Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

JJPF: uno strumento per calcolo parallelo con JINI

Candidato
Patrizio Dazzi

Relatore
Prof. Marco Danelutto

Controrelatrice
Dott.ssa Chiara Bodei

Anno Accademico 2003/2004

Dedicata a:

I miei genitori,
perchè da quasi venticinque anni mi supportano e sopportano,
perchè mi hanno sempre lasciato libero di fare le mie scelte,
perchè spero che un giorno, i miei figli, provino nei miei confronti
la stessa stima che io nutro per loro.

Chiara,
perchè è la musa ispiratrice e la principessa del mio cuore,
perchè senza di lei "Amore" sarebbe soltanto una parola.

I miei parenti,
perchè nessun uomo potrebbe sperare in una famiglia migliore.

I miei amici più cari,
perchè è grazie a loro se non mi sono mai sentito un "Figlio Unico".

Ringraziamenti

Questa tesi è il risultato di una bellissima esperienza accademica che dura ormai dall'ottobre 1998. In questi anni sono molte le persone che mi hanno aiutato, compreso e sopportato, purtroppo poche righe non sono sufficenti per ricordale tutte.

Desidero cominciare ringraziando il mio relatore, il Prof. Marco Danelutto, sia per avermi aperto gli occhi davanti al fantastico mondo delle architetture parallele, sia per l'incredibile disponibilità manifestata nei miei confronti.

Un grande, infinito grazie va ai miei genitori, se quel primo maggio di trentatre anni fa non avessero deciso di andare a ballare, adesso non ci sarebbe nessun Patrizio e nessuna tesi.

Un pensiero speciale va a Chiara, perchè quel tramonto di quasi otto anni fa ha fatto da cornice al più bel giorno della mia vita, senza di lei vivrei in un mondo in bianco e nero.

Infine, un grandissimo grazie lo devo a Christian, Francesco e Valerio per essermi stati vicini quando ne avevo più bisogno, non dimenticherò ciò che hanno fatto per me.

Patrizio Dazzi

Indice

Elenco delle figure

Elenco dei Frammenti di Codice

Capitolo 1

Introduzione

L'obiettivo di questa tesi è valutare la possibilità di realizzare un ambiente per il calcolo parallelo, utilizzando Jini [JOR04] come middleware di supporto; in particolare per testare l'adeguatezza di Jini nell'ambito del calcolo parallelo strutturato.

La tecnologia della programmazione parallela strutturata [Col89] prevede che ogni applicazione venga espressa come una composizione di moduli ognuno dei quali realizza una forma di parallelismo. Fra le forme di parallelismo [Pel98] che normalmente si utilizzano, ricordiamo per esempio, farm, pipeline, data parallel e divide-and-conquer.

Nel nostro caso si è scelto di esaminare la realizzabilità di un paradigma di tipo farm stateless e di una sua variante che permetta, invece, la condivisione di dati fra i worker.

Questa scelta è stata guidata principalmente dalla considerazione che il paradigma farm, avendo una struttura logica semplice, permette una analisi dei risultati ottenuti (valori di performance) più immediata rispetto alle altre forme di parallelismo. Inoltre esistono molte applicazioni reali che utilizzano questo paradigma, basti pensare alle applicazioni embarassingly parallel e parameter sweeping, come ad esempio i sistemi di elaborazioni di foto satellitari o quelli per l'analisi di dati raccolti dai radiotelescopi.

Per verificare quanto effettivamente Jini si presti alla realizzazione delle due varianti delle forme di parallelismo farm, abbiamo realizzato un framework per la programmazione parallela e distribuita [SCK94, MB01, ZJ03] basato su tale tecnologia, il Java & Jini Parallel Framework (JJPF). JJPF permette la realizzazione di applicazioni parallele di tipo task farm. In particolare si occupa di:

- cercare e arruolare le risorse computazionali disponibili;

- gestire le comunicazioni;
- eseguire le operazioni di staging e di spawning;
- bilanciare dinamicamente il carico;
- garantire un adeguato grado di fault tolerance.

Quindi lo sviluppatore che usa JJPF può concentrare la sua attenzione sul problema da risolvere e sulla sua decomposizione, senza doversi preoccupare esplicitamente dei diversi aspetti legati al controllo della concorrenza.

JJPF dà la possibilità di realizzare semplicemente un'applicazione parallela secondo un paradigma di tipo farm o secondo una sua variante che consenta la condivisione di dati fra i worker.

L'astrazione che Jini offre è quella di una *"federazione"* di un insieme di clienti e servizi comunicanti tra di loro.

Il framework realizzato, per sfruttare le possibilità offerte da Jini, fornisce una visione astratta di ogni risorsa computazionale mostrandola ai clienti come un "Servizio". Gli utilizzatori di questi servizi sono detti "Clienti".

Affinchè un calcolatore possa essere arruolato come worker in una computazione parallela realizzata utilizzando JJPF, è sufficiente eseguire alcuni semplici passi:

1. istanziare un oggetto che realizzi il "Servizio"
2. istanziare un oggetto che si occupi di registrare il "Servizio" presso un *LookupService* (l'entità che tiene traccia dei "Servizi" attivi).

La realizzazione di un *cliente* è leggermente più complessa, per realizzarne uno è necessario:

1. Scrivere una classe che implementi i metodi dichiarati nell'interfaccia `ProcessoIf`:
   - `setData` e `getData` per il trasferimento dei dati tra cliente e servizio;
   - `run` per mandare in esecuzione il programma che il cliente ha trasferito sul servizio;
   - `setSharedObject` per indicare al worker l'identificatore dell'entità condivisa da utilizzare;
2. Istanziare un oggetto di tipo `BasicClient` passando come parametri al costruttore:

- una `collection` Java contenente l'insieme dei dati in input;
- una `collection` java dove JJPF memorizzerà i risultati;
- il vettore delle classi che saranno usate dal worker;
- (solo nel caso del farm con stato) una classe che implementi l'interfaccia `SharedObjectIf`: l'entità condivisa che i worker dovranno utilizzare;

Il framework sviluppato dimostra che la tecnologia Jini costituisce un valido strumento per la realizzazione delle forme di parallelismo fornite da JJPF.

JJPF garantisce, infatti, ottimi valori di scalabilità [FB87] (§ 3.1.1 a pagina 35), affidabilità (§ 3.1.2 a pagina 35) e dinamicità (§ 3.1.3 a pagina 36).

Le prove sono state realizzate presso il centro di calcolo del Dipartimento di Informatica dell'Università di Pisa e saranno descritte dettagliatamente nel paragrafo 4.2.4 a pagina 62. I test hanno dimostrato che il JJPF garantisce buoni livelli di **scalabilità** fino all'impiego di cinquantacinque calcolatori.

Il framework realizzato è stato progettato per gestire correttamente l'improvvisa scomparsa delle risorse di calcolo e permettere il recupero dei dati spediti ai calcolatori non più attivi. Le prove effettuate hanno dimostrato che JJPF riesce a garantire, anche durante eventi critici un ottimo grado di **affidabilità**.

Inoltre i test hanno evidenziato che il framework sviluppato possiede un'ottima capacità di reazione all'aggiunta e alla rimozione di risorse computazionali. Nelle prove in cui, a computazione iniziata, è stato raddoppiato il numero di macchine a disposizione, JJPF ha reagito di conseguenza, fornendo (dopo un breve transitorio) prestazioni adeguate al nuovo numero di macchine.

Abbiamo organizzato la tesi in modo da analizzare in maniera organica tutti gli aspetti fin qua accennati. Abbiamo quindi deciso di strutturarla nella maniera seguente:

- nel capitolo 2 descriviamo brevemente gli strumenti utilizzati, illustrando le motivazioni che ci hanno portato al loro uso;
- nel capitolo 3 vediamo il progetto logico del JJPF, ne descriviamo gli obiettivi, gli attori e i possibili scenari d'uso; illustriamo inoltre i problemi che abbiamo affrontato durante la sua realizzazione;
- nel capitolo 4 mostriamo i dettagli implementativi del framework sviluppato, analizzando dettagliatamente ogni singolo package in cui è suddiviso;

- nel capitolo 5 analizziamo i test effettuati descrivendo l'ambiente di prova e gli strumenti utilizzati, valutandone i risultati;
- nel capitolo 6 traiamo infine le nostre conclusioni sul lavoro svolto.

Inoltre sono presenti tre appendici:

- nell'Appendice A sono contenute le informazioni inerenti l'installazione, la configurazione e l'uso del framework JJPF;
- nell'Appendice B mostriamo i diagrammi UML, sia secondo la vista architetturale che quella di dettaglio.
- nell'Appendice C riportiamo il codice sorgente del JJPF;

Capitolo 2

Strumenti Utilizzati

Nell'introduzione abbiamo illustrato lo scopo della nostra tesi: valutare quanto Jini si presti ad essere utilizzato come middleware di supporto per la realizzazione di un ambiente per il calcolo parallelo. In questo capitolo vedremo cos'è Jini, ne descriveremo gli attori principali e il funzionamento di base.

2.1 Jini: Network Plug&Play

Jini è un sistema distribuito, scritto in Java, finalizzato alla costruzione di reti mutevoli e dinamiche, formate da componenti, dispositivi e servizi eterogenei [JNJ03, JFA04, BP01, UL00, Sci97, Jin].

Il suo obiettivo è la minimizzazione dei compiti di amministrazione, configurazione e gestione dei dispositivi connessi. Jini consente l'utilizzo di risorse eterogenee in modo uniforme, astraendo la loro natura fisica come oggetti.

Uno dei punti di forza di Jini è quello di consentire la configurazione e registrazione di una risorsa in modo automatico [WJT99, JRW03, JBE, JK01, CSGG01]. Ogni dispositivo o servizio che viene connesso alla rete e dichiara la sua presenza può essere trovato e utilizzato dai clienti.

L'uso di Jini è particolarmente interessante nelle reti caratterizzate da un alto grado di dinamicità [Cre04], in cui è frequente l'aggiunta o la rimozione di elementi (stampanti, calcolatori, memorie di massa,...). Analizziamo alcuni scenari tipici al fine di rendere più chiaro questo concetto:

- Una nuova stampante viene connessa alla rete, annuncia la sua presenza e funzionalità: da questo momento qualsiasi cliente può utilizzarla senza dover essere appositamente riconfigurato;

- Una macchina fotografica digitale può essere connessa alla rete e fornire

un'interfaccia che consenta di effettuare foto e di spedirle ad una stampante di rete;

- Una stampante che finisca l'inchiostro a colori può rendere nota in maniera automatica la sua transizione di stato da `stampante-a-colori` a `stampante-in-bianco-e-nero`;
- Una stampante che finisca la carta può cambiare il suo stato in `offline`;

Un sistema basato su Jini, detto anche *"federazione"*, è un insieme di clienti e servizi comunicanti tra di loro tramite i protocolli compatibili col framework: nella maggior parte dei casi questo significa applicazioni Java comunicanti tramite RMI.

2.2 Remote Method Invocation (RMI)

RMI è un meccanismo che consente l'invocazione di metodi su oggetti remoti, mascherando la tecnologia rete sottostante[WW04, SRW04, Eck03, JBM+].

Dato un oggetto X, in esecuzione su una macchina A, che metta a disposizione alcuni dei suoi metodi e data una applicazione Y, in esecuzione su una macchina B, grazie a RMI Y può utilizzare i metodi pubblicati da X come se entrambi gli oggetti si trovassero sulla stessa macchina (vedi Figura 2.1 a fronte in alto).

RMI utilizza la serializzazione Java [PB02], una tecnologia che permette di trasformare un oggetto in un insieme di byte pronti per essere trasferiti da una JVM ad un'altra. L'entità che si occupa della gestione delle comunicazioni tra X ed Y è chiamata "proxy" (letteralmente "procura").

La parte bassa della Figura 2.1 nella pagina successiva mostra un esempio del funzionamento di RMI in cui un oggetto Y invoca il metodo m1 esposto dall'oggetto remoto X. Il suo comportamento può essere riassunto nelle sei fasi seguenti:

1. l'oggetto Y in esecuzione sulla macchina B invoca il metodo m1 sul proxy stub dell'oggetto X;
2. il proxy stub, che si trova su B, redirige al proxy skel sito sulla macchina A la chiamata al metodo m1;
3. il proxy skel invoca il metodo m1 sull'oggetto X;
4. X esegue il metodo e restituisce il risultato al proxy skel;
5. il proxy skel spedisce i risultati al proxy stub;

Schema Logico del Funzionamento
dell'invocazione remota di metodi

Schema Logico del Funzionamento di Java RMI

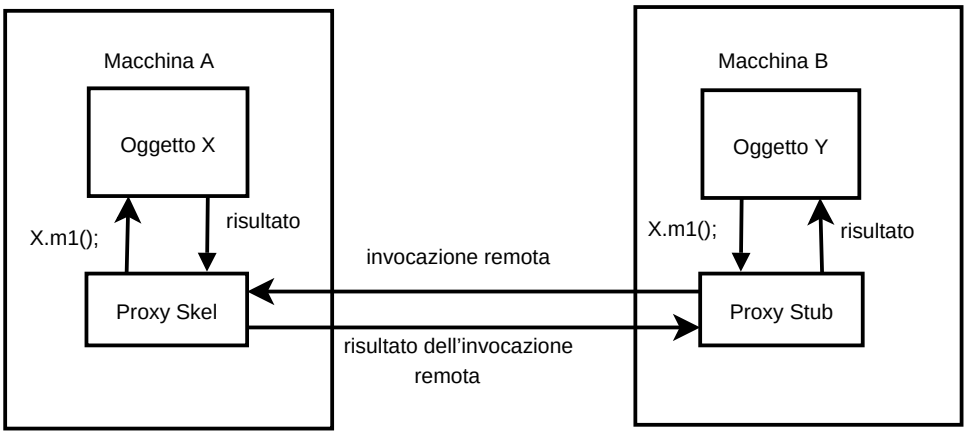


Figura 2.1: Invocazione remota dei metodi

6. il proxy stub si occupa di inoltrare i risultati ricevuti ad Y;

La prima implementazione di RMI utilizzava come protocollo di trasporto il Java Remote Method Protocol (JRMP), basato su TCP.

In seguito sono nate diverse implementazioni di RMI che hanno consentito l'utilizzo di altri protocolli, come ad esempio HTTP [TBL96, RF99], IIOP [RMI04a] (il protocollo di trasporto di CORBA [COR04]), SSL [RMI04b] e Firewire (IEEE 1394 [IEE04]). Un punto dolente è che ognuna di queste implementazioni espone un'interfaccia di programmazione distinta e mette a disposizione un insieme di classi realizzate ad-hoc. Quindi una applicazione scritta utilizzando un particolare protocollo diventa di fatto dipendente da esso.

Un altro grosso problema di RMI è la creazione dei proxy: per generarli è necessario passare in input all'applicazione `rmic`, il file risultante dalla compilazione della classe di cui si vogliono esporre i metodi, questo vincolo limita molto la dinamicità delle applicazioni create poichè non consente la generazione di proxy a run-time.

Per risolvere questo problema Sun Microsystems ha sviluppato il Jini Extensible Remote Invocation (JERI), per la cui trattazione rimandiamo al paragrafo successivo.

2.2.1 Jini Extensible Remote Invocation (JERI)

JERI [Ven02, Som03] è un'evoluzione di RMI (§ 2.2 a pagina 20) in quanto ne consente una totale personalizzazione.

I punti chiave di JERI sono la possibilità di generare i proxy a tempo di esecuzione, tramite il meccanismo di "esportazione", e la possibilità di scegliere il protocollo del livello di trasporto da utilizzare senza dover riscrivere o ricompilare il codice.

Esportazione RMI classico prevede che un oggetto, per rendersi visibile dall'esterno della JVM sulla quale è istanziato, debba registrarsi presso di essa eseguendo un'operazione chiamata "esportazione".

Da questo momento in poi la JVM conosce i metodi esportati verso l'esterno e sa quando sostituire un oggetto vero e proprio con il suo proxy.

Il modo più semplice per utilizzare RMI è dichiarare una classe che estenda `UnicastRemoteObject` ed implementi l'interfaccia `Remote` (frammento di codice 1 nella pagina successiva). In questo caso, la maggior parte dei compiti legati all'esportazione sono svolti dal costruttore della classe `UnicastRemoteObject`; l'unica attività richiesta all'utente è la definizione di un costruttore senza parametri. Durante l'operazione di esportazione il runtime Java crea il proxy corrispondente alla classe da esportare. Affinchè il runtime

Frammento di Codice 1 RMI tramite UnicastRemoteObject

```
public class EsmpioRmi extends UnicastRemoteObject implements
Remote
{

   public static void main(String[] args) throws Exception
   {
      // questa istanziazione esporta il proxy nel java runtime
      // e fa partire un thread per mantenerlo in esecuzione
      new EsempioRmi();
   }

   // Il costruttore senza parametri è necessario al runtime
   // per generare il proxy
   public EsempioRmi() throws java.rmi.RemoteException
   {
   }

}
```

possa istanziare l'oggetto proxy, è necessario che sia disponibile la classe che ne definisce la struttura. Per generare questa classe è necessario, dopo la normale compilazione della classe, utilizzare l'RMI compiler (`rmic`).

Utilizzando il Jini Extensible Remote Invocation, non è più necessaria la generazione delle classi proxy a tempo di compilazione, il Jeri runtime si occupa della loro creazione a tempo di esecuzione.

Il frammento di codice 2 nella pagina seguente mostra l'esempio di una classe che esporta il proprio proxy utilizzando JERI. In questo caso l'operazione di esportazione deve essere fatta esplicitamente: si istanzia un oggetto della classe `BasicJeriExporter` sul quale invocare il metodo export passandogli per parametro l'oggetto da esportare.

Configurazione L'altro grosso vantaggio di JERI rispetto all'RMI tradizionale è la possibliità di scegliere a run-time il protocollo del livello di trasporto da utilizzare. Solitamente il protocollo di trasporto utilizzato da un'applicazione dipende dalle classi di comunicazione utilizzate durante la codifica, al contrario, JERI, tramite un sistema di indirezioni, offre la possibilità di cambiare il sistema di trasporto senza che sia necessario mettere mano al codice. JERI legge le informazioni necessarie da un file di configurazione.

Il frammento di codice 3 nella pagina successiva mostra un possibile file di configurazione.

Come abbiamo precedentemente illustrato JERI consente la creazione di classi proxy a runtime, senza che sia necessario ricorrere all'RMI compiler. Il

Frammento di Codice 2 RMI tramite Export

```
public class EsempioExport implements Remote
{

   public static void main(String[] args) throws Exception
   {
      // creazione di un exporter JERI
      Exporter exp =
         newBasicJeriExporter(TcpServerEndpoint.getInstance(0)
                              new BasicILFactory());

      // esportazione di un oggetto di questa classe
      Remote proxy = exp.export(new EsempioExport());
   }

}
```

Frammento di Codice 3 File di Configurazione dell'Exporter

```
import net.jini.jrmp.*;
import net.jini.iiop.*;

JrmpEsempioExport {
   exporter = new JrmpExporter();
}
IiopEsempioExport {
   exporter = new IiopExporter();
}
```

Frammento di Codice 4 File di Configurazione dell'Exporter Jeri

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;

JeriEsempioExport {
   exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                    new BasicILFactory());
}
```

frammento di codice 4 mostra un file di configurazione che sfrutta la tale possibilità: anzichè utilizzare l'exporter di uno specifico protocollo utilizza l'esportatore standard di JERI, il `BasicJeriExporter`.

I passi da eseguire a programma per poter leggere ed utilizzare le informazioni contenute nei file di configurazione sono pochi e semplici:

1. invocare il metodo statico `getInstance` della classe `ConfigurationProvider` per leggere il file di configurazione;

2. utilizzare le informazioni ottenute al punto precedente per creare un exporter;

3. esportare l'oggetto utilizzando l'exporter creato al punto 2;

Il frammento di codice 5 nella pagina successiva mostra come sono lette le informazioni di configurazione dal file rappresentato nel frammento di codice 4.

2.3 Jini: Componenti

All'interno di una federazione possiamo individuare tre tipi di attori:

Frammento di Codice 5 Programma che carica a runtime la configurazione Jeri

```
import java.rmi.*;
import net.jini.config.*;
import net.jini.export.*;

public class EsempioConfigurazioneExport implements Remote
{
   // In questo esempio il nome del file di configurazione è
   // scritto direttamente nel codice, in realtà è possibile
   // utilizzare un altro livello di indirezione per indicare
   // il percorso dove trovare il file di configurazione
   private static String CONFIG_FILE = "jeri/jeri.config";

   public static void main(String[] args) throws Exception
   {
      String[] configArgs = new String[] {CONFIG_FILE};

      // leggo la configurazione dal file "jeri/jeri.config"
      Configuration config =
         ConfigurationProvider.getInstance(configArgs);

      // uso le informazioni lette per creare un exporter
      Exporter exp =
         (Exporter) config.getEntry("JeriEsempioExport",
                                    "exporter",
                                    Exporter.class);

      // esporto un oggetto di tipo EsempioConfigurazioneExport
      Remote proxy =
         exp.export(new EsempioConfigurazioneExport());

