOutputStreamAPI	PFileOutputStreamAPI	QuotedFileOutputStream
RandomAccessFile	RandomAccessFileAPI	PRandomAccessFileAPI	QuotedRandomAccessFile
FileWriter	FileWriterAPI	PFileWriterAPI	QuotedFileWriter

Figura 2.3: Corrispondenza fra gli API di LocalStorage e le classi Java.

2.4 Funzionalità di LocalStorage

LocalStorage ha tre funzionalità peculiari. La prima consiste nella gestione dello spazio su disco occupato da ciascun plugin con lo scopo di limitarlo. Questa funzione è utile se si caricano moduli come distributed Backup. Tale plugin usa dello spazio della macchina per salvare frammenti dei file di backup degli altri utenti. Tale spazio può essere quindi limitato in base alle esigenze regolando la quota di spazio del plugin. Se un plugin eccede la quota massima prestabilita, viene lanciata un'eccezione (`QuotaExeededException`) e non viene occupato ulteriore spazio per la scrittura. La seconda controlla il numero di file utilizzati da un plugin, impostando un limite prestabilito in modo da arginare errori di codice che generino file indiscriminatamente o plugin malevoli che tentino di bloccare la macchina. In caso si ecceda il numero di file prestabilito viene lanciata un'eccezione (`FileNumberExceededException`). Ultima, ma non per importanza, per ogni plugin LocalStorage crea una gerarchia di directory nella cartella principale di PariPari: la più esterna prende il

nome dal plugin (ad esempio Mulo) e ne funge da home. Dentro questa cartella vengono create altre due directory, una chiamata ”data” e una ”config”. Quando un plugin richiede un file senza specificarne il percorso questo viene messo da LocalStorage dentro la cartella ”data” relativa al plugin. Ogni modulo ha libero accesso alla sua direcory home. Nel caso in cui venga richiesta l'esecuzione di un'operazione su qualche file all'esterno di tale directory, viene visualizzata una finestra di dialogo che richiede il consenso esplicito dell'utente (figura 2.4). La gestione dell'accesso ai file è resa necessaria per impedire che i plugin possano, all'insaputa dell'utente, accedere a dati sensibili o comunque avere libero accesso al disco, creando un potenziale problema di sicurezza e privacy. I file esterni vengono richiesti massicciamente da plu-

Figura 2.4: La finestra di dialogo per la richiesta di consenso all'utente.

gin come Mulo, Torrent e Distributed Backup, i quali si trovano tipicamente a dover leggere o scrivere dati sparsi per il filesystem. Per evitare il continuo presentarsi di finestre di dialogo si è deciso di dare all'utente la possibilità di dare il permesso di lettura/scrittura ad alcune cartelle esterne attraverso conferma esplicita tramite pop up. Inoltre è possibile dare la totale fiducia ad un dato modulo, disabilitando ogni notifica riguardante gli accessi al filesystem operati dallo stesso, attraverso uno specifico comando della console (Vedere il capitolo 5). Per default i plugin non possono accedere alla cartella di LocalStorage per evitare modifiche inopportune al *fileset.xml*. Il *fileset.xml* viene utilizzato per mantenere le informazioni che LocalStorage perderebbe

nel momento dell'uscita dal programma, cioè il nome dei plugin conosciuti e i loro attributi (spazio e numero di file massimo, i file esterni e le directory esterne di cui ha la fiducia, e la fiducia al plugin). Esso viene salvato su disco ogni 15 minuti da un PariPariTimerTask[2] e ovviamente alla chiusura di PariPari.

[2] `http://www.paripari.it/javadoc/paripari/core/PariPariTimerTask.html`

Capitolo 3

Architettura di LocalStorage

Per il funzionamento del sistema di crediti è importante che i plugin non possano effettuare chiamate di sistema direttamente, ma debbano rivolgersi agli specifici plugin della cerchia interna. Il problema che si presenta per il Security Manager è dunque quello di distinguere le chiamate di LocalStorage da quelle di altri plugin in modo da autorizzare le une e bloccare le altre, come mostrato in figura 3.2. La soluzione attualmente implementata consiste nella separazione degli stack delle chiamate, ovvero attraverso un sistema di messaggi i plugin affidano dei compiti a LocalStorage che invocherà i metodi di cui ha i privilegi, in modo che un plugin che non sta passando per LocalStorage venga individuato facilmente. Questo problema, almeno per quanto riguarda i moduli della cerchia interna (principalmente Connectivity e LocalStorage), comporta la progettazione di un'architettura composta da un sistema di ricezione e soddisfacimento di "ordini".

3.1 StorageListener

Quando un plugin richiede un API al Core (step a) questo, previo il controllo sulla disponibilità di crediti del richiedente (step b e c), ne veicola la richiesta al plugin che implementa tale API (step d ~h). Se il plugin non può affrontare l'acquisto di risorse lo stato del Messaggio spedito dal plugin viene impostato su `IMessage.NOT_SENT` e il thread richiedente viene svegliato. Ad ogni plugin il Core assegna due code. La prima è detta

