Università degli Studi di Padova
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi di laurea

PARICORE: MODULARIZZAZIONE DEI PLUGIN E GESTIONE DELLE LIBRERIE A RUNTIME

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

Laureando: Alberto Rubin

A.A. 2009-2010

Introduzione

In questa tesi parlerò del mio lavoro all'interno del gruppo *Core* nell'ambito del progetto di *PariPari*, cioè la gestione del meccanismo di caricamento delle librerie a runtime, finalizzato a rendere possibile l'utilizzo delle classi messe a disposizione da una libreria in modo dinamico. In questo modo è possibile, ad esempio, cambiare la versione della libreria da usare anche dopo l'avvio dell'applicazione.

Primo capitolo: si introduce la rete PariPari rivolgendo particolare attenzione al Core;

Secondo capitolo: si descrive il funzionamento del launcher e la procedura che ha portato alla sua realizzazione;

Terzo capitolo: si descrivono i ragionamenti che hanno portato alla progettazione della struttura delle librerie e se ne spiega il funzionamento;

Indice

1 PARIPARI
1.1 La rete …… 1
1.2 I plugin …… 2
1.3 L'architettura a plugin del client…… 3
1.4 Il Core …… 5
1.5 La cerchia interna …… 5

2 LAUNCHER
2.1 Avvio di PariPari …… 7
2.2 Compiti del Launcher …… 8
2.3 Class loader …… 9
2.4 Progettazione del Launcher …… 10
2.4.1 Classe LocalJar …… 12
2.4.2 Classe Launcher …… 15

3 STRUTTURA LIBRERIE
3.1 Librerie in PariPari …… 21
3.2 Struttura descriptor.xml …… 21
3.3 Progettazione librerie …… 24
3.4 XML Parser …… 27
3.5 Aggiornamento lista librerie …… 29
3.6 Caricamento librerie …… 30

CONCLUSIONI …… 32

ELENCO DELLE FIGURE …… 33

BIBLIOGRAFIA …… 34

1. PariPari

In questo capitolo verranno esposte le caratteristiche principali della rete PariPari, affrontando in maniera più approfondita il funzionamento del Core.

1.1 La rete

PariPari è una rete peer to peer (o P2P) in fase di sviluppo presso il Dipartimento di Ingegneria dell'Informazione dell'Università degli studi di Padova.

Letteralmente peer-to-peer significa “da pari a pari”.
Con questo termine generalmente si indica l’architettura di una rete che non possiede nodi gerarchizzati come client o server fissi, ma un numero di nodi equivalenti (peer) che fungono sia da cliente che da servente verso altri nodi della rete.

Le caratteristiche principali di PariPari sono le seguenti:

- è una rete *servless* nella quale tutti gli utenti sono trattati in maniera paritaria. Questo le permette di funzionare in maniera indipendente dal numero dei nodi connessi e dalla loro identità e soprattutto non richiede loro di rimanere collegati in modo permanente.

- possiede un’architettura espandibile attraverso l’implementazione di plugin, che le permette di offrire qualsiasi tipo di servizio.

- dispone di un servizio di gestione dei crediti che regola il rapporto di collaborazione tra i vari nodi.

- garantisce un sistema di anonimato tra i nodi partecipanti, permettendo loro di scambiarsi risorse senza conoscere l’indirizzo IP l’uno dell’altro.

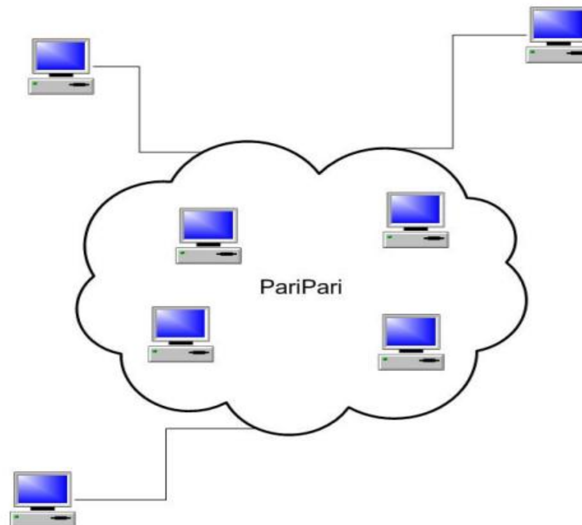

Figura 1.1 La rete PariPari e gli host esterni.

L’intera applicazione è scritta in Java. La scelta di questo linguaggio è dovuta al fatto che esso si adatta molto bene a quelle che sono le esigenze di PariPari. Infatti Java:

- è indipendente dalla piattaforma utilizzata.
- mette a disposizione una vasta collezione di librerie per il networking.
- permette attraverso la tecnologia *Java Web Start* (JWS) di rendere trasparente all’utente l’uso della rete: è infatti possibile accedere a PariPari direttamente dal browser web ed usufruire delle sue funzionalità in modo analogo all'utilizzo di un programma installato in locale.
- è un linguaggio sicuro e consente ai suoi applicativi di esserlo a loro volta.

1.2 I plugin

Prima di parlare dell’architettura a plugin del client di PariPari è utile dare una definizione formale al concetto di plugin.

Possiamo pensare ad un plugin come ad un software accessorio che aumenta le funzionalità di un programma. In PariPari un plugin a livello fisico è rappresentato da un file JAR. Un file JAR è un

archivio compresso (ZIP) usato per distribuire raccolte di classi Java. Tali file sono associabili anche al concetto di libreria.

L'utilizzo di questi archivi ha molteplici benefici:

- *Compressione:* con l'archiviazione si riesce a ridurre sensibilmente il numero di file arrivando anche ad unico file jar. Questo metodo, infatti, è largamente diffuso per applicazioni di tipo Applet, dato che diminuisce il tempo di caricamento e quindi il carico di rete e server.
- *Firma:* stabilisce l'identità del creatore del pacchetto. Essa è criptata con SHA (*Secure Hash Algorithm*) e mediante checksum garantisce l'immunità di un JAR firmato (se viene manomesso il pacchetto viene lanciata una SecurityException).
- *Estensione:* in genere l'archivio può essere espanso e quindi generare da esso altri package, classi e interfacce. Se l'archivio però è firmato non potrà più essere modificato.
- *Portabilità:* i file compressi restano in tutto e per tutto uguali agli originali e quindi utilizzabili su tutte le piattaforme che contengono la Java Virtual Machine (JVM).
- *Documentazione:* il programma JavaDoc è comunque in grado di autodocumentare il contenuto dell'archivio compresso.

Per poter lavorare con i plugin è essenziale creare degli oggetti in Java che li rappresentino. Esiste un procedimento rigoroso per creare un plugin che non verrà esposto, ma è importante sapere che ogni plugin è realizzato secondo una specifica interfaccia che ne descrive le funzionalità.

1.3 L'architettura a plugin del client

PariPari è definita una rete *multifunzionale*. Come già detto, essa nasce come piattaforma per la condivisione di svariati servizi: ad ogni servizio viene associato uno specifico plugin.
La rete inoltre permette di offrire, attraverso ciascuno dei suoi nodi, qualsiasi tipo di servizio anche ad host esterni non facenti parte di PariPari.
Osservando la figura 1.2 si può notare che ciascun client (ovvero ogni nodo della rete PariPari sulla quale viene eseguita l'omonima applicazione) è composto da una parte centrale detta *Core* e da due cerchie: una definita interna ed una esterna.

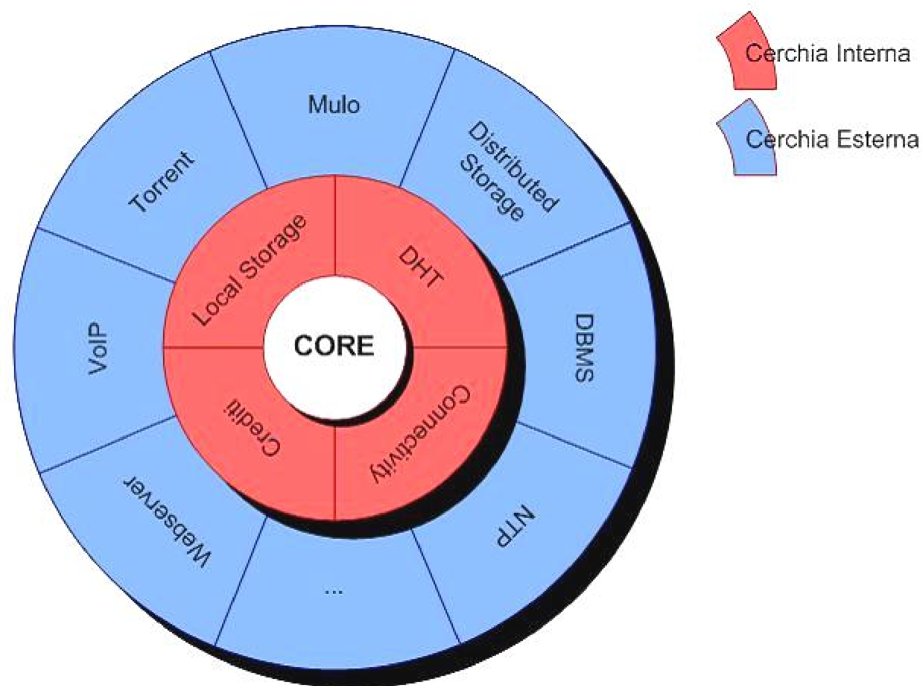

Figura 1.2 Struttura di un client.