      // rimuovo l'esportazione quando ho finito
      exp.unexport(true);
   }

}
```

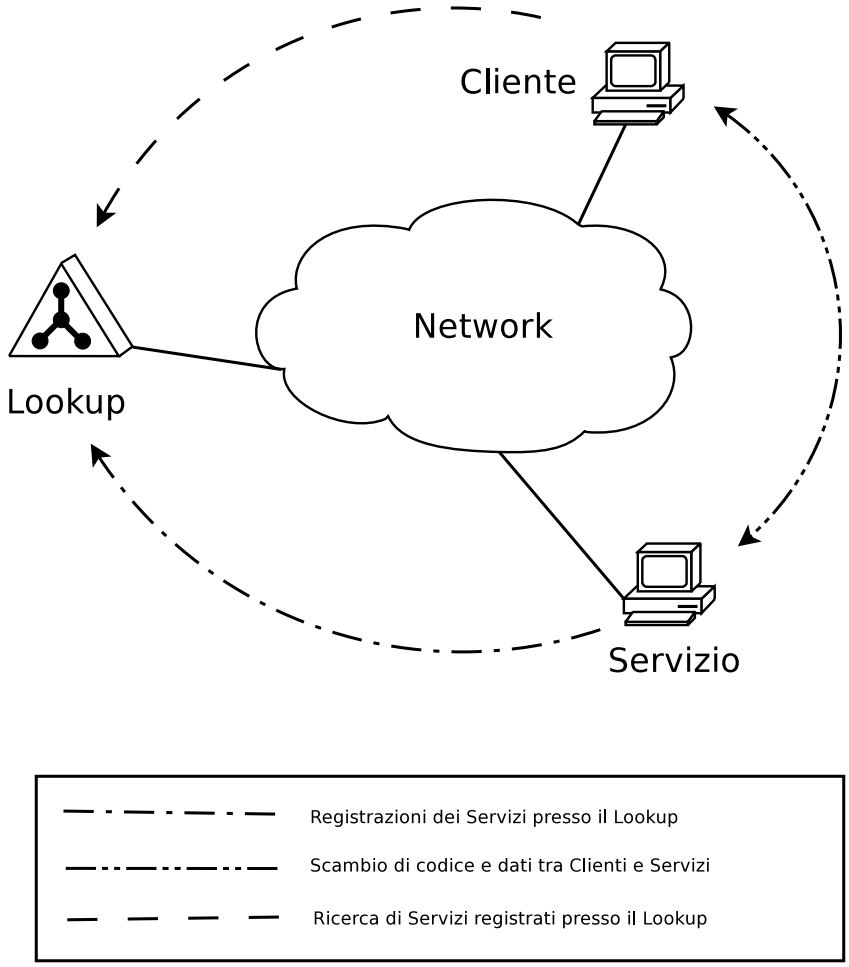


Figura 2.2: Struttura semplificata di Jini

1. I Servizi: una stampante, un diskarray, un computer o più in generale qualsiasi entità che fornisca funzionalità;

2. I Clienti: gli utilizzatori dei servizi

3. Il Servizio di Lookup (in seguito *LookupService*) : memorizza le funzionalità dei servizi, e le comunica ai clienti; il suo ruolo è quello di intermediario.

Codice e dati vengono trasferiti tra i tre attori tramite la rete di interconnessione (generalmente TCP/IP), per spostare gli oggetti tra una Java Virtual Machine ed un'altra si utilizzano i meccanismi di serializzazione messi a disposizione da Java.

Gli elementi trasferiti possono essere oggetti veri e propri oppure proxy (RMI e non) che a loro volta eseguono invocazioni di metodi remoti.

2.3.1 Il Lookup Service

Il *lookup service[JNJ03, JOT99]* è il gestore centralizzato dei servizi disponibili all'interno della federazione, ne memorizza le caratteristiche e li rende visibili agli altri membri. Per ogni servizio registrato, il *LookupService*, mantiene una propria rappresentazione: il *service item*.

Ogni *service item* è composto da tre oggetti:

- `serviceID`: l'identificatore del servizio (rappresentato su 128 bit);
- `service`: un'istanza del servizio (molto spesso un proxy);
- `attributeSets`: un insieme di attributi che descrivono le caratteristiche del servizio registrato.

Un cliente che vuole cercare un certo servizio, fa una richiesta al *LookupService*, indicando nel messaggio le funzionalità di cui necessita. Il *LookupService* esamina le informazioni contenute nella richiesta del cliente e le confronta con quelle memorizzate, se trova dei *service item* compatibili, li spedisce al cliente.

2.3.2 I Servizi

Con il termine Servizio indichiamo un'entità che mette a disposizione dei clienti un insieme di funzionalità. In Jini un servizio è rappresentato da un'interfaccia Java.

All'interno di una federazione è possibile trovare, per ogni servizio, più implementazioni distinte. Affinchè ognuna di esse possa essere trovata ed utilizzata dai clienti, è necessario eseguire preventivamente un operazione di "registrazione". Tale operazione, portata a termine da un'entità chiamata "*service provider*", si occupa di registrare un'implementazione di un servizio presso un *LookupService*.

Il *service provider* è l'entità che si occupa di:

1. istanziare l'implementazione di un servizio;
2. registrare l'oggetto creato in un *LookupService*;
3. restare attivo eseguendo alcuni compiti di gestione;

Un *service provider*, per poter registrare un servizio, deve prima di tutto localizzare e contattare almeno un *LookupService* (Figura 2.3 nella pagina successiva), per farlo esistono due modi diversi:

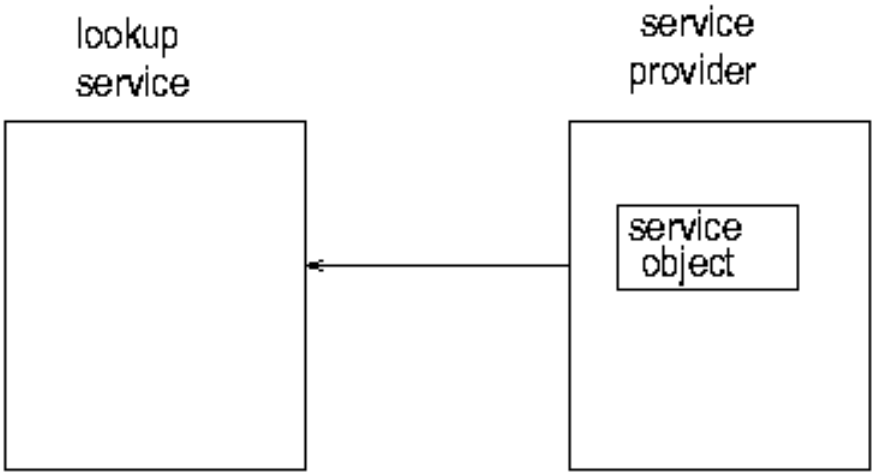


Figura 2.3: Contattato Registrar

- stabilire una connessione TCP direttamente col *LookupService*;
- eseguire una richiesta UDP su alcuni canali di Multicast;

Nel primo caso è necessario conoscere a priori l'indirizzo IP della macchina che ospita il *LookupService*, nell'altro caso no.

Quando un *LookupService* riceve una richiesta, spedisce all'host mittente un oggetto chiamato *registrar* (Fig. 2.4 nella pagina seguente), quest'ultimo funge da proxy tra il *service provider* e il *LookupService* (Fig. 2.5 nella pagina successiva).

2.3.3 I Clienti

Il cliente è l'utilizzatore delle funzionalità che uno o più servizi mettono a disposizione.

Le operazioni che un cliente deve portare a termine per cercare i servizi nei *LookupService*, sono simili a quelle compiute da un *service provider* per la registrazione di un'istanza di servizio.

La fase di ricerca del *LookupService* e quella di ricezione del *registrar* sono praticamente le stesse. Le differenze si basano sui diversi ruoli assolti dalle due parti, infatti, mente i *service provider* si preoccupano di mettere a disposizione nuove funzionalità, l'obiettivo del cliente è localizzare e utilizzare le risorse disponibili.

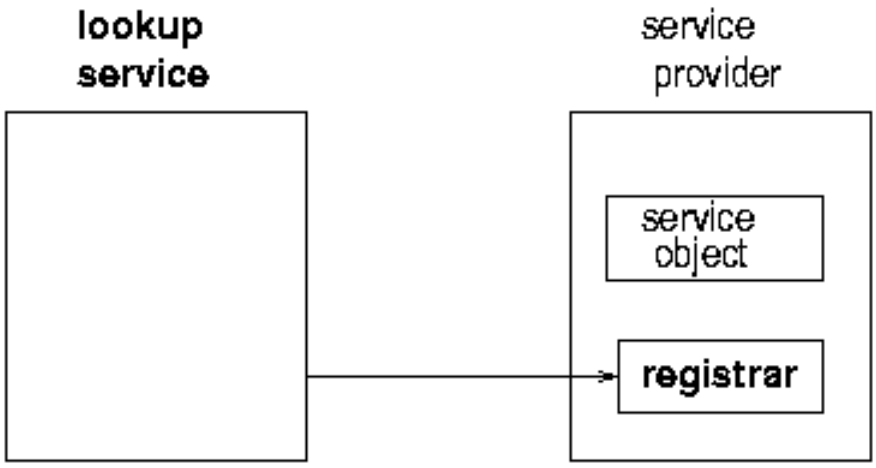


Figura 2.4: Ottenuto Registrar

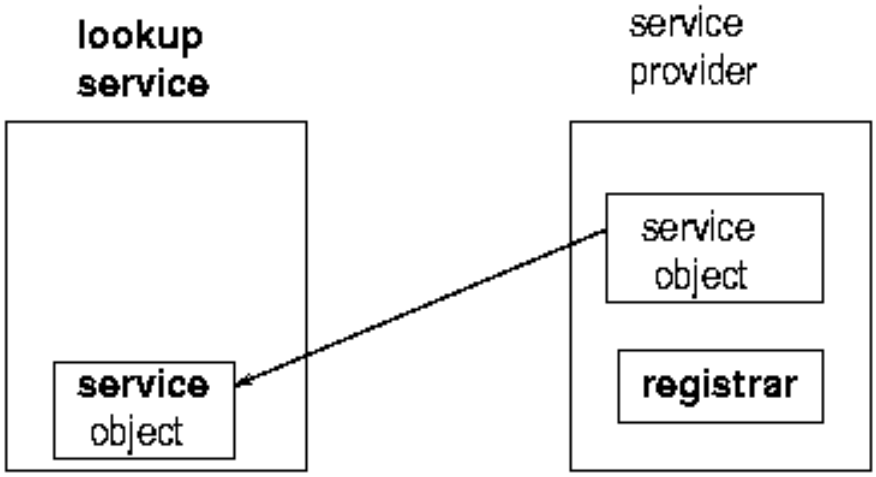


Figura 2.5: Registrato Servizio

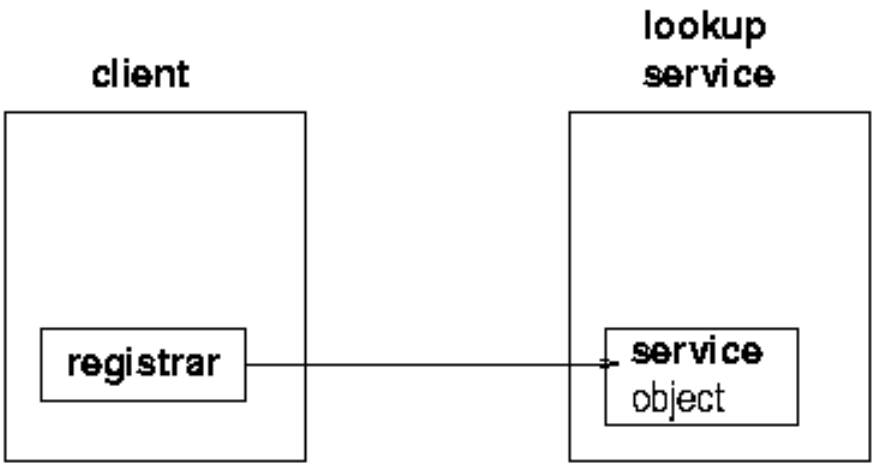


Figura 2.6: Il cliente richiede

Come è possibile vedere in Figura 2.6, il cliente chiede al *LookupService* i service item che hanno caratteristiche compatibili con quelle di cui necessita, se il *LookupService* riesce a trovarne, li serializza e li trasferisce attraverso la rete fino alla JVM del cliente (Fig. 2.7 nella pagina successiva). Da questo momento in poi il cliente può utilizzare il servizio.

2.3.4 I Proxy

Gli oggetti trasferiti dal *LookupService* al cliente, possono essere i servizi veri e propri (Es. algoritmi particolarmente efficienti e/o precisi per particolari calcoli matematici) o come più spesso accade, proxy che fungono da intermediari tra il cliente ed il servizio.

Vediamo alcuni esempi in cui l'oggetto trasferito è un proxy:

- un servizio che controlla e tiene traccia del traffico di rete che passa su un router;
- un servizio che controlla lo stato di una stampante;
- un servizio che consente l'esecuzione di codice su di una macchina remota;

In tutti questi casi gli oggetti in esecuzione per ogni istanza di servizio sono due:

- un proxy sul computer in cui è in esecuzione il cliente;
- il servizio vero e proprio in esecuzione sulla macchina in cui è ospitato il servizio (che chiameremo anche *service backend* [JNJ03]).

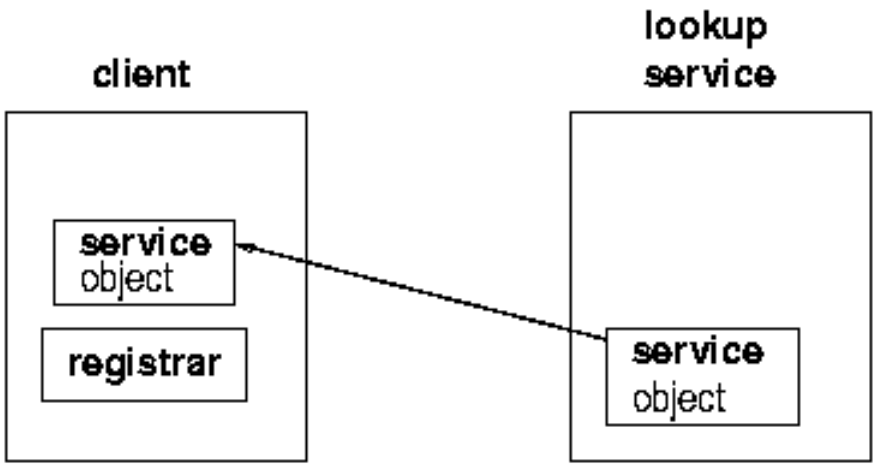


Figura 2.7: Il servizio viene trasferito nella JVM del cliente

Grazie all'utilizzo del proxy, non è necessario che il cliente sia a conoscenza dell'indirizzo di rete del service backend. Per utilizzare le funzionalità, che quest'ultimo mette a disposizione, è sufficiente che conosca l'interfaccia di programmazione implementata dall'istanza di servizio. Quindi, è chiaro che per consentire il corretto funzionamento del meccanismo di invocazione remota, il proxy deve conoscere l'indirizzo di rete della macchina che ospita il servizio, tale informazione, viene inserita nel proxy al momento della sua creazione.

2.3.5 Servizi di supporto

Le parti in gioco in un ambiente Jini sono tre: i Clienti, i Servizi e i *LookupService*. Poichè ognuna di esse è generalmente in esecuzione su una macchina diversa, per permettere lo scambio di codice e di oggetti viene utilizzata la serializzazione di Java. Per il trasferimento di oggetti da una JVM ad un'altra è necessario che sia il mittente che il destinatario conoscano la struttura della classe dell'oggetto serializzato. In Java, questo tipo di informazioni è contenuto nei classfile, per la condivisione dei quali, Jini utilizza il protocollo HTTP.

Server HTTP Ogni oggetto Java è formato da codice e dati, durante la serializzazione vengono trasferiti solo questi ultimi, quindi per ricostruire l'oggetto, il ricevente, deve conoscere le metainformazioni sulla struttura della classe. Tuttavia, se fosse necessario, al cliente, conoscere a priori la definizione di tutte le classi che è possibile trovare, Jini perderebbe quasi tutta la sua flessibilità poichè

a tal punto, non sarebbe possibile aggiungere dinamicamente nuovi servizi e dispositivi alla rete.

La soluzione che adotta Jini è scaricare le definizioni delle classi direttamente dalla rete, in particolare dai dispositivi appena connessi o piu in generale da un server HTTP o FTP il cui indirizzo è specificato nella property `java.rmi.server.codebase` del *service provider*. Il cliente si collega a quell'indirizzo per ottenere il bytecode necessario alla deserializzazione dell'oggetto proxy ricevuto.

2.3.6 Ulteriori strumenti di supporto

ClassServer È un semplice server HTTP che può essere facilmente integrato all'interno di un'applicazione Jini. E' in grado di soddisfare richieste di file JAR e `class`.

ClassDep `ClassDep` è un'applicazione Jini che prende in input un insieme di classi e partendo da esse crea ricorsivamente un grafo di dipendenze.

Il grafo di dipendenze è restituito come un vettore di nomi di classi da cui quelle in input dipendono. In questa tesi `ClassDep` viene utilizzato per determinare quali classi il cliente deve spedire al servizio per consentire l'esecuzione remota del codice.

Capitolo 3

JJPF: Progetto logico

3.1 Obiettivi

Gli obiettivi che ci siamo posti per la realizzazione di JJPF possono essere riassunti da queste quattro parole: **semplicità, affidabilità, scalabilità** e **dinamicità**.

Nonostante si tratti di termini che, molto probabilmente, il lettore già conosce, si è scelto, per completezza, di riproporre brevemente il significato degli ultimi tre.

3.1.1 Scalabilità

La scalabilità rappresenta la predisposizione di una architettura ad essere espansa, garantendo un certo aumento di banda di elaborazione [FB87].

Un'architettura ideale, da questo punto di vista, è quella in cui la banda varia proporzionalmente con il suo grado di parallelismo.

Nei casi reali tale obiettivo può essere raggiunto solo approssimativamente. JJPF ha dimostrato, nei test effettuati, di riuscire a garantire buoni livelli di scalabilità.

3.1.2 Affidabilità

Una gestione ottimale dell'affidabilità è un requisito essenziale per un ambiente di calcolo parallelo e distribuito.

Per comprendere perchè l'affidabilità sia così importante, è necessario precisare che JJPF è stato progettato per consentire la realizzazione di calcolo parallelo anche quando si utilizzino risorse computazionali il cui numero è fortemente variabile nel tempo, ovvero nelle situazioni in cui si utilizzino calcolatori che potrebbero essere spenti o riavviati da un istante all'altro.

Un framework studiato e realizzato per esser utilizzato in tali circostanze deve rispettare requisiti di robustezza e tolleranza ai guasti, in altre parole deve riuscire a portare a termine una computazione anche quando buona parte delle risorse iniziali non siano più disponibili.

L'analisi delle prove effettuate ha dimostrato che JJPF garantisce buoni livelli di affidabilità.

3.1.3 Dinamicità

Con questo termine indichiamo la capacità di reazione di un sistema all'aggiunta e alla rimozione di risorse computazionali [APJ02].

Quando nuove macchine dichiarano la loro disponibilità ad essere utilizzate JJPF reagisce nel seguente modo:

1. attiva i meccanismi che consentano l'arruolamento delle nuove risorse di calcolo;
2. predispone le adeguate strutture dati necessarie a:
   - garantire l'affidabilità;
   - consentire un appropriato ribilanciamento del carico.

Nel progetto di JJPF si è tenuto conto di queste necessità e sono stati predisposti meccanismi che gli consentano di reagire rapidamente all'aggiunta e alla rimozione di risorse di calcolo.

3.2 Descrizione

Passiamo ora a descrivere il progetto logico di JJPF, soffermandoci sugli scenari chiave, sui principali attori e sulle possibilità di utilizzo offerte.

3.2.1 Gli scenari chiave

JJPF permette di realizzare, su cluster di workstation[1], applicazioni parallele che implementino uno fra i due seguenti paradigmi di programmazione parallela: il farm stateless (§ 3.2.1.1 a fronte) e il farm con stato (§ 3.2.1.2 nella pagina successiva).

Secondo un paradigma task farm[Pel98], una serie di task in ingresso $(x_1 ... x_n)$ viene trasformata in una serie di risultati $(f(x_1) ... f(x_n))$ calcolando in parallelo fra di loro i task $(x_i ... x_{i+nw})$.

[1] Maggiori informazioni sulla struttura dei cluster di workstation e sulle strategie di scheduling utilizzabili con essi possono essere trovate in: [Dan01, NOW, FNO]

Analizzando la struttura logica di un farm stateless è possibile identificare tre entità distinte: **emitter**, **worker** e **collector**[2]. Nel farm con stato questo numero sale a quattro, infatti, alle tre entità precedenti si aggiunge un ulteriore elemento, condivisa fra i **worker**, in questa tesi ci riferiremo ad esso anche chiamandolo **Shared Object** (§ 3.2.2.3 a pagina 43).

3.2.1.1 Farm privo di stato

Questo tipo di farm è probabilmente il più comune fra i vari approcci paralleli, essenzialmente vi si possono individuare tre ruoli: **emitter**, **worker** e **collector**.

La Figura 3.1 nella pagina successiva schematizza la struttura e il comportamento di un generico farm privo di stato: l'emitter si occupa di distribuire i task da elaborare fra gli N worker a sua disposizione, questi ultimi eseguono il calcolo che gli è stato affidato e comunicano il risultato ottenuto al collector.

Le diverse modalità di distribuzione dei task ai worker sono raggruppabili in due categorie: scheduling statico e scheduling dinamico [Dan03]. L'esempio tipico di stategia di scheduling statica è il round-robin: l'emitter richiede ciclicamente a tutti i worker di calcolare un task. In altre parole, il primo task viene inviato per il calcolo al primo worker, il secondo al secondo, ... l'ennesimo al'ultimo worker, l'enne piu' unesimo al primo worker, e così via. La strategia dinamica per antonomasia è il self-scheduling: il worker che ha terminato l'elaborazione di un task manda il risultato all'emitter. Quest'ultimo provvede, di conseguenza, all'invio di un nuovo task da calcolare.

In JJPF i ruoli di **emitter** e **collector** sono assolti da un'unica entità, il *cliente* (§ 3.2.2.2 a pagina 41).

I **worker** mostrati in Figura 3.1 nella pagina successiva rappresentano il cuore pulsante del farm, ovvero le risorse computazionali sulle quali il collector distribuisce il calcolo, la componente JJPF che ricopre il ruolo di worker è il *Servizio*.

3.2.1.2 Farm con Stato

Nel paradigma farm privo di stato: i task giungono all'emitter e questi li distribuisce fra i worker a sua disposzione. A questo punto, i worker, indipendentemente l'uno dall'altro, eseguono il calcolo commissionato loro e, terminata la computazione, spediscono il risultato al collector.

Da quanto detto, appare chiaro che in un farm stateless ogni worker comunica soltanto con due entità: emitter e collector; in altre parole, per un worker, non è prevista nessuna sorgente di dati diversa dall'emitter e nessuna destinazione

[2] per una trattazione completa dei termini emitter, worker e collector si rimanda a libri specializzati fra i quali [FB87] e [Pel98].

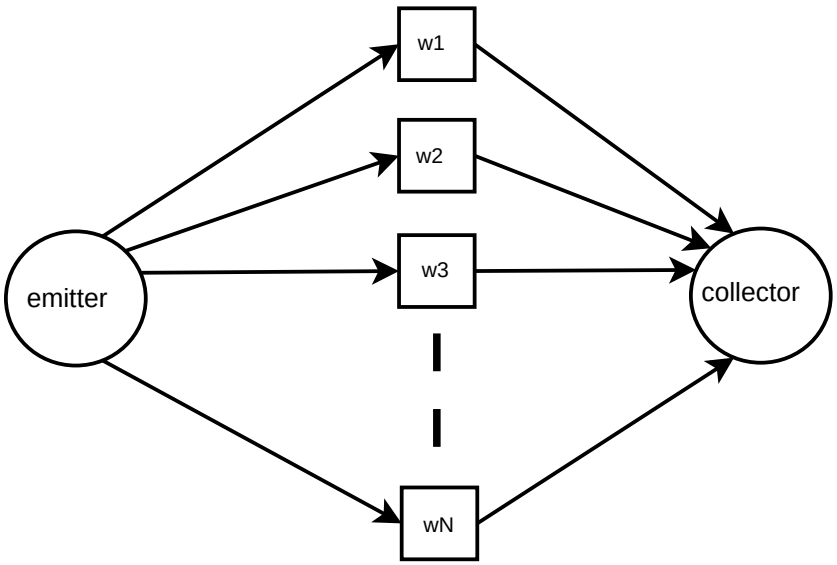


Figura 3.1: Schematizzazione Farm

dei dati diversa dal collector. Il farm con stato è una variante del farm stateless che prevede l'esistenza di un'entità condivisa fra i worker.

Interagendo con l'entità condivisa, i worker, possono scabiarsi dati e sincronizzarsi.

La Figura 3.2 a fronte mostra lo schema logico di funzionamento di un farm con stato: sulla sinistra vediamo un emitter/collector che dirige la computazione dei worker (w1...w6), sulla destra vediamo questi ultimi interagire con l'oggetto condiviso.

JJPF lascia molta libertà al programmatore che strutturi la sua applicazione parallela secondo lo schema del farm con stato. Il framework realizzato, infatti, delega all'utilizzatore la responsabilità di gestire correttamente le informazioni ricevute (o spedite) dall'entità condivisa.

3.2.2 Gli Attori

Come già accennato in precedenza, nel framework JJPF, i ruoli di **emitter** e **collector** sono assolti da un'unica entità, il *cliente*; i "Servizi" (§ 3.2.2.1 nella pagina successiva), invece, svolgono le attività che fanno capo ai worker. Solitamente in una applicazione Client-Server i ruoli sono rovesciati, infatti, generalmente i Server rivestono un ruolo di "controllori" e i Client di "controllati". Nel nostro caso la struttura rovesciata consente di utilizzare al meglio il supporto offerto da Jini. Infatti, in un sistema orientato ai servizi, al fine di ottenere un sistema

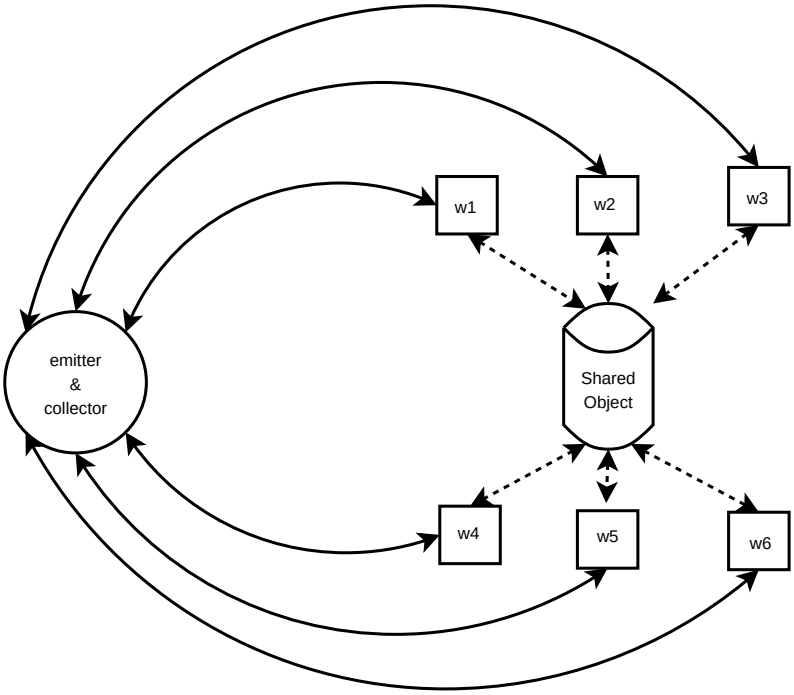


Figura 3.2: Schematizzazione Farm con Stato

articolato, il compito di comporre le risorse disponibili spetta all'utilizzatore dei servizi.

3.2.2.1 I Servizi

Una delle attività principali svolte dal framework realizzato, è l'arruolamento delle risorse computazionali disponibili. Nel contesto di questa tesi, ci riferiremo a tali risorse come a "Servizi".

Nel paragrafo 4.2.2 a pagina 56 analizzeremo dettagliatamente la loro struttura, nel Manuale Utente descriveremo il loro funzionamento ed uso (Appendice A.4 a pagina 103), qui di seguito ci accontenteremo di accennare alla loro struttura logica e funzionamento.

La Figura 3.3 nella pagina seguente mostra il diagramma di flusso di un generico *servizio* che implementi il paradigma farm stateless, dove è possibile individuare: due stati ("In attesa di utilizzo" e "In Uso"), tre azioni (Collegamento, Registrazione e DeRegistrazione da un *LookupService*) e due entità esterne con cui collaborare (*cliente* e *LookupService*).

Il comportamento di questo tipo di *servizio* può essere schematizzato nel seguente modo:

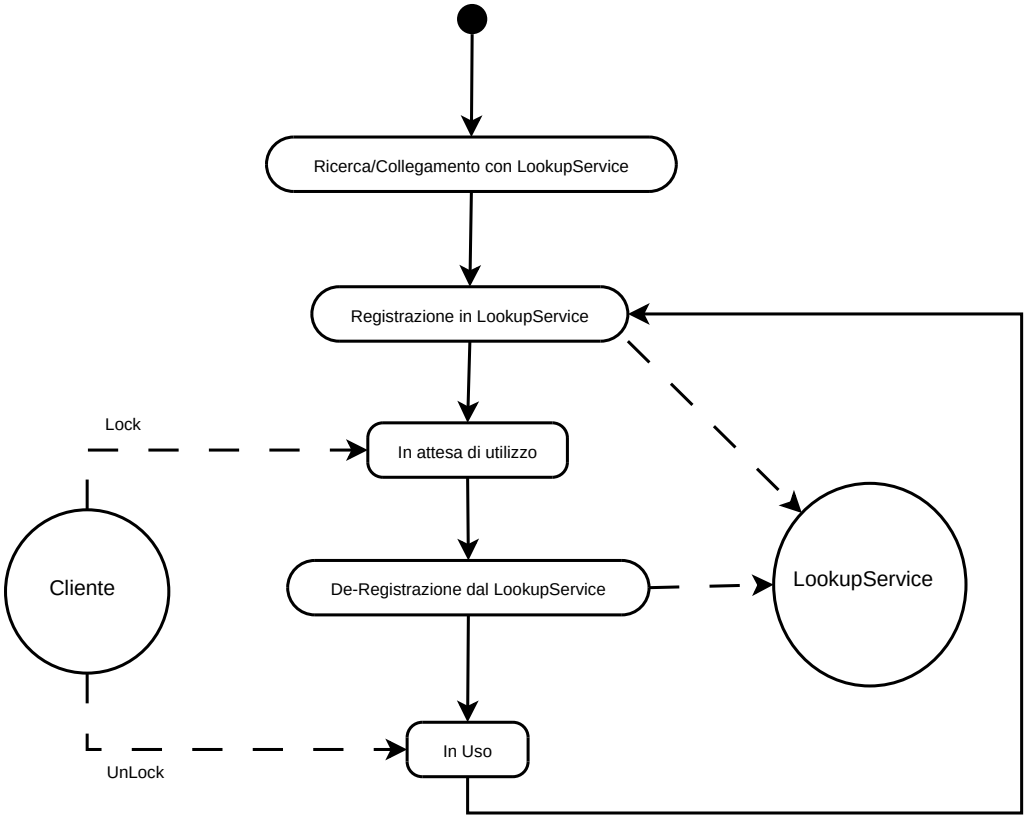


Figura 3.3: Diagramma semplificato degli stati di un generico *servizio*

- appena viene creato, un *servizio*, cerca di registrarsi presso un *LookupService* così da permettere la sua localizzazione;
- una volta registrato, resta in attesa, aspettando che qualche *cliente* ne richieda l'uso;
- ricevuta una richiesta, si rende irreperibile cancellando dal *LookupService* l'oggetto che lo rappresenta e si mette a disposizione del *cliente* fino a quando quest'ultimo non ne rilasci l'uso.
- quando il *cliente* lo "libera, il *Servizio*, torna nello stato di attesa.

La Figura 3.4 nella pagina successiva mostra il diagramma di flusso che rappresenta un *servizio* realizzato seguendo la struttura logica del farm con stato.

Ciò che differenzia quest'ultimo diagramma dal precedente è la presenza di un'ulteriore azione: la ricerca dell'entità condivisa, usata per permettere ai servizi di sincronizzarsi e scambiarsi informazioni. Nel paragrafo 4.2.3 a pagina 60 è illustrato il meccanismo utilizzabile dai Servizi per interagire con l'entità condivisa

3.2.2.2 Il Cliente

L'utilizzatore dei Servizi è il *cliente*.

Come già accennato in precedenza, il *cliente* è l'entità che svolge quelle attività che in un paradigma parallelo di tipo farm sono eseguite da emitter e collector. Si occupa di:

- cercare le risorse computazionali libere;
- riservarsene l'uso esclusivo;
- caricare il codice su di esse;
- spedirvi e riceverne dati;
- rilasciarne l'uso al termine della computazione;

La Figura 3.5 a pagina 44 illustra lo schema semplificato del comportamento di un *cliente* multi-threaded [SO99]; in questo caso non si prevede che ulteriori risorse computazionali possano essere trovate a computazione iniziata. Questo tipo di limitazione è presente in molti framework per la programmazione parallela, ad esempio Lithium [ADT03, DT02].

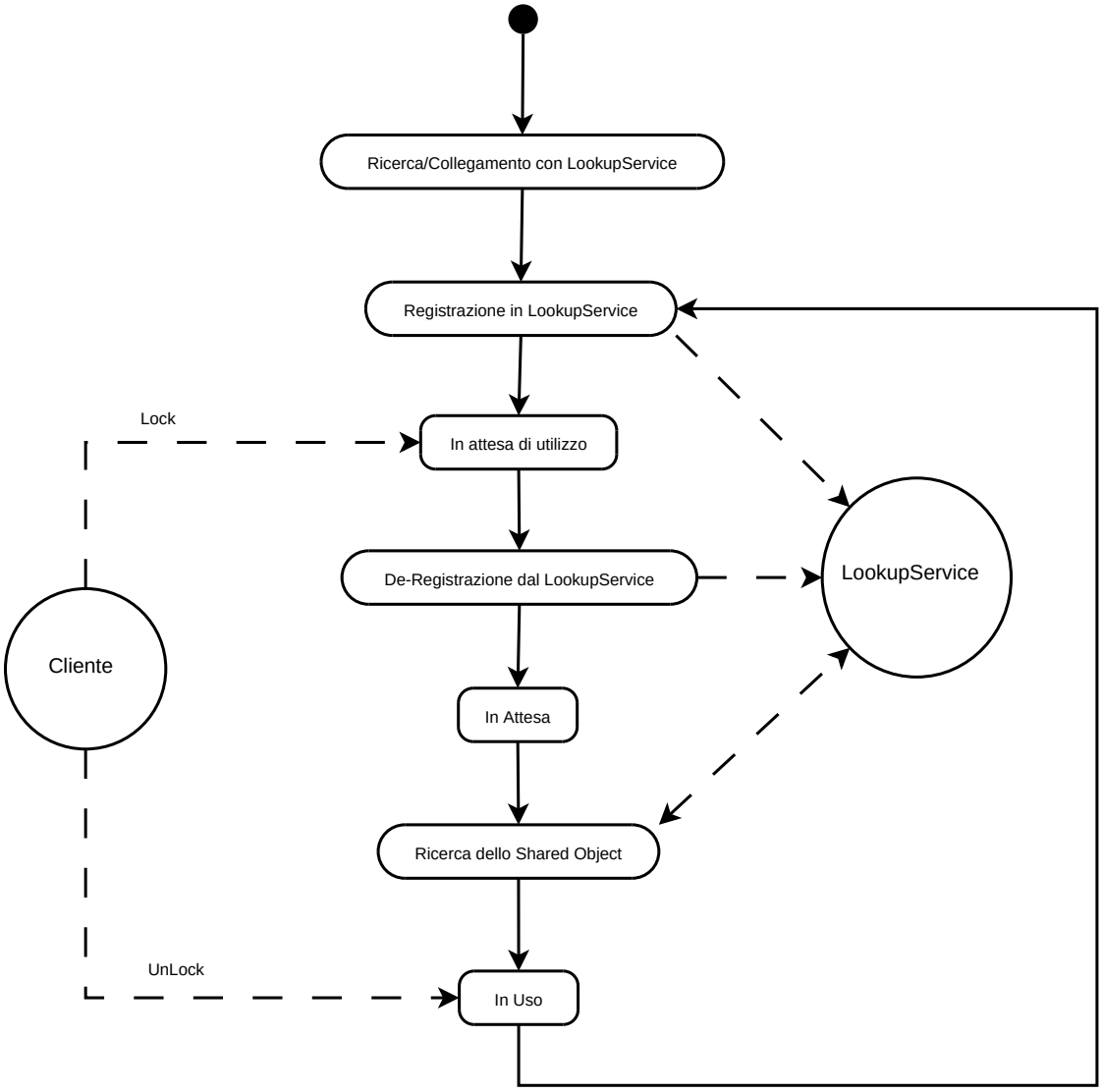


Figura 3.4: Diagramma semplificato di un generico *servizio* con Shared Object

In questo schema semplificato il *cliente* cerca di contattare un *LookupService* al quale richiedere servizi che corrispondano al template specificato dall'utilizzatore, se non esistono risorse computazionali disponibili il *cliente* rinuncia ad eseguire la computazione parallela e termina.

Se trova un *servizio* disponibile, il *cliente*, prova a contattarlo e a richiederne l'uso esclusivo; se l'operazione va a buon fine, carica il codice da eseguire sul *servizio* e comincia a spedirgli i dati da calcolare.

Terminata la computazione, il *cliente*, rilascia l'uso delle risorse computazionali arruolate e termina.

Nella Figura 3.6 a pagina 45 vediamo un secondo schema di funzionamento del *cliente*, qui si tiene conto della possibile scoperta di risorse computazionali anche a calcolo cominciato: il *cliente* chiede al *LookupService*, di essere avvisato qualora nuovi servizi conformi al template indicato comunichino la loro disponibilità ad essere arruolati.

Se nuovi calcolatori si registrano come disponibili per il calcolo,
il *LookupService* avvisa il *cliente*. Quest'ultimo, se ancora non ha terminato la computazione, li arruola e li usa creando un nuovo thread di gestione per ognuno di essi.

Come già accennato, JJPF permette la realizzazione di programmi paralleli sia secondo il paradigma farm stateless sia secondo il paradigma farm con stato. Per creare applicazioni parallele strutturate secondo quest'ultimo schema, il *cliente*, deve realizzare i meccanismi necessari alla ricerca dell'entità condivisa e occuparsi di comunicare ai worker l'identificatore Jini dell'entità trovata.

La Figura 3.7 a pagina 46 mostra lo schema del *cliente* quando sia richiesto l'uso dell'entità condivisa.

3.2.2.3 Lo Shared Object

Nei paragrafi precedenti, mentre illustravamo il funzionamento dei clienti e dei servizi, abbiamo più volte fatto riferimento all'esistenza di un'entità condivisa tra i servizi: lo Shared Object. Come già accennato, i worker possono utilizzare tale entità per scambiarsi i dati e sincronizzarsi.

Per consentire a JJPF di trovare l'entità software da utilizzare come Shared Object, è necessario che l'oggetto condiviso implementi l'interfaccia `SharedObjectIf`. I worker accedono all'entità condivisa utilizzando i metodi `set` e `get` dichiarati nell'interfaccia `SharedObjectIf`.

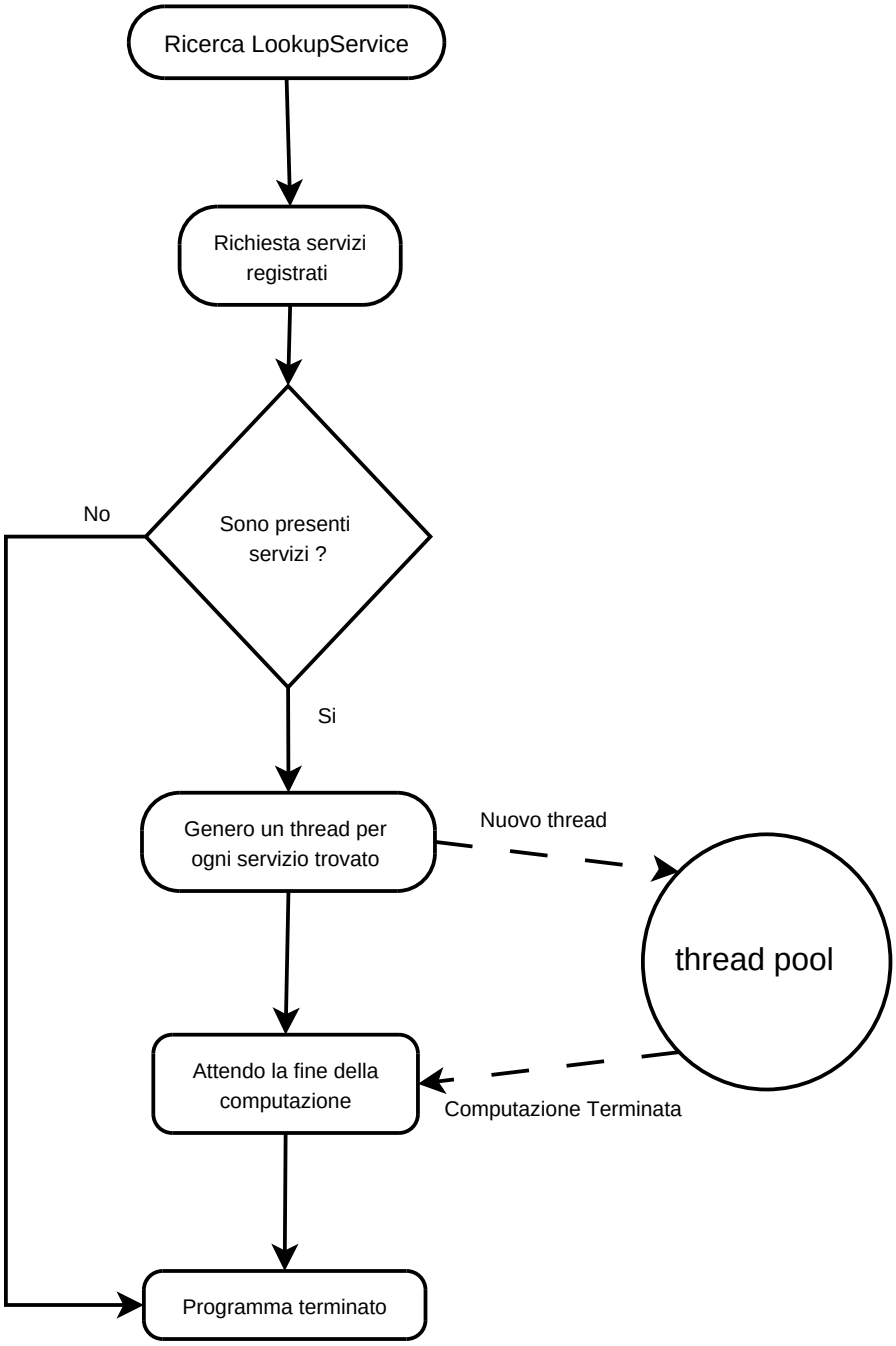


Figura 3.5: Schema semplificato del *cliente*

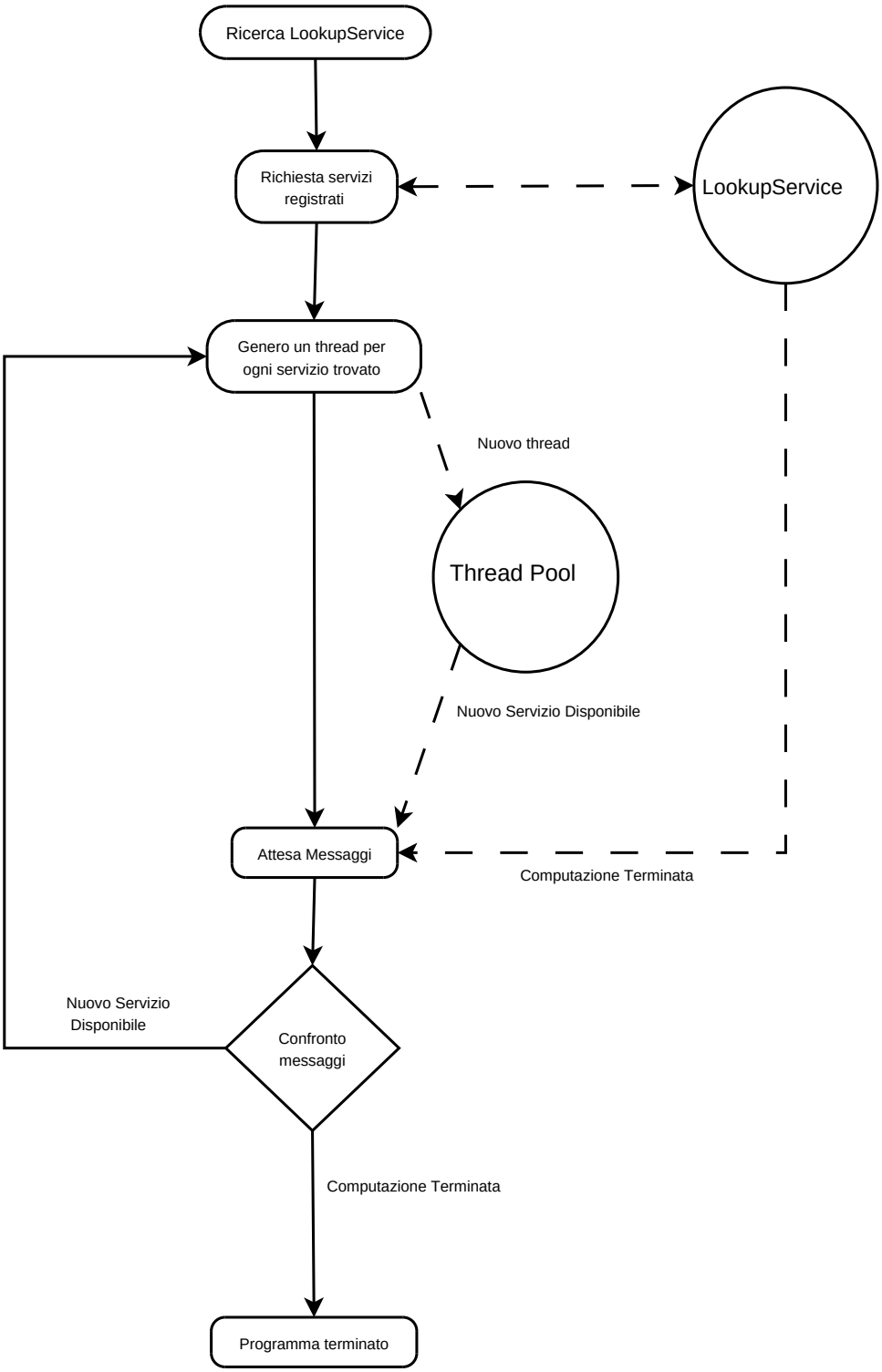


Figura 3.6: Schema del *cliente* senza SharedObject

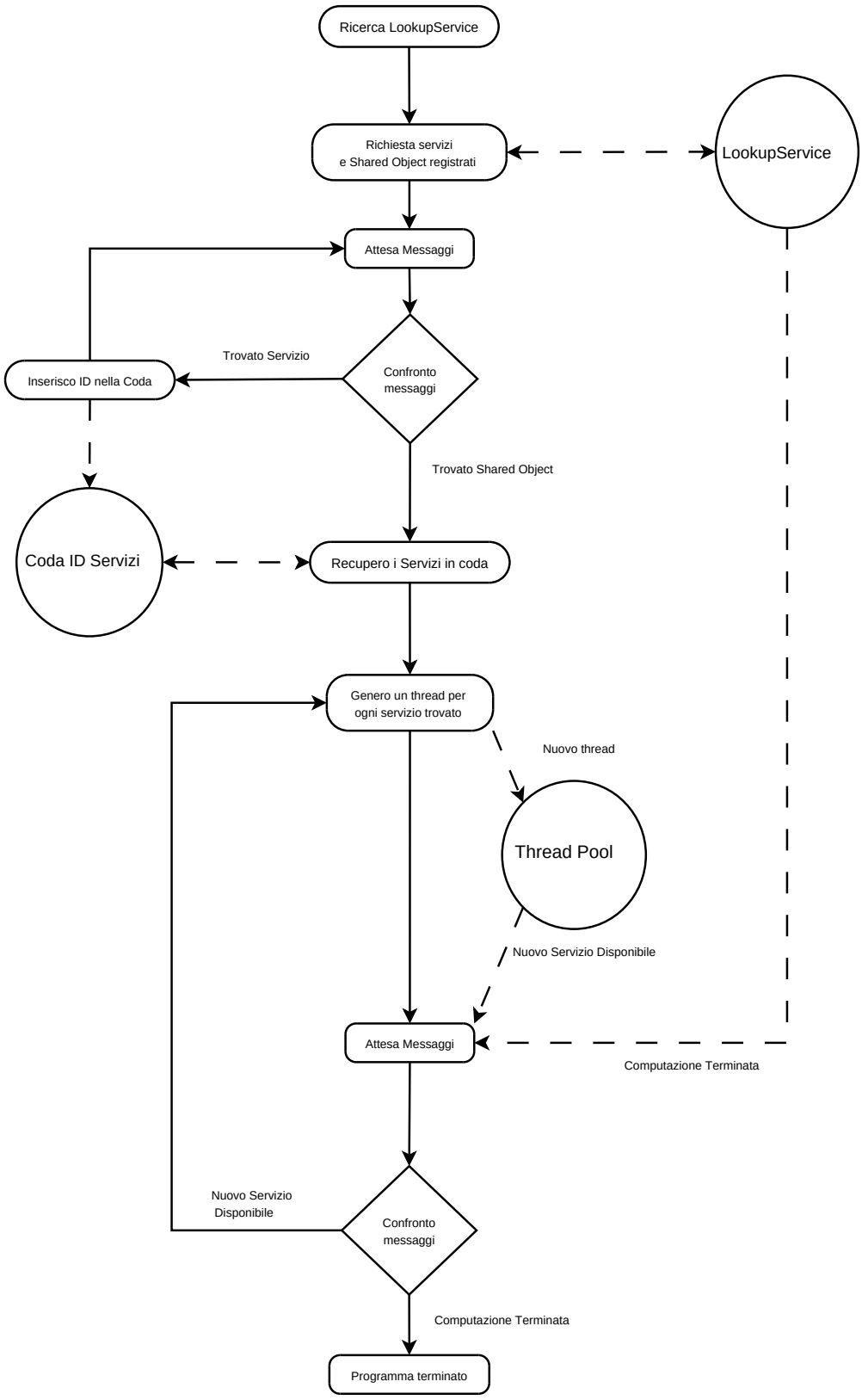


Figura 3.7: Schema del *cliente* con SharedObject

3.2.3 Le possibilità di utilizzo

Lo sviluppo di applicazioni distribuite richiede la presenza di un meccanismo che consenta al client di trasferire il codice da eseguire sui worker.

Solitamente questa operazione (staging) viene portata a termine trasferendo il codice eseguibile dalla macchina che ospita il *cliente* a quelle dove sono in esecuzione i worker. Il passo successivo è quello di mandare in esecuzione il codice trasferito (spawning).

JJPF mette a disposizione due diversi meccanismi per il caricamento dinamico delle classi: `rmi` e `hybrid`. Il primo utilizza direttamente le funzionalità messe a disposizione dall'estensione Jini di RMI: in questo caso è richiesta la presenza, sulla macchina *cliente*, di un server web che permetta il trasferimento dei file contenenti il bytecode delle classi da caricare.

Esistono casi in cui un *cliente* potrebbe non potere o comunque non volere avviare, sulla propria macchina, un webserver. Questo è il motivo principale che ci ha spinto a realizzare il meccanismo hybrid per il caricamento dinamico delle classi.

3.2.3.1 RMI puro

Solitamente le applicazioni Jini, per effettuare il caricamento dinamico distribuito delle classi, utilizzano un'estensione di RMI [JNJ03, Ven02].

Vediamo un esempio che illustra i passi salienti compiuti da questo meccanismo:

1. PreCondizioni: sul computer A esiste la definizione della classe Java `programma`, una applicazione in esecuzione su A vuol trasferire un'istanza di `programma`, l'oggetto `prog1`, sul computer B;

2. A apre un `ObjectOutputStream` verso B e serializza `prog1` nello stream, A "annota" lo stream con il nome della classe e con l'indirizzo del mittente[3] (più precisamente con il valore della property `java.rmi.server.codebase`);

3. l'applicazione sita su B prova a deserializzare l'oggetto ricevuto, chiedendo al classloader "primordiale" [Sch03, Bet98] di caricare in memoria le adeguate metainformazioni;

4. se la definizione non si trova ancora in memoria o se il classloader non riesce a caricarla dal filesystem locale, allora il classloader RMI prova a scaricarla utilizzando le informazioni contenute nelle annotazioni descritte nel punto 2;

[3] Se la definizione delle classi non si trova sul mittente, lo stream deve essere annotato con l'indirizzo del calcolatore da cui la macchina B può scaricare le metainformazioni necessarie.

5. B riesce a deserializzare correttamente `prog1`, e lo manda in esecuzione, durante la sua computazione ad ogni richiesta di classe presente su A ma non su B la procedura riprende dal passo 4.

Punti a Favore:

- vengono trasferite solo le classi strettamente necessarie al flusso di esecuzione attivo;

Punti a Sfavore:

- richiede un WebServer attivo sul *cliente*;

3.2.3.2 RMI "ibrido"

Hybrid è stato realizzato per fornire un'alternativa al sistema di caricamento dinamico illustrato nel paragrafo precedente [LB98]. L'idea alla base di questo meccanismo è quella di trasferire sul computer ospite tutte e sole le classi di cui la nostra applicazione mobile avrà bisogno una volta in esecuzione. Per poter identificare a priori quali siano queste classi è necessario analizzare il bytecode java cercando tutti i riferimenti alle classi da caricare.

Una volta che tali classi siano state identificate, è necessario trasferirle, assieme all'applicazione mobile, sul computer che ospiterà la computazione. Terminato il trasferimento, quest'ultimo provvederà a mandare in esecuzione il codice ricevuto.

Vediamo un esempio che illustri il funzionamento di RMI "ibrido":

1. PreCondizioni: sul computer A esiste la definizione della classe Java `programma`, una applicazione in esecuzione su A vuol trasferire un'istanza di `programma`, l'oggetto `prog1`, sul computer B;

2. A apre un `ObjectOutputStream` verso B e serializza nello stream `prog1`, `programma` e tutte le classi da cui l'oggetto serializzato dipende[4];

3. l'applicazione sita su B prova a deserializzare l'oggetto ricevuto, a questo punto interviene l'hybrid classloader: se il classloader primordiale fallisce il caricamento, la definizione della classe viene cercata nel vettore di classi trasferite da A su B;