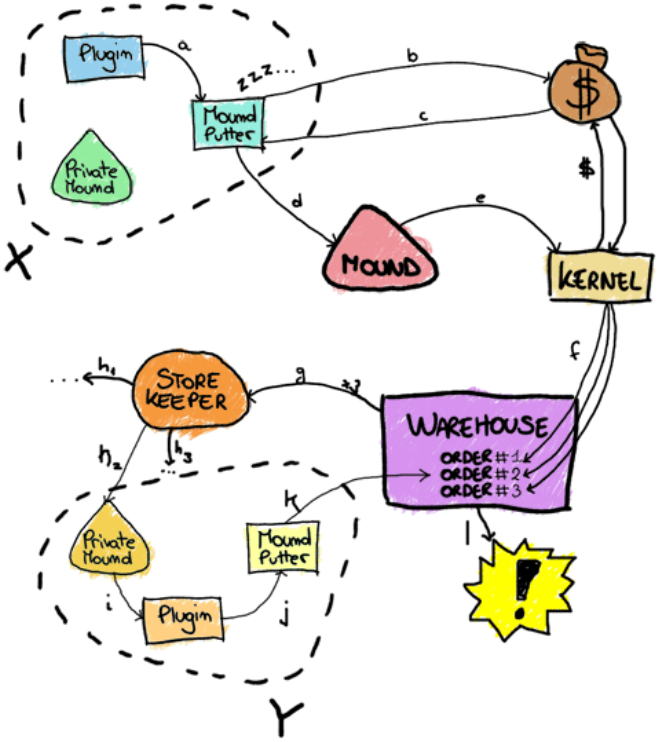

Figura 3.1: Schema comunicazione fra Plugin attraverso il Core

`PrivateMound` ed è dove il Core deposita tutti i messaggi per quel plugin. Attraverso l'estensione della classe `Listener` (step i), i destinatari ricevono il messaggio e inseriscono la risposta, sotto forma di oggetti, dentro il campo "data" del Messaggio ricevuto dal Core, ne aggiornano lo stato e infine lo inseriscono nella seconda coda, detta `MoundPutter` (step j), dalla quale il Core la preleva e la ritorna al mittente. Come si può evincere dal nome, `StorageListener` è l'estensione di LocalStorage della classe `Listener` che provvede a prelevare, gestire e rispondere ai messaggi che arrivano dal Core. Quando un messaggio viene prelevato, StorageListener lo apre e aggiunge dei dati, necessari per il funzionamento di LocalStorage, che vengono utilizzati dai costruttori delle implementazioni degli API. Questi dati comprendono il nome del plugin richiedente, la referenza al `PluginSet` e quella

al `StorageOrderDispatcher`. Tramite le funzionalità della reflection[1] viene creata una nuova istanza della classe che implementa lo API richiesto, e inserita come risposta nel messaggio. Una volta impostato il corretto stato che descrive la riuscita dell'operazione `IMessage.OK`, o l'avvenuto verificarsi di un'eccezione `IMessage.NOT_SATISFIABLE` o `Imessage.BAD_REQUEST` (a seconda delle condizioni di errore), viene spedita la risposta al Core. Dentro al costruttore delle classi che implementano gli API viene generato un ordine, evaso dal `Satisfier`, con la specifica di essere un costruttore. In questo modo è possibile creare un'istanza della classe che opera realmente su disco. Un riferimento a tale oggetto viene inserito in una `HashMap` detta `instanceHandler`. Questo argomento verrà trattato in dettaglio nel prossimo paragrafo.

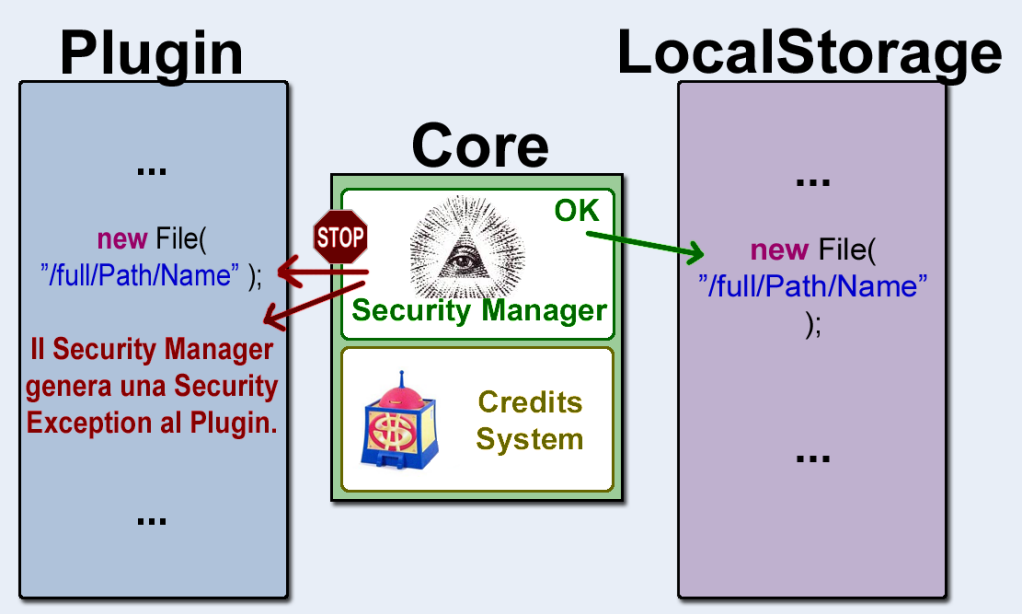

Figura 3.2: Schema del funzionamento del Security Manager

[1] `http://it.wikipedia.org/wiki/Riflessione_(informatica)`

3.2 Il sistema degli ordini

È necessario che le implementazioni delle classi API non effettuino invocazioni a metodi che richiedono l'intervento del Security Manager, in quanto tali invocazioni verrebbero interpretate come provenienti dal plugin richiedente che, non possedendo i permessi necessari ad operare su disco, si troverebbe ad avere a che fare con una `SecurityException`. Per evitare ciò è stato progettato un sistema di invio ordini, detti `StorageOrder`. Una volta che un plugin ottiene dal Core un'istanza dell'implementazione dell'API desiderata, può utilizzare i suoi metodi (le cui firme ricalcano quelle dei relativi metodi Java, vedasi figura 2.3). All'interno di tali metodi vengono generati gli ordini e consegnati allo `StorageOrderDispatcher` (figura 3.6). Il processo di esaudimento delle richieste contenute negli ordini sarà trattato nei paragrafi 3.3 e 3.4 intitolati StorageOrderDispatcher e Satisfier. La struttura di