Il Core si occupa di far comunicare fra loro i plugin quando essi ne hanno la necessità; lo sfruttamento da parte dei plugin delle risorse del computer (ad esempio il disco fisso) sul quale viene eseguito il client è gestito da alcuni moduli che sono essi stessi dei plugin, ma che svolgono funzioni delicate (gestione del disco, della banda, eccetera) e di conseguenza con privilegi maggiori.

Questo approccio fornisce una serie di vantaggi che rendono più facile l'attività di sviluppo: il programmatore vedrà tutti gli altri plugin come "scatole chiuse" di cui non conosce l'implementazione, ma solo i parametri che deve fornire per ottenere i servizi messi a disposizione.

Della cerchia interna fanno parte i moduli necessari al funzionamento basilare della rete; essi hanno lo scopo di garantire la gestione del file-system locale della macchina (`LocalStorage`), l'accesso alla rete (`Connectivity`), la reperibilità dei nodi della rete (`DHT`) e l'economia della stessa (`Crediti`).

Della cerchia esterna fanno parte tutti gli altri plugin, i quali offrono i propri servizi appoggiandosi sui moduli della cerchia interna.

1.4 Il Core

Il Core è il nucleo centrale di qualsiasi client e come già accennato assolve a tre funzioni principali: caricare le classi già compilate di cui si compongono i plugin, gestire la comunicazione tra plugin e gestire i crediti attribuiti a ciascun plugin a seconda delle richieste da esso soddisfatte.

Una volta lanciato il Core la classe `CoreLoader` si occupa di istanziare gli oggetti necessari al funzionamento del Core e di aggiungere i plugin richiesti dall'utente. Inoltre al caricamento dell'applicazione il Core verifica quali sono i plugin da caricare all'avvio attraverso la lettura del file di configurazione globale di PariPari (paripari.conf), e per ognuno di essi vengono istanziati i seguenti oggetti:

- un `paripari.core.PluginAccount`: oggetto utilizzato per gestire i crediti in possesso dello specifico plugin e per permettere la consultazione in tempo reale della propria situazione economica (crediti di cui dispone il plugin);

- un `paripari.core.PrivateMound`: immagazzina le richieste arrivate dal Core (provenienti dal Core stesso o da altri plugin);

- un `paripari.core.MoundPutter`: oggetto che permette di inoltrare le richieste al Core, e quindi ad ogni altro plugin;

- un `paripari.core.Plugin`: un'istanza della classe principale dichiarata dal plugin.

1.5 La cerchia interna

I plugin della cosiddetta cerchia "interna" sono privilegiati rispetto a tutti gli altri: sono gli unici adibiti a gestire direttamente le risorse della macchina (oltre al Core) ripartendole tra tutti i plugin e di conseguenza possiedono maggiore libertà di azione.

Crediti

I crediti in PariPari rappresentano una sorta di moneta con la quale vengono negoziate tutte le richieste effettuate dai diversi plugin presenti nel client. Nel momento in cui una richiesta ha luogo, è compito del plugin offerente stimarne il prezzo, il quale andrà sottratto al conto che ciascun modulo possiede sin dal suo caricamento. Nel caso in cui il costo sia troppo elevato, la richiesta può essere rifiutata.

In PariPari esistono due tipologie di crediti: quelli interni si riferiscono allo scambio di servizi tra plugin presenti nella stessa macchina, mentre gli esterni sono relativi allo scambio tra client diversi.

Local storage

Con Local storage si può gestire ed eventualmente limitare l'utilizzo della memoria di massa presente nella macchina. Eccetto il Core, chiunque necessiti di creare od utilizzare un file deve passare attraverso questo modulo. Local storage comprende inoltre per ogni singolo plugin degli elementi di statistica e di sicurezza quali la verifica dello spazio utilizzato su disco, il numero di file e le modalità di accesso a quest'ultimi.

Connettività

Per quanto riguarda tutti gli aspetti relativi allo sfruttamento della rete, è stato deciso di includere Connettività nella cerchia interna, dal momento che la banda è a tutti gli effetti una risorsa della macchina.

La materia di sua competenza comprende la creazione e la gestione dei socket, il mantenimento dell'anonimato nelle comunicazioni inter-client, la comunicazione multicast ed unicast, il routing geografico, il controllo della latenza e la gestione di molte altre problematiche.

DHT

Questo plugin permette di distribuire e localizzare nodi e risorse nella rete, utilizzando e mantenendo una tabella hash distribuita.

2. Launcher

In questo capitolo verrà esposto il funzionamento del launcher di PariPari. Inoltre verranno esposte le modifiche da me apportate e le motivazioni che mi hanno spinto verso determinate scelte implementative.

2.1 Avvio di PariPari

Il procedimento di avvio dell'applicazione che verrà di seguito esposto è riportato in figura 2.1.

Quando gli utenti arrivano sul sito di PariPari (www.paripari.it) e cliccano sul link di avvio dell'applicazione, viene automaticamente scaricato un file JNLP (fase 1).

Il Java Network Launch Protocol (JNLP) consente ad un'applicazione di essere avviata su un desktop client utilizzando risorse ospitate da un server web remoto mediante download automatico delle stesse. Esso specifica, tra le altre cose, il nome del JAR principale.

Nel nostro caso, infatti, una volta avviato Java Web Start viene scaricato nel client il file Launcher.jar (fase 2), che è una semplice applicazione java di piccole dimensioni che contatta i server degli aggiornamenti di PariPari al fine di recuperare le informazioni sull'ultima versione di Core.jar ed Interfaces.jar (fase 3).

Nel caso in cui almeno uno tra Core.jar e Interfaces.jar non sia aggiornato ne viene scaricata la versione più recente dal server degli aggiornamenti e caricata in un class loader che è figlio del class loader del Launcher (fase 4).

Questo è fatto in modo che le librerie possano essere caricate in fase di runtime, aggiungendo la risorsa puntata dall'URL all'*URLClassLoader* del Core; quando gli utenti vogliono caricare un plugin, come IM nell'esempio (fase 5), questo verrà recuperato dal server degli aggiornamenti, insieme a tutti gli altri plugin che da cui esso dipende (come specificato nel *descriptor.xml* del plugin).

Ogni plugin è caricato da un class loader proprio, in modo che i plugin non possano tentare di accedere ad altre classi di plugin attraverso l'utilizzo della *reflection* (fase 6). Tutti questi class

loader sono figli del class loader del Core: quando hanno bisogno di caricare qualche libreria che può essere condivisa con altri plugin il Core la carica nel proprio ClassLoader, per far sì che sia utilizzabile da tutti i plugin. In genere si cerca di rendere condivise le librerie in modo da risparmiare memoria sul client.