[4] In realtà è sufficiente trasferire soltanto le classi che il classloader primordiale della macchina remota non può caricare.

4. B riesce a deserializzare correttamente `prog1`, e lo manda in esecuzione, durante la computazione ad ogni richiesta di classe si riparte dal punto 3.

Nel paragrafo 3.3.2 nella pagina seguente saranno descritte più dettagliatamente le procedure che abbiamo dovuto realizzare per rendere possibile questo tipo di caricamento.

Punti a Favore:

- non richiede nessun Server aggiuntivo destinato al trasferimento delle classi;

Punti a Sfavore:

- vengono trasferite tutte le classi che il codice potrà richiedere;

3.3 Problemi Affrontati

I problemi che abbiamo dovuto risolvere durante lo sviluppo del JJPF sono stati diversi, sia per numero che per natura. Di seguito riportiamo le principali difficoltà affrontate, descrivendole brevemente.

3.3.1 La Scarsità di Documentazione

3.3.1.1 Jini 2.0

Per lo sviluppo del JJPF abbiamo utilizzato la seconda versione del framework Jini, rilasciata da Sun nella seconda metà dello scorso anno. Quando abbiamo iniziato l'attività di documentazione non esisteva (e al momento della scrittura di questa tesi ancora non esiste) un manuale che illustrasse le possibilità offerte da questa nuova versione di Jini. Inoltre, la documentazione reperibile on-line si è dimostrata spesso frammentaria e lacunosa.

L'unica guida a Jini 2.0 ben organizzata, seppur incompleta, che siamo riusciti a trovare e consultare è stata la: "Jan Newmarch's Guide to JINI Technologies" [JNJ03]. Nonostante l'aiuto di questa preziosa guida, per una migliore comprensione di alcuni aspetti legati alla tecnologia Jini, ci è capitato più volte di consultare direttamente il codice sorgente del framework in oggetto.

3.3.1.2 Il Class loading dinamico distribuito

Quasi tutti i manuali Java dedicano una parte più o meno consistente alla trattazione del meccanismo per il caricamento dinamico delle classi. Tuttavia,

trovare un libro che affronti in modo esauriente la trattazione del sistema di caricamento dinamico **distribuito** offerto da RMI, è molto meno semplice. Inoltre, considerando l'ingente numero di modifiche apportate da Sun a RMI nel corso degli ultimi due anni e tenendo presente che Jini utilizza una versione modificata ed estesa di questo middleware (JERI), appare chiaro che la documentazione reperibile è ben lungi dall'essere organica ed esauriente.

3.3.2 Il Class Loading Dinamico Distribuito

Nel paragrafo 3.2.3.1 a pagina 47 abbiamo introdotto il meccanismo utilizzato da Java per il caricamento a runtime delle classi [Sch03], soffermandoci in particolare sul supporto offerto da RMI al caricamento dinamico **distribuito**. Adesso ci occuperemo dei problemi legati alla personalizzazione del meccanismo di classloading nel modello di invocazione remota utilizzato da Jini.

Il meccanismo RMI "ibrido" realizzato in questa tesi fornisce un'alternativa al sistema di caricamento e trasferimento delle classi proposto da RMI standard. Per farlo implementa un proprio classloader, l'`hybridclassloader`. Quest'ultimo riceve le richieste dal deserializzatore che si occupa di ricostruire il codice cliente e restituisce la definizione delle classi necessarie al riassemblaggio (o all'esecuzione dei metodi) degli oggetti provenienti dal *cliente*.

Da quanto detto, sembrerebbe che non esistano problemi di sorta per la personalizzazione del sistema di classloading. In realtà cercare di convincere un programma a rivolgersi, per il caricamento dinamico delle classi, ad un classloader diverso da quello con cui è stato caricato, è un'operazione tutt'altro che semplice.

Quando un programma Java deve deserializzare un oggetto, normalmente utilizza la classe `ObjectOutputStream`. Quest'ultima costruisce sullo stream in input un'astrazione che consente al ricevente di leggere oggetti anzichè byte. Un `ObjectOutputStream` legge il nome della classe associata all'oggetto da ricostruire e ne chiede, al classloader, la definizione. Nel caso di una applicazione Jini standard, il caricatore di classi a cui `ObjectOutputStream` si rivolge è l'`RMIClassLoader`, ovvero quello con cui lui il deserializzatore stesso è stato caricato. Quindi, il problema della sostituzione dell'`RMIClassLoader` con l'`hybridclassloader` è legato al fatto che Jini svolge autonomamente diversi compiti di gestione. In altre parole, se da un lato Jini semplifica lo sviluppo, dall'altro ne riduce la flessibilià.

Grazie alle funzionalità introdotte nella nuova versione di RMI e, più in particolare, per merito del nuovo Jini Extensible Remote Invocation (JERI), attualmente esiste una soluzione semplice ed elegante a questo cervellotico prob-

lema [JNJ03]. Infatti, nelle applicazioni Jini basate sull'uso di JERI, è possibile specificare al runtime di Java quale classloader utilizzare per la deserializzazione degli oggetti passati come parametro ai metodi invocabili da remoto.

Un esempio di configurazione JERI che prevede l'utilizzo di un classloader alternativo si trova nel file `jjpf.service.hybrid.config`.

3.3.3 Il Debugging Distribuito

Una fase fondamentale del ciclo di vita del software riguarda la prova del prodotto realizzato: il suo scopo è quello di individuare e correggere eventuali difetti (o bug) così da evitare possibili malfunzionamenti durante il normale utilizzo del software.

Al manifestarsi di un malfunzionamento si rende necessaria l'individuazione del difetto che lo ha provocato, tale attività, detta anche debugging, è spesso svolta con l'ausilio di un apposito software (debugger).

Per identificare un bug all'interno di un normale programma sequenziale si procede seguendone passo passo il flusso di esecuzione, cercando di capire quale sia la porzione di codice che lo genera.

Quando si ha a che fare con un programma concorrente o parallelo le cose si fanno molto più complesse e articolate, non è più sufficiente esaminare un singolo flusso di istruzioni ma è necessario tener conto dell'interazione esistente tra il software in oggetto e gli altri componenti, spesso siti su macchine remote.

Capitolo 4

JJPF: Implementazione

Nel primo paragrafo di questo capitolo illustriamo la stuttura generale del framework realizzato, nei successivi passiamo alla descrizione dei package che lo compongono, soffermandoci sulle strutture dati e i metodi più "interesanti".

4.1 Struttura generale

Il codice sorgente del JJPF è suddiviso in 5 package: client, common, service, shared e util. In Figura 4.1 è mostrata la struttura del framework realizzato.

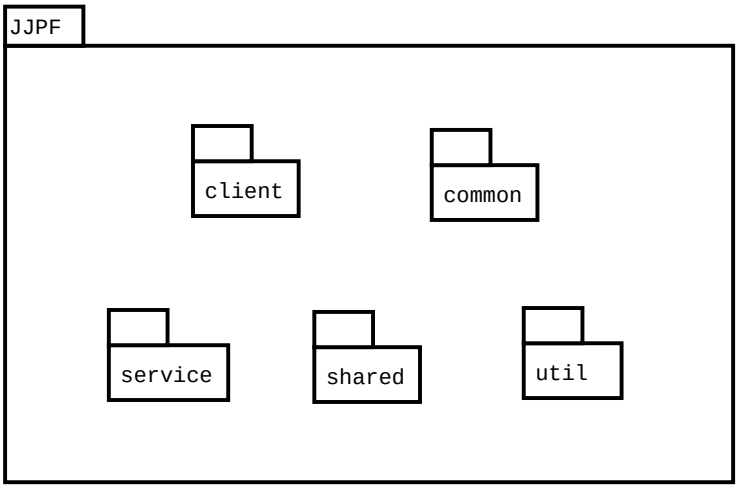


Figura 4.1: JJPF: struttura package

Il package `common` contiene la definizione delle interfacce JJPF più importanti. Interfacce che descrivono la struttura di: servizi, entità condivise e codice

mobile. Inoltre, in `common`, sono contenute le classi corrispondenti alle eccezioni lanciate dal JJPF.

Come suggerito dal nome, `client`, è il package dove sono contenuti gli strumenti che JJPF mette a disposizione per la creazione di clienti (§ 3.2.2.2 a pagina 41). Utilizzando le classi definite in questo package è possibile realizzare, semplicemente, clienti relativamente complessi.

In `shared` troviamo le classi dedicate allo sviluppo di entità condivise (§ 3.2.2.3). Se vogliamo realizzare con JJPF una computazione parallela seguendo il paradigma del farm con stato, l'importanza di questo package è fuori discussione.

Le classi JJPF destinate allo sviluppo semplificato di Servizi (§ 3.2.2.1 a pagina 39), sono contenute nel package `service`; quest'ultimo è a sua volta suddiviso in tre subpackage: `core`, `rmi` e `hybrid`. Essenzialmente `core` definisce le classi astratte da estendere per poter realizzare i servizi JJPF. `rmi` e `hybrid` sono i package messi a disposizione per consentire una rapida e semplice creazione di servizi.

Il package `util`, contiene due classi. La prima, `SimplestFormatter`, implementa un semplice logger, l'altra, `DependencyResolver`, è utilizzabile per risolvere le dipendenze di una classe Java: aiuta il programmatore semplificando la creazione del vettore delle classi che il *cliente* trasferirà sui servizi per l'esecuzione. `DependencyResolver` risulta essere uno strumento utilissimo quando si utilizza RMI ibrido (§ 3.2.3.2 a pagina 48) come modello di caricamento dinamico delle classi .

4.2 Descrizione dei Package

4.2.1 Common

Come già accennato durante la presentazione della struttura generale di JJPF (§ 4.1 nella pagina precedente), il package Common contiene le interfacce Java che definiscono la struttura di servizi, entità condivise e codice mobile.

In particolare:

- `ServiceIf` indica le funzionalità che un *servizio* deve mettere a disposizione ai clienti. Tali funzionalità sono rappresentate da sette metodi: un metodo per consentire ad un *cliente* di riservarsi la risorsa ed un altro per rilasciarla (`lock` e `unlock`), due metodi destinati al trasferimento dati dal *cliente* al *servizio* e viceversa (`setData` e `getData`), un metodo che permetta ad un *cliente* di indicare ad un *servizio* l'identificatore dell'entità condivisa con cui interfacciarsi (`setSharedObject`) ed un metodo per avviare l'esecuzione del codice *cliente* all'interno del *servizio* (`run`). Infine,

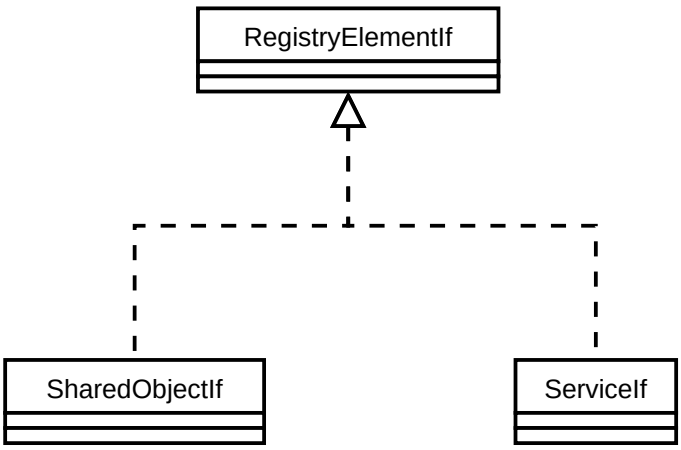


Figura 4.2: Gerarchia delle principali interfacce di Common

definisce un ultimo metodo (`exec`) per l'esecuzione di un ciclo completo di copia dati - esecuzione - trasferimento risultati dei risultati.

- `ProcessoIf` è l'interfaccia da implementare per la creazione di codice mobile da far eseguire ai servizi. I metodi qui definiti sono quattro: `getData`, `setData`, `run` e `setSharedObject`; la semantica di questi ultimi è analoga a quella descritta per `ServiceIf`.

- `SharedObjectIf` indica i metodi che uno SharedObject (l'entità condivisa del farm con stato) deve implementare affinchè i servizi possano interfacciarcisi. I metodi dichiarati in questa interfaccia sono due: `set` e `get`. Il primo prende due parametri in ingresso: il dato da memorizzare sull'entità comune e il suo indice; l'altro prende in input un indice e restituisce il dato corrispondente memorizzato nello SharedObject.

- `RegistryElementIf` è l'interfaccia vuota che tutti gli oggetti da registrare presso un *LookupService* devono implementare; `SharedObjectIf` e `ServiceIf` estendono questa interfaccia.

In JJPF sono dichiarate due eccezioni specifiche del framework:

- `ServiceAlreadyInUseException`;

- `ServiceRegistrationException`;

entrambe sono contenute nel sottopackage exception all'interno di Common.

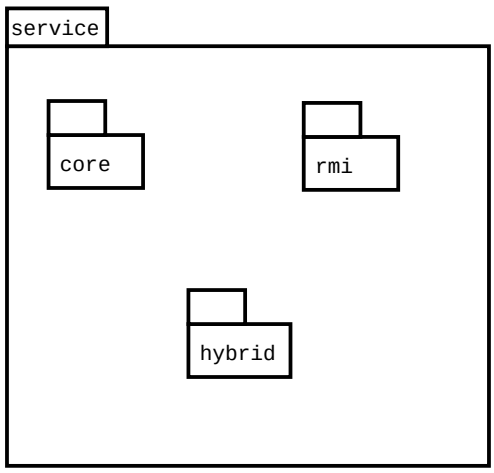


Figura 4.3: Gerarchia del Package service

4.2.2 Service

Se è vero che le reti di interconnessione sono la spina dorsale di una computazione parallela, è anche vero che le risorse di calcolo ne costituiscono la massa muscolare.

I servizi sono l'astrazione che il JJPF usa per rappresentare questi "muscoli", nei paragrafi che seguiranno ne vedremo in dettaglio la struttura.

4.2.2.1 core

`core` è il package che contiene l'insieme delle classi fondamentali per la realizzazione e la registrazione di un *servizio* JJPF.

Le principali classi qui definite sono:

- `ServiceRegistrationManager`;
- `AbstractLockableService`;
- `AbstractServiceRegister`;

`ServiceRegistrationManager` è la classe che si occupa di portare a termine i principali compiti per la registrazione di un *servizio*; per dialogare con il *LookupService* implementa i metodi definiti nelle interfacce `DiscoveryListener` e `LeaseListener`.

In questa classe sono definite due importanti strutture dati: `registrars` e `registrations`. Il compito affidato alla prima (`registrars`) è la memorizzazione dei proxy utilizzati per la comunicazione verso i *LookupService*. L'altra

(`registrations`) si occupa di tenere traccia delle informazioni ottenute durante la registrazione del *servizio* sui *LookupService*. Entrambe le strutture dati sono di tipo `java.util.Vector`.

Per maggiori dettagli sul significato di registrar e registration si rimanda alla documentazione delle API Jini (in particolar modo alle classi `ServiceRegistrar` e `ServiceRegistration` del package `net.jini.core.lookup`).

La classe `AbstractLockableService` estende `ServiceRegistrationManager`, a quest'ultima aggiunge alcune funzionalità di gestione riassumibili coi tre principali metodi implementati: `lock`, `unlock` e `searchInKnownRegistrar`.

Il primo (`lock`) può essere usato, da un *cliente*, per garantirsi l'uso esclusivo di un *servizio*.

Quando un *cliente* non ha più bisogno di una data risorsa computazionale, invoca sul *servizio* che la astrae il metodo `unlock`, così da consentirne l'utilizzo agli altri clienti. Il metodo `searchInKnownRegistrar` può essere utilizzato, sempre da un *cliente*, per indicare ai servizi arruolati l'identificatore dello `SharedObject` da ricercare. Se la ricerca non ha esito positivo, `AbstractLockableService` chiede ai *LookupService* di essere avvisato qualora l'oggetto cercato si registri presso di loro. Il meccanismo di avvisi è realizzato tramite gli "observer": particolari tipi di oggetti che un *cliente* o un *servizio* registra presso un *LookupService* affinchè quest'ultimo lo avvisi al verificarsi di particolari eventi.

Nella fattispecie, tutti gli observer usati da `AbstractLockableService` istanziano la classe `SharedObjectObserver`.

`AbstractServiceRegister` è una classe astratta da estendere quando si vogliano realizzare le entità destinate alla registrazione di servizi presso i *LookupService* (Figura 4.4 nella pagina successiva).

Affinchè i clienti possano utilizzare un *servizio* è essenziale che la sua presenza venga segnalata ad uno o più *LookupService*. Perchè ciò avvenga è necessario eseguire una particolare operazione di "registrazione", che prevede due passi:

1. la creazione del proxy che il *cliente* scaricherà da un *LookupService* e che utilizzerà per comunicare col *servizio*;

2. il trasferimento del suddetto proxy dal *servizio* al *LookupService*;

Solitamente ogni *servizio* possiede caratteristiche diverse dagli altri, per questo motivo è necessario che ad ogni tipo di *servizio* faccia capo un registratore ad-hoc.

Per semplificare il lavoro agli sviluppatori, la classe `AbstractServiceRegister` mette a disposizione il metodo protetto `getRMIProxy`, quest'ultimo genera un

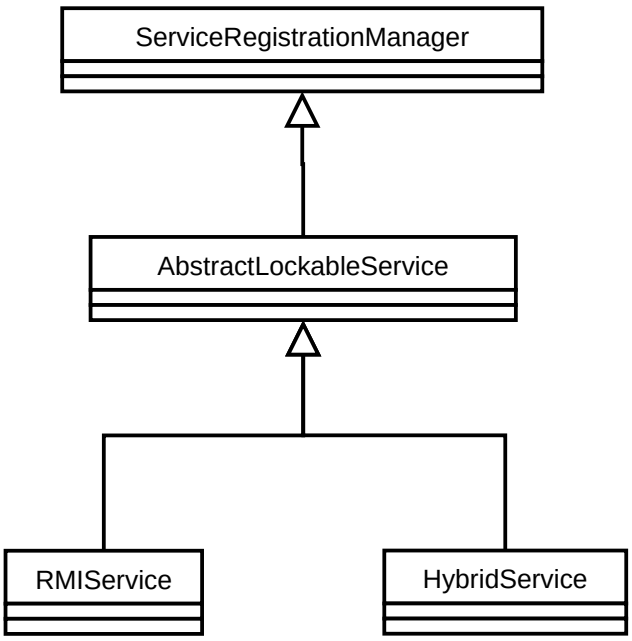


Figura 4.4: Gerarchia delle classi Servizi

proxy RMI del *servizio* da registrare. Questo proxy può essere utilizzato dagli sviluppatori come punto di partenza nella personalizzazione dei servizi JJPF.

Il resto del package `core` è costituito per lo più da interfacce, di queste l'unica degna di nota è `ServiceMgmtIf`, qui sono definite le firme dei metodi che un *servizio* deve implementare per riuscire a portare a termine la ricerca dei *LookupService*.

Come abbiamo già accennato in precedenza (§ 4.1 a pagina 53) JJPF offre la possibilità di creare due diversi tipi di servizi (`rmi` e `hybrid`), distinti dal meccanismo col quale, a tempo di esecuzione, caricano le classi.

4.2.2.2 rmi

Il package `rmi` contiene due classi: `RMIService` e `RMIServiceRegister`.

La prima definisce il *servizio* vero e proprio, l'altra specializza `AbstractServiceRegister` ed è utilizzabile per la registrazione di oggetti di tipo `RMIService`.

Entrambe le classi sono molto semplici e non aggiungono molto alle funzionalità offerte dalle classi che estendono.

Più in dettaglio:

- `RMIService`, implementando l'interfaccia `ServiceIf`, espone il metodo `setCode`, quest'ultimo istanzia un oggetto la cui classe è contenuta nel primo elemento del vettore in input, tale operazione consente la creazione, sul *servizio*, del codice cliente. Ricevendo la richiesta di invocazione del metodo `setSharedObject`, `RMIService` avvia la ricerca di un'entità condivisa (nella modalità già illustrata nel § 4.2.2.1 a pagina 56) utilizzando il metodo `searchInKnownRegistrar`, definito nella classe padre `AbstractLockableService`. Tutti gli altri metodi di `RMIService` si occupano semplicemente di inoltrare al codice mobile in esecuzione le richieste provenienti dal *cliente*.

- In `RMIServiceRegister` si possono identificare due metodi, uno è il costruttore della classe, l'altro, chiamato "exporting", genera il proxy da registrare presso i *LookupService*. Nella fattispecie, si limita a inoltrare la chiamata al metodo `getRMIProxy` (vedi § 4.2.2.1 a pagina 56).

4.2.2.3 hybrid

Nel paragrafo precedente abbiamo descritto la struttura di `jjpf.service.rmi`, un semplice *servizio* che, di fatto, non aggiunge molto alle funzionalità delle classi astratte che estende.

Qui esemineremo, invece, il *servizio* definito dal package `hybrid`, un *servizio* più complesso e articolato che sfrutta alcune interessanti possibilità offerte da Jeri (§ 2.2.1 a pagina 22): `hybrid`, propone un meccanismo di caricamento dinamico distribuito diverso da quello realizzato da RMI standard.

Le principali classi di questo package sono: `HybridService`, `HybridServiceRegister`, `HybridServiceClassLoader`, `HybridServiceProxy` e `WorkerClasses`.

- Buona parte dei metodi della classe `HybridService` si comportano in modo analogo agli omonimi definiti nella classe `RMIService`. D'altra parte è stato specificato più volte che la differenza fra i suddetti servizi è identificabile, quasi esclusivamente, nel diverso meccanismo implementato per il caricamento dinamico delle classi. Infatti, il metodo `setCode`, in questo caso non si ferma all'istanziazione di un oggetto ma si occupa anche dell'inizializzazione delle strutture dati interne al `HybridServiceClassLoader`. Quest'ultimo memorizza, all'interno di un `Vector`, il bytecode delle classi che, il codice cliente in esecuzione sul *servizio*, tenterà di caricare durante la computazione.

- `HybridServiceClassLoader` è la classe che sta alla base del caricamento dinamico distribuito proposto da `hybrid`. I bytecode contenuti nel vettore passato dal *cliente*, in input al metodo `setCode`, vengono memorizzati in una struttura dati interna alla classe `HybridServiceClassLoader`. Da li sono richiamati quando il codice cliente in esecuzione sul *servizio* tenta di istanziare una classe che il classloader primordiale non conosce.

- La registrazione, presso un *LookupService*, di un oggetto di tipo `HybridService` è affidata alla classe `HybridServiceRegister`. Come già detto in precedenza un "registratore di servizi", realizzato estendendo la classe `AbstractServiceRegister`, è personalizzabile implementando il metodo astratto `exporting`. Il metodo exporting implementato nella classe `HybridServiceRegister` si occupa di incapsulare il proxy RMI restituito dall'invocazione del metodo `getRMIProxy` all'interno di un oggetto della classe `HybridServiceProxy`.

- Analizzando la classe `HybridServiceProxy` è possibile osservare che quasi tutti i metodi in essa definiti si limitano a richiamare il loro corrispettivo remoto, implementato sul lato *servizio*. La comunicazione tra *cliente* e *servizio* avviene utilizzando i metodi del proxy RMI incapsulato. Il metodo `setCode`, invece, anzichè trasferire sul *servizio* il vettore di classi, esegue localmente, quindi sul *cliente*, le operazioni necessarie alla creazione del vettore di bytecode java da passare all'`HybridServiceClassLoader` sito sul *servizio*.

4.2.3 Shared

In precedenza abbiamo ripetuto più volte che per realizzare un farm con stato è necessaria la presenza di un'entità condivisa fra i vari servizi. Nel package `shared` sono definite le classi che semplificano la realizzazione di tale entità. Le classi che analizzeremo in questo packege sono due: `SharedObject` e `SharedObjectRegister`.

- Per realizzare un'entità condivisa è sufficiente estendere la classe `SharedObject`, e implementare i due metodi per l'accesso ai dati definiti in `SharedObjectIf`. `SharedObject` anzichè seguire la catena di ereditarietà usata dai servizi `rmi` e `hybrid`, estende direttamente

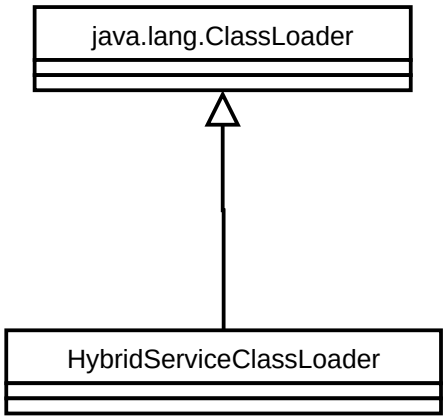


Figura 4.5: HybridServiceClassLoader

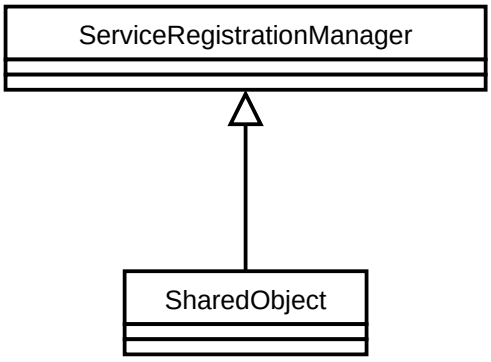


Figura 4.6: Gerarchia della Classe SharedObject

`ServiceRegistrationManager` (Figura 4.6 nella pagina precedente). Infatti, non avendo bisogno né dei metodi `lock` e `unlock` né di quelli destinati alla ricerca di entità condivise, non ha nessun motivo di estendere la classe `AbstractLockableService`.

- La registrazione presso i *LookupService*, degli oggetti di tipo `SharedObject`, è affidata alla classe `SharedObjectRegister`. Il compiti svolti da quest'ultima sono analoghi a quelli tenuti dalla classe `RMIServiceRegister` nei confronti di `RMIService`.

4.2.4 Client

Abbiamo cominciato questo capitolo descrivendo la struttura generale del JJPF, in seguito siamo passati ad analizzare i package per la creazione dei servizi e delle entità condivise tra questi ultimi, adesso ci occuperemo dell'altro grande soggetto di JJPF: il *cliente*.

Nel capitolo 3 a pagina 35 abbiamo descritto la struttura logica del *cliente* e siamo poi passati all'analisi del suo comportamento. Adesso vedremo in dettaglio la sua implementazione.

Le principali classi definite in questo package sono: `AbstractClient`, `BasicClient`, `AbstractClientThread`, `TaskHolder` e `ServiceObserver`.

`AbstractClient` è la colonna portante del *cliente*, gestisce le comunicazioni con il *LookupService*, ricerca i servizi e le entità condivise fra questi ultimi. Inoltre, si occupa, tramite un sistema di indirezione basato su chiamate a metodi astratti, di avvertire le classi che la estendono al verificarsi di alcuni eventi. Qui di seguito riportiamo i più importanti:

- la scoperta di nuove risorse computazionali (`workerDiscovered`);
- la scomparsa di alcuni servizi precendemente arruolati (`workerLeft`);
- la localizzazione di un'entità condivisa che rispetti il template specificato (`foundSharedObject`);

Le principali strutture dati contenute in questa classe sono:

- `workerHash`: una tabella hash che contiene gli identificatori Jini dei servizi arruolati nella computazione;
- `serviceObservers`: un vettore in cui sono memorizzati i riferimenti agli "observer";

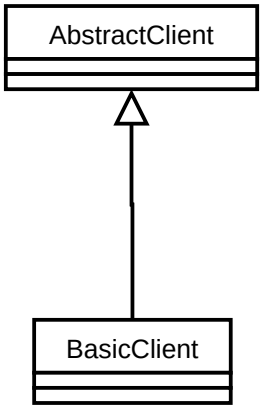


Figura 4.7: BasicClient

- `sharedObjectObservers`: una lista il cui scopo è molto simile a `serviceObservers` ma anzichè monitorare lo stato dei servizi, controlla i cambiamenti di stato delle entità condivise.

`AbstractClient` integra un server `HTTP`, utilizzato dal *cliente* per il trasferimento del java bytecode. Senza un webserver il *cliente* non potrebbe utilizzare correttamente i servizi che prevedono il caricamento dinamico realizzato secondo il meccanismo RMI "puro" (§ 3.2.3.1 a pagina 47, 4.2.2.2 a pagina 58). Il server HTTP contenuto nel *cliente* è realizzato utilizzando la classe `com.sun.jini.tool.ClassServer`.

`AbstractClient` è una classe astratta che per questo motivo non può essere istanziata. Per consentire un rapido e semplice lo sviluppo di applicazioni parallele JJPF fornisce una classe concreta che estende `AbstractClient` alla quale aggiunge alcune utili funzionalità, il suo nome è `BasicClient`. Le principali strutture dati definite in questa classe sono:

- `waitingQueue`: una coda usata per memorizzare gli identificatori dei servizi disponibili non ancora in uso; è realizzata con una java `LinkedList`;
- `taskHolder`: la struttura dati che memorizza e gestisce i dati da calcolare nel contesto della nostra computazione parallela;

La classe `BasicClient` implementa tutti i metodi astratti della sua superclasse, vediamo i dettagli dei più importanti fra di essi:

`workerDiscovered`: all'atto della scoperta di un nuovo *servizio*, `BasicClient` aggiunge l'identificatore del nuovo arrivato in `workerHash`, a quel punto se

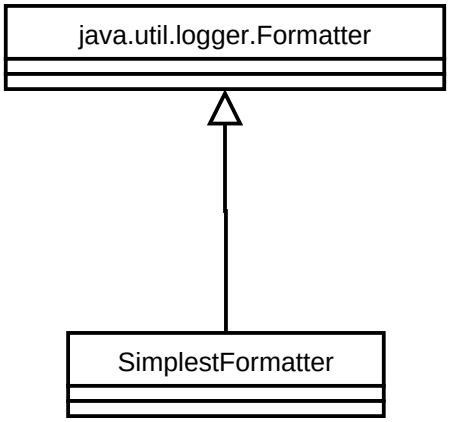


Figura 4.8: SimplestFormatter

non è stata richiesta la localizzazione di un'entità condivisa (o se tale localizzazione è già stata portata a termine con successo), `BasicClient` genera un `BasicClientThread` che si occupa della gestione della nuova risorsa computazionale. Se, al contrario, stiamo ancora cercando uno `SharedObject` l'identificatore del *servizio* viene inserito nella coda `waitingQueue`.

`workerLeft:` se si interrompe il collegamento con un calcolatore arruolato per la nostra computazione parallela, `BasicClient` rimuove da `workerHash` l'identificatore corrispondente al *servizio* rimosso. Se a quel punto non ci sono più thread di gestione attivi e la computazione è terminata, viene chiamato il metodo `stopServiceDiscovery` definito in `AbstractClient` che causa l'interruzione della ricerca di nuove risorse computazionali.

`foundSharedObject:` quando un'entità condivisa corrispondente al template specificato dall'utente viene trovata, `BasicClient` scorre la coda `waitingQueue` e per ogni elemento in essa contenuto fa partire un thread di gestione.

Il *cliente* JJPF è in grado di interfacciarsi simultaneamente ed efficentemente con un numero elevato di servizi grazie alla sua struttura multithreaded. Per ogni *servizio* arruolato viene fatto partire, sul *cliente*, un thread di gestione, ogni thread è un oggetto della classe `BasicClientThread`. Il frammento di codice 6 a fronte ne mostra sommariamente il comportamento.

Frammento di Codice 6 Thread del *cliente* per la gestione del *servizio*