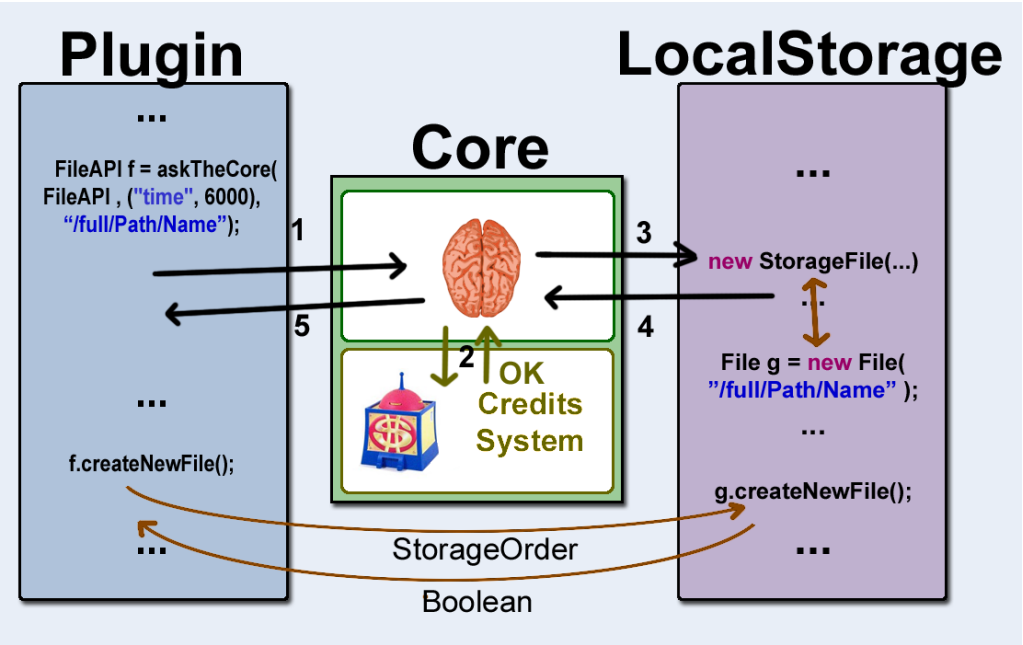

Figura 3.3: Schema della creazione di un'API

uno `StorageOrder` è rappresentata in figura 3.4, e permette alla classe chiamante di descrivere in maniera precisa il metodo da invocare, e alla classe ricevente di comunicare il valore di ritorno o l'eventuale eccezione generata.

Di seguito un esempio del codice per generare un nuovo ordine. In questo caso si tratta del metodo `createNewFile()` della classe `StorageFile`.

```
StorageOrder order = new StorageOrder(orderKey, target,
"createNewFile", null);
Orders.sendOrder(order, queue);
Orders.manageExceptions(order, IOException.class);
return (Boolean) order.getResults()[0];
```

Si può notare come vangano passati la chiave del plugin (`orderKey`), che contiene il nome del plugin, il path completo del file a cui fa riferimento, la classe a cui fa riferimento (per distinguere se il metodo appartiene ad un `FileAPI` oppure a una delle altre API), la priorità e una variabile `long` r per discriminare fra vari thread dello stesso plugin che operino sullo stesso file. Inoltre viene passata la classe di riferimento, il nome del metodo e gli eventuali parametri (in questo caso non ci sono parametri e viene quindi passato `null`). Poi viene spedito con il metodo `sendOrder()` della classe `Orders`.

```
public static void sendOrder(StorageOrder order,
            StorageOrderDispatcher dispatcher) {
// delivery order to the dispatcher
synchronized (dispatcher) {
    dispatcher.add(order);
}
// wait until this request is satisfied
synchronized (order) {
    while (order.getStatus() == Status.TO_SATISFY) {
        try {
            order.wait();
        } catch (InterruptedException e) { ... }
    }
}
}
```

Come si può vedere prima l'ordine viene consegnato allo spedizioniere e successivamente si mette in attesa che l'ordine venga soddisfatto.

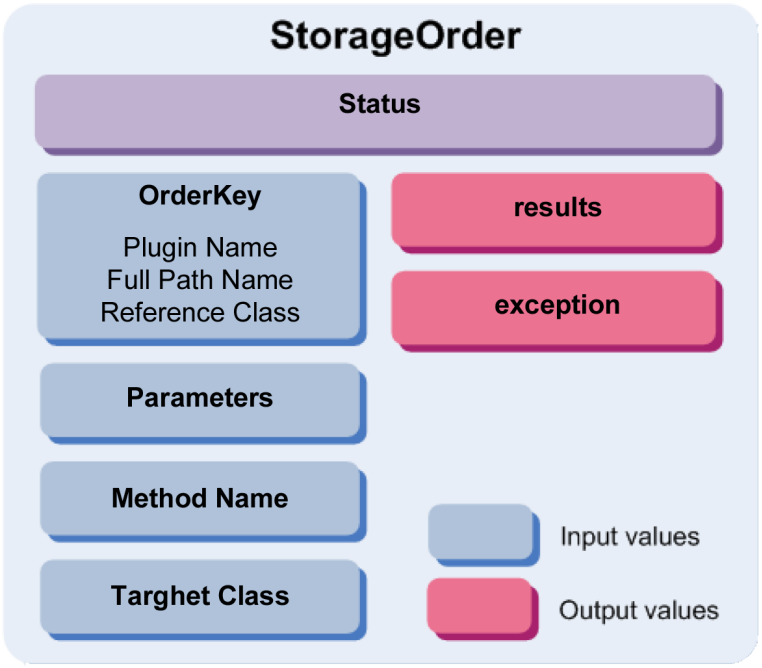

Figura 3.4: Struttura di uno StorageOrder

3.3 StorageOrderDispatcher

Ogni volta che viene invocato un metodo delle classi implementanti le API, viene generato un nuovo ordine. Quando un nuovo ordine giunge a `StorageOrderDispatcher` esso controlla se ci sono istanze di `Satisfier` libere, in caso affermativo assegna l'ordine al thread libero e lo segna come occupato, altrimenti si trova di fronte a due situazioni: nella prima l'ordine è stato generato all'interno del metodo `invoke()` di un `Satisfier`, il quale si è messo in attesa del soddisfacimento dell'ordine. Se questo succedesse per tutte le istanze di `Satisfier` si avrebbe una situazione di *Deadlock*[2]. Per evitare questa situazione di stallo viene creata una nuova istanza di `Satisfier` e gli viene assegnato il lavoro. Nel secondo caso l'ordine non è "bloccante" e viene quindi inserito in una coda, in base alla sua priorità[3]. Gli ordini in coda vengono evasi man mano che gli `Satisfier` portano a termine il loro