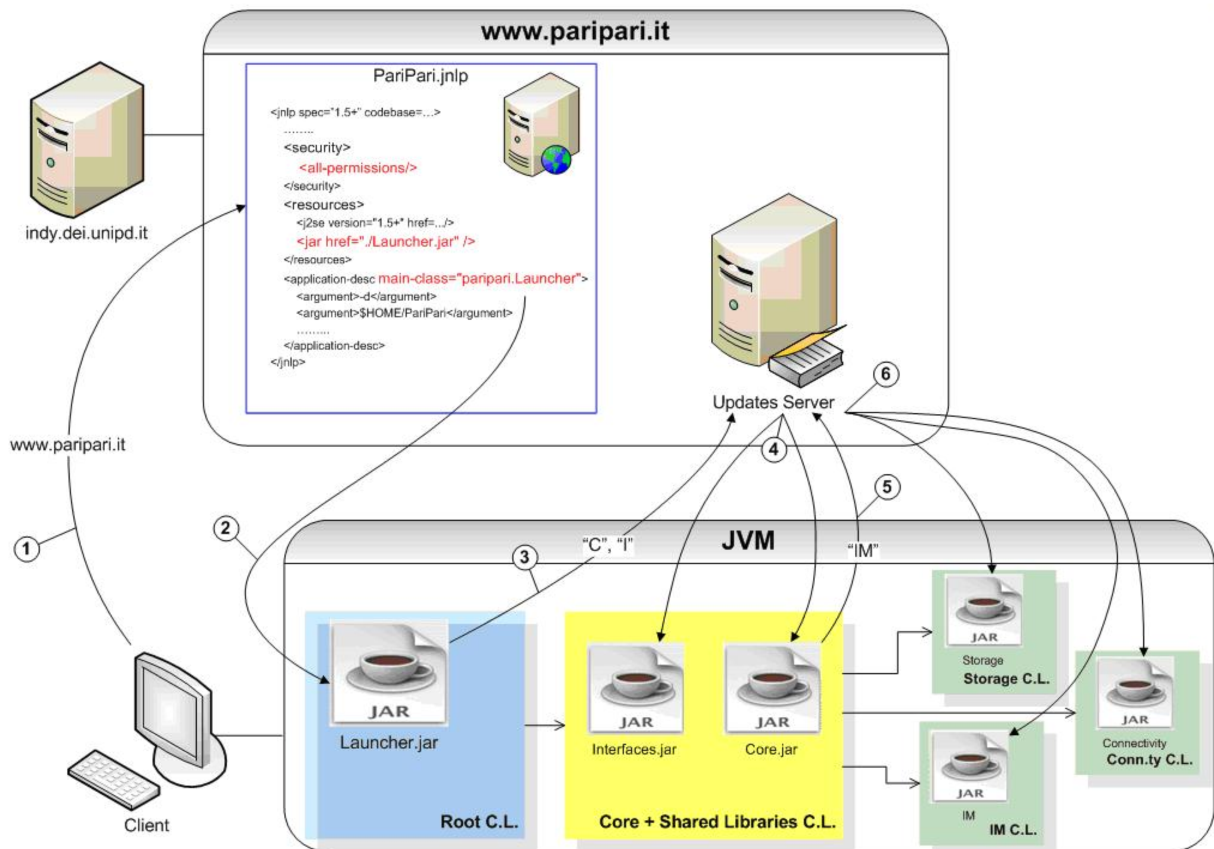

Figura 2.1 Avvio di PariPari

2.2 Compiti del Launcher

Possiamo riassumere i compiti principali del launcher in 4 punti:

- controllare che la versione locale di *Core.jar* salvato nella cartella di PariPari sia aggiornata e, nel caso non lo sia, scaricare la versione più recente dal server degli aggiornamenti di PariPari;

- analoga procedura dev’essere seguita per il file *Interfaces.jar*;

- creare un PariPariClassLoader che servirà per poter aggiungere le librerie a runtime;

- lanciare il core.

Il launcher presente prima del mio arrivo si limitava semplicemente a scaricare ogni volta i file *Core.jar* ed *Interfaces.jar* evitando qualsiasi controllo sulle versioni salvate in locale. Quindi in sostanza ho preferito progettare da zero la struttura delle interfacce che avrebbero permesso la realizzazione del launcher.

2.3 Class loader

Fino adesso si è citato più volte il class loader senza mai soffermarsi troppo su cosa esso sia e sui ruoli che deve svolgere, quindi si ritiene opportuno aprire una piccola parentesi per spiegare meglio perché esso è così importante.

Java permette di costruire applicazioni *estendibili dinamicamente*, nel senso che un'applicazione è in grado di caricare a tempo di esecuzione (*runtime*) nuovo codice e di eseguirlo, incluso del codice che nemmeno esisteva quando l'applicazione è stata scritta.

Java infatti effettua il caricamento dinamico delle classi, cioè carica le informazioni relative alle classi durante l'esecuzione del programma. Di questo caricamento si occupa un oggetto *ClassLoader*, una classe messa a disposizione da Java. Un *ClassLoader* si occupa quindi di importare i dati binari che definiscono le classi (e le interfacce) di un programma mentre esso è in esecuzione.

Nella Java Virtual Machine è presente il cosiddetto *class loader primordiale* che viene utilizzato per caricare le classi dal file system locale (comprese quelle delle API di Java). Poiché fa parte della Java Virtual Machine, tale class loader è implementato in C. Il comportamento di questo class loader di default sarà quello di cercare un file *.class* per ogni classe da caricare nel file system locale, nei path indicati dalla variabile d'ambiente CLASSPATH.

2.4 Progettazione del Launcher

Per la progettazione del launcher siamo partiti analizzando attentamente i compiti da esso svolti e cercando di scomporre ciascun compito in compiti più piccoli. Si pensi, ad esempio, che per il confronto delle versioni di *Core.jar* si dovrà prima risalire alla versione del file che si ha in locale, per poi interrogare il server per farsi fornire l'ultima versione del jar desiderato ed infine confrontare le due versioni. Risulta quindi evidente che esiste la necessità di creare un'interfaccia che conterrà tutti i metodi utili per la comunicazione col server degli aggiornamenti ed un'altra interfaccia che si occuperà di risalire alle informazioni dei file che si hanno in locale.

Seguendo le indicazioni dell'Extreme Programming (XP), al quale PariPari si attiene, si è preferita la scrittura di un numero maggiore di metodi, ciascuno composto da poche righe, piuttosto che sviluppare pochi metodi molto articolati; questo per poter rendere più leggibile il codice e per rendere più facili le operazioni di testing.

L'interfaccia di comunicazione col server dovrà contenere anche i metodi che permettono il download dei due file. In sostanza la prima interfaccia creata, chiamata appunto *IServerCommunication*, sarà così strutturata:

IServerCommunication
getLatestCore
getLatestInterfaces
isCoreUpToDate
isInterfacesUpToDate
downloadCore
downloadInterfaces

Per quanto riguarda la gestione dei file locali è stata creata l'interfaccia *IJarInfo* che dovrà risalire alla versione di un jar di PariPari dato il suo nome. Sono quindi sufficienti due soli metodi: uno per accedere al jar e uno per estrarre la versione. La struttura dell'interfaccia sarà la seguente:

ILocalJar
getta
getJarVersion

Il launcher sicuramente dovrà interrogare il server degli aggiornamenti: i metodi dell'interfaccia *IServerCommunication* dovranno perciò essere implementati all'interno del launcher. È però

possibile creare un'ulteriore interfaccia chiamata *ILauncher* che estenderà *IServerCommunication* e in più avrà il metodo `launchCore` che servirà appunto per avviare il Core.

La struttura finale delle interfacce è riportata in figura 2.2 in cui i rettangoli tratteggiati rappresentano interfacce, mentre quelli dai bordi continui rappresentano classi. Le frecce con tratteggio più lungo indicano che una classe implementa l'interfaccia a cui punta; quelle con tratteggio più corto indicano che una classe usa al suo interno oggetti della classe puntata; infine, le frecce continue indicano che una classe (o interfaccia) estende la classe (o interfaccia) puntata.

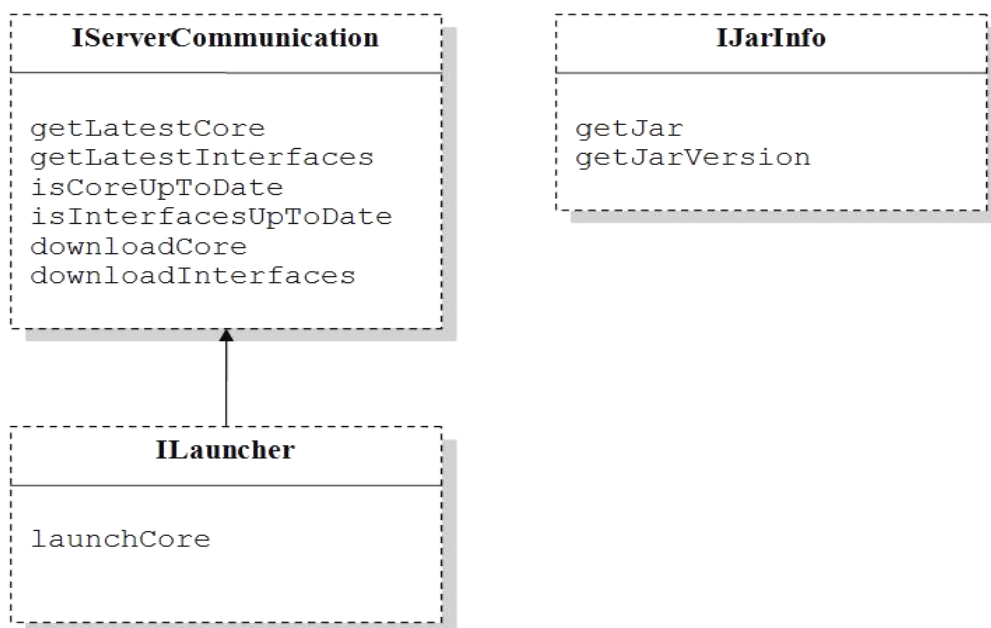

Figura 2.2 Interfacce utilizzate per la realizzazione del launcher

Partendo da queste interfacce risulta semplice implementare la classe *Launcher* che di fatto rappresenta il launcher. Quindi la classe implementerà *ILauncher* ed userà un oggetto di tipo *LocalJar*, ovvero l'implementazione dell'interfaccia *IJarInfo*. I concetti espressi sono riportati in figura 2.3.