```
// mi riservo l'uso esclusivo del servizio;
service.lock();

// trasferisco il codice da eseguire sul servizio
service.setCode(CodiceDaEseguire);

// comunico al servizio l'identificatore di una eventuale
// entità condivisa da cercare e utilizzare per la realizzazione
// del farm con stato
service.setSharedObject(SharedObjectID);

// entro nel ciclo di esecuzione vero e proprio
while( not( lavoroFinito ) )
{
     // spedisco i dati, eseguo il calcolo e ricevo il risultato
     risultato[i] = service.exec( datiDaElaborare[i] );
}

// a lavoro finito rilascio la risorsa computazionale
service.unlock();
```

4.2.5 Util

All'interno del package `util` sono definite due classi: `SimplestFormatter` e `DependencyResolver`. La prima delle due implementa un logger, realizzato estendendo la classe `java.util.logger.Formatter`. Per chi fosse interessato ai dettagli per la realizzazione di un java Formatter si rimanda alla documentazione Sun delle API java.

La classe `DependencyResolver` è stata realizzata per aiutare il programmatore JJPF nella costruzione del vettore delle classi che il *cliente* trasferirà sui servizi. Analizzando il bytecode di una data classe A, `DependencyResolver` riesce a determinare quali sono le classi che A cercherà di caricare una volta in esecuzione. La classe `DependencyResolver` da noi realizzata è un wrapper ad uno strumento Sun chiamato `ClassDep`, distribuito col Jini toolkit 2.0.

`DependencyResolver` risulta essere uno strumento utilissimo quando si utilizza il meccanismo di caricamento dinamico delle classi realizzato da RMI ibrido (§ 3.2.3.2 a pagina 48).

4.3 Esempio d'uso

Dopo aver descritto in dettaglio l'implementazione del JJPF, adesso illustriamo un esempio che mostra quanto sia semplice realizzare un'applicazione parallela di tipo farm utilizzando il framework sviluppato.

L'applicazione implementata è semplicissima, si tratta di un farm stateless in cui il *cliente* spedisce un numero intero ai worker i quali dopo aver convertito il numero in una stringa le concatenano "TestAff".

Di seguito descriviamo i passi necessari all'implementazione del *Servizio* e del *cliente* JJPF, illustrando per ogni fase dello sviluppo il codice da scrivere per creare l'applicazione. Il codice completo è riportato al termine della descrizione.

Cominciamo mostrando il codice minimo necessario per costruire un Servizo JJPF. Qui viene illustrata la classe `SempliceServizio` nella quale è definito soltanto il metodo `main`.

```
public class SempliceServizio
{
   public static void main(String[] args)
   {
      ...
```

Il primo passo riguarda la creazione di un *service provider*, in questo caso si tratta di un oggetto `RMIServiceRegister`, il quale si occupa di creare un oggetto di tipo `RMIService` ovvero un *servizio* che per il caricamento dinamico delle classi utilizza il meccanismo stadard di Jini.

```
      ...
      RMIServiceRegister srvPrv =
                  new RMIServiceRegister(RMIService.class);
      ...
```

A questo punto è necessario indicare al *service provider* gli indirizzi dove cercare i *LookupService* presso i quali registrare l'oggetto *servizio* creato. In questo caso viene fatta una ricerca multicast, `LookupDiscovery.ALL_GROUPS` indica a Jini di cercare su tutti gli indirizzi di questo tipo.

```
      ...
      try{
         srvPrv.setMulticastGroups(LookupDiscovery.ALL_GROUPS);
      }
      catch(ServiceRegistrationException sre)
      {
         System.out.println("Errore durante Registrazione sul LookupService");
         System.exit(1);
      }
      ...
```

Dopo aver indicato al framework, tramite il *service provider*, gli indirizzi presso i quali ricercare i *LookupService*, è sufficiente invocare il metodo `start` per dare il via al *servizio*.

```
      ...
      srvPrv.start();

   }
}
```

Per sviluppare un *cliente* cominciamo definendo la classe `TestAff` che implementa l'interfaccia `ProcessoIf`.

```
public TestAff implements ProcessoIf
{

```

Di seguito passiamo a definire due attributi privati della classe destinati a contenere rispettivamente il numero intero e la stringa "TextAff".

```
   ...
   private int numeroIntero = 0;
   private String strConcatenata = null;
   ...
```

Il passo successivo è l'implementazione del metodo `setData` col quale JJPF passa i dati ai worker sotto forma di `java.lang.Object`. In questo caso convertiamo `obj` al tipo `Integer`.

```
   ...
   public void setData(Object obj)
   {
      numeroIntero = ((Integer) obj).intValue();
   }
   ...
```

Passiamo ora a implementare `run`: questo si occupa di concatenare la il numero in input alla stringa "TestAff".

```
   ...
   public void run()
   {
      strConcatenata = new String("TestAff");
      strConcatenata = numeroIntero + strConcatenata;
   }
   ...
```

Infine definiamo il metodo invocato dal framework per recuperare i risultati della computazione dal *cliente*.

```
   ...
   public Object getData()
   {
      return strConcatenata;
   }
   ...
```

Per ultimo dobbiamo implementare il metodo `setSharedObject`, usato da JJPF per assegnare al worker il proxy da utilizzare per le comunicazioni con l'entità condivisa. Nel nostro caso vuoto in quanto stiamo realizzando un farm stateless.

```
   ...
   public setSharedObject(SharedObjectIf ssi)
   {
   }

}
```

Dopo aver implementato il codice da eseguire sui servizi, passiamo alla definizione della classe *cliente*, ovvero l'entità che si occupa di trovare e gestire le risorse computazionali.

Per creare un semplice *cliente* cominciamo definendo la classe `SempliceCliente` nella quale è dichiarato il solo metodo `main`.

```
public SempliceCliente
{
   public static void main(String[] args)
   {
      ...
```

A questo punto cominciamo definendo la Collection con i dati da passare in input all'applicazione parallela e dichiarandone un'altra destinata a contenere i risultati del calcolo.

```
      ...
      Collection input = new Vector();
      Collection output = new Vector();
      ...
```

Il passo successivo consiste nel riempire la Collection di input con i dati, in questo caso un insieme di mille oggetti di tipo Integer il cui valore è calcolato in modo pseudocasuale.

```
...
Random rand = new Random();

for(int i=0;i<1000;i++)
{
   Integer intero = new Integer(rand.nextInt());
   input.add(intero);
}
...
```

A questo punto dobbiamo passare a costruire il vettore delle classi richieste da `TextAff` per la sua esecuzione. In questo caso la classe da trasferire sui worker dipende soltanto da quelle che fanno già parte del Java SDK, quindi il vettore delle classi da trasferire contiene soltanto la classe da noi sviluppata.

```
...
Class[] arrayClassi = new Class[1];
arrayClassi[0] = TextAff.class;
...
```

Arrivati a questo punto è necessario istanziare il *cliente* vero e proprio: il modo più semplice è utilizzare la classe `BasicClient`.

Ovviamente essendo un farm stateless il secondo argomento del costruttore è `null`.

```
...
BasicClient cm = null;

try
{
   cm = new BasicClient(arrayClassi,null, input, output);
}
catch(IOException ioe)
{
   System.out.println("Errore di I/O durante la creazione del Client:" +ioe);
}
catch(ConfigurationException ce)
{
   System.out.println("Errore di Configurazione durante la creazionedel Client: "+ ce);
}
...
```

Dopo aver creato l'oggetto *cliente* è necessario indicargli quali indirizzi utilizzare per la ricerca del *LookupService*.

```
...
cm.setLookup(AbstractClient.ALL_GROUPS);
...
```

A questo punto è sufficiente dare il via al *cliente* utilizzando il metodo `start`.

```
...
cm.start();
...
```

L'invocazione del metodo `start` sul *cliente* provoca la creazione di un thread. Da questo momento in poi il flusso d'esecuzione di `BasicClient` e di `SempliceCliente` sono separati, per fare in modo che quest'ultimo attenda la terminazione dell'esecuzione del *cliente* e recuperare i dati derivanti dall'esecuzione della computazione è necessario invocare dopo il metodo `start`, il metodo `join`.

```
...
try{
   cm.join();
}
catch(InterruptedException ie)
{
   System.out.println("Interruzione imprevista: "+ie);
   System.exit(1);
}
...
```

Una volta terminato `BasicClient` si passa all'esame e alla stampa dei risultati calcolati.