[2] `http://it.wikipedia.org/wiki/Deadlock`
[3] Vedasi capitolo 2.1

lavoro e comunicando al `StorageOrderDispatcher` di essere tornati liberi, ricevono il nuovo lavoro da svolgere. Normalmente sono presenti tre istanze di `Satisfier` in attesa, nel caso in cui siano state create più di tre istanze, se non sono più utili `StorageOrderDispatcher` ferma i thread in eccesso.

3.4 Satisfier

`Satisfier` è un `PariPariThread`[4] in attesa su se stesso. Quando viene risvegliato chiama il metodo delegato all'evasione degli ordini (`satisfyOrder()`) che usa la già citata tecnica della reflection per invocare i metodi corrispondenti nell'opportuna classe Java. La tecnica della reflection è stata utilizzata per minimizzare le righe di codice, in sostituzione ad una struttura `switch-case` con svariate centinaia di casi, che risultava difficile da leggere e mantenere. L'uso della reflection ha però introdotto dei problemi, analizzati nel tutorial della Sun sulla reflection[5], che si possono riassumere in tre punti:

- Degrado di prestazioni dovuto alla natura dinamica delle strumento
- Sono richiesti permessi runtime che potrebbero causare problemi con il Security Manager
- Dal momento che la riflessione consente al codice di eseguire operazioni come l'accesso a campi e metodi privati, l'uso della riflessione può portare a effetti collaterali inattesi, che possono rendere il codice disfunzionale.

Detto questo è bene notare come le invocazioni effettuate dal thread `Satisfier`, essendo parte di LocalStorage, vengano interpretate dal Security Manager come provenienti dallo stesso LocalStorage: questo comportamento è chiaramente ciò che permette di avere la separazione dello stack delle chiamate. Nel caso in cui la chiamata sia ad un costruttore viene creata un'istanza della classe di riferimento (ad esempio `FileWriter`, per le `FileWriterAPI`) e viene inserita in una `HashMap` (`instanceHendler`) che usa come chiave l'`OrderKey`, che viene passato con lo `StorageOrder`. Inoltre il plugin viene

[4] `http://www.paripari.it/javadoc/paripari/core/PariPariThread.html`
[5] `http://download.oracle.com/javase/tutorial/reflect/index.html`

inserito, se non lo è già, in una struttura dati chiamata `PluginSet` (figura 3.5). Questa struttura non è altro che una mappa che associa i vari plugin ad una variabile denominata `PluginProperties`. Essa contiene quattro variabili: `FileNameSet` che contiene l'insieme dei vari file usati dal plugin; `trustedDirectory` che è l'insieme delle cartelle "esterne" per le quali l'utente ha impostato la fiducia; `PluginQuota` che tiene traccia dello spazio occupato e del numero di file usati, e una variabile booleana che tiene traccia della scelta dell'utente di fidarsi del plugin. La fiducia viene richiesta ogniqualvolta un plugin tenta di accedere ad un file in una posizione esterna alla directory home del plugin stesso o all'esterno delle cartelle fidate. L'utente può scegliere se bloccare l'azione o consentirla. In alternativa può impostare la fiducia alla cartella contenente il file (figura 2.4). Se l'ordine

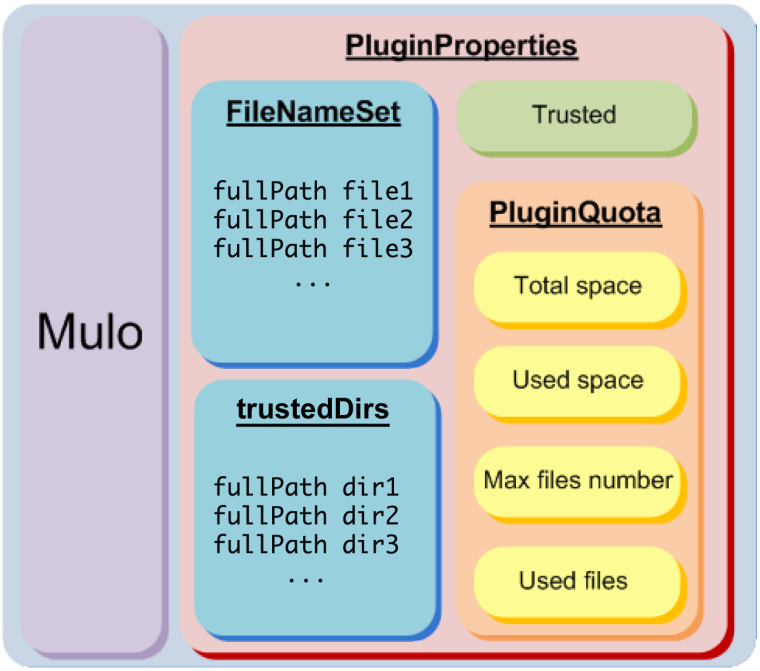

Figura 3.5: La struttura del PluginSet

non contiene un costruttore, ma un metodo, allora tale metodo viene invocato attraverso il metodo `invoke()` della reflection sull'istanza recuperata dall'`instanceHendler`. Prima e dopo l'`invoke()` viene controllata l'esistenza e la dimensione del file su cui viene chiamato il metodo: per differenza