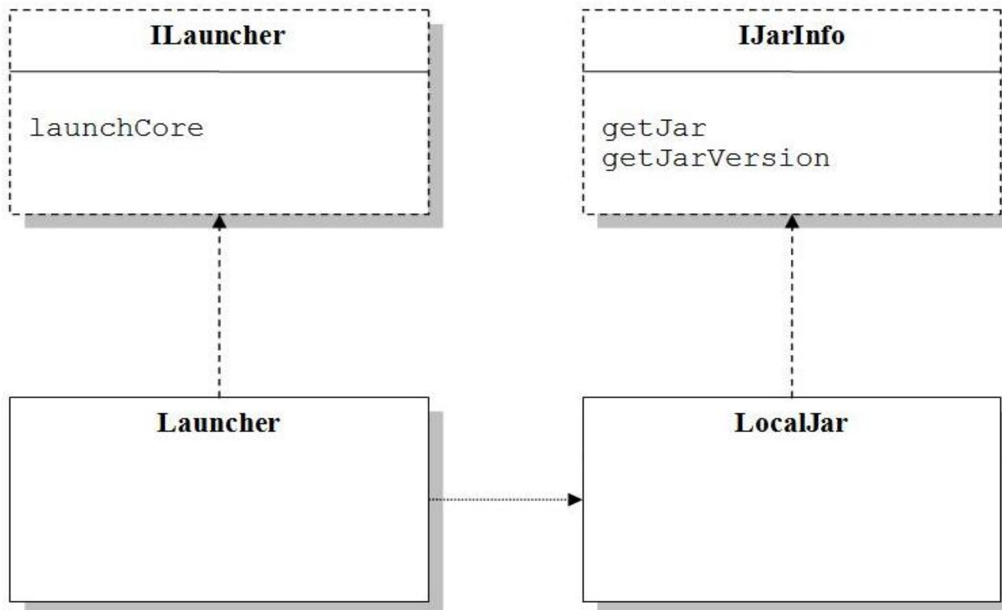

Figura 2.3 Classi per la realizzazione del launcher

Nei paragrafi successivi andremo ad analizzare dettagliatamente le classi appena prodotte per capirne meglio la loro utilità ai fini della realizzazione del launcher.

2.4.1 Classe LocalJar

Questa classe è stata creata con lo scopo di ottenere informazioni riguardo alla versione dei file Core.jar ed Interfaces.jar, ma il suo uso può essere esteso anche ad altri jar. Essa è costituita semplicemente da due metodi: getJar e getJarVersion.

getJar

Questo metodo ha lo scopo di restituire un oggetto di tipo `JarFile` associato al jar che si trova nella cartella principale di PariPari. Il nome del jar è passato come parametro al metodo.

getJarVersion

Questo metodo ha lo scopo di ricavare la versione del jar attraverso il suo manifest. Il manifest viene ottenuto chiamando il metodo getJar e passandogli come parametro il nome del jar di cui si vuole conoscere la versione.

getJarVersion

```
public String getJarVersion(String jarName) throws IOException {

        String version=null;
        String temp="";


               Manifest manifest = getJar(jarName).getManifest();

               // Get the main attributes in the manifest
            Attributes attrs = (Attributes)manifest.getMainAttributes();

            // Enumerate each attribute
            for (Iterator it=attrs.keySet().iterator(); it.hasNext(); ) {
                // Get attribute name
                Attributes.Name attrName = (Attributes.Name)it.next();

                // Get attribute value
                temp = attrs.getValue(attrName);
                if(attrName.toString().compareTo("Implementation-Version")==0)
                       version=temp.split(" ")[0];

            }
               return version;
}
```

Come già detto nei paragrafi precedenti, un file JAR è in grado di supportare una vasta gamma di funzionalità, compresa la firma elettronica, il controllo della versione e altre ancora. Ma cosa dà al file JAR la capacità di essere così versatile? La risposta è inclusa nel *manifest* del file JAR.

Il *manifest* è un file speciale che può possedere informazioni relative ai file contenuti all'interno del JAR. Impostando opportunamente queste "meta" informazioni, si crea un file JAR da utilizzare per una grande varietà di scopi.

A seconda del ruolo che si desidera far assumere al file JAR, potrebbe essere necessario modificare le impostazioni predefinite del *manifest*. Se si è interessati solo alla compressione ed alla archiviazione di contenuti, non ci si deve preoccupare di modificare il *manifest*.

Quando si crea un file JAR, esso automaticamente riceve un file *manifest* predefinito. Ci può essere solo un file *manifest* all'interno di un archivio, ed ha sempre il percorso:

```
META-INF/MANIFEST.MF
```

Ad esempio, quando un file JAR viene creato a partire dalla versione 1.2 del Java Development Kit, il file manifest predefinito è molto semplice. Eccone il contenuto integrale:

```
Manifest-Version: 1.0
```

La riga mostra che le voci di un *manifest* assumono la forma di coppie del tipo "`header: valore`".

Uno degli usi che si può fare di un file JAR è quello di "contenitore" per un'applicazione Java. In questo caso sorge la necessità di indicare in qualche modo quale classe dentro al file JAR è quella che avvia l'applicazione, cioè specificare qual è la classe principale, tra tutte quelle aventi il metodo con firma `public static void main(String[] args)`.

Un esempio di *manifest* di PariPari è il seguente:

esempio di *manifest* di PariPari

```
Manifest-Version: 1.0
Specification-Title: Java PariPari Core Classes
Specification-Vendor: PariPari Team
Specification-Version: 1.0
Implementation-Title: paripari
Implementation-Vendor: PariPari Team
Implementation-Version: Build 01
```

Dove le sezioni più importanti sono:

Specification-Version: 1.0

la versione delle specifiche (indicativamente quando si aggiunge qualcosa di nuovo)

Implementation-Version: Build 01

la versione dell'implementazione (indicativamente quando si correggono bug o si migliorano le prestazioni)

Il metodo `getJarVersion`, come si può vedere, cerca proprio il valore associato al campo *Implementation-Version* per risalire sia alla versione di *Core.jar* che di *Interfaces.jar*. Nel caso non venga trovato, il metodo restituisce il valore `null`.

2.4.2 Classe Launcher

Il launcher vero e proprio è rappresentato dalla classe `Launcher`. Per svolgere in modo completo ed efficiente i propri compiti ad essa sono stati aggiunti alcuni metodi oltre a quelli precedentemente citati. In questo paragrafo verrà esposta una breve panoramica dei metodi e del loro utilizzo.

getInstance

Attualmente la classe CoreStarter è realizzata seguendo un pattern Singleton abbastanza semplice. Il Singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.

Il Singleton in questione utilizza l'inizializzazione lazy: l'idea è quella di includere nella classe che implementa il singleton una classe-contenitore avente, come attributo statico, una istanza del singleton stesso: il primo accesso a tale attributo statico (e la contestuale inizializzazione) verrà quindi effettuato durante l'inizializzazione della classe-contenitore, e quindi sempre in modo serializzato. In questo modo l'istanza del singleton viene creata solo alla prima chiamata del metodo *getInstance*, e non prima.

Singleton del Launcher di PariPari

```
public class Launcher {

  /**
   * Costruttore privato, in quanto la creazione dell'istanza deve essere controllata.
   */
  private Launcher() {}

  /**
   * La classe Contenitore viene caricata/inizializzata alla prima esecuzione di
getInstance()
   * ovvero al primo accesso a Contenitore.ISTANZA, ed in modo thread-safe.
   * Anche l'inizializzazione dell'attributo statico, pertanto, viene serializzata.
   */
  private static class SingletonLauncher {
    private final static Launcher LAUNCHER = new Launcher();
  }

  /**
   * Punto di accesso al Singleton. Ne assicura la creazione thread-safe
```

```
   * solo all'atto della prima chiamata.
   * @return il Singleton corrispondente
   */
  public static Singleton getInstance() {
    return SingletonLauncher. LAUNCHER;
  }
}
```

parseArgs

Il Core accetta una serie di parametri che gli venivano passati direttamente dal file JNLP. Ora però, con l'aggiunta di un nuovo livello fra Core e JNLP si è pensato di estrarre i parametri di cui si necessita a livello del launcher e di passarli al Core. Questo è stato possibile con l'aggiunta del metodo *parseArgs*.

I parametri accettati dal Core sono riportati nella seguente tabella:

Comando	**Cosa fa**
-d *directory*	Imposta l'home directory di PariPari. Tutti i file di configurazione e jar utilizzati per la sessione corrente verranno salvati all'interno di questa cartella (questo parametro deve essere passato ogni volta che si desidera utilizzare una cartella diversa da user_home / PariPari).
-j	Usato da PariPari.jnlp per permettere al Core di sapere che è stato lanciato da Java Web Start.
-p *portNumber*	Imposta la porta locale da utilizzare per ascoltare i comandi remoti (quando il Core viene eseguito in modalità remota).
-r	Imposta il Core in modalità remota.
-s *serverName*	Imposta l'hostname (o l'indirizzo IP) del server che il Core sta per interrogare per scaricare plug-in e aggiornamenti.
-t	Imposta il Core in modalità testuale.
-u *portNumber*	Imposta la porta remota del Server degli aggiornamenti.