```
...
Iterator iter = output.iterator();
while(iter.hasNext())
{
   String result = (String) iter.next();
   System.out.println("Il Risultato e': "+result);
}

  }
}
```

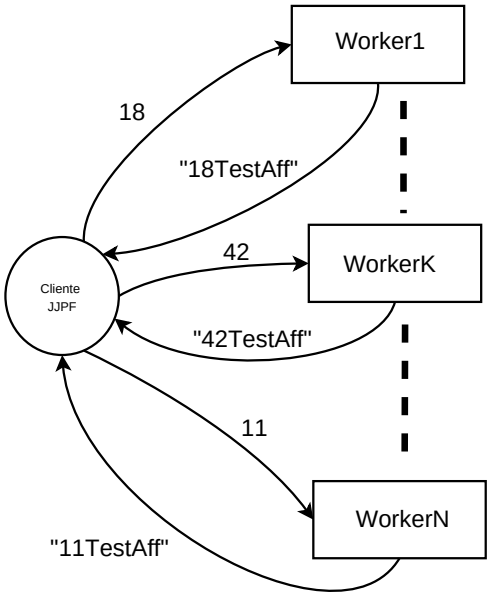


Figura 4.9: Struttura dell'applicazione di esempio TestAff

4.3.1 Codice completo dell'esempio

4.3.1.1 SempliceServizio

```
import jjpf.service.rmi.RMIServiceRegister;
import jjpf.service.rmi.RMIService;
import jjpf.common.exception.ServiceRegistrationException;
import net.jini.discovery.LookupDiscovery;

public class SempliceServizio
{
   public static void main(String[] args)
   {
      // Creazione del Service Provider
      RMIServiceRegister srvPrv =
                  new RMIServiceRegister(RMIService.class);

      // Ricerca dei LookupService
      try{
         srvPrv.setMulticastGroups(LookupDiscovery.ALL_GROUPS);
      }
      catch(ServiceRegistrationException sre)
      {
          System.out.println("Errore durante Registrazione sul LookupService");
          System.exit(1);
      }

      // Avvio del Servizio
      srvPrv.start();
   }
```

```
}
```

4.3.1.2 SempliceCliente

```
import java.util.Collection;
import java.util.Vector;
import java.util.Random;
import java.util.Iterator;
import java.io.IOException;

import net.jini.config.ConfigurationException;

import jjpf.client.BasicClient;
import jjpf.client.AbstractClient;


public class SempliceCliente
{
   public static void main(String[] args)
   {
      // Creazione Collection di input e di output
      Collection input = new Vector();
      Collection output = new Vector();

      // Inizializzazione del generatore
      // di numeri PseudoCasuali
      Random rand = new Random();

      // Inserimento dati nella Collection di input
      for(int i=0;i<1000;i++)
      {
         // Creazione di un oggetto intero da numero intero
         Integer intero = new Integer(rand.nextInt());

         // Inserimento dell'oggetto intero nella Collection
         input.add(intero);
      }

      // Generazione dell'array di classi da passare
      // ai worker
      Class[] arrayClassi = new Class[1];
      arrayClassi[0] = TestAff.class;

      // Dichiaro un Cliente
      BasicClient cm = null;

      try
      {
         // Istanziazione di un oggetto Cliente
         cm = new BasicClient(arrayClassi,null, input, output);
      }
      catch(IOException ioe)
      {
```

```
            System.out.println("Errore di I/O durante la creazione del Client:" +ioe);
         }
         catch(ConfigurationException ce)
         {
            System.out.println("Errore di Configurazione durante la creazionedel Client: "+ ce);
         }

         // Ricerca dei LookupService
         cm.setLookup(AbstractClient.ALL_GROUPS);

         // Avvio del Cliente
         cm.start();

         // Attesa della terminazione del Cliente
         try
         {
            cm.join();
         }
         catch(InterruptedException ie)
         {
            System.out.println("Interruzione imprevista: "+ie);
            System.exit(1);
         }

         // Analisi e stampa dei risultati
         Iterator iter = output.iterator();
         while(iter.hasNext())
         {
            String result = (String) iter.next();
            System.out.println("Il Risultato e': "+result);
         }

      }
   }
```

4.3.1.3 TestAff

```
import jjpf.common.ProcessoIf;
import jjpf.common.SharedObjectIf;

public class TestAff implements ProcessoIf
{
   // Variabili Globali per la memorizzazione
   // del numero ottenuto e della stringa
   // da restituire
   private int numeroIntero = 0;
   private String strConcatenata = null;

   // Metodo tramite il quale ci vengono
   // passati i dati di input
   public void setData(Object obj)
   {
```

```
      // Conversione da tipo Integer a tipo
      // intero primitivo
      numeroIntero = ((Integer) obj).intValue();
   }

   // Metodo che esegue la concatenazione
   public void run()
   {
      strConcatenata = new String("TestAff");
      strConcatenata = numeroIntero + strConcatenata;
   }

   // Metodo col quale questa classe restituisce
   // il risultato del calcolo
   public Object getData()
   {
      return strConcatenata;
   }