è poi possibile ricostruire se il file è stato creato, cancellato, e la dimensione scritta o eliminata dal file in modo da poterne gestire la relativa quota. Se la quota di spazio disponibile al plugin è sufficiente per completare l'operazione, i risultati dell'invocazione, siano essi valori di ritorno o eccezioni lanciate, vengono inseriti all'interno dell'oggetto `StorageOrder`, ne viene settato lo status in modo opportuno e notificata la disponibilità del risultato alla classe che implementa gli API, di modo che questa possa restituire il valore al plugin chiamante o lanciare la corretta eccezione. Giova ricordare il fatto che i metodi vengono invocati direttamente sulle classi Java, come `File`, `FileOutputStream`, ecc...: in questo modo per aggiungere nuove classi basterà aggiungere l'implementazione che consiste nel dichiarare gli stessi metodi della classe che si sta implementando, definendoli come generatori di ordini. Si è accennato in precedenza al fatto che ogni oggetto API aper-

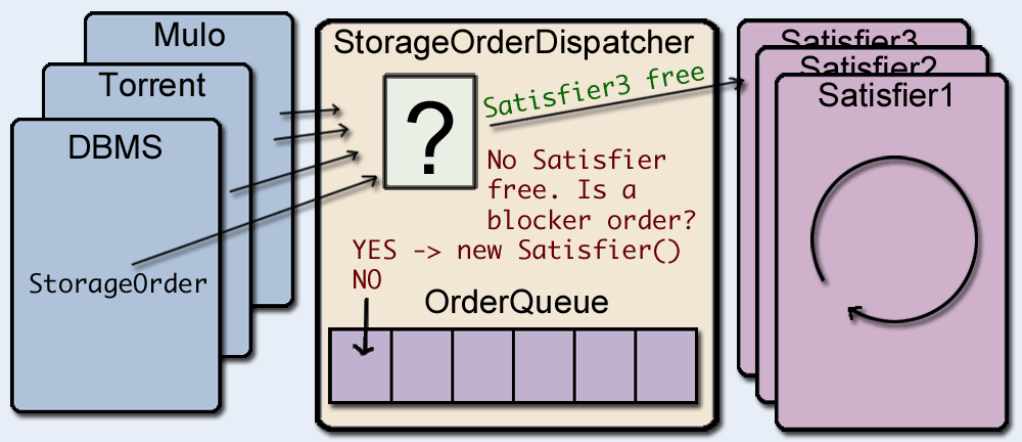

Figura 3.6: La struttura interna del plugin LocalStorage

to corrisponde ad una istanza presente nell'`instanceHandler`. Questo può effettivamente rappresentare un problema in quanto ad un numero maggiore di elementi corrisponderà un aumento della memoria occupata e del tempo di ricerca dell'istanza corretta sulla quale invocare i metodi. Il problema viene risolto grazie al Credit System che si occupa di richiamare il metodo `destroy()` sui file per cui sia scaduto il quanto temporale acquistato: alla chiamata di `destroy()` LocalStorage chiude il file, costringendo i programmatori a stare attenti a chiudere i file che non utilizzano più, attraverso il

metodo `close()`, che è stato aggiunto anche alla classe `FileAPI`.

Capitolo 4

Utilizzo degli API

Come già riportato, in PariPari ogni plugin che necessiti di un servizio offerto da qualche altro modulo è obbligato a farne richiesta al Core. Supponiamo che il plugin Mulo desideri richiedere un oggetto `FileAPI` per salvare il contenuto di un file che è attualmente in download. La chiamata effettuata da Mulo sarà di questo tipo, ipotizzando che il file in questione abbia nome "lubuntu-10.04.iso" e si voglia posizionare nella home del plugin stesso:

```
IFeatureValue[] values = { new FeatureValue("time", 6000)};
IReply ireply = askTheCore (new Request ( FileAPI.class,
new ConstructorParameters( values, "lubuntu-10.04.iso",
AccessMode.RW)));
FileAPI file = (FileAPI) ireply.getAPIs()[0];
```

Questa chiamata viene interpretata dal Core come richiesta, da parte di Mulo, di un oggetto appartenente ad una classe che implementi lo API `FileAPI` (si suppone infatti che in futuro ci possano essere più plugin in grado di offrire contemporaneamente implementazioni diverse degli stessi API, e che questi operino in regime concorrenziale in base a quantità di crediti richiesti e qualità del servizio prestato). Il Core, al momento, può individuare solamente l'implementazione offerta da LocalStorage, ed in questo caso la classe `StorageFile`, il cui costruttore viene quindi invocato. Mulo potrà poi usare l'oggetto `file` come fosse un'istanza della classe `File`, per esempio invocando `file.crateNewFile()`. Il programmatore dovrà tenere in considerazione il fatto che, passato il tempo indicato nella variabile `values` (nel nostro

caso 6.000 s), il Credit System invocherà il metodo `destroy()` sull'oggetto, provocandone la chiusura. Per evitare la distruzione delle risorse il Credit System mette a disposizione dei metodi per rinnovarle. È inoltre disponibile una libreria, `PluginSender`, che si occupa di gestire in automatico le richieste al Core delle risorse e il loro rinnovo. L'elenco completo di tutti i costruttori disponibili è riportato nella pagina del plugin sulla mediawiki[1] di PariPari, mentre per i metodi disponibili è possibile fare riferimento alla documentazione[2].

4.1 FileExtension

Per la classe `FileAPI` sono disponibili due metodi aggiuntivi, rispetto alla classe `File`.

```
boolean moveTo(File destination, boolean keepOriginal)
```

Questo metodo permette di spostare uno `StorageFile` in un altro. Se `keepOriginal` è impostato a `true` il metodo farà una copia del file, altrimenti verrà semplicemente invocato il metodo `renameTo(File dest)`, che delega al sistema operativo l'onere di spostare il file nella locazione destinazione.

```
Vector<String> getFileList(Boolean recurse)
```

Restituisce un vettore di tutti i file in possesso del plugin, cioè dei file contenuti nel `pluginSet`.