Di questi, al launcher interessano solo: la working directory, l'URL e la porta remota del server degli aggiornamenti.

refreshRemoteJarList

Per riuscire a risalire alla versione dei due file presenti nel server degli aggiornamenti, inizialmente si è pensato di aggiungere un comando al server per poterlo interrogare e farsele restituire. In realtà poi si è optato per una soluzione migliore. Poiché il Core all'avvio già scarica il file *plugin.list*, contenente l'elenco di tutti i plugin con versioni annesse, sarebbe stato più semplice includere in questo file anche le versioni dei due jar. In questa maniera, una volta scaricato il file, si può accedere alle righe desiderate ed ottenere la versione. Questo metodo si occupa quindi di chiedere al server la lista delle informazioni associate ai plugin, a *Core.jar* e a *Interfaces.jar* e le salva nel file *plugin.list*. Quest'ultimo è composto da un certo numero di righe, ciascuna delle quali rappresenta un plugin (più avanti vedremo che in realtà ci saranno anche librerie). Di ogni plugin si hanno quattro informazioni separata da ":", così strutturate:

firma del jar **:** *nome plugin* **:** *versione plugin* **:** *nome file plugin*

Il metodo è costituito da un thread che si mette in ascolto sulla porta del server degli aggiornamenti. Successivamente viene inoltrata la richiesta della lista al server tramite il comando "L" ed il file di testo, passato come risposta dal server, tramite uno *stream* letto con un BufferedReader (e quindi una riga alla volta), viene salvato nel file *plugin.list*.

Durante la fase di scrittura del metodo si è riscontrato un piccolo problema dovuto al fatto che il thread creato era di tipo "*daemon thread*". Un thread daemon fornisce un servizio generale e non essenziale in background mentre il programma esegue altre operazioni. Dunque il programma termina e di conseguenza la JVM può uscire, quando tutti i thread non-daemon terminano. Nel nostro caso è stata necessaria una particolare attenzione alla sincronizzazione dei thread, perché nella versione stand alone il daemon thread era più lento rispetto al thread principale (quello del main) e quindi il programma terminava prima di aver completato la lettura della risposta del server. La sincronizzazione tra thread è stata possibile tramite l'introduzione di un semaforo opportunamente gestito. In questo modo, dopo che il thread viene lanciato, il *main* si mette in attesa.

getLatestCore

Dopo aver scritto il metodo *refreshRemoteJarList* ottenere la versione remota di *Core.jar* non è più un problema, in quanto è sufficiente accedere al file *plugin.list* all'interno della working directory e cercare tra le sue righe quella rappresentante il file interessato. Analogo ragionamento va fatto anche per il metodo *getLatestInterfaces*.

isCoreLocalUpToDate

Questo metodo si limita semplicemente a confrontare se le due stringhe restituite dopo aver chiamato i metodi *getLatestCore* e *getJarVersion* sono uguali: nel caso lo siano ritorna *true*, nel caso non lo siano ritornano *false*. Analogo ragionamento si può fare per il metodo *isInterfacesLocalUpToDate*.

launchCore

È il metodo che si occupa di avviare il Core di PariPari. Inoltre svolge l'importante funzione di creare e di ottenere il class loader del Core stesso.

Abbiamo già parlato dell'esistenza di un *class loader primordiale*, ma questo non è l'unico che possiamo utilizzare, infatti è possibile creare class loader personalizzati, ad esempio quello del Core.

Il class loader del Core è un oggetto di tipo *CoreClassLoader*, cioè una classe che estende *URLClassLoader*. Quest'ultima viene utilizzata per caricare classi e risorse da un URL che si riferisce sia a JAR che a directory. Qualsiasi URL che termina con un '/' si presume che faccia riferimento a una directory. In caso contrario, l'URL è assunto come riferimento a un file JAR. Per avere una panoramica completa va inoltre specificato che *URLClassLoader* estende *SecureClassLoader*, che a sua volta estende *ClassLoader*.

Il metodo della classe `ClassLoader` utilizzato per caricare in memoria una classe è `loadClass`, al quale va fornito il nome (comprensivo della definizione del package di appartenenza) della classe da caricare. Nel nostro caso la classe che il launcher deve caricare in memoria è la classe `paripari.core.CoreStarter`. Una volta ottenuto l'oggetto `Class` restituito dal metodo *loadClass* è possibile ottenere un'istanza della classe chiamando, tramite

riflessione, il metodo `getInstance` presente in `CoreStarter`. Questo metodo semplicemente chiama il main di *CoreStarter*. Poiché *CoreStarter* è singleton, la prima chiamata a `getInstance` implica la creazione dell'istanza *CoreStarter* solo per la sessione corrente di PariPari.

E' interessante soffermarsi sul concetto di *riflessione*. Il supporto alla riflessione consente ad un programma di ispezionarsi ed operare su se stesso. Tale supporto comprende la classe `Class` nel package `java.lang` e l'intero package `java.lang.reflect` che introduce le classi `Method`, `Constructor` e `Field`.

La riflessione si usa per:

- ottenere informazioni su una classe e sui suoi membri;
- manipolare oggetti.

In particolare:

- la classe `Field` permette di scoprire e impostare valori di singoli campi;
- la classe `Method` consente di invocare metodi;
- la classe `Constructor` permette di creare nuovi oggetti.

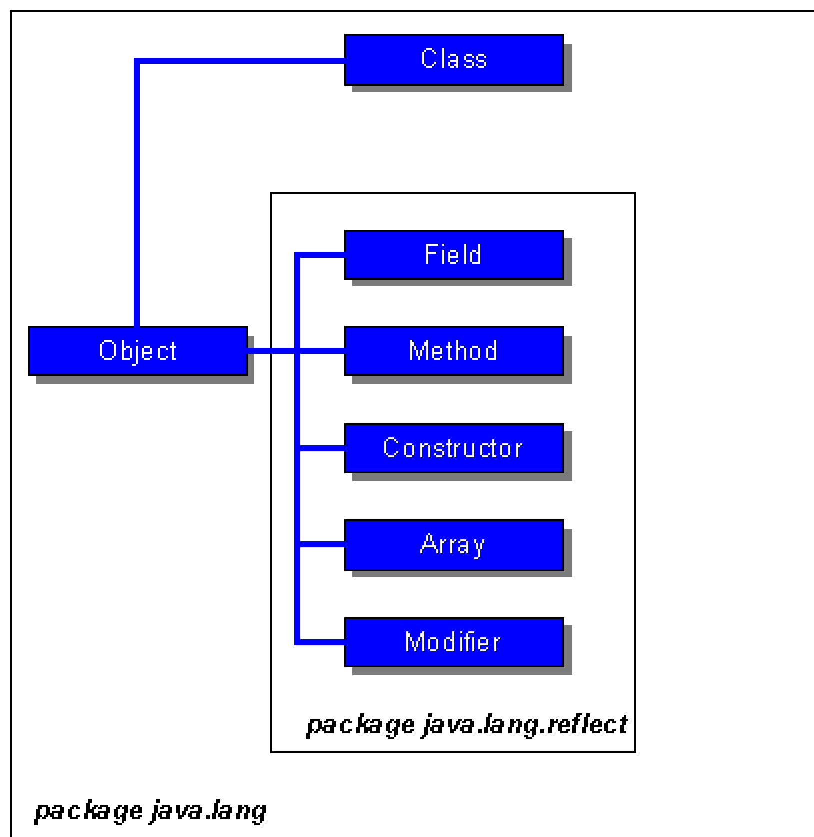

Figura 2.4 Classi componenti la riflessione

La riflessione permette di invocare indirettamente un metodo passato per nome.
Lo schema di invocazione diretta è:

```
res = m.invoke(oggettoTarget, args)
```

dove *m* è un'istanza di `Method`, *args* è un array di `Object` che rappresentano gli argomenti da passare al metodo, e *res* è un `Object` che rappresenta il risultato del metodo.

Essendo il metodo `getInstance` statico non è necessario passare come parametro un oggetto, quindi al posto di *oggettoTarget* metteremo `null`, mentre come *args* passeremo l'array di stringhe passato come parametro da riga di comando al main del launcher.

Ma come si può ottenere l'oggetto m di tipo Method che rappresenta il metodo da chiamare?
L'oggetto viene recuperato tramite una chiamata a `getMethod`:

```
m = c.getMethod(nomeMetodo, parametri);
```

dove *c* è un'istanza di *Class* che rappresenta la classe dell'oggetto target sul quale verrà invocato il metodo, *nomeMetodo* è una String che contiene il nome del metodo, e *parametri* è un array di *Class* che rappresenta la lista dei parametri formali del metodo.

Nel nostro caso *c* è un'istanza di CoreStarter, *nomeMetodo* è `getInstance` e *parametri* è il tipo runtime che corrisponde ad un'array di String (`String[].class`).

3. Struttura librerie

In questo capitolo verrà esposta l'importanza delle librerie all'interno di PariPari. Inoltre verrà illustrata la loro struttura, argomentando le motivazioni delle scelte effettuate.