   // Metodo usato per ottenere il proxy
   // per le comunicazioni con una entità
   // condivisa. In questo caso vuoto perchè
   // stiamo realizzando un farm stateless
   public void setSharedObject(SharedObjectIf ssi){}
}
```

Capitolo 5

Test Effettuati e Risultati Ottenuti

Per testare e valutare le effettive potenzialità del JJPF sono state eseguite alcune batterie di test.

Lo scopo dei test è quello di controllare se il framework realizzato rispetta i requisiti di performance e affidabilità che un ambiente per la programmazione parallela deve soddisfare.

In particolar modo sono stati effettuati test di: scalabilità (§ 5.3.1 a pagina 78), affidabilità (§ 5.3.2 a pagina 88) e dinamicità (§ 5.3.3 a pagina 89).

5.1 Ambiente di prova

Tutte le prove sono state effettuate utilizzando le macchine del centro di calcolo del Dipartimento di Informatica dell'Università di Pisa.

Di seguito riportiamo la configurazione hardware e software dei computer utilizzati.

5.1.1 Configurazione Hardware

- **CPU:**
  - AMD Athlon XP 2600+
  - frequenza di clock 2,13 Ghz
  - Cache di primo livello associativa su insiemi a 2 vie da 128 KB
  - Cache di secondo livello associativa su insiemi a 16 vie da 256 KB
  - Linux Bogomips 4259

- **Memoria:**
  - 512 MB di RAM
- **Interfaccia di rete:**
  - Scheda di rete Ethernet da 100 Mbit

5.1.2 Software di Base Installato

- **Sistema Operativo:**
  - RedHat Linux
  - Kernel Linux 2.4.20-8
  - glibc 2.3.2
- **Java:**
  - Java2 SDK 1.4.0-01
  - Jini 2.0.001

5.2 Strumenti Utilizzati

Per effettuare i test e analizzare i risultati che questi ultimi ci hanno fornito, abbiamo utilizzato alcuni utili strumenti : un linguaggio di scripting (il Perl), un programma per la creazione di grafici (gnuplot) e una libreria per l'elaborazione delle immagini.

5.2.1 Perl

L'attività di prova di un sistema distribuito comprende alcuni compiti che ben si prestano ad essere svolti attraverso l'uso di un linguaggio di scripting. Nel nostro caso, ad esempio, abbiamo realizzato uno script per trasferire il nostro framework sui calcolatori da arruolare nella computazione, un altro per avviare i servizi sulle macchine da arruolare, uno per far partire il *LookupService* e uno per avviare l'esecuzione del *cliente*. Abbiamo inoltre sviluppato script per l'analisi dei dati sperimentali ottenuti e script per la formattazione dei file di dati. Infine, abbiamo realizzato uno script per la generazione dei file da passare come input a Gnuplot (§ 5.2.2 nella pagina successiva). Tutti gli script sono stati sviluppati in Perl.

Perl[1] è un linguaggio di scripting finalizzato alla trattazione di stringhe e file di testo [Dan99, TC98, ES98]. Con Perl è estremamente facile fare ricerche di

[1] Processing Estraction Report Language

sequenze di caratteri all'interno di stringhe (pattern matching), sostituire parti di stringhe (pattern substitution) e operare su file di testo strutturati.

5.2.2 Gnuplot

Al termine di una prova, per capire quanto i risultati ottenuti siano vicini a quelli previsti, è necessario analizzare i dati contenuti nel file di log generato dal *cliente* JJPF.

Per studiare ed analizzare i dati raccolti, spesso è necessario esaminare minuziosamente file di log lunghi diverse centinaia di righe. Un approccio più immediato e sicuramente più comodo è quello di usare un software in grado di visualizzare graficamente i dati ottenuti sperimentalmente.

Gnuplot è un programma per la visualizzazione di dati e di funzioni, configurabile tramite appositi script [Cra98]. Tutti i grafici relativi all'analisi dei risultati presenti in questa tesi sono stati realizzati con l'ausilio di questo strumento.

5.2.3 JIU

Per i test abbiamo sviluppato alcuni programmi di prova, in particolare abbiamo realizzato applicazioni per l'image processing che applicano una serie di filtri grafici (Median e Blur) allo stream di immagini in input. La libreria grafica usata si chiama JIU.

5.3 Dati ottenuti

Prima di cominciare a descrivere i test effettuati e analizzare i dati sperimentali ottenuti, è utile definire un sistema ideale, ovvero un sistema che offra massimi valori teorici sia in termini di scalabilità che di dinamicità.

Sia N il numero totale di task da calcolare e t_i il tempo necessario al calcolo dell'i-esimo task, allora il tempo totale sequenziale di completamento vale:

$$T_{TOTALE} = \sum_{i=1}^{i=N} t_i$$

In un sistema che garantisca una **scalabilità** ideale, utilizzando P calcolatori, la formula diventa:

$$T_{TOTALE}^{Parallelo} = \frac{\sum_{i=1}^{i=N} t_i}{P}$$

In altre parole, utilizzando un sistema ideale il tempo necessario al calcolo degli N task diminuisce in modo direttamente proporzionale all'aumento delle

risorse computazionali.

Nel paragrafo 3.1.3 a pagina 36 abbiamo introdotto il significato del termine **dinamicità**, adesso, esemplificheremo il comportamento di un sistema che offra un grado di dinamicità ideale.

Cominciamo suddividendo il calcolo degli N task[2] in due parti uguali,

$$T^{Parallelo}_{TOTALE} = T^{Parallelo}_{PARZIALE_1} + T^{Parallelo}_{PARZIALE_2} = \frac{\sum_{i=1}^{i=\frac{N}{2}} t_i}{P} + \frac{\sum_{i=\frac{N}{2}+1}^{i=N} t_i}{P}$$

se durante il calcolo dei primi $\frac{N}{2}$ task lasciamo il numero P di calcolatori invariato, e lo raddoppiamo durante il calcolo degli $\frac{N}{2}$ task rimanenti, poichè stiamo utilizzando un sistema con scalabilità ideale il tempo necessario al calcolo del secondo addendo diventa

$$T^{Parallelo}_{PARZIALE_2} \frac{\sum_{i=\frac{N}{2}+1}^{i=N} t_i}{2P} = \frac{1}{2} \cdot \frac{\sum_{i=\frac{N}{2}+1}^{i=N} t_i}{P}$$

quindi supponendo $t_i$costante possiamo riscrivere il nuovo tempo totale T' come

$$T' = \frac{\frac{N}{2} \cdot t}{P} + \frac{\frac{N}{2} \cdot t}{2P} = \frac{(N \cdot t) + (\frac{N}{2} \cdot t)}{2P} = \frac{t \cdot (N + \frac{N}{2})}{2P} = \frac{\frac{3}{2} \cdot N \cdot t}{2P} = \frac{3}{4} \cdot \frac{N \cdot t}{P}$$

dalla formula è facile ricavare che $T' = \frac{3}{4} \cdot T^{Parallelo}_{TOTALE}$, ovvero raddoppiando il numero di calcolatori nelle seconda metà della computazione, il tempo necessario al calcolo degli ultimi $\frac{N}{2}$ task dimezza e il tempo totale si riduce di un quarto.

Quindi la performance di un sistema ottimo in termini di dinamicità varia in modo direttamente proporzionale al:

- numero di calcolatori utilizzati;
- tempo per il quale le risorse computazionali sono utilizzate.

5.3.1 Test di scalabilità

Un framework per la programmazione parallela deve garantire un buon livello di scalabilità (§ 3.1.1 a pagina 35), in questo paragrafo descrivremo i test che abbiamo eseguito per valutare JJPF relativamente a questa caratteristica.

Per le prove abbiamo scritto, utilizzando JJPF e JIU, alcuni programmi per il processing di immagini digitali. Più precisamente abbiamo realizzato dei programmi per l'applicazione di filtri grafici.

[2] per semplicità supponiamo N pari

I test sono stati eseguiti su un numero variabile di macchine, compreso tra uno e cinquantacinque. I programmi realizzati per i test sono stati provati sia strutturando la computazione come un farm stateless, sia strutturandola come un farm con stato, in entrambi i casi è stato testato sia il modello di caricamento standard di Jini sia il modello hybrid proposto in questa tesi.

I primi quattro test realizzati prevedevano l'applicazione del filtro grafico Blur (effetto sfuocatura) ad uno stream di 320 immagini. La dimensione media delle immagini processate è di circa 40 Kbyte e il tempo medio richiesto per l'elaborazione di una singola immagine è approssimativamente un secondo. Il tempo di trasmissione necessario per il trasferimento, dal *cliente* al *servizio*, di una singola immagine[3] è di circa 0.017 secondi. Mettendo a rapporto il tempo di comunicazione e il tempo di calcolo di una singola immagine otteniamo il valore di grana utilizzato:

$$grana = \frac{Tempo\,di\,Calcolo}{2 \cdot Tempo\,di\,Comunicazione)} = \frac{1\,sec}{2 \cdot (0.017)\,sec} \approx 29.411$$

I risultati di questi test sono illustrati nelle figure 5.2, 5.5, 5.7 e 5.9. In ognuna di queste figure è possibile distinguere tre linee, rappresentative di altrettanti valori misurati: tempo totale, tempo totale di calcolo e funzione obiettivo. Quest'ultima indica l'andamento dei valori di scalabilità di un sistema ideale rispetto a questa caratteristica. Le altre due linee rappresentano rispettivamente il tempo totale di calcolo, e il tempo totale. Il primo non tiene conto del tempo necessario per l'arruolamento e il disarruolamento delle risorse computazionali, l'altro invece considera anche tali overhead. L'incidenza, espressa in percentuale, di questi ultimi due valori è rappresentata nelle figure 5.3, 5.6, 5.8 e 5.10.

La Figura 5.2 a pagina 81 mostra l'andamento della scalabilità ottenuta strutturando la nostra applicazione di prova come un farm stateless (il cui schema è mostrata in Figura 5.1) e utilizzando per il trasferimento delle classi il meccanismo standard di Jini. Osservando la figura vediamo che il grafico della funzione obiettivo rimane molto vicina alle altre due linee per la quasi totalità del grafico, il distacco è apprezzabile soltanto quando il numero di macchine arruolate è maggiore di sedici. Infatti, l'overhead misurato, mostrato in Figura 5.3, supera la soglia dell'8% soltanto quando i worker arruolati nella computazione parallela salgono a trentadue. Questa degradazione della performance è legata al sovradimensionamento del farm rispetto alla velocità della struttura di interconnessione. Infatti, come mostrato nella formula del calcolo della "grana", il tempo di comunicazione necessario per trasferire le informazioni sui worker e recuperare i risultati del calcolo è pari a circa 35 millisecondi men-

[3]Misurato a rete scarica, con un solo worker arruolato nella computazione

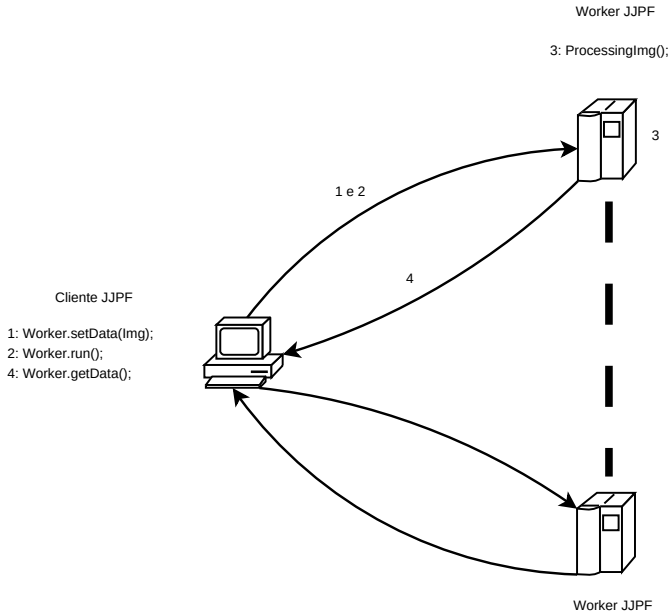


Figura 5.1: Schema Logico della prova di scalabilità strutturata come farm stateless

tre il tempo che impiega un worker a calcolare un singolo task è pari a circa un secondo. Mettendo a rapporto i due tempi otteniamo il numero di worker necessari al dimensionamento ottimo del farm: circa ventinove. Risulta chiaro che abbiamo arruolato per la nostra computazione parallela più worker di quanti ne fossero necessari, di conseguenza, durante il calcolo esistono costantemente alcune risorse computazionali inutilizzate, ciò comporta la perdita di efficienza registrata.

In Figura 5.5 a pagina 82 è rappresentato l'andamento dei valori di scalabilità quando si struttura l'applicazione parallela come un farm senza stato utilizzando per il caricamento dinamico e distribuito delle classi il meccanismo hybrid. Anche in questo caso JJPF garantisce ottimi valori di scalabilità fino all'utilizzo di sedici calcolatori, superata questa soglia, la scalabilità decade sensibilmente. In Figura 5.6 è chiaramente distinguibile un'impennata dell'overhead quando si passa all'uso di trentadue worker, anche in questo caso il degrado della performance è attribuibile allo squilibrio esistente tra la potenza computazionale delle risorse di calcolo disponibili e la velocità della struttura di interconnessione tra esse.

Le figure 5.7 a pagina 84 e 5.9 a pagina 85, mostrano la scalabilità ottenuta sviluppando l'applicazione di prova secondo il paradigma del farm con stato (il cui schema è mostrata in Figura 5.4). Nel primo caso (Figura 5.7) utilizzando

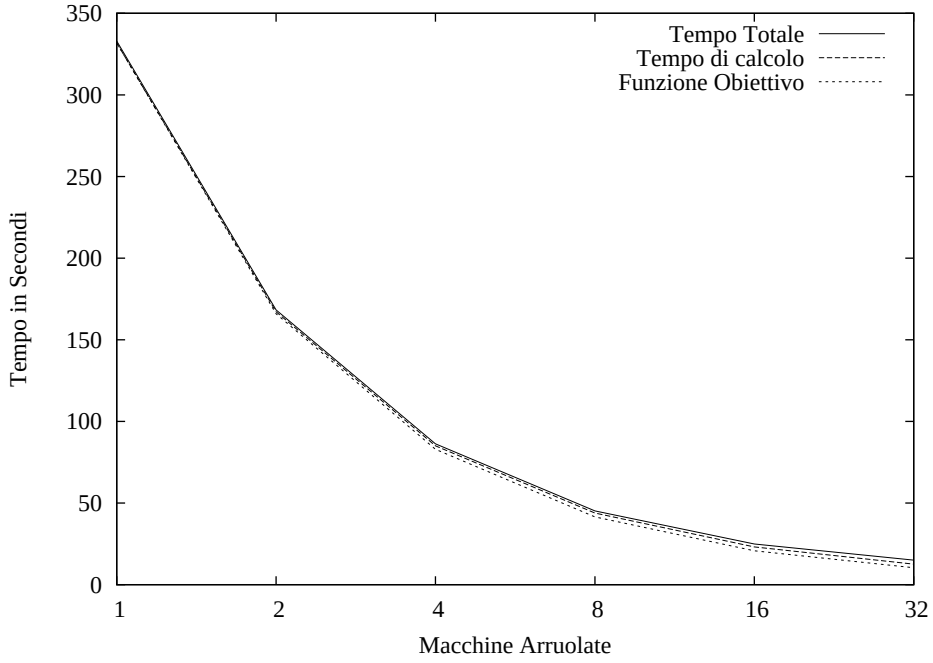


Figura 5.2: Test di Scalabilità del farm stateless RMI: 320 immagini processate col filtro grafico Blur

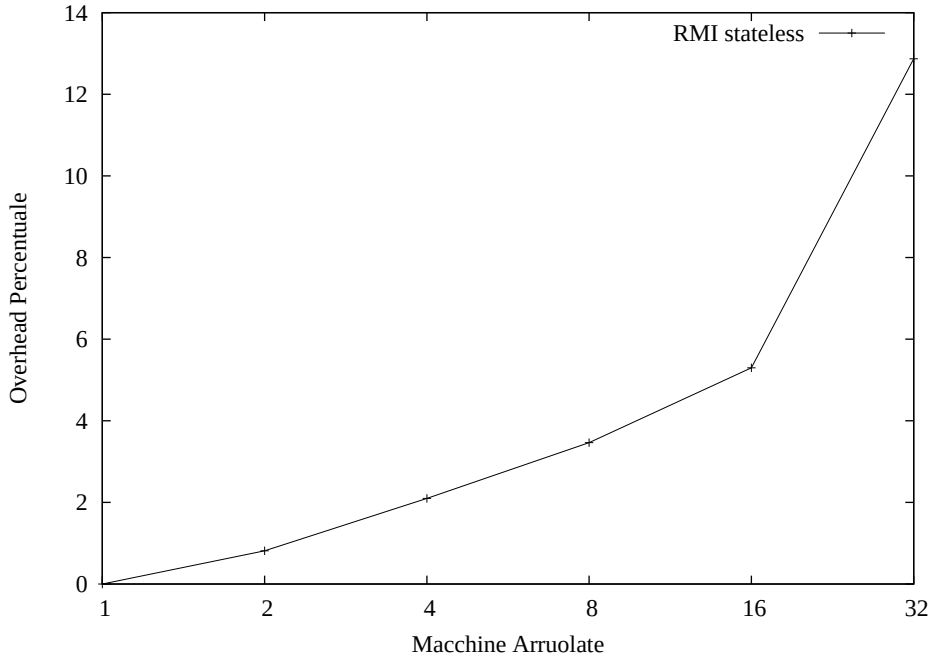


Figura 5.3: Test dell'overhead introdotto nel farm stateless RMI: 320 immagini processate con filtro grafico Blur

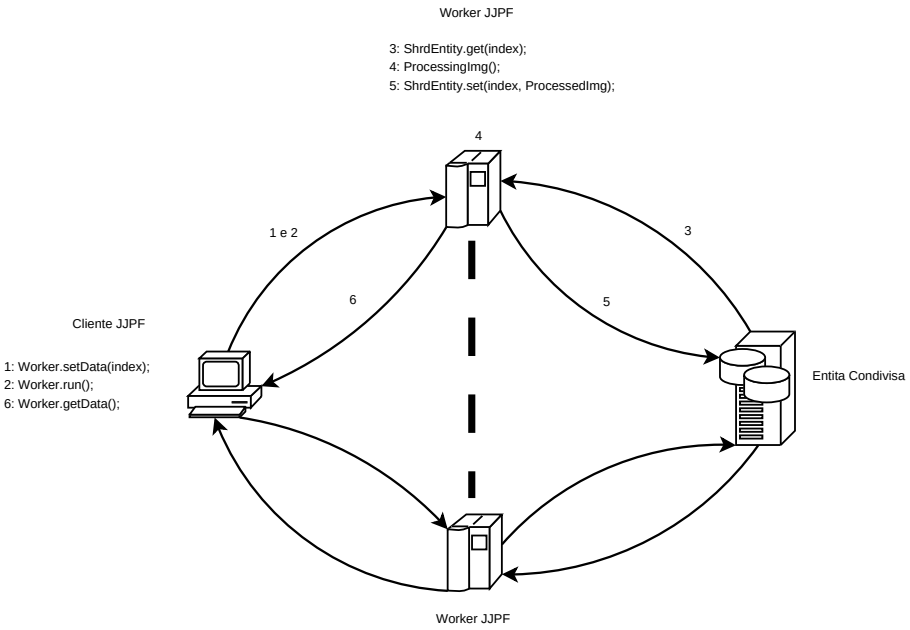


Figura 5.4: Schema Logico della prova di scalabilità strutturata come farm con stato

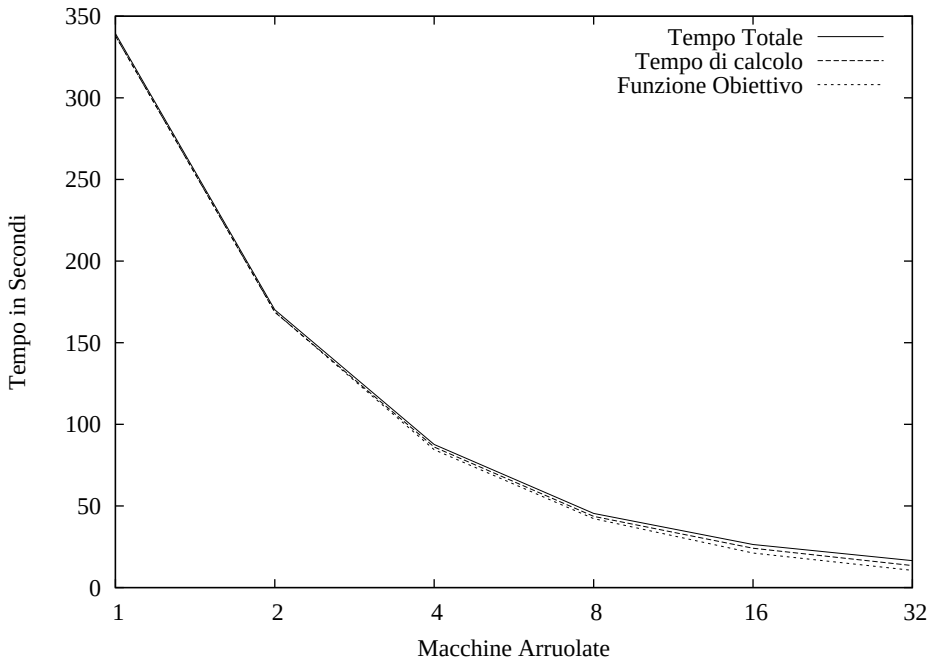


Figura 5.5: Test di Scalabilità del farm stateless Hybrid: 320 immagini processate con filtro grafico Blur

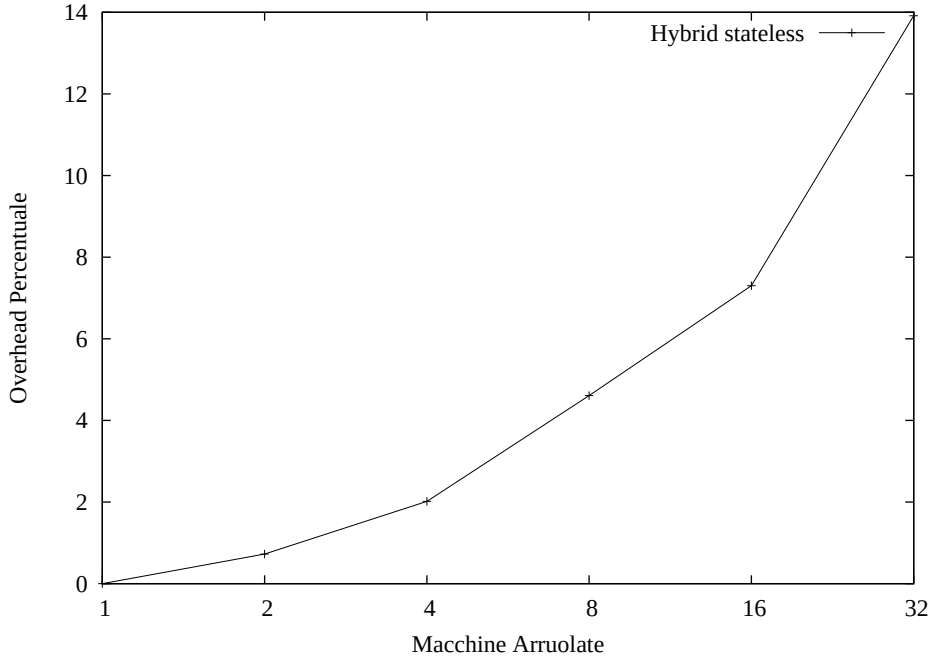


Figura 5.6: Test dell'overhead introdotto nel farm stateless Hybrid: 320 immagini processate con filtro grafico Blur

il meccanismo di caricamento dinamico distribuito offerto da Jini, nel secondo (Figura 5.9 a pagina 85) basandosi sul sistema hybrid da noi sviluppato. Osservando attentamente entrambe le figure è possibile notare che il tempo richiesto per portare a termine la computazione è leggermente superiore rispetto ai casi precedenti. Infatti strutturando l'applicazione parallela come un farm con stato si introduce l'overhead necessario alla ricerca dell'entità condivisa fra i worker; inoltre alle comunicazioni tra i worker e il gestore del farm vanno ad aggiungersi quelle tra i worker e l'entità condivisa. Anche in questi due casi i valori di scalabilità misurati sono soddisfacenti. Riguardo agli overhead registrati (Figure 5.8 e 5.10), la situazione è analoga ai due casi precedenti, infatti anche qui JJPF riesce a garantire una buona efficienza fino all'uso parallelo di sedici macchine. Superata questa soglia l'overhead sale a circa il 18%. In questi ultimi due test l'efficenza ottenuta quando si utilizzano trentadue macchine in parallelo è inferiore a quella degli esempi precedenti a causa del maggior numero di messaggi in transito sulla rete.

In Figura 5.11 è mostrata la banda di elaborazione garantita dalla nostra applicazione parallela, espressa in task calcolati per secondo. Come è possibile vedere, le applicazioni strutturate secondo il paradigma farm stateless garantiscono prestazioni superiori; tuttavia quando il numero di calcolatori arruolati

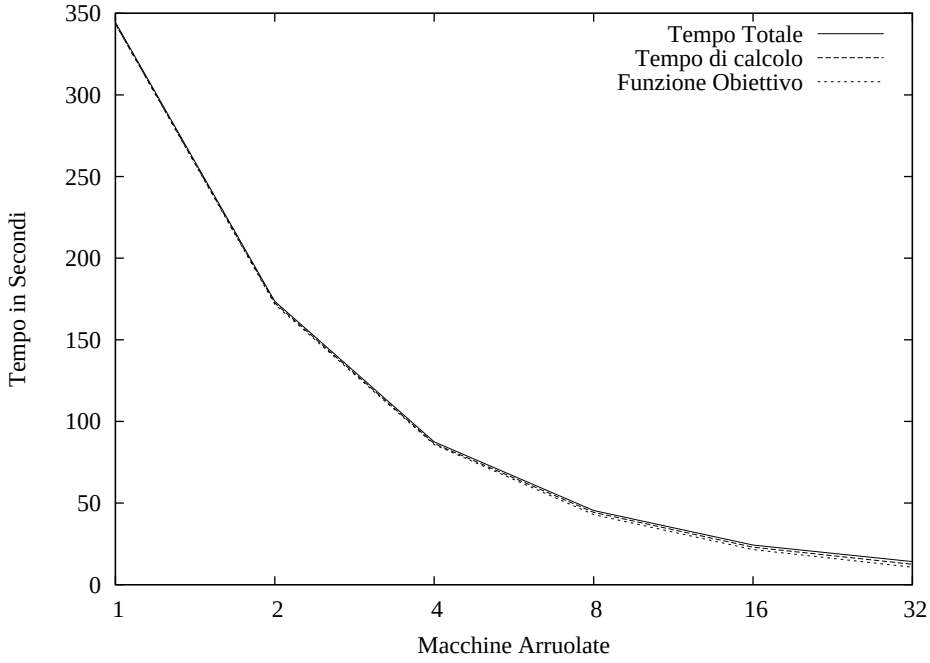


Figura 5.7: Test di Scalabilità del farm con stato RMI: 320 immagini processate con filtro grafico Blur

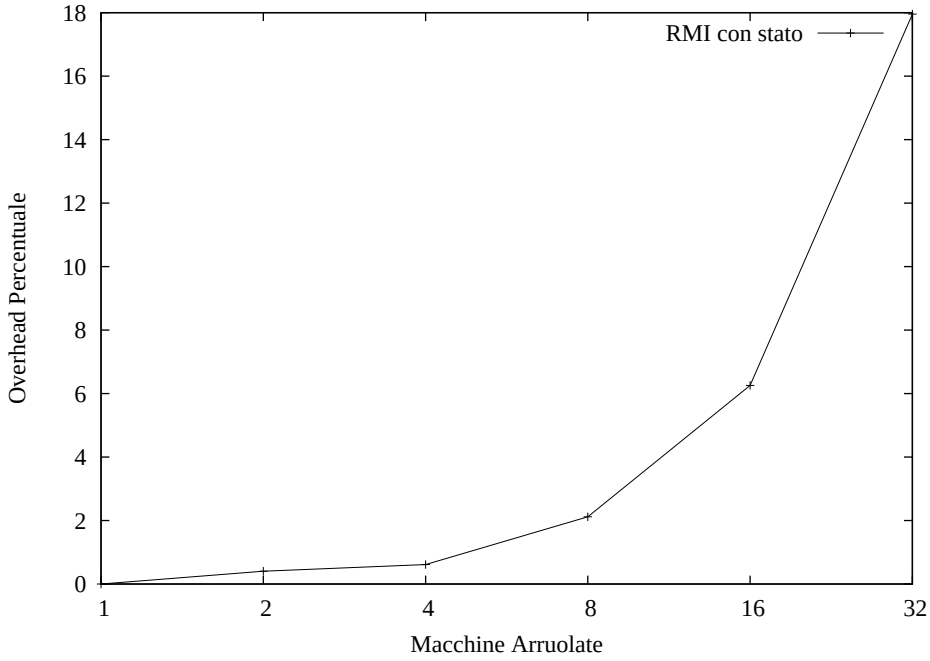


Figura 5.8: Test dell'overhead introdotto nel farm con stato RMI: 320 immagini processate con filtro grafico Blur

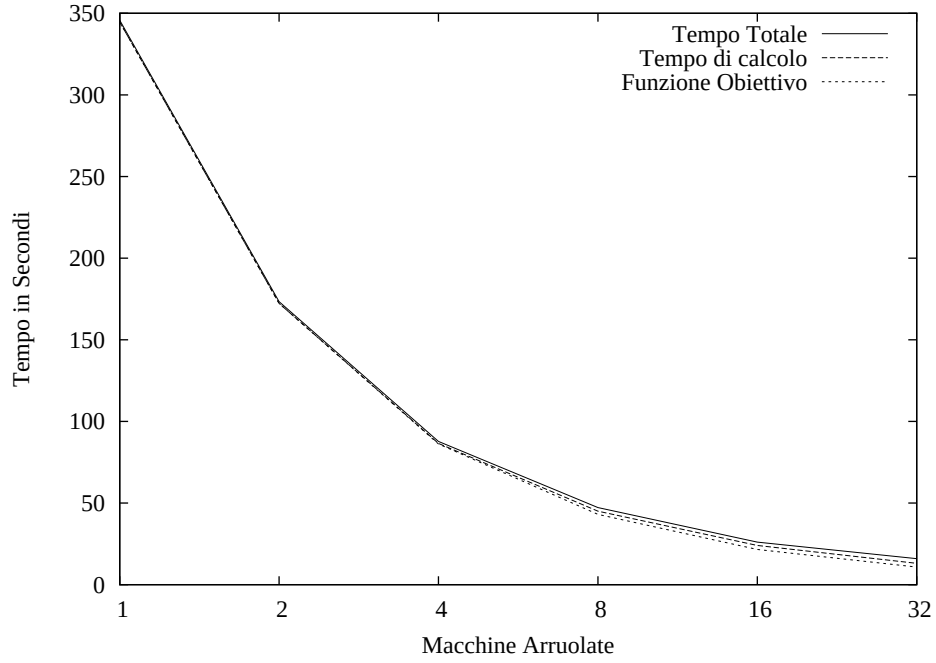


Figura 5.9: Test di Scalabilità del farm con stato Hybrid: 320 immagini processate con filtro grafico Blur

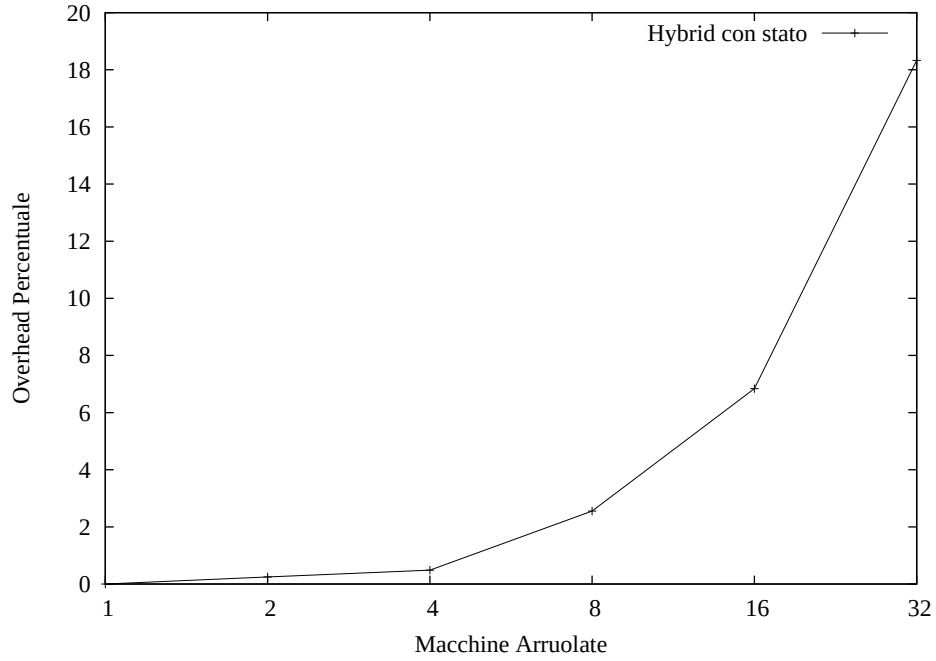


Figura 5.10: Test dell'overhead introdotto nel farm con stato Hybrid: 320 immagini processate con filtro grafico Blur

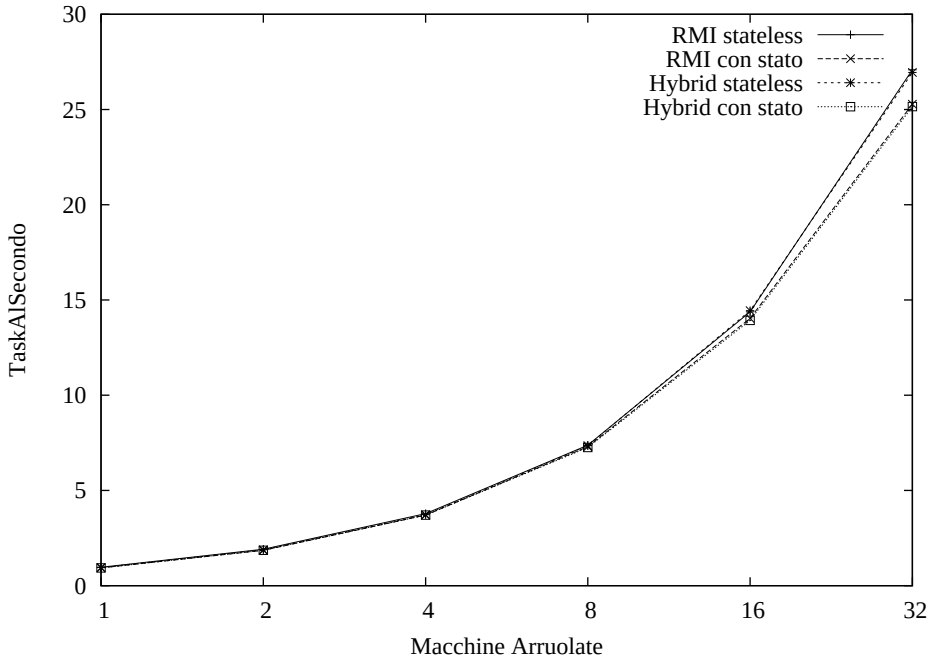


Figura 5.11: Task al Secondo

rimane sotto a sedici la differenza tra la banda offerta dai due schemi paralleli è esigua. Ciò accade perchè fino all'uso di sedici calcolatori in parallelo la struttura di interconnessione tra i worker non costituisce un collo di bottiglia.

Per l'ultimo dei test di scalabilità abbiamo deciso di sviluppare un programma leggermente diverso dai precedenti: siamo passati da una grana 29 ad una grana 185, aumentando il numero di filtri applicati alle immagini, ed abbiamo portato ad 800 il numero delle immagini da processare. L'applicazione è stata strutturata come un farm stateless ed ha utilizzato per il caricamento dinamico distribuito delle classi il sistema standard di Jini.

I risultati di scalabilità ottenuti in questo test sono buoni: osservando la Figura 5.12 nella pagina successiva è possibile notare che la linea del tempo totale e quella della funzione obiettivo avanzano quasi di pari passo, restando molto vicine anche quando il numero di worker arruolati sale a cinquantacinque. In Figura 5.13 è mostrato l'andamento dell'overhead relativo a quest'ultimo test. Come è possibile notare, una grana piu "grossa" ha contribuito a ridurre la degradazione della performance dovuta alle comunicazioni. Inoltre un maggior numero di task da calcolare ha reso meno incidenti i tempi per l'arruolamento e il disarruolamento delle risorse computazionali sulle performance totali.

I test di descritti in questa sezione hanno evidenziato i valori di scalabilità

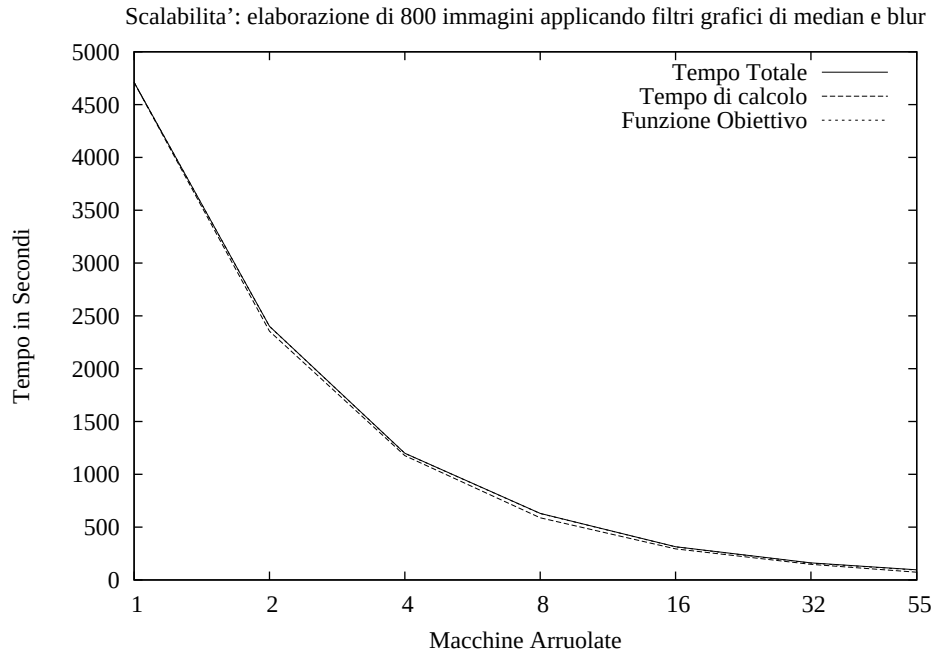


Figura 5.12: Test di Scalabilità del farm stateless RMI: 800 immagini processate con filtro Blur e Median

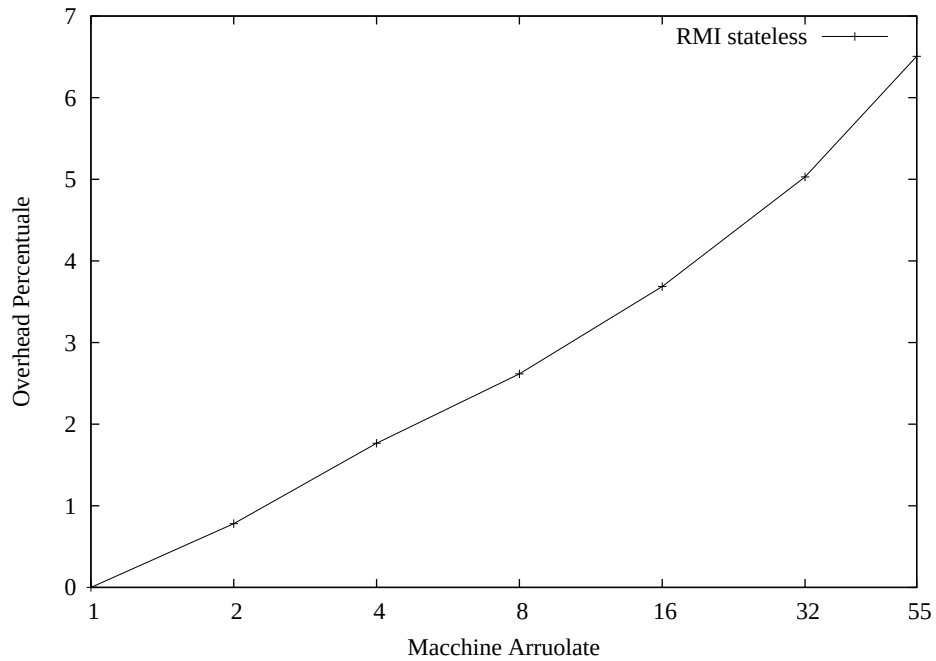


Figura 5.13: OverHeadTest dell'overhead introdotto nel farm stateless RMI: 800 immagini processate con filtro grafico Blur e Median

ottenuti dalle nostre applicazioni di prova realizzate utilizzando JJPF. I risultati ottenuti sono buoni, il framework realizzato è riuscito ad offrire un buon livello di scalabilità anche quando il rapporto tra il tempo di calcolo e quello di comunicazione è sceso sotto a trenta. Inoltre utilizzando una grana di calcolo adeguata all'ambiente in cui abbiamo eseguito i test, dove le macchine utilizzate come worker non sono dedicate esclusivamente a tale scopo, abbiamo ottenuto ottimi valori di scalabilità anche quando il numero di risorse computazionali arruolate ha superato la soglia delle cinquanta macchine.

5.3.2 Test di affidabilità

In ambienti fortemente variabili, dove la rimozione e l'aggiunta di risorse computazionali sono eventi frequenti, un framework di programmazione parallela e distribuita deve riuscire a gestire correttamente la disconnessione dei worker arruolati. In particolare deve riuscire a recuperare i dati spediti alle macchine scomparse e portare comunque a termine la computazione.

Per verificare se JJPF offre queste funzionalità abbiamo portato a termine alcuni test di affidabilità. Per i test abbiamo implementato una semplice classe Java, `TestAff` (§ 4.3), il cui comportamento è banale: prende in input un numero intero e lo restituisce in output concatenato alla stringa "TestAff".

Utilizzando il *cliente* JJPF abbiamo caricato la classe di test nei worker disponibili ed abbiamo avviato la computazione.

Abbiamo scelto di effettuare tre tipi di test: nel primo abbiamo rimosso un esiguo numero di worker, nel secondo un numero elevato di worker e nel terzo il *LookupService*.

- Il primo tipo di prove era destinato a controllare che JJPF rispettasse minimi requisiti di affidabilità: durante lo svolgimento della prova abbiamo rimosso un numero di worker non superiore al 10% del totale delle risorse computazionali arruolate. JJPF ha reagito all'eliminazione dei worker recuperando i dati spediti ai Servizi scomparsi e portando a termine correttamente la computazione.

- Dopo aver verificato che JJPF rispettasse i requisiti di affidabilità descritti al punto 1, ci siamo spinti oltre, provando ad eliminare dalla computazione un elevato numero di worker, anche in questo caso JJPF ha mostrato di tollerare bene la scomparsa di risorse di calcolo. In alcune prove sono stati eliminati tutti i worker arruolati dal *cliente*, in questo caso JJPF ha sospeso la computazione e l'ha ripresa appena si sono rese disponibili nuove risorse di calcolo.

- L'ultima classe di prove aveva come obiettivo l'analisi dell'affidabilità qualora venisse meno il supporto offerto dal *LookupService*. In questo caso i test effettuati sono stati due, nel primo abbiamo rimosso soltanto il *LookupService*, nel secondo sia il *LookupService* sia una buona parte di risorse di calcolo. In entrambe le situazioni la computazione è stata portata a termine con successo.

5.3.3 Test di dinamicità

Dopo esserci assicurati che il framework realizzato riesca a portare a termine la computazione anche quando parte delle risorse di calcolo iniziali non siano più disponibili, siamo passati all'analisi delle performance offerte da JJPF nei frangenti di aggiunta e rimozione di risorse computazionali.

Durante i test si è passati da cinque a dieci calcolatori per esaminare la reazione di JJPF all'aggiunta di risorse computazionali e da dieci a cinque worker per valutare quanto velocemente il framework realizzato riesca ad adeguarsi alla scomparsa di risorse di calcolo.

Le prove di dinamicità sono state realizzate utilizzando uno dei programmi di image processing introdotti nel paragrafo sui test di scalabilità. I test effettuati sono stati di quattro tipi, diversificati dal paradigma parallelo utilizzato e dal numero di task calcolati. La prime due prove (figure 5.14 e 5.15) sono state mirate a valutare la dinamicità offerta dal paradigma farm stateless di JJPF nel processing di 800 immagini.

Per la prima delle due siamo partiti utilizzando cinque calcolatori e aggiungendone altri cinque a circa metà del calcolo, nell'altra siamo partiti con dieci e siamo passati a cinque approssimativamente a metà della computazione.

I risultati ottenuti dalla prima prova (figura 5.14 a pagina 91) sono soddisfacenti, infatti, per portar a termine la prima metà degli 800 task, utilizzando cinque calcolatori, JJPF ha impiegato circa 490 secondi, dopo aver raddoppiato il numero di risorse computazionali il calcolo dei task rimanenti è stato portato a termine in circa 250 secondi. Se assumiamo che il tempo necessario al calcolo di ogni task sia costante, per completare l'intera computazione utilizzando cinque calcolatori, sarebbero stati necessari $490 \cdot 2$ secondi, quindi un sistema ottimo in termini di dinamicità che fosse passato dall'uso di cinque a dieci macchine, per portare a termine l'intera computazione avrebbe impiegato $(490 \cdot 2) \cdot \frac{3}{4}$. Rapportando i valori ideali a quelli ottenuti sperimentalmente otteniamo un'efficienza pari a:

$$\epsilon = \frac{(490 \cdot 2) \cdot \frac{3}{4}}{(490 + 250)} \approx \frac{735}{740} \approx 0.99$$

Nella seconda prova (figura 5.15 nella pagina successiva) abbiamo iniziato la computazione utilizzando dieci calcolatori, a circa metà del calcolo abbiamo ridotto questo numero, dimezzando il numero di risorse computazionali arruolate.

Anche in questo caso i risultati sperimentali ottenuti sono buoni. Durante l'esecuzione di questa prova, JJPF ha portato a termine il calcolo della prima metà dei task approssimativamente in 246 secondi, mentre per il calcolo degli ultimi 400 ha impiegato circa 496 secondi.

Se paragoniamo i valori ottenuti a quelli del sistema ideale di riferimento definito nella prova precedente, l'efficenza calcolata vale:

$$\epsilon = \frac{(490 \cdot 2) \cdot \frac{3}{4}}{(496 + 246)} \approx \frac{735}{742} \approx 0.99$$

Esistono alcune differenze tra i tempi parziali di questa prova e la precedente, le cause sono essenzialmente due:

1. quando un worker viene rimosso dalla computazione, il lavoro parziale compiuto sull'ultimo task ricevuto viene perso;

2. all'avvio della seconda prova, erano registrati presso il *LookupService* dieci calcolatori, nella prima prova, invece, dall'istante in cui abbiamo cominciato ad introdurre gli ulteriori worker, a quello in cui tutti e cinque si sono registrati presso il *LookupService* sono trascorsi alcuni secondi.

Negli due test successivi (figure 5.17 a pagina 93 e 5.16 a pagina 92), abbiamo valutato il grado di dinamicità offerto dall'implementazione JJPF del farm con stato. L'algoritmo usato per questi due test è il medesimo utilizzato in precedenza mentre i dati in input sono stati ridotti, da 800 a 200.

Nella prima della due prove siamo partiti utilizzando cinque calcolatori e aggiungendone altri cinque a circa metà della computazione. Nella seconda siamo partiti da dieci worker e giunti a metà del calcolo abbiamo ridotto il loro numero a cinque. Le considerazioni fatte per le prove precedenti, valgono anche per queste due, tuttavia, avendo notevolmente ridotto il numero di dati in input, il transitorio necessario per passare da cinque a dieci worker e da dieci a cinque, influisce in maniera più significativa.

Nella prima di queste ultime due prove, JJPF ha portato a termine il calcolo della prima metà dei task in circa 125 secondi, mentre per il calcolo dei 100 rimanenti ha impiegato approssimativamente 65 secondi.

Quindi l'efficienza ottenuta è:

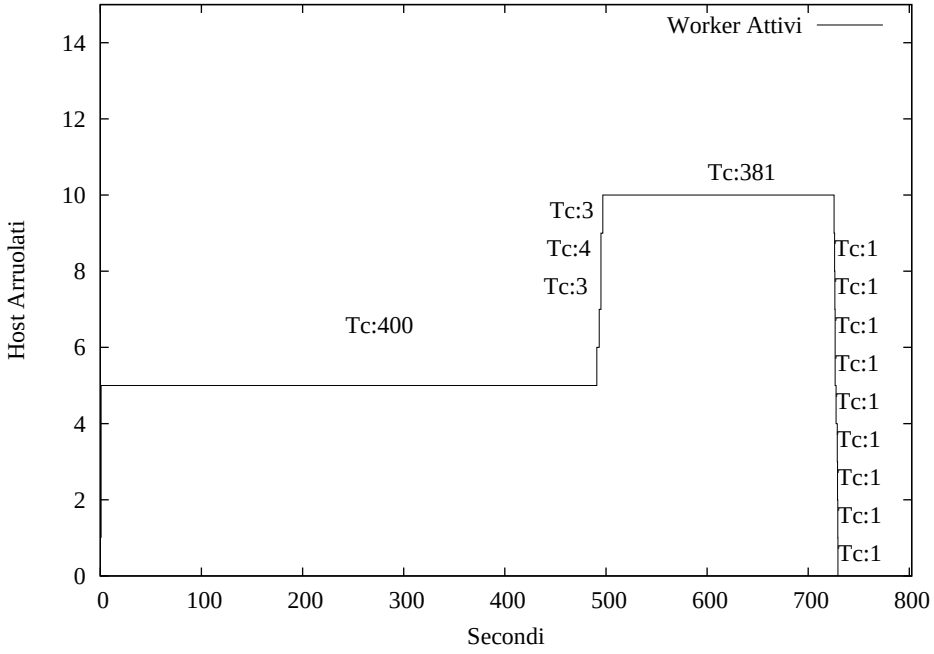


Figura 5.14: 5 servizi iniziali più 5 a metà del calcolo - farm stateless RMI

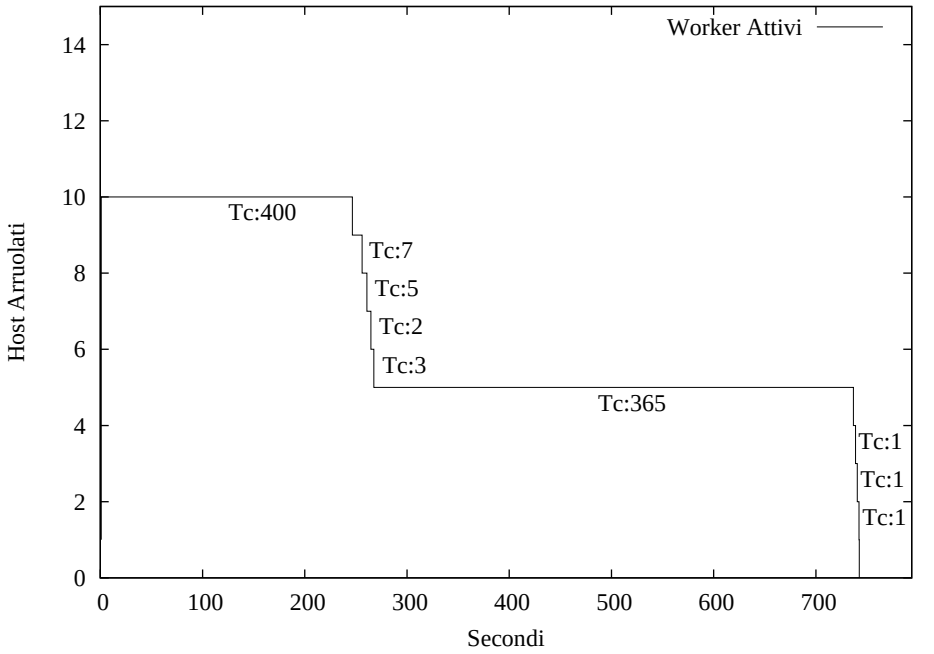


Figura 5.15: 10 servizi iniziali di cui 5 persi a metà della computazione - farm stateless RMI

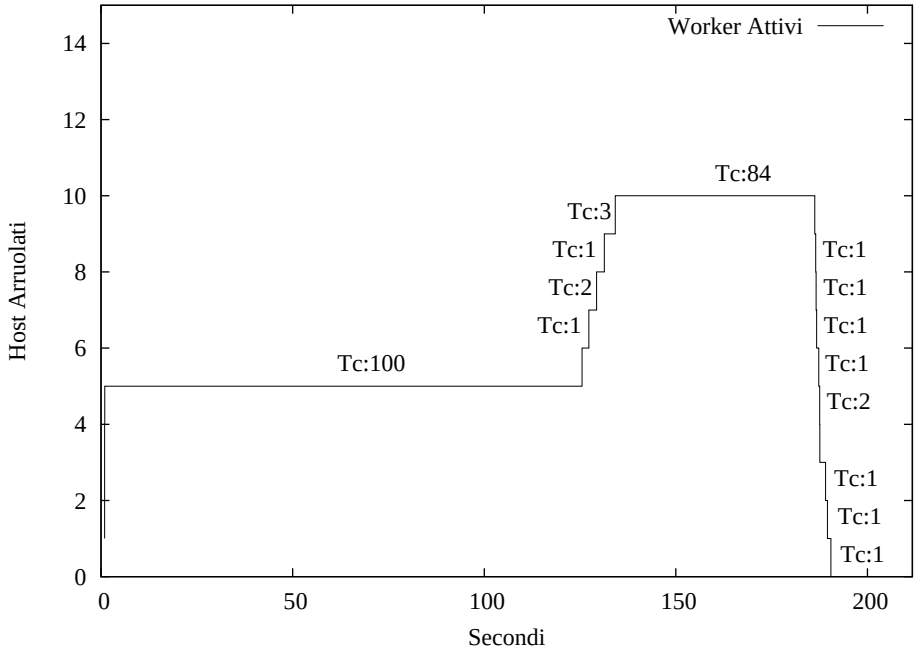


Figura 5.16: 5 servizi iniziali più 5 a metà della computazione - farm con stato RMI

$$\varepsilon = \frac{(125 \cdot 2) \cdot \frac{3}{4}}{125 + 65} = \frac{187.5}{190} \approx 0.986$$

Nella seconda prova, per completare i primi 100 task sono stati necessari circa 63 secondi mentre per portare a termine il calcolo dei task rimanenti sono stati necessari 128 secondi.

Prendendo come sistema ideale di riferimento quello definito nella prova precedente, otteniamo un'efficenza pari a:

$$\varepsilon = \frac{(125 \cdot 2) \cdot \frac{3}{4}}{128 + 63} = \frac{187.5}{191} \approx 0.981$$

Le prove di dinamicità eseguite hanno dimostrato che il framework realizzato reagisce prontamente all'aggiunta e alla rimozione di risorse computazionali. Il paragone con un sistema ideale ha mostrato che JJPF offre un grado di dinamicità elevato.

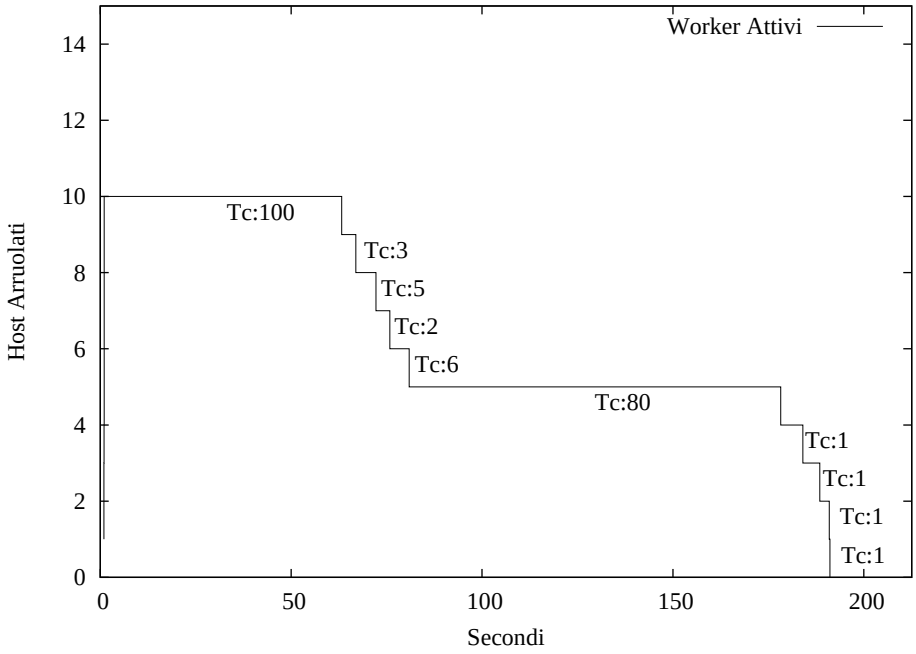


Figura 5.17: 10 servizi iniziali di cui 5 persi a metà calcolo - farm con stato RMI

Capitolo 6

Conclusioni

In questa tesi abbiamo valutato la possibilità di realizzare un ambiente per il calcolo parallelo strutturato che, utilizzando Jini come middleware di supporto, consenta di sviluppare applicazioni parallele secondo il paradigma farm stateless ed una sua variante che permetta la condivisone di dati fra i worker.

Per valutare oggettivamente quanto Jini si presti ad essere utilizzato in ambito parallelo e distribuito, abbiamo sviluppato un framework, il JJPF, che permette di realizzare, in modo semplice e immediato, programmi che sfruttino il parallelismo nella forma farm e farm con stato.

I risultati sperimentali ottenuti sono incoraggianti: JJPF ha dimostrato di soddisfare ampiamente i requisiti di scalabilità che ci eravamo posti come obiettivo. In particolare il framework realizzato ha garantito ottimi valori di scalabilità fino all'uso contemporaneo di cinquantacinque calcolatori. Inoltre, i test hanno mostrato che JJPF reagisce rapidamente all'aggiunta e alla rimozione di risorse computazionali, evidenziando un elevato livello di affidabilità e dinamicità.

Un altro importante risultato di questa tesi è la proposta, analisi e realizzazione di hybrid, un meccanismo di caricamento delle classi dinamico distribuito alternativo a quello usato da Jini Extensible Remote Invocation. Nonostante i risultati di performance offerti dal modello proposto siano allineati, a quelli offerti da JERI, hybrid offre una maggiore flessibilità in quanto non presuppone, sulla macchina *cliente*, l'esistenza di un webserver per il trasferimento delle classi. L'eliminazione di questo vincolo, congiuntamente all'uso del paradigma farm con stato, nel quale i worker sono in grado di ottenere i dati di input da una sorgente di informazioni diversa dal *cliente*, apre la strada a nuove interessanti possibilità, quali ad esempio l'esecuzione del processo *cliente* su palmari o semplici dispositivi mobili che dispongano di banda di comunicazione e potenza

computazionale ridotta.

La realizzazione di hybrid ha richiesto una consistente attività di documentazione nel settore del java classloading distribuito. Tale attività, resa difficile dalla scarsità di documentazione reperibile, ha portato alla realizzazione di un classloader destinato al caricamento delle classi *cliente* sui worker arruolati nella computazione parallela.

Un'ulteriore attività svolta nel contesto di questa tesi, riguarda la progettazione e lo sviluppo di un buon numero di script Perl. Le motivazioni che hanno portato alla loro realizzazione sono molteplici: alcuni sono stati scritti per l'analisi dei dati sperimentali raccolti, altri per la ricerca delle macchine disponibili, altri per avviare l'esecuzione dei servizi JJPF da usare durante i test.

JJPF lascia aperte di fronte a se numerose vie di espansione. Una di queste riguarda l'implementazione di un maggior numero di paradigmi paralleli, sia del tipo task parallel sia data parallel. Per quanto riguarda i primi una interessante strada percorribile è l'introduzione del concetto di "nodo successore", ovvero dare la possibilità al *cliente* di specificare per ogni worker il destinatario dei risultati della sua computazione, in modo da riuscire a realizzare il paradigma pipeline. Invece relativamente ai paradigmi data parallel è auspicabile una evoluzione del farm con stato che consenta la comunicazione diretta tra worker riducendo il numero di accessi concorrenti ai dati condivisi così da migliorarne le performance. Un'ultima, ma non meno importante, direzione è data dallo sviluppo in JJPF dello "skeleton generico" (parmod) introdotto in ASSIST [MD01], in questo modo si offrirebbe al programmatore un ampia possibilità di scelte e di specializzazioni nella strutturazione di un programma parallelo.

Infine, un'ulteriore via percorribile nello sviluppo del JJPF è l'integrazione con CORBA: ciò permetterebbe ai Servizi realizzati con JJPF di essere utilizzati anche da clienti scritti con altri linguaggi di programmazione e quindi renderlo utilizzabile anche da macchine sulle quali non sia disponibile una JVM.

Bibliografia

[ADT03] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[APJ02] Arpad Andics, Szabolcs Pota, and Zoltan Juhasz. JM: A jini framework for global computing. July 10 2002.

[Bet98] L. Bettini. Il classloader. http://www.mokabyte.com, March 1998.

[BP01] Jyoti Batheja and Manish Parashar. Adaptive cluster computing using javaspaces. July 20 2001.

[Col89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[COR04] Corba website. http://www.corba.org, 2004.

[Cra98] D. Crawford. *Gnuplot Guide*, 1998.

[Cre04] D. Creswell. Getting started with jini 2.0. http://v2getsmart.jini.org/, 2004.

[CSGG01] Autospaces L. L. C, Frank Sommers, Shahram Gh, and Shan Gao. Cluster-based computing with active, persistent objects on the web. July 31 2001.

[Dan99] M. Danelutto. Introduzione al perl. SEU - Pisa, 1999.

[Dan01] M. Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1), March 2001.

[Dan03] M. Danelutto. Adaptive task farm implementation strategies. volume 1, pages 100–107, 2003.

[DT02] M. Danelutto and P. Teti. Lithium: A structured parallel programming environment in Java. *Lecture Notes in Computer Science*, 2330, 2002.

[Eck03] B. Eckel. *Thinking in Java.* 2003.

[ES98] N. Patwardhan E. Siever, S. Spainhour. *Perl in a NutShell.* O'Reilly, first edition, 1998.

[FB87] M. Vanneschi F. Baiardi, A. Tomasi. *Architettura dei sistemi di elaborazione.* Franco Angeli, 1987.

[FNO] Network of workstations. http://www.lri.fr/ fci/RS-Anglais.html.

[IEE04] Ieee 1394. http://grouper.ieee.org/groups/1394/1/Documents/, 1995 – 2004.

[JBE] Jini by example. http://www.cswl.com/whiteppr/tutorials/jini.html.

[JBM+] Ceriel Jacobs, Henri Bal, Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Rutger Hofman, and Thilo Kielmann. Efficient java RMI for parallel programming.

[JFA04] Jini faq. http://www.artima.com/jini/faq.html, 2004.

[Jin] Distributed computing for plug-and-play network service configuration.

[JK01] Zoltan Juhasz and Laszlo Kesmarki. A Jini-based prototype metacomputing framework (research note). *Lecture Notes in Computer Science*, 1900:1171–??, 2001.

[JNJ03] *Jan Newmarch's Guide to JINI Technologies.* 2003.

[JOR04] Jini technology website. http://www.jini.org, 2004.

[JOT99] Jini technology architectural overview. Technical report, Sun Microsystems, 1999.

[JRW03] Jini resources. http://www.jini.org/resources/, 2003.

[LB98] D. Cappetta L. Bettini. Networkclassloader in java. http://www.mokabyte.com, 1998.

[MB01] C. A. G. Ferraz M. B.D'Amorim. Designing jini distributed services - a framework to support the development of reliable component networks. 2001.

[MD01] L. Vaglini M. Vanneschi D. Guerri M. Lettere M. Danelutto, P. Ciullo. *Ambiente ASSIST: modello di programmazione e linguaggio di coordinamento ASSIST-CL (versione 1.0).* ASI-PQE2000, 2001.

[NOW] Berkeley network of workstations. http://now.cs.berkeley.edu/.

[PB02] Constantine D. Polychronopoulos and Fabian Breg. Java virtual machine support for object serialization. February 09 2002.

[Pel98] S. Pelagatti. *Structured development of parallel programs.* Taylor I& Francis, Inc., 1998.

[RF99] J. Mogul H. Frystyk L. Masinter P. Leach T. Berners-Lee R. Fielding, J. Gettys. Hypertext transfer protocol – http/1.1. RFC2616, June 1999.

[RMI04a] Rmi over iiop. http://java.sun.com/products/rmi-iiop/, 2004.

[RMI04b] Rmi over ssl. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/socketfactory/index.html, 2004.

[Sch03] A. Schaefer. Inside class loaders. http://www.onjava.com/pub/a/onjava/2003/11/12/classloader.html, 2003.

[Sci97] Department Of Computer Science. Object-based distributed systems. December 08 1997.

[SCK94] A. Wollrath G. Wyant S. C. Kendall, J. Waldo. A note on distributed computing. Technical report, Sun Microsystems, 1994.

[SO99] H. Wong S. Oaks. *Java Threads.* O'Reilly, second edition, 1999.

[Som03] F. Sommers. Call on extensible rmi - an introduction to jeri. http://www.javaworld.com, December 2003. jiniology.

[SRW04] Java remote method invocation. http://java.sun.com/products/jdk/rmi/, 2004.

[TBL96] H. Frystyk T. Berners-Lee, R. Fielding. Hypertext transfer protocol – http/1.0. RFC 1945, May 1996.

[TC98] N. Torkington T. Christiansen. *Perl Cookbook.* O'Reilly, first edition, August 1998.

[UL00] Tampere University and Tommi Lukkarinen. Tools for distributed software. August 16 2000.

[Ven02] B. Venners. Jini extensible remote invocation, a conversation with bob scheifler. http://www.artima.com/intv/jeri.html, 2002.

[WJT99] Why jini technology now? Technical report, Sun Microsystems, 1999.

[WW04] J. Waldo and A. Wollrath. An overview of rmi applications. http://java.sun.com/docs/books/tutorial/rmi/overview.html, 2004.

[ZJ03] P. Kacsuk Z. Juhasz. Jgrid a jini-based universal service grid. http://www.irt.vein.hu/jgrid, 2003.

Appendice A

Manuale Utente

A.1 Requisiti

Per utilizzare JJPF è necessario avere installato sulla propria macchina:

- la versione 1.4.0 di Java SDK o una successiva;
- una versione del Jini toolikit maggiore o uguale alla 2.0;

A.2 Installazione

Per il corretto funzionamento del framework JJPF, è necessario eseguire alcune semplici operazioni di installazione:

1. copiare l'intero package `jjpf` nella directory in cui è contenuta l'applicazione che fa uso del nostro framework;
2. assicurarsi che la variabile d'ambiente `CLASSPATH` contenga i riferimenti agli archivi Java (JAR) necessari al corretto funzionamento di Jini: `jini-core.jar, jini-ext.jar, sun-util.jar`;
3. aggiungere a questi ultimi `start.jar` e `tools.jar`, entrambi contenuti nella sottodirectory `lib` del Jini toolkit;
4. aggiungere inoltre nella variabile `CLASSPATH` un riferimento al JAR `tools.jar` contenuto nella directory `lib` del Java Software Development Kit;

Di seguito è riportato un esempio che mostra come portare a termine i punti 2, 3 e 4 sopra descritti se si utilizza una shell Bash. Nell'esempio si assume che lo SDK Java sia installato nella directory `/usr/lib/java` e che il Jini toolkit sia

contenuto in `/usr/lib/java/jini2`.