[1] `http://www.paripari.it/mediawiki/index.php/Local_Storage_en#Incoming_Request`

[2] `http://www.paripari.it/javadoc/`

Capitolo 5

La Console

Per rendere più user-friendly l'utilizzo e la configurazione di LocalStorage, sono stati implementati nella classe `StorageConsole` alcuni comandi, utili anche per il testing, utilizzabili direttamente dalla console di PariPari. Tali comandi sono elencati nel seguito con una breve spiegazione sulle finalità e relativa sintassi:

space SINTASSI: space [plugin-name] [new-space]. Questo comando permette all'utente di impostare lo spazio disco massimo utilizzabile da parte di un plugin. Se la dimensione che si vuole impostare è minore dello spazio che il plugin già occupa, il comando viene ignorato.

files SINTASSI: files [plugin-name] [new-files-number]. Questo comando permette all'utente di impostare il numero massimo di file utilizzabili da un plugin, se il numero impostato è minore di quello attualmente utilizzato il comando viene ignorato.

info SINTASSI: info [plugin-name]. Questo comando permette di visualizzare alcune informazioni di interesse riguardanti un plugin; viene visualizzato lo stato di trustiness del plugin, le informazioni relative alla quota, le dimensioni dei file posseduti e le informazioni sugli stream correntemente aperti sugli stessi file.

fileSet SINTASSI: fileSet. Questo comando stampa il fileSet per ogni plugin visualizzando il percorso di tutti i file contenuti in esso.

trustDir SINTASSI: trustDir [plugin-name] [fullPathToDir] [true/false]. Questo comando imposta o revoca la fiducia totale alla cartella specificata dal `fullPathToDir`, per il plugin passato come parametro.

trust SINTASSI: trust [plugin-name] [true/false]. Questo comando imposta o revoca la fiducia totale al plugin passato come parametro.

bug SINTASSI: bug [on/off]. Questo comando trova la sua utilità quando si necessita di conoscere esattamente quali invocazioni a metodi degli API di LocalStorage vengono effettuate e se i relativi ordini vengono o meno soddisfatti. In modalità bug hunting, infatti, si fa in modo che tali informazioni vengano stampate in console. Ciò è particolarmente utile per individuare la porzione di codice responsabile di eventuali freeze o malfunzionamenti; d'altra parte non è raccomandabile eseguire PariPari con tale funzionalità attivata in quanti risulta particolarmente difficoltoso seguire il normale andamento dei plugin a causa dell'eccessiva mole di informazioni stampata a console.

Capitolo 6

Changelog

Una delle regole dell'XP è il "Simple Design": i programmatori dovrebbero utilizzare un approccio del tipo "semplice è meglio" alla progettazione software. Una volta raggiunto un buon punto di copertura con i test mi sono concentrato sulla semplificazione del codice. Da quando sono entrato in LocalStorage le righe di codice sono passate da 16.315 a 10.233 (-37%). Riassumendo in punti fondamentali i cambiamenti:

- Ho ordinato il codice dividendo le classi in vari package tematici. Un maggiore ordine facilita la manutenzione del codice, permettendo di individuare classi non più utilizzate o classi che possono essere fuse insieme evitando ridondanza dei dati e spreco di memoria.

- Gli "ordini" dai vari plugin venivano evasi da LocalStorage tramite reflection su classi che estendevano quelle di java in cui veniva fatto il controllo della quota del plugin. Queste classi avevano molto codice di gestione in comune. Una volta individuati i punti fondamentali, riassunti nella tabella qui sotto, ho spostato questo codice all'interno di `Satisfier` prima e dopo l'invocazione del metodo ed eliminato le classi.

Prima	Dopo	Dim. Dopo - Dim. Prima	Used Space	Used Files
E	E	> 0	Incrementa	-
E	E	< 0	Decrementa	-
E	E	=	-	-
E	N	<= 0	Decrementa	Decrementa
N	E	>= 0	Incrementa	Incrementa
N	N	=	-	-

Dove E = Esiste, N = Non Esiste.

Meno codice e meno classi comportano risparmio di memoria e meno probabilità di errori. Di seguito il codice che viene eseguito prima e dopo la chiamata ad `invoke()` al metodo `m`, sull'istanza recuperata del già citato `instanceHandler`. Dopodiché viene chiamata la funzione `manageQuota(pluginName, f2, e1, e2, l1, l2)` che si occupa di gestire la quota di spazio su disco disponibile e il numero di file usati dal plugin.

```
File f1 = new File(fullPathName);
// the space occupied by the file before the call
long l1 = f1.length();
// i need to know if they are creating a new file...
boolean e1 = f1.exists();
// invoke method on the instance of the class
// with parameters
order.setResults(m.invoke(instanceHandler.get(key),
            order.getParameters()));
File f2 = new File(fullPathName);
// the space occupied by the file after the call
long l2 = f2.length();
// i need to know if they are deleting the file...
boolean e2 = f2.exists();
```