3.1 Librerie in PariPari

L'obiettivo finale del mio lavoro è dare la possibilità ai plugin di caricare librerie a runtime. Risulta quindi essenziale definire delle classi che lavorino sulle librerie e che diano la possibilità di memorizzare tutte le informazioni necessarie al Core.

Oltre a definire nuove classi, esiste la necessità di modificare alcune di quelle già caricate. Si pensi ad esempio alle dipendenze che un plugin può avere da librerie. Risulta quindi necessaria l'aggiunta di una variabile d'istanza nella classe `PluginJar` contenente tutte le librerie da cui esso dipende.

In una prima analisi possiamo affermare che le informazioni essenziali che ci interessa memorizzare di una libreria sono il nome, il nome del jar associato alla libreria, la versione ed infine se essa può essere condivisa con altri plugin.

Inoltre bisogna tener presente che possono nascere dipendenze (sia per plugin che librerie) da librerie che si trovano al di fuori del sito di PariPari. Per questo tipo di librerie esiste la necessità di fornire anche l'indirizzo al quale poterle reperire.

Risulta quindi necessario definire lo schema del *descriptor.xml* che i plugin (ma anche le librerie) useranno per dichiarare quali librerie possono voler usare.

3.2 Struttura descriptor.xml

Prima di esporre e motivare la struttura delle librerie da me scelta è utile fare qualche cenno all'xml, in quanto risulta essenziale capire com'è strutturato un file xml per poter accedervi.

XML (eXtensible Markup Language) è un meta-linguaggio per definire la struttura di documenti e dati. Nella terminologia XML, il termine documento è utilizzato più in generale come contenitore di

informazioni. Concretamente, un documento XML è un file di testo che contiene una serie di tag, attributi e testo secondo regole sintattiche ben definite. È utile ora fare una breve introduzione sui concetti fondamentali di questo meta-linguaggio.

Un documento XML è intrinsecamente caratterizzato da una **struttura gerarchica**. Esso è composto da componenti denominati **elementi**. Ciascun elemento rappresenta un componente logico del documento e può contenere altri elementi (sottoelementi) o del testo.

Gli elementi possono avere associate altre informazioni che ne descrivono le proprietà. Queste informazioni sono chiamate **attributi**.

L'organizzazione degli elementi segue un ordine gerarchico o arboreo che prevede un elemento principale, chiamato **root element** o semplicemente root o radice.

La radice contiene l'insieme degli altri elementi del documento. Possiamo rappresentare graficamente la struttura di un documento XML tramite un albero, generalmente noto come **document tree**.

La struttura logica di un documento XML dipende dalle scelte progettuali. Siamo noi a decidere come organizzare gli elementi all'interno di un documento XML. Non esistono regole universali per l'organizzazione logica di un documento, quindi nello svolgere il mio lavoro ho avuto libertà pressoché assoluta.

La struttura logica di un documento XML viene tradotta in una corrispondente struttura fisica composta di elementi sintattici chiamati **tag**. Questa struttura fisica viene implementata tramite un file di testo creato con un qualsiasi editor.

Un'applicazione che voglia accedere ai dati nel file XML deve avanzare come all'interno di un file system: partendo dall'origine, lungo un percorso che indica la posizione ricercata. Un programma in grado di fare tutto questo è detto **parser** (dal verbo inglese "to parse", analizzare). Esso è risultato essenziale per il lavoro da svolgere, in quanto una volta stabilita la struttura del descriptor, si è dovuto implementare un parser che permettesse di estrapolare le informazioni dal file stesso.

Nella seguente tabella sono riportate le informazioni di interesse e la corrispondente componente xml scelta.

Informazione	Componente XML
nome	elemento
Versione	attributo del tag library

Condivisa	attributo del tag library
Esterna	attributo del tag library
Firma	attributo del tag library
Dipendenze	elemento
url	elemento

Come si può notare, il nome della libreria è stato impostato non come attributo, ma come elemento perché si è ritenuto che il nome della libreria fosse più importante degli altri attributi. Inoltre è stata aggiunta una rappresentazione della firma come attributo della libreria, in quanto si è ritenuto utile inserire anch'essa tra le informazioni della libreria. Un altro concetto aggiunto è l'url delle librerie esterne; esso rappresenta l'unico modo per scaricare la libreria esterna nel caso in cui non sia già stata scaricata.

A questo punto bisogna chiarire che il *descriptor.xml* non potrà essere contenuto nelle librerie esterne, in quanto non sono gestite dal gruppo di PariPari. Esso infatti sarà presente solo nelle librerie definite "interne", ovvero presenti nel nostro sito. Risulta quindi già evidente una separazione, seppur parziale, tra le librerie in due categorie: interne ed esterne.

Un altro aspetto importante è la definizione delle dipendenze da librerie. Per un plugin è essenziale poter memorizzare le informazioni associate alle librerie di cui necessita. Quindi è stato necessario aggiungere nell'elemento *dependencies* del *descriptor* dei plugin la possibilità di definire l'elemento *library*. Per permettere la definizione di questo tipo di dipendenza ho dovuto aggiungere nella classe `XMLParser` il metodo `getLibraryDependencies` che sostanzialmente restituisce una lista di tutte le librerie da cui il plugin in questione dipende. Questa funzione risulta fondamentale per descrivere la dipendenza da librerie esterne, ma altrettanto importante per librerie interne, in quanto si dà la possibilità in un futuro di implementare il *versioning*: ad esempio un plugin può voler usare la libreria *log4j* versione 1.1 perché sono cambiati i metodi nelle versioni successive e gli sviluppatori non hanno ancora aggiornato il codice.
Inoltre è stata data la possibilità di far dipendere librerie interne da API di PariPari permettendo l'aggiunta dell'elemento *API* tra gli elementi di dependencies di una libreria.