```
export JINI_HOME=/usr/lib/java/jini2

export CLASSPATH=$CLASSPATH:$JINI_HOME/lib/jini-core.jar

export CLASSPATH=$CLASSPATH:$JINI_HOME/lib/jini-ext.jar

export CLASSPATH=$CLASSPATH:$JINI_HOME/lib/sun-util.jar

export CLASSPATH=$CLASSPATH:$JINI_HOME/lib/start.jar

export CLASSPATH=$CLASSPATH:$JINI_HOME/lib/tools.jar

export CLASSPATH=$CLASSPATH:/usr/lib/java/lib/tools.jar
```

A.3 Configurazione

Sia il *cliente* che i Servizi di JJPF implementano un semplice Server Web. Per la modificare la configurazione di quest'ultimo non è necessario apportare cambiamenti al codice sorgente di JJPF, è sufficiente accedere al file di configurazione (ad esempio `jjpf.service.hybrid.config` per la configurazione dei servizi realizzabili utilizzando le classi contenute nel package `jjpf.service.hybid`) per poter impostare:

- la porta su cui far partire il server Web;
- la directory base del server Web.

Il protocollo di invocazione remota utilizzato dagli attori del JJPF è il JERI. Accedendo ai file di configurazione è possibile cambiare il protocollo di trasporto da utilizzare senza dover modificare e ricompilare i sorgenti del JJPF.

Infine, dai file di configurazione, è possibile scegliere il nome del file contenente le politiche di sicurezza da utilizzare. Il Frammento di codice mostra un esempio di file di configurazione

Apportando alcune modifiche a questo file è possibile, ad esempio, cambiare la porta su cui far partire il server Web semplicemente modificando il valore assegnato alla variabile `classServerPort` o decidere quale politiche di sicurezza adottare, assegnando il nome del file dove sono definite alla variabile `securPolicy`.

Frammento di Codice 7 File di Configurazione del *servizio* rmi

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import com.sun.jini.config.ConfigUtil;
Service{
   exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                    new BasicILFactory());
   securPolicy = new String("jjpf/service/rmi/rmi.start.policy");
   localhost = ConfigUtil.getHostName();

   classServerPort = "8080";
   classServerBaseDir = ".";
   codebase = ConfigUtil.concat(new String[]{
                                             "http://",
                                             localhost,
                                             ":",
                                             classServerPort,
                                             "/"
                                            }
                                );
}
```

A.4 Guida d'uso

JJPF mette a disposizione del programmatore le classi necessarie alla creazione di clienti (§ 3.2.2.2 a pagina 41), servizi (§ 3.2.2.1 a pagina 39) ed entità condivise (§ 3.2.2.3 a pagina 43). In questa guida illustreremo il modo corretto di utilizzare tali classi.

A.4.1 Creazione di Servizi

Per creare in modo semplice e rapido un *servizio* JJPF, è possibile utilizzare le classi dei package `jjpf.service.rmi` e `jjpf.service.hybrid`. Nel primo caso otteniamo un *servizio* che, per il caricamento dinamico e distribuito delle classi, utilizza il meccanismo standard di Jini. Nell'altro il *servizio* creato sfrutta il meccanismo hybrid proposto in questa tesi.

Le procedure da portare a termine per la realizzazione di servizi del primo o del secondo tipo sono analoghe.

Nel Frammento di codice 8 sono mostrate le istruzioni necessarie alla creazione di un *servizio* `rmi`.

- Per prima cosa, è necessario istanziare un oggetto di tipo `RMIServiceRegister` che ricopre il ruolo di *service provider* (§ 2.3.2 a pagina 28). Per la creazione di questo è necessario passare come input al costruttore un

Frammento di Codice 8 Creazione di un *Servizio*

```
import jjpf.service.rmi.RMIService;
import jjpf.service.rmi.RMIServiceRegister;

public class NuovoServizioJJPF
{

   public static void main(String args[])
   {
      ...
      // Creazione del service provider
      RMIServiceRegister srv =
         new RMIServiceRegister(RMIService.class);

      // Ricerca dei LookupService
      srv.setMulticastGroups(LookupDiscovery.ALL_GROUPS);

      // Avvio del service provider
      srv.start();

      ...
   }

}
```

oggetto di tipo `class` associato alla classe a cui appartiene l'oggetto che definisce il *servizio* da attivare.

- Il passo successivo è la scelta degli indirizzi da utilizzare per cercare e contattare i *LookupService*, tale ricerca può esser fatta con protocolli multicast (metodo `setMulticastGroups`) o unicast (metodo `addUnicastLookup`).

- A questo punto è sufficiente invocare il metodo `start` sull'istanza del *service provider* creata, così da far partire il *servizio*.

Per creare un *servizio* che utilizza il meccanismo di caricamento dinamico e distribuito hybrid si procede nello stesso modo, l'unica differenza è data dal nome delle classi da utilizzare: `RMIService` è sostituito da `HybridService`, mentre `HybridServiceRegister` prende il posto di `RMIServiceRegister`.

A.4.2 Creazione di Entità Condivise

Per creare un'entità condivisa è sufficiente estendere la classe `jjpf.shared.SharedObject` e implementare i seguenti due metodi astratti:

- `void set(int index, Object obj)`

- `Object get(int index)`

I due metodi sono utilizzati dai worker per accedere all'entità condivisa. Il primo consente ad un generico worker di trasferire l'oggetto `obj` nello `SharedObject`, l'intero `index` è utilizzabile per specificare l'indice col quale riferire successivamente i dati salvati.

Per fare in modo che i worker trovino e usino l'entità condivisa è necessario effettuare l'operazione di *esportazione* e quella di *registrazione* di questa presso i *LookupService*. Il frammento di codice 9 mostra come portare a termine le procedure illustrate. Come è possibile notare è necessario creare un oggetto di tipo `SharedObjectRegister`: quest'ultimo svolge le medesime funzioni del *service provider* (§ 2.3.2 a pagina 28). Una volta creato tale oggetto è necessario specificare a JJPF dove cercare i *LookupService* tramite l'invocazione del metodo `setMulticastGroups`, che in input riceve un array di `java.lang.String` contenenti i nomi dei canali multicast, oppure `addUnicastLookup`, che in input riceve o un array di `java.lang.String` contenenti gli indirizzi delle macchine sulle quali è possibile trovare il *LookupService* o un oggetto di tipo `java.lang.String` ed un numero di porta a rappresentare la posizione del *LookupService* nella rete. Il passo successivo è la chiamata al metodo `start`, che da il via all'esecuzione dell'entità condivisa.

A.4.3 Creazione di Clienti

La realizzazione di un *cliente* passa attraverso allo sviluppo di due classi: una che implementa l'interfaccia `ProcessoIf` ed un'altra che si occupa di istanziare il *cliente* vero e proprio.

Il Frammento di codice 10 a pagina 107 mostra i passi da compiere per l'istanziazione di un *cliente*.

Per prima cosa è necessario creare i due oggetti di tipo `java.util.Collection`, destinati a contenere rispettivamente i dati di input e di output della applicazione parallela. Dopo aver creato questi due contenitori è necessario riempire la `Collection` di input con le informazioni da far processare ai worker che saranno arruolati nella computazione parallela.

Il secondo passo da compiere è la creazione dell'array delle classi che sono istanziate e invocate dai worker per l'esecuzione del codice scritto dallo sviluppatore che utilizza il *cliente* di JJPF. Per costruire tale insieme è possibile utilizzare il metodo statico `classDep` della classe `DependencyResolver`. Questo analizza la classe passata come parametro di input e restituisce un array contenente le classi che il codice del cliente tenterà di caricare una volta in esecuzione.

Se l'applicazione parallela che si vuole realizzare è di tipo farm con sta-

Frammento di Codice 9 Creazione di un'Entità condivisa

```
public class Main
{

   public static void main(String[] args)
   {
      try {

         // Creazione del Register del server (o Worker)
         SharedObjectRegister shr =
            new SharedObjectRegister(ClassCheEstSharedObject.class);


         // Ricerca dei LookupService
         shr.setMulticastGroups(LookupDiscovery.ALL_GROUPS);

         // Avvio dell'entità condivisa
         shr.start();

      } catch(Exception e)
      {
      }

   }

}
```

Frammento di Codice 10 Creazione di un *cliente*

```
import jjpf.client.BasicClient;
import jjpf.client.AbstractClient;
...

public class NuovoClienteJJPF
{

   public static void main(String args[])
   {
      ...

      // creazione vettori di input - output
      Collection datiInput = new Vector();
      Collection datiOutput = new Vector();

      ...

      // riempimento vettore con dati in input
      riempiCollectionConDatiInInput(datiInput);

      ...

      // creazione vettore di classi cliente
      Class[] arrayClassiCliente =
         DependencyResolver.classDep(classeCheEstendeProcessoIf.class);

      mettiComePrimoNellArrayLaClasseProcessoIf(arrayClassiCliente);

      ...

      BasicClient bc = new BasicClient(arrayClassiCliente,
                                       classeSharedObject,
                                       datiInput,
                                       datiOutput);

      bc.setLookup(AbstractClient.ALL_GROUPS);
      bc.start();
      bc.join();

      ...

      // analisi dei dati ottenuti
      esaminaRisultatoDellaComputazione(datiOutput);

      ...
   }

}
```

to, è necessario specificare al *cliente* JJPF l'oggetto `class` associato all'entità condivisa che i worker dovranno utilizzare.

A questo punto si passa alla creazione dell'oggetto *cliente* vero e proprio. Per semplificare e velocizzare la realizzazione dei *clienti*, JJPF mette a disposizione una classe di nome `BasicClient`, la quale è un `java.lang.Thread`. Questa si occupa della ricerca e dell'arruolamento delle risorse computazionali disponibili. Inoltre, `BasicClient` gestisce la totalità delle comunicazioni con i worker. Istanziando questa classe è necessario passare al costruttore i seguenti parametri:

- l'array delle classi cliente;
- l'oggetto `class` dell'entità condivisa se l'applicazione parallela è di tipo farm con stato, `null` altrimenti;
- la `Collection` in input;
- la `Collection` di output.

Dopo aver istanziato un oggetto di tipo `BasicClient` è necessario specificare al *cliente* JJPF a quali indirizzi cercare un *LookupService*. I metodi a disposizione sono `setLookup` e `addLookup`: il primo va usato per gli indirizzi di tipo multicast, l'altro per quelli di tipo unicast.

Dopo aver indicato al *cliente* dover cercare i *LookupService* possiamo finalmente far partire il *cliente* invocando il metodo `start`.

Infine, anche se non è necessario, è consigliabile invocare sull'oggetto appartenente alla classe `BasicClient` il metodo `join`, in questo modo il flusso di controllo dell'applicazione si ferma fino a quando il thread è attivo.

Quando l'array delle classi scritte dallo sviluppatore che utilizza il *cliente* di JJPF viene trasferito sui Worker, i Servizi JJPF istanziano un oggetto della classe che implementa `ProcessoIf`, contenuta nella prima posizione dell'array in input. Il Frammento di codice 11 mostra lo scheletro di una classe con questa struttura.

L'interfaccia `ProcessoIf` contiene i seguenti metodi:

- `Object getData()`
- `void setData(Object o)`
- `void run()`
- `void setSharedObject(SharedObjectIf soi)`

Frammento di Codice 11 Creazione di una classe che estende `ProcessoIf`

```
public class Processo implements ProcessoIf
{
   ...
   // Metodo tramite il quale ci vengono
   // passati i dati di input
   public void setData(Object obj)
   {
      ...
   }

   // Metodo che esegue la concatenazione
   public void run()
   {
      ...
   }

   // Metodo col quale questa classe restituisce
   // il risultato del calcolo
   public Object getData()
   {
      return strConcatenata;
   }

   // Metodo usato per ottenere il proxy
   // per le comunicazioni con una entità
   // condivisa.
   public void setSharedObject(SharedObjectIf ssi)
   {
      ...
   }

}
```

Il metodo `setData` è utilizzato dal *cliente* JJPF per passare le informazioni al codice in esecuzione sui worker. Dopo aver trasferito queste informazioni, il *cliente*, chiama il metodo `run`, che provoca il processing dei dati spediti. Infine utilizzando `getData`, il *cliente* scarica dal worker i risultati del calcolo.

Il metodo `setSharedObject` viene usato dal worker sull'oggetto di tipo `ProcessoIf` per assegnargli l'oggetto che opera da *proxy* fra il worker e lo `SharedObject`.

A.5 Concetti Avanzati

JJPF consente di realizzare clienti e servizi diversi da quelli descritti fino ad adesso: ciò semplicemente estendendo le classi astratte `AbstractLockableService`, `ServiceRegistrationManager` e `AbstractClient`. Ad esempio è possibile far condividere i medesimi worker a più applicazioni realizzando un sistema di locking e unlocking differente rispetto a quello definito in `AbstractLockableService`. All'interno del nostro framework questa possibilità è sfruttata da `jjpf.shared.SharedObject` in quanto, come detto in § 4.2.3, salta un passo nella catena di ereditarietà ed eredita da `ServiceRegistrationManager`.

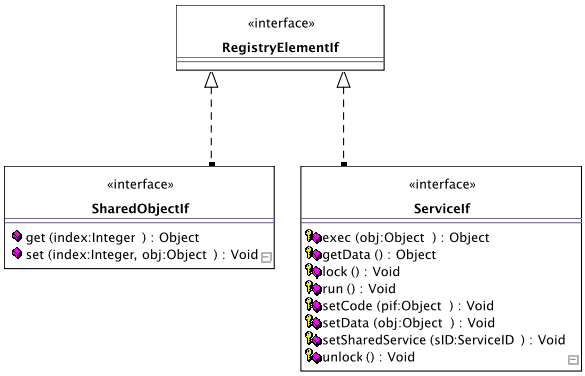
Per quanto riguarda il *cliente* JJPF esiste la possibilità di estendere la classe `AbstractClient` anzichè utilizzare `BaiscClient`, ciò consente:

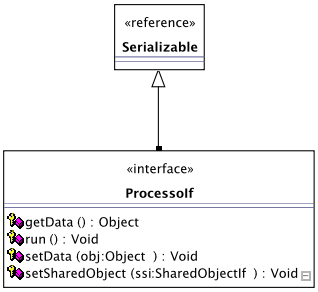
- una gestione personalizzata della fault tolerance;
- la creazione di un logger più complesso che, ad esempio, memorizzi le informazioni registrate in formato XML.