- L'insieme dei dati relativi ai plugin caricati almeno una volta in PariPari è chiamato globalmente `PluginSet`; esso era mantenuto in maniera

consistente anche su disco (nel file `PluginSet.dat` presente nella directory "data" di LocalStorage) attraverso l'uso di `ObjectOutputStream` ad ogni ordine evaso da LocalStorage. Questo degradava in modo pesante le prestazioni del gestore proporzionalmente alla dimensione del `PluginSet`. Ho pensato che tenere sul disco tutta questa struttura fosse inutile. Visto che LocalStorage all'avvio effettuava un controllo di consistenza del `PluginSet` per accertare che non fossero state apportate modifiche ai file mentre LocalStorage non era in esecuzione. Ho pensato che il `PluginSet` poteva essere creato durante questa fase, inserendo i file e le loro dimensioni prese dalla scansione dei file e delle cartelle interne ai plugin che erano stati aggiunti nelle precedenti sessioni di LocalStorage. Mentre i file esterni e le altre opzioni come il `trusted`, che prima erano inseriti in una struttura dati separata e salvata su un altro file, devono essere scritte su disco. Ho scelto di scriverle in XML per semplicità. Un esempio del nuovo `pluginset.xml` è riportato nella figura 6.1. Il nuovo file XML viene salvato su file da un `PariPariTimerTask` ogni 15 minuti e all'uscita di PariPari. Questo ha portato ad un miglioramento di performance attorno al 40% per la classe `StorageFile` e fino all'80% per le altre classi. (confrontando il tempo medio su 5 prove effettuate con diverse funzioni del plugin dummystorage).

- Dopo un'attenta analisi, tramite test e documentazione, ho pensato di poter semplificare l'architettura multi-threading di LocalStorage per cui se si superavano più di λ;[1] richieste contemporaneamente veniva lanciato un altro thread che se ne occupava, dopo 2λ; un altro ancora e così via. Questo sistema era deleterio per vari motivi: essendo il disco una unità seriale I/O bound, avendo più di due thread concorrenti che cercano di scrivere/leggere dal disco, si ha un notevole rallentamento dovuto al continuo cambio di traccia (seek time + latency time) del disco, anche se gli ultimi dischi hanno algoritmi di ottimizzazione che limitano questo comportamento. Inoltre veniva speso del tempo nella sincronizzazione delle varie strutture dati utilizzate. Il vecchio comportamento per cui

[1] λ; era un parametro posto uguale a tre secondo osservazioni empiriche aleatorie

prima un ordine veniva messo in coda e poi un apposito thread si occupava di assegnare il lavoro ad un `Satisfier` è stato cambiato nel modo descritto nel paragrafo 3.4. Anche in questo caso le prestazioni sono migliorate, raggiungendo un tempo di risposta molto vicino ai tempi ottenuti utilizzando direttamente le classi Java.

- Il riordinamento e la semplificazione del codice mi ha permesso di inserire con semplicità una nuova feature per LocalStorage. Si tratta della priorità degli ordini. Ho duplicato le classi astratte presenti in `Interfaces.jar` mettendo una "P" davanti alle classi prioritarie (es. `PFileAPI`). Le due classi astratte vengono implementate dalla stessa classe, con la sola differenza del valore `priority` salvato in un parametro intero dell'istanza. Così gli ordini fatti a LocalStorage possono essere discriminati ed inseriti nell'apposita lista e serviti con priorità diverse.

```
<?xml version="1.0" ?>
<FILENAMESETS>
    <PLUGIN PLNAME="dummystorage" TRUST="false"
        TOTALSPACE="1000000000000" MAXFILENUM="1000">
        <FILE>
            <PATH>/Users/darkfrank</PATH>
            <NAME>fileProva2.txt</NAME>
        </FILE>
        <FILE>
            <PATH>/Users/darkfrank</PATH>
            <NAME>fileProva.txt</NAME>
        </FILE>
        <FILE>
            <PATH>/Users/darkfrank</PATH>
            <NAME>fileProva3.txt</NAME>
        </FILE>
        <DIR>/Users/darkfrank</DIR>
    </PLUGIN>
</FILENAMESETS>
```

Figura 6.1: Esempio di pluginset.xml

Capitolo 7

Testing

"Il testing è un procedimento utilizzato per individuare le carenze di correttezza, completezza e affidabilità delle componenti software in corso di sviluppo. Consiste nell'eseguire il software da collaudare, da solo o in combinazione ad altro software di servizio, e nel valutare se il comportamento del software rispetta i requisiti."[1]

7.1 Testing Driven Development

Uno dei punti fondamentali dello sviluppo di PariPari, che deriva dai dettami dell'Extreme Programming, è il Testing Driven Development. Come prima cosa da quando sono entrato nel progetto ho imparato a testare il codice già presente in LocalStorage tramite lo unit testing[2]. Per unità si intende genericamente la minima parte testabile di un codice sorgente: nel nostro caso la più piccola unità è il metodo. Lo Unit Testing si articola in test case ciascuno dei quali dovrebbe essere indipendente dagli altri. Eclipse gestisce lo unit testing sfruttando la libreria JUnit[3], un semplice framework per scrivere ed eseguire test automatici. Al termine di ogni esecuzione JUnit riporta il numero di test eseguiti, il numero dei test andati a buon fine, il numero dei test falliti ed il numero dei test errati indicandone la causa del fallimento o dell'insuccesso di quest'ultimi. Un esempio di output di JUnit è in figura 7.1. Tramite il plugin

[1] http://it.wikipedia.org/wiki/Collaudo_del_software
[2] http://it.wikipedia.org/wiki/Unit_testing
[3] http://www.junit.org