Un esempio di descriptor.xml di un'immaginaria libreria *library1* è riportato in figura 3.1.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
- <library version="1.2.20100729" external="false" shared="false" signature="-1749617103">
    <name>library1</name>
  - <dependencies>
    - <API>
        <abstract>paripari.API.connectivity.socket</abstract>
      </API>
    - <API>
        <abstract>paripari.API.storage.File</abstract>
      </API>
    - <library version="1.1.20100727" external="true" shared="true">
        <name>library2</name>
        <url>http://www.apache.org/dyn/logging/log4j/apache-log4j-1.2.16.jar</url>
      </library>
    - <library version="1.1.20100727" external="false" shared="true" signature="457550124">
        <name>library3</name>
      </library>
    </dependencies>
  </library>
```

Figura 3.1 Esempio di descriptor.xml per librerie

3.3 Progettazione librerie

Fin dall'inizio della fase di progettazione è sorta una necessità. Come abbiamo già detto nei paragrafi precedenti, bisogna poter memorizzare in appositi oggetti le informazioni che riguardano le librerie intese come dipendenze. Allo stesso tempo però si deve tenere traccia delle librerie che effettivamente si hanno a disposizione, o meglio di quelle che sono fisicamente presenti nel file system all'interno della directory di *PariPari*. Proprio da questi due diversi modi di concepire una libreria è nata l'idea di creare oggetti separati per rappresentare ognuno di essi. Entrambe le rappresentazioni, seppur diverse, hanno delle caratteristiche in comune: sono caratterizzate da un nome, da una versione, possono essere condivise o meno, possono dipendere da altre librerie.

A livello strutturale, risulta evidente che le due implementazioni discendano da una stessa interfaccia, chiamata `ILibrary`.

La classe che si occupa di rappresentare una libreria come dipendenza a livello astratto è chiamata `LibraryRequest` e contiene le informazioni estratte dal *descriptor* del plugin (o della libreria) che la dichiara come dipendenza.

Per quanto riguarda le librerie fisicamente presenti, abbiamo già accennato al fatto che esiste un'ulteriore suddivisione concettuale di una libreria, infatti una libreria può essere interna o esterna. È importante far notare che a livello astratto possiamo non scindere i concetti in due classi separate, in quanto è sufficiente aggiungere un attributo *isExternal* e gestire con un minimo di attenzione gli eventuali altri parametri, mentre per gli oggetti che rappresentano file cambia molto gestire l'uno o l'altro tipo, si pensi ad esempio allo scaricamento della libreria. Per questi motivi si è deciso di

creare due classi separate `InternalLibrary` ed `ExternalLibrary`, che implementano la stessa interfaccia `Library` che estende `ILibrary`.

La struttura di interfacce e classi che permettono la gestione delle librerie è rappresentata in figura 3.2.

LibraryRequest
getAPIDependencies
getSignature
getLibraryURL
isExternal
setVersion
setShared
setIsExternal
setAPIDependencies
setLibraryDependencies
setLibraryURL
setSignature

ILibrary
getLibraryDependencies
getSimpleName
getVersion
isShared

Library
isLocalUpToDate
downloadLibrary
getPublicKey
getFile
getJarName

InternalLibrary
getAPIDependencies
getLibrarySignature

ExternalLibrary
setVersion
setShared
setLibraryURL
setLibraryDependencies

Figura 3.2 Struttura gestione librerie

Come si può notare, nell'interfaccia `ILibrary` sono presenti i metodi che permettono di risalire alle informazioni delle librerie, mentre nell'interfaccia `Library` sono presenti i metodi per la gestione fisica delle librerie.

Questa struttura sfrutta le potenzialità del polimorfismo in Java, in quanto la parte del Core che si occuperà della risoluzione delle dipendenze chiamerà sempre gli stessi metodi, ma il modo in cui verranno effettivamente eseguiti sarà diverso (ad esempio per il metodo downloadLibrary, se la libreria è esterna viene aperta una `HTTPURLConnection`, mentre se la libreria è interna viene contattato il server degli aggiornamenti di PariPari).

Per quanto riguarda le librerie esterne, sappiamo che esse non hanno un *descriptor.xml* da cui poter estrapolare le informazioni che ci interessano. L'unico modo di conoscere i dati caratterizzanti

le librerie stesse è quello di estrarre i vari `LibraryRequest` dalla lista di dipendenze di un plugin (o di una libreria) e di impostare i campi dell'oggetto `ExternalLibrary` facendosi restituire i corrispettivi valori per ogni `LibraryRequest` che rappresenta una libreria esterna.

Inoltre si è considerato difficile valutare se una libreria esterna è aggiornata o no, cioè è sufficiente controllare se la dimensione in byte del jar in locale è differente da quella remota oppure basta semplicemente controllare se i nomi coincidono? Per far fronte a questo problema si è supposto che una libreria esterna è sempre aggiornata. Questo a livello di codice si traduce facendo sempre restituire al metodo `isLibraryLocalUpToDate` della classe `ExternalLibrary` il valore `true`. Inoltre si è assunto che gli sviluppatori delle librerie adottino una gestione delle risorse di tipo RESTful (*Representational Transfer State*), ovvero che in sostanza ogni URL rappresenta un determinato oggetto.

Nell'interfaccia `ILibrary` è stata introdotta una costante molto importante: *allLibraries*. Questa costante ha il compito importante di tenere traccia di tutte le librerie utilizzate da PariPari che sono fisicamente presenti nel file system. Infatti sia il costruttore di `InternalLibrary` che di `ExternalLibrary` aggiungono l'oggetto appena creato a questa lista.

Di seguito verranno analizzate in maniera più dettagliata le classi sopra citate.

InternalLibrary

La classe serve per rappresentare una libreria interna fisicamente presente nella directory di PariPari e possiede una variabile d'istanza per ogni attributo caratteristico di tale libreria. I metodi più utilizzati sono i metodi *getter* che permettono di risalire a tutti questi valori. Tra questa categoria di metodi, di particolare importanza sono `getLibraryDependencies` e `getAPIDependencies` che permettono di risalire a tutte le dipendenze che una libreria può avere rispettivamente nei confronti di altre librerie o di API di PariPari.

Al costruttore vengono passati un oggetto di tipo *File,* contenente la libreria, ed il nome del file ad essa associata. Dopo aver impostato alcune variabili, il costruttore richiama `loadFromJar`. Quest'ultimo è essenziale per ricavare le informazioni della libreria, in quanto dopo aver associato il parser alla libreria vengono settate le variabili d'istanza ai valori restituiti dai metodi del parser.

ExternalLibrary

Come precedentemente affermato, ExternalLibrary rappresenta una libreria esterna fisicamente presente nella directory di PariPari. La struttura è analoga a quella della classe qui sopra descritta. Una delle differenze di rilievo è l'informazione aggiuntiva che rappresenta la caratteristica principale delle librerie esterne, cioè l'attributo *libraryURL*, al cui interno è contenuto l'indirizzo URL dove risiede la libreria.

Un'altra differenza importante è la mancanza di un parser associato alla libreria, perché, come già detto, le librerie esterne non dispongono di un *descriptor.xml*. L'unico modo di ottenere i valori delle variabili d'istanza è quello di leggere dal descriptor della libreria interna che necessita della nostra libreria esterna tutte le informazioni di quest'ultima ed impostarle, tramite i metodi *setter*.

3.4 XML Parser

Abbiamo già detto che il parser svolge un ruolo importante in quanto permette di estrarre i valori di attributi ed elementi associati alle librerie dal *descriptor.xml*. All'interno del progetto esisteva già un parser che svolgeva un ruolo analogo per i plugin rappresentato dalla classe `XMLParser`.

In maniera analoga alla realizzazione di tutte le altre classi si è partiti dall'implementazione di un'interfaccia, `IXMLLibraryParser`, che fornisse i metodi di cui il nostro parser necessitava. Per stabilire questi metodi è stato sufficiente guardare la struttura del *descriptor.xml* sul quale si doveva lavorare. Analizzando la struttura di quest'ultimo risulta evidente che sicuramente si devono estrarre i vari attributi presenti nel tag `<library>`, gli elementi `<name>` e, se la libreria in questione è esterna, `<url>`. La parte più interessante però è rappresentata dal tag `<dependencies>`. Infatti ci possono due tipi di dipendenze: da libreria o da plugin. Per la dipendenza da plugin è sufficiente ricavare il nome dell'API di cui si necessita, mentre per la dipendenza da libreria si devono estrarre tutte le informazioni ad essa associate. Il metodo che si occupa di tale compito è `getLibraryDependencies` ed è riportato qui di seguito.

getLibraryDependencies

```
public ArrayList<LibraryRequest> getLibraryDependencies() {
        ArrayList<LibraryRequest> returnThis = null;
        Element dependencies = getFirstLevelElement("dependencies");
        if (dependencies != null) {
                NodeList libraries = dependencies.getElementsByTagName("library");
                if (libraries != null && libraries.getLength() > 0) {
                        // there are declared dependencies
                        returnThis = new ArrayList<LibraryRequest>();
                        for (int i = 0; i < libraries.getLength(); i++) {
                                // scan through all declared dependencies
                                Element thisDependency = (Element) libraries.item(i);

                                //now extracts the parameter of our library
                                boolean isExternal=
Boolean.parseBoolean(thisDependency.getAttribute("external"));
                                boolean isShared=
Boolean.parseBoolean(thisDependency.getAttribute("shared"));
                                String version=thisDependency.getAttribute("version");
                                String signature=thisDependency.getAttribute("signature");
                                String libraryName = getLibraryAttribute(thisDependency,"name");
                                LibraryRequest temp=new LibraryRequest(libraryName);
                                temp.setIsExternal(isExternal);
                                temp.setShared(isShared);
                                temp.setVersion(version);
                                if(isExternal){
                                        try {
                                                String libraryURL=
getLibraryAttribute(thisDependency,"url");
                                                temp.setLibraryURL(libraryURL);