Appendice B

Diagrammi UML di JJPF

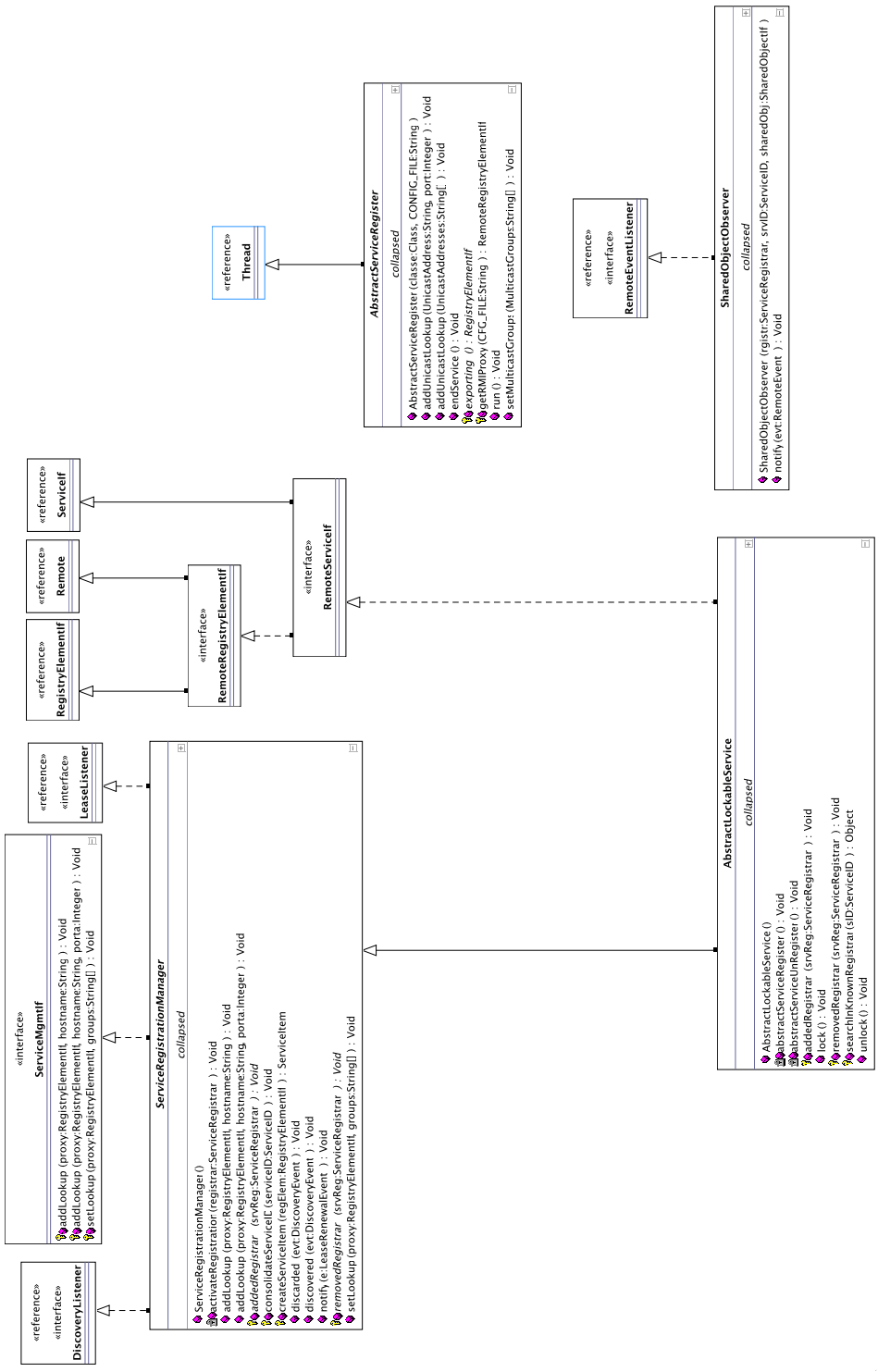
B.1 Package Common



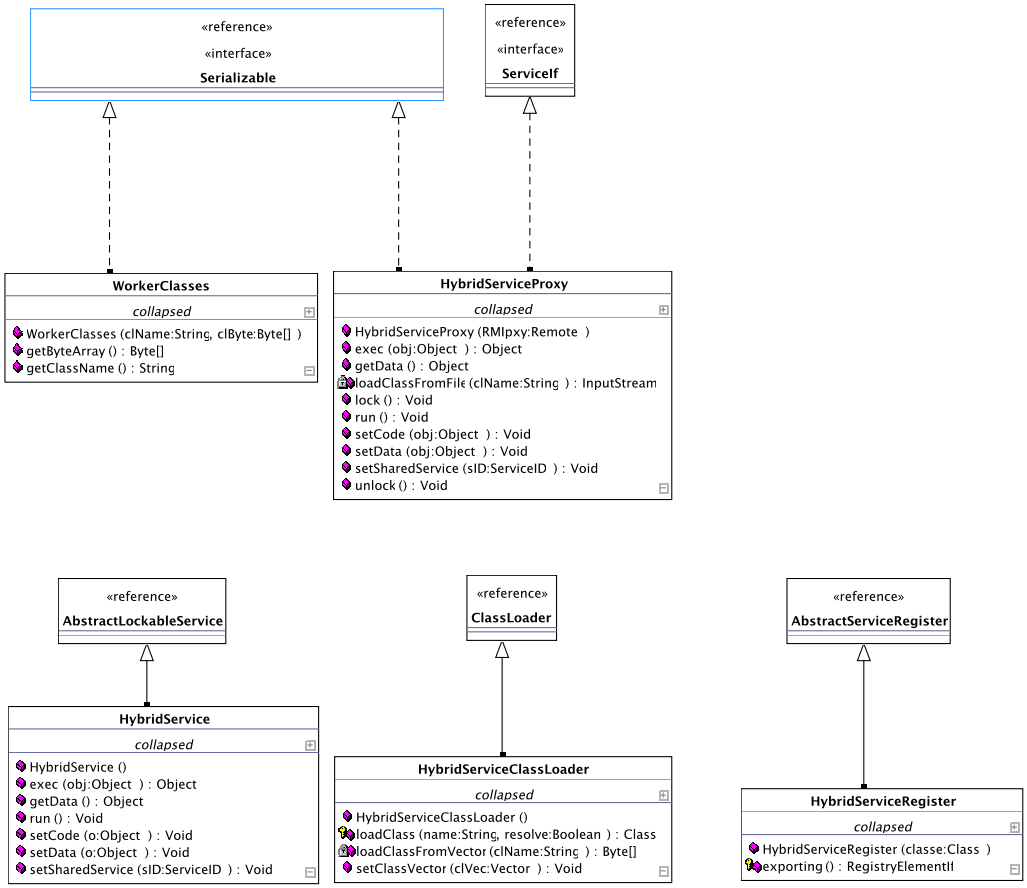


B.2 Package Service

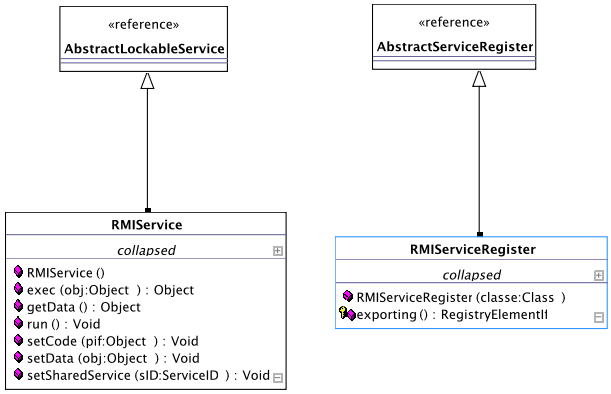
B.2.1 Package core



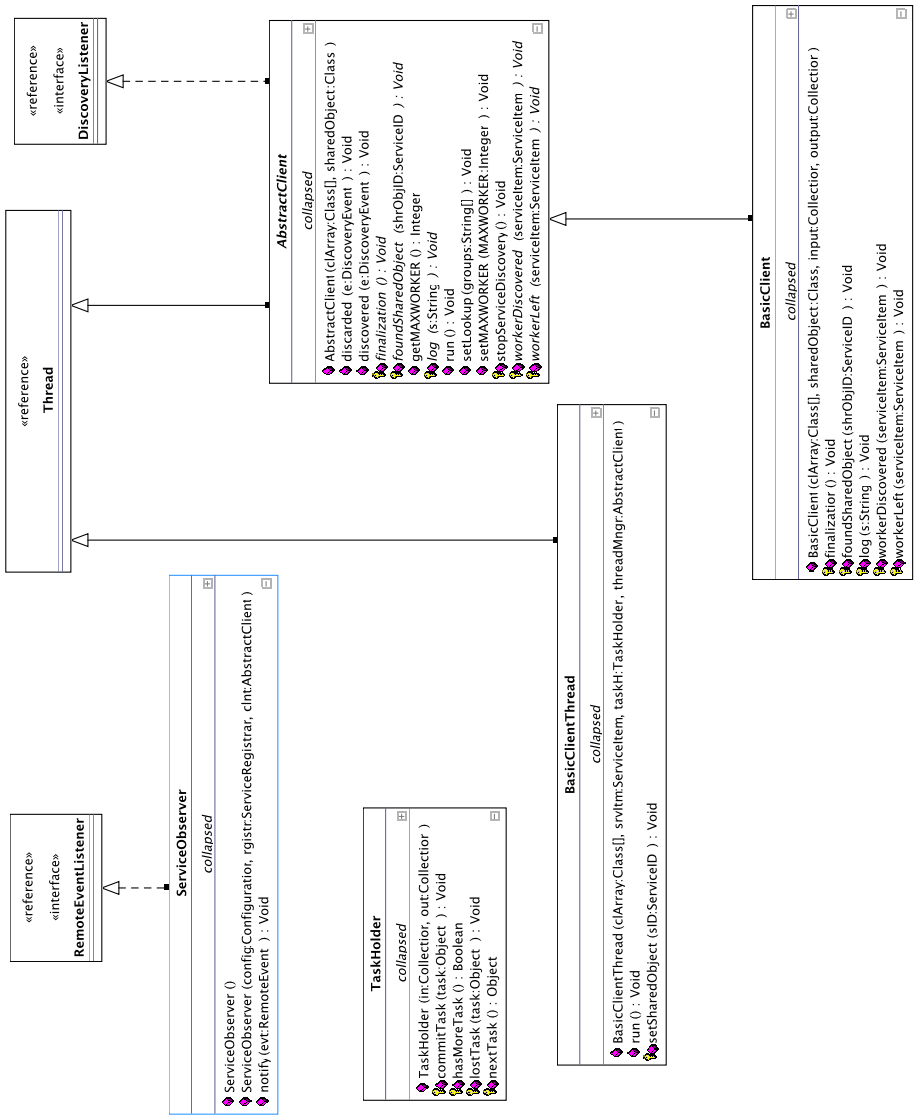
B.2.2 Package hybrid



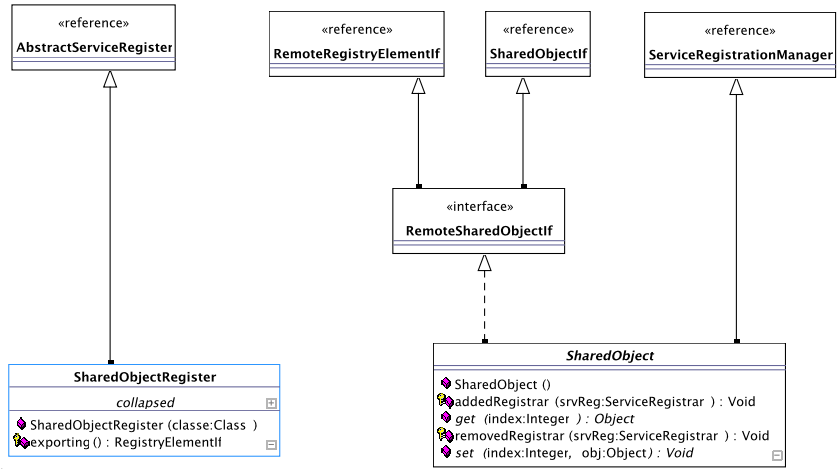
B.2.3 Package rmi



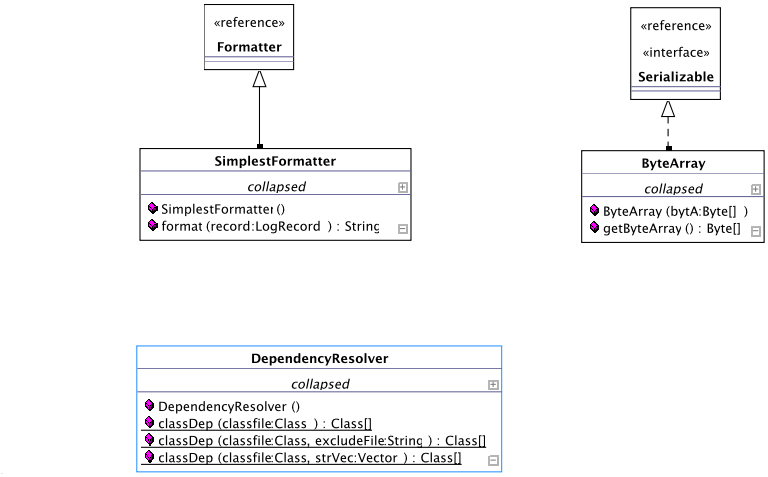
B.3 Package Client



B.4 Package Shared



B.5 Package Util



Appendice C

Codice Java

C.1 jjpf.common.ProcessoIf.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.common;

import java.io.IOException;
import java.io.Serializable;


/** This Interface must be implemented
 *  by the Object that needs
 *  to be executed on a Service
 *  implementing ServiceIf interface.
 * @author Patrizio Dazzi
 */
public interface ProcessoIf extends Serializable {

    /** This method sets the SharedObject
     *  that will be used by ProcessoIf
     * @param ssi SharedObject object
     */
    void setSharedObject(SharedObjectIf ssi);

    /** Gets the execution result
     * @return the <CODE>Object</CODE>
     *  that contains execution result
     */
    Object getData();

    /** Sends data to the <CODE>Object</CODE>
     *  running on the Service
     * @param obj data for the running
     *  <CODE>Object</CODE>
     */
    void setData(Object obj);
```

```
    /** Starts the Execution */
    void run();
}
```

C.2 jjpf.common.RegistryElementIf.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.common;

/**
 *
 * @author  Patrizio Dazzi
 *
 * This interface must be
 *  implemented by every object
 *  that needs to be Registred on a
 *  Lookup Service
 */
public interface RegistryElementIf {

}
```

C.3 jjpf.common.ServiceIf.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.common;

import java.rmi.RemoteException;

import java.util.Vector;

import net.jini.core.lookup.ServiceID;


/**
 * Every Service that wants to
```

```
 *  be found with JJPF needs to
 *  implement this interface
 * @author Patrizio Dazzi
 */
public interface ServiceIf extends RegistryElementIf {

    /** Reserve the Service for the
     *  locking Client. Lock ends when
     *  Client invoke  unlock method.
     * @throws RemoteException When a
     *          locking error occours,
     *          a <CODE>RemoteException</CODE>
     *          is sent back to the
     * Client.
     */
    void lock() throws RemoteException;

    /**
     * Unlocks the Service
     * @throws RemoteException
     */
    void unlock() throws RemoteException;

    /**
     * Sets the <CODE>Object</CODE> to be
     *  executed on the Service
     *  @param pif the Object to send
     *  @throws RemoteException if an
     *          execution error occours
     *          a Remote Exception is thrown
     */
    void setCode(Object pif) throws RemoteException;

    /**
     * Sets the <CODE>ServiceID</CODE> object
     * that permits to a Service the searching
     * for a Shared Object
     *
     * @param sID the ServiceID Object
     * @throws RemoteException if an error
     *          occours a Remote Exception is thrown
     */
    void setSharedService(ServiceID sID) throws RemoteException;

    /**
     *  Wraps the method with the same
     *  name defined in ProcessIf
     *  @param obj the data to send to
     *          the running object nested
     *          in the service
     *  @throws RemoteException an exception
     *          occured during data passing
     */
    void setData(Object obj) throws RemoteException;

    /**
     * Wraps the method with the same
     *  name defined in ProcessIf
     *  @throws RemoteException an exception
     *           occured during data getting
     *  @return the data obtained from running
     *           object to send to the client
     */
    Object getData() throws RemoteException;

    /**
     * Wraps the method with the same name
     * defined in ProcessIf
     * @throws RemoteException an exception
     *          occured during execution
     */
    void run() throws RemoteException;

    /**
     * Invoke in sequence <CODE>setData(obj) -
     *                         run() -
     *                         getData()</CODE>
     * @param obj the data to send to the running
     *     object nested in the service
     * @throws RemoteException an exception occured
     *      during data setting, getting or execution
     * @return the data obtained from running object
     *      to send to the client
```

```
     */
    Object exec(Object obj) throws RemoteException;
}
```

C.4 jjpf.common.SharedObjectIf.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.common;

import java.rmi.RemoteException;


/** Every SharedObject must to implements
 *  this interface if wants to be found
 *  by JJPF Framework.
 * @author Patrizio Dazzi
 */
public interface SharedObjectIf extends RegistryElementIf {

    /** Sets data to a <CODE>SharedObject</CODE>
     * @param obj the <CODE>Object</CODE>
     *              to send to SharedObject
     * @throws RemoteException If an error
     *          occours during data setting,
     *          a <CODE>RemoteException</CODE>
     *          is thrown
     */
    public void set(int index, Object obj) throws RemoteException;

    /** Gets data from <CODE>SharedObject</CODE>
     * @return data from the SharedObject
     * @throws RemoteException If an error
     *          occours during data getting,
     *          a <CODE>RemoteException</CODE>
     *          is thrown
     */
    public Object get(int index) throws RemoteException;
}
```

C.5 jjpf.common.exception.ServiceAlreadyInUseException.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.common.exception;

/**
 * An Exception is occurred during Service Locking
 * @author Patrizio Dazzi
 */
public class ServiceAlreadyInUseException
    extends java.lang.Exception {

    /**
     * Creates a new instance of
     *  <code>ServiceAlreadyInUseException</code>
     *  without detail message.
     */
    public ServiceAlreadyInUseException() {

    }

    /**
     * Constructs an instance of
     *  <code>ServiceAlreadyInUseException</code>
     *  with the specified detail message.
     * @param msg the detail message.
     */
    public ServiceAlreadyInUseException(String msg) {
        super(msg);

    }

}
```

C.6 jjpf.common.exception.ServiceRegistrationException.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.common.exception;

/** An Exception is occurred during
 *  Service Registration
 * @author Patrizio Dazzi
 */
public class ServiceRegistrationException
    extends java.lang.Exception {

    /** Contains the generating exception */
    private Throwable cause;

    /**
     * Creates a new instance of
     *  <code>ServiceRegistrationException</code>
     *  without details message.
     */
    public ServiceRegistrationException() {
```

```
    }

    /**
     * Constructs an instance of
     *  <code>ServiceRegistrationException</code>
     *  with the specified detail message.
     * @param msg the detail message.
     */
    public ServiceRegistrationException(String msg) {
        super(msg);

    }

    /**
     * Constructs an instance of
     *  <code>ServiceRegistrationException</code>
     *  with the specified detail message and
     *  generating cause. This Constructor
     *  prevent further cause initialization.
     *
     * @param s the detail message.
     * @param exc the generating cause
     */
    public ServiceRegistrationException(String s, Throwable exc) {
        super(s);


        // prevent further initCause
        initCause(null);
        cause = exc;

    }

    /**
     * Return the superclass Exception Message
     *
     * @return the exception message
     */
    public String getMessage() {

        if(cause == null) {

            return super.getMessage();

        } else {

            return super.getMessage() +
                   "; nested exception is: \n\t" +
                   cause.toString();

        }

    }

    /**
     * Return the nested exception
     *
     * @return the nested exception
     */
    public Throwable getCause() {

        return cause;

    }
}
```

C.7 jjpf.service.core.AbstractLockableService.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
```

```
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.core;

import java.io.IOException;

import java.rmi.RemoteException;

import java.util.Iterator;
import java.util.LinkedList;

import jjpf.common.SharedObjectIf;

import jjpf.common.exception.ServiceAlreadyInUseException;

import jjpf.service.core.RemoteServiceIf;

import net.jini.config.ConfigurationException;

import net.jini.core.lease.Lease;
import net.jini.core.lease.UnknownLeaseException;
import net.jini.core.lookup.ServiceID;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceTemplate;


/** Class used to create JJPF Services,
 *  a Lockable Service permit to a
 *  client to lock the service.
 * @author Patrizio Dazzi
 */
public abstract class AbstractLockableService
    extends ServiceRegistrationManager implements RemoteServiceIf {

    /* Excpetion Management variables */
    private Exception       alUE                    = null;
    private RemoteException re;
    private ServiceID       srvID                   = null;
    private LinkedList      observers               = null;
    private SharedObjectIf  sharedObjectForObservers = null;

    /** Creates a new instance of
     *  AbstractService
     * @throws IOException Input/Output
     *         Exception during Service creation
     */
    public AbstractLockableService() throws IOException {

        observers = new LinkedList();

    }

    /** A client that invoke this method
     *  lock the service for its purpose
     * @throws RemoteException thrown if
     *         the service has been already locked
     */
    public final void lock() throws RemoteException {
        synchronized(alreadyInUse) {

            if(alreadyInUse_bool) {

                alUE      = new ServiceAlreadyInUseException();
                re        = new RemoteException();
                re.detail = alUE;
                throw re;

            } else {

                alreadyInUse_bool = true;

            }

        }
```

```
        // Deregistrazione dal servizio
        abstractServiceUnRegister();

    }

    /** When a client does no longer require
     *  the service invoke this method to unlock the service.
     * @throws RemoteException after the unlocking a service
     *       try to register itself in a LookupService, if it
     *       fail this exception will be thrown
     */
    public final void unlock() throws RemoteException {


        // Ri-registrazione nel servizio
        abstractServiceRegister();
        synchronized(alreadyInUse) {

            alreadyInUse_bool = false;

        }

    }

    /**
     * Method used to find a SharedObject
     * @param sID ServiceID to find
     * @return a SharedObject
     */
    protected Object searchInKnownRegistrar(ServiceID sID) {

        int            i       = 0;
        boolean        trovato = false;
        SharedObjectIf so      = null;

        // If a null ServiceID object is passed,
        //  a null object is returned
        if(sID == null)

            return null;

        srvID = sID;

        // Creates searching template
        ServiceTemplate srvTmpl = new ServiceTemplate(sID, null, null);
        synchronized(registrars) {

            while((i < registrars.size()) && (!trovato)) {

                try {

                    so = (SharedObjectIf) ((ServiceRegistrar) registrars.elementAt(i))
                                                              .lookup(srvTmpl);

                } catch(RemoteException re) {

                    continue;

                }

                if(so != null) {


                    // SharedObject found
                    trovato = true;

                    // Removes all SharedObjectObserver
                    for(int j = 0; j < observers.size(); j++) {

                        try {

                            ((SharedObjectObserver) observers.removeFirst()).reg
                                                    .getLease().cancel();

                        } catch(UnknownLeaseException ule) {

                            continue;

                        } catch(RemoteException re) {

                            continue;

                        }
```

```
                }
            } else {
                try {

                    // Adds an SharedObjectObserver
                    observers.add(
                            new SharedObjectObserver(
                                    (ServiceRegistrar) registrars.elementAt(i),
                                    srvID,
                                    sharedObjectForObservers));
                } catch(RemoteException re) {
                    continue;
                }
            }
            i++;
        }
    }
    // if no Shared Object has
    //  been found, it suspend itself
    if(so == null) {
        synchronized(sID) {
            try {
                srvID.wait();
                so = sharedObjectForObservers;
            } catch(InterruptedException ie) {
                throw new RuntimeException(
                        "Interrupted AbstractLockableService during wait",
                        ie);
            }
        }
        // Removes the observers
        //  that it had put
        for(int j = 0; j < observers.size(); j++) {
            try {
                ((SharedObjectObserver) observers.removeFirst()).reg.getLease()
                                                    .cancel();
            } catch(UnknownLeaseException ule) {
                continue;
            } catch(RemoteException re) {
                continue;
            }
        }
    }
    return so;
}
/** Superclass defined method,
 *  called if a new Lookup Services
 *  is found
 * @param srvReg LookupService
 *          proxy handler
 * @throws IllegalStateException
 */
protected final void addedRegistrar(ServiceRegistrar srvReg)
                            throws IllegalStateException {
    if(srvID == null)
```

```
            return;

        try {

            SharedObjectObserver soo = new SharedObjectObserver(
                                                srvReg, srvID,
                                                sharedObjectForObservers);

        } catch(RemoteException re) {

        }

    }

    /** Superclass defined method, called
     *  if a Lookup Services is lost
     * @param srvReg lost LookupService
     */
    protected final void removedRegistrar(ServiceRegistrar srvReg) {

    }

    private void abstractServiceRegister() {

        ServiceRegistration registration = null;
        ServiceRegistrar    registrar = null;
        Iterator            iterV     = registrars.iterator();

        while(iterV.hasNext()) {

            registrar = (ServiceRegistrar) iterV.next();

            try {

                registration = registrar.register(sItem,
                                                  Lease.FOREVER);


                // Adds registration object
                //  to the vector registrations
                registrations.add(registration);

            } catch(RemoteException re) {


                // Removes the register
                //  that cause problems
                iterV.remove();

                continue;

            }


            // it register itself
            //  in the lease manager
            leaseManager.renewUntil(registration.getLease(),
                                    Lease.FOREVER,
                                    ((registration.getLease()
                                                  .getExpiration() -
                                                  System.currentTimeMillis()) / 2),
                                    this);

        }

    }

    private void abstractServiceUnRegister() {

        ServiceRegistration registration = null;
        Iterator            iterV = registrations.iterator();

        while(iterV.hasNext()) {

            registration = (ServiceRegistration) iterV.next();

            try {

                registration.getLease().cancel();

            } catch(RemoteException re) {

                System.err.println("Errore nella Deregistrazione: " +
```

```
                                re.toString());
                continue;
            } catch(UnknownLeaseException ule) {
                System.err.println("Errore nella Deregistrazione: " +
                                re.toString());
                continue;
            }
        }

        // stop to renews the leases
        registrations.clear();
    }
}
```

C.8 jjpf.service.core.AbstractServiceRegister.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.core;

// Jini ClassServer
import com.sun.jini.tool.ClassServer;

// I/O exception
import java.io.IOException;

// RMI security manager
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;

// JJPF common classes
import jjpf.common.RegistryElementIf;
import jjpf.common.ServiceIf;

import jjpf.common.exception.ServiceRegistrationException;

import jjpf.service.core.RemoteServiceIf;

// JJPF service core classes
import jjpf.service.core.ServiceMgmtIf;

// Jini config
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;

// Jini Service Exporting
import net.jini.discovery.DiscoveryEvent;

import net.jini.export.Exporter;

// Leasing Management
import net.jini.lease.LeaseRenewalEvent;
```

```
/** This class is used to create in a very
 *  simply manner a class to use as a Service
 *  Register
 * @author Patrizio Dazzi
 */
public abstract class AbstractServiceRegister extends Thread {

    private Object                  keepAlive;
    private Configuration           config;
    private RegistryElementIf       proxy;
    private RemoteRegistryElementIf serv;
    private ServiceMgmtIf           mgmtService;
    private Object                  servizio;

    /** Creates a new instance of
     *  AbstractServiceRegister
     * @param classe the class object
     *          representative of the
     *          object to register
     * @param CONFIG_FILE Configuration filename
     */
    public AbstractServiceRegister(Class classe, String CONFIG_FILE) {

        try {

            // open configuration file
            config = ConfigurationProvider.getInstance(
                            new String[]{ CONFIG_FILE });

            // reads entry from
            //  configuration  file
            String baseDir = (String) config.getEntry("Service",
                                                       "classServerBaseDir",
                                                       String.class,
                                                       null);
            String classServerPort = (String)  config.getEntry(
                                              "Service",
                                              "classServerPort",
                                              String.class, null);

            // starts a ClassServer
            ClassServer cs = new ClassServer(
                                        new String[]{
                "-port", classServerPort, "-dir", baseDir
            }, null);

            // reads and sets
            //  security policies
            String securityPolicy = (String) config.getEntry(
                                             "Service", "securPolicy",
                                             String.class, null);
            System.setProperty("java.security.policy", securityPolicy);

            // reads and sets
            //  RMI codebase property
            String codebase = (String) config.getEntry("Service",
                                                        "codebase",
                                                        String.class,
                                                        null);
            System.setProperty("java.rmi.server.codebase", codebase);

        } catch(ConfigurationException ce) {

            throw new RuntimeException(
                    "Configuration File Missing or Incorrect: " + ce);

        } catch(IOException ioe) {

            throw new RuntimeException("ClassServer Launch Error: " +
                                       ioe);

        }


        // RMI Security Manager
        System.setSecurityManager(new RMISecurityManager());

        // builds a Worker
        try {
```

```
            servizio    = classe.newInstance();
            serv        = (RemoteRegistryElementIf) servizio