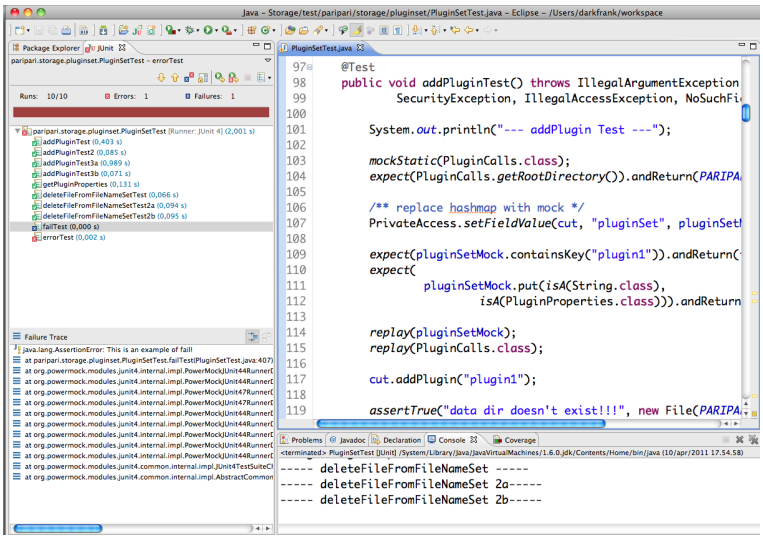

Figura 7.1: Screenshot di Eclipse che mostra l'output di JUnit

eclemma[4] è poi possibile visualizzare la percentuale di codice coperto dai test e colorare le righe di codice coperte (verde), coperte parzialmente (giallo) e non coperte (rosso). Attualmente il codice coperto di LocalStorage si attesta intorno al 68% (+45% da quando ho iniziato a fare testing).

7.1.1 Unit Testing

Lo scopo dello Unit testing è quello di verificare il corretto funzionamento di parti di programma permettendo così una precoce individuazione dei bug. Uno unit testing accurato può dare una prova certa se un pezzo di codice funziona correttamente, con importanti vantaggi:

- Semplifica le modifiche: Lo unit testing facilita la modifica del codice del modulo in momenti successivi (refactoring) con la sicurezza che il modulo continuerà a funzionare correttamente. Il procedimento consiste nello scrivere test case per tutte le funzioni e i metodi, in modo che se una modifica produce un fallimento del test, si possa facilmente individuare la modifica responsabile. Unit test già predisposti semplificano la vita

[4] http://www.eclemma.org

al programmatore nel controllare che una porzione di codice funzioni sempre correttamente.

- Semplifica l'integrazione: Lo unit testing semplifica l'integrazione di moduli diversi perché limita i malfunzionamenti dovuti a problemi nella interazione tra i moduli e non nei moduli stessi, rendendo i test di integrazione più semplici.

- Supporta la documentazione: Lo unit testing fornisce una documentazione del codice rappresentando le specifiche per il successo di un'unità di codice. Tali caratteristiche indicano l'uso appropriato dell'unità e i comportamenti errati che devono essere identificati nel suo funzionamento.

- Separazione dell'interfaccia dall'implementazione: Alcune classi possono far riferimento ad altre classi esterne. Un esempio è una classe che interagisce con un database: testare la classe spesso implica la scrittura del codice che interagisce con il database. Questo è un problema perché ogni unit test non dovrebbe mai varcare i confini della classe. Di conseguenza per isolare la classe da analizzare, si individua l'interfaccia con il database e la si implementa con un mock object, una simulazione dell'oggetto reale che può essere effettuata in condizioni controllate. I mock object vengono creati attraverso la libreria OpenSource EasyMock[5]. Per intercettare le chiamate a metodi statici viene invece usata la libreria PoweMock[6].

7.1.2 Limitazioni dello unit testing

In generale il testing non riesce ad identificare tutti gli errori in un programma e lo stesso vale per lo Unit Testing che, analizzando per definizione le singole unità, non può identificare gli errori di integrazione, problemi legati alla performance e altri problemi legati al sistema in generale. Per ottenere i benefici dallo unit test, è richiesto un rigoroso senso di disciplina durante tutto il processo di sviluppo. È essenziale mantenere traccia non solo dei test

[5] `http://easymock.org`
[6] `http://code.google.com/p/powermock`

che non sono stati sviluppati ed eseguiti, ma anche di tutte le modifiche effettuate al codice funzionale dell'unità in esame e di tutte le altre. L'uso di un sistema di controllo versione è essenziale. Se una versione successiva di una unità fallisce un test che aveva passato in precedenza, il sistema di controllo versione permette di evidenziare le modifiche al codice intervenute nel frattempo. Per questo il codice di PariPari è ospitato presso un server SVN[7], e viene gestito in Eclipse attraverso il plugin subclipse[8]. Per garantire una facile coordinazione fra team nel risolvere i bug di integrazione fra plugin è stato aperto un sito[9] BugZilla[10]. Vengono inoltre usati dei gruppi su Google Groups[11] per le comunicazioni fra tutti i partecipanti al progetto oltre che una wiki[12] appositamente messa in piedi. Un team preposto si occupa di eseguire tutti i test con cadenze predefinite e pubblica i risultati e i nuovi obiettivi in un'apposita pagina[13] della wiki.

[7] http://it.wikipedia.org/wiki/Subversion
[8] http://subclipse.tigris.org
[9] http://www.paripari.it/bugzilla
[10] http://www.bugzilla.org
[11] https://groups.google.com
[12] http://www.paripari.it/mediawiki
[13] http://www.paripari.it/mediawiki/index.php/StatusOfTesting

Conclusioni

Una corretta gestione del disco è importante per permettere a tutti i plugin di funzionare in maniera efficiente e sicura. Nondimeno, è importante che il plugin sia molto efficiente per non rallentare il lavoro degli altri plugin rendendo l'esperienza utente frustrante. Anche se scrivere test può risultare un po' noioso, la loro utilità si vede subito in caso di modifiche del codice in quanto si può vedere immediatamente se il nuovo codice scritto si comporta come si crede o se produce errori di qualche tipo. Questo rende il programmatore più sicuro di quello che fa, velocizzando i tempi di sviluppo dell'applicazione. Concludo sottolineando quanto sia stato formativo partecipare a questo progetto, in quanto non si è trattato di svolgere un lavoro prettamente personale come avviene in moltissime tesi, ma è stato necessario imparare a lavorare in gruppo acquisendo in questo modo delle metodologie che saranno utili nel mondo del lavoro. Inoltre con un progetto così grande si imparano ad usare strumenti e metodologie che sono il meglio dell'ingegneria del software degli ultimi anni.

Elenco delle figure