                                        } catch (MalformedURLException e) {
                                                e.printStackTrace();
                                        }
                                }
                                else{
                                        //If the library is internal, also has a signature
                                        temp.setSignature(signature);
                                }
                                returnThis.add(temp);
                        }
                }
        }
        return returnThis;
}
```

Il metodo, implementato nella classe `XMLLibraryParser`, recupera l'elemento *dependencies* all'interno del documento XML e successivamente si fa restituire un `NodeList` con tutti gli elementi discendenti aventi il tag *library*, secondo l'ordine in cui compaiono. Quindi di ogni singolo elemento della lista vengono restituiti i valori che andranno ad impostare i vari campi dell'oggetto `LibraryRequest`. Da notare la gestione dei parametri che differenziano le librerie esterne da quelle interne. Infine ogni oggetto creato viene aggiunto all'`ArrayList` di tipo `LibraryRequest` che verrà restituito.

In maniera analoga gli altri metodi ricavano le informazioni specifiche.

La classe `XMLLibraryParser` andrà collocata anche sul server degli update (oltre che nel client nel quale viene eseguito PariPari) perché il server ha la necessità di costruirsi il proprio archivio delle librerie di PariPari. Quindi deve estrarre tutte le informazioni per poter esaudire le richieste dei client. I core "locali" devono però poter sapere se le librerie che hanno in locale sono aggiornate, o se è il caso di scaricarle dal server degli update. La cosa funziona allo stesso modo dei plugin, ovvero la classe che fa da parser è proprio la stessa sia nel jar del core che nel jar del Server, quello che cambia è l'uso che il core e il server fanno delle informazioni recuperate.

3.5 Aggiornamento lista librerie

Come avviene per i plugin, è necessario mantenere una lista aggiornata di tutte le librerie contenute nella sub directory *lib* della working directory di PariPari.
Il metodo che si occupa di questo compito è `refreshLocalLibraryList` della classe `PluginUpdater`. Questa classe contiene tutti i metodi necessari per mantenere aggiornati i plugin e le librerie.
Il metodo per prima cosa accede alla sub directory *lib* e si fa restituire, attraverso l'uso di un `CustomFileFilter`, un array contenente tutti i file jar contenuti in essa. Se l'array non è nullo, per ogni elemento si controlla che il file sia una libreria valida, ovvero il nome non deve cominciare per *old_*, *bad_* e non deve essere una directory. Se rispetta questa serie di vincoli allora ci si fa restituire il path assoluto. Attraverso questo valore si controlla se il file è già presente nella variabile d'istanza *jars* della classe `PluginTable` che rappresenta la lista dei jar conosciuti (esiste un'unica lista per plugin e librerie). Se è già presente si controlla che esista un solo jar associato a questo path; nel caso in cui ne esista più di uno si segnala l'errore all'utente. Altrimenti se il file non è presente nella lista dei jar conosciuti, si controlla attraverso il metodo hasDescriptor se il jar contiene un descriptor.xml: in caso affermativo si desume che la libreria è interna, mentre in caso contrario la libreria sarà esterna.

Per le librerie interne si crea il relativo oggetto `InternalLibrary`, che in modo automatico aggiunge la libreria sia nella lista delle librerie di cui si dispone, sia in quella dei jar conosciuti.

Una delle scelte prese in fase di sviluppo è stata quella di considerare i jar messi a disposizione dal server degli aggiornamenti tutti allo stesso modo: la lista che andrà a formare il file plugin.list contiene infatti oltre a plugin, Core.jar ed Interfaces.jar anche le librerie interne.

Quindi si è dovuto tener presente che le informazioni contenute in *plugin.list* non si riferissero più solo a plugin, ma anche librerie. Di conseguenza si è modificato il metodo **`addJar(PluginJar jar)`**, che aggiungeva alla variabile d'istanza *jars* un nuovo jar trovato nella working directory di PariPari. Esso è diventato **`addJar(AbstractJar jar)`**. `AbstractJar` è la classe astratta estesa da PluginJar e contiene dei metodi utili per lavorare con i jar a livello astratto.

La soluzione più elegante da adottare è stata quella di prendere *AbstractJar* come superinterfaccia comune a PluginJar ed InternalLibrary e farla implementare da entrambe le classi. Poiché non tutti i metodi hanno un motivo di esistere in entrambe le classi, si è scelto di creare una classe `NullJar` che estende *AbstractJar*. Questa classe sovrascrive tutti i metodi, e non fa altro che lanciare una `UnsupportedOperationException` per ognuno. Le due classi quindi estenderanno *NullJar* invece che *AbstractJar* e sovrascriveranno solo i metodi di cui intendono fornire un'implementazione, sfruttando il cosiddetto *overriding* di Java.

ha innescato un inevitabile effetto domino che ha portato a modificare e a gestire in modo opportuno gran parte dei metodi che sfruttano la variabile d'istanza *jars*.

3.6 Caricamento librerie

Una volta progettata ed implementata la struttura per la gestione delle librerie si è passati alla scrittura del metodo necessario per la realizzazione del caricamento delle librerie a runtime. Il metodo in questione è stato chiamato `loadLibrary` ed è stato definito come metodo statico della classe `PluginCalls`, una classe contenente i metodi che rappresentano tutte le chiamate possibili che i plugin possono effettuare al Core.

Le informazioni di cui necessitiamo sono principalmente l'identità del plugin che richiama il metodo e il nome della libreria che vogliamo caricare. In realtà ci servono altre informazioni riguardanti la libreria, ma il lavoro svolto fino a questo punto ci viene incontro. È stato infatti aggiunto nella classe *PluginJar* il metodo che restituisce le varie dipendenze da altre librerie. Con questo possiamo scandire tutte le librerie dalle quali il nostro plugin dipende e cercare quella con il nome dato. Una volta ottenuto l'oggetto *LibraryRequest* opportuno disponiamo di tutte le informazioni riguardanti la libreria.

Possiamo quindi risalire alla firma (*signature*) della libreria, e se questa è interna il campo dovrà avere valore non nullo.

La prima cosa da controllare è vedere se si possiede già nella subdirectory *lib* di *PariPari* la libreria da caricare. E' necessario innanzitutto quindi scandire tutti gli elementi della costante

allLibraries e stabilire se sono di tipo *InternalLibrary* o di tipo *ExternalLibrary*. Una volta fatta questa separazione si deve guardare, per le librerie interne, se esiste nella lista una libreria con *signature* uguale a quello estratto dalla *LibraryRequest*. In caso affermativo significa che possediamo già la libreria e che quindi sarà sufficiente controllare se la libreria è aggiornata e nel caso non lo sia scaricarla.

Analogo procedimento va seguito per le librerie esterne, con la differenza che per esse sarà sufficiente confrontare se in *allLibraries* c'è una libreria con lo stesso nome.

Se la libreria non è stata trovata nella lista delle librerie possedute, allora si dovrà scaricarla.

La parte più complicata della realizzazione di questo metodo è sicuramente la risoluzione delle dipendenze. Infatti se si deve caricare una libreria *library1* che necessita di una libreria *library2*, la quale a sua volta necessita di un'altra libreria *library3* bisognerà controllare se si possiedono già, oltre a *library1* anche le altre due librerie. Nel caso in cui non si disponga di tutte le librerie, si dovrà scaricarle e controllare anche le eventuali dipendenze di quest'ultime. L'approccio che si è scelto è quello della creazione di un metodo che richiama se stesso e per ogni libreria di una dipendenza controlla se questa è presente in locale: nel caso lo sia, la aggiunge alla lista dei file da caricare nel class loader, mentre nel caso contrario la scarica ed istanzia un oggetto di tipo *InternalLibrary* o *ExternalLibrary* e richiama se stesso passando le dipendenze da risolvere di quest'ultime come parametro.

Appena accertata la presenza della libreria cercata, si deve verificare se la libreria da caricare può essere condivisa con altri plugin. Questo è molto importante in quanto se essa può essere condivisa, dev'essere aggiunta al class loader del Core; in caso contrario dev'essere aggiunta al class loader del plugin.

Conclusioni

Oggetto di questa relazione è stata l'analisi del launcher e della struttura delle librerie che andranno a far parte di PariPari.

Il launcher è stato testato completamente e con successo in tutte le sue funzioni attraverso l'utilizzo diretto dello stesso all'interno del progetto. I test sono stati eseguiti con particolare attenzione in quanto il ruolo svolto dal launcher è essenziale per l'intera applicazione: un avvio sbagliato comporterebbe all'errato funzionamento dell'intera applicazione.

La struttura delle librerie è stata testata con un programma stand alone che simula in maniera esclusiva il funzionamento delle librerie mediante l'invocazione del metodo *loadLibrary*. Particolari attenzioni sono state rivolte alla risoluzione delle dipendenze delle librerie che di fatto è stata una delle fasi più impegnative del lavoro svolto. I test sono stati eseguiti più volte ed in situazioni diverse, in modo da verificare il corretto funzionamento anche nei casi particolari.

L'introduzione della gestione delle librerie rappresenta un importante passo avanti perché, a partire da questa, è possibile realizzare una serie di funzionalità non poco rilevanti. Tra queste, le due principali sono sicuramente la possibilità di caricare a runtime moduli e il *versioning* delle librerie.

Inoltre si deve tener presente che un'applicazione Web deve sempre considerare la larghezza di banda come la risorsa più preziosa e più rara. Quindi la possibilità di dividere grandi file jar in parti più piccole che possono essere scaricate *on demand* contribuisce a potenziare la reattività di PariPari.

Elenco delle figure

Figura 1.1 La rete di PariPari e gli host esterni ………………………………………………… 2
Figura 1.2 Struttura di un client ………………………………………………………………... 4
Figura 2.1 Avvio di PariPari …………………………………………………………………… 8
Figura 2.2 Interfacce utilizzate per la realizzazione del launcher ……………………………… 11
Figura 2.3 Classi per la realizzazione del launcher …………………………………………….. 12
Figura 2.4 Classi componenti la riflessione ……………………………………………………. 19
Figura 3.1 Esempio di descriptor.xml per librerie ……………………………………………... 24
Figura 3.2 Struttura gestione librerie …………………………………………………………... 25

Bibliografia

[1] PARIPARI CORE TEAM, *PariPari core's wiki pages*, http://paripari.it/mediawiki/index.php/Core_en

[2] PARIPARI CORE TEAM, *Paripari core's javadoc page*, http://www.paripari.it/jws/core_doc

[3] M. BONAZZA, *PariPari: progettazione e realizzazione in JavaTM di un web server distribuito*, Tesi di laurea triennale, Università degli studi di Padova, 2007.

[4] M. BONAZZA, *Paricore*, Tesi di laurea specialistica, Università degli studi di Padova, 2009.

[5] P. BERTASI, *Progettazione e realizzazione in Java di una rete peer to peeranonima e multifunzionale*, Tesi di laurea specialistica, Università degli studi di Padova, 2005.

[6] "*Wikipedia*", http://it.wikipedia.org/wiki/P2p.

[7] "*Listing the Main Attributes in a JAR File Manifest*", http://www.exampledepot.com/egs/java.util.jar/ GetMainAttr.html

[8] "*Struttura dei documenti XML*", http://xml.html.it/guide/lezione/1842/struttura-dei-documenti-xml/

[9] "*RIFLESSIONE (reflection)*",http://nicchia.ingce.unibo.it/oop/web/21-riflessione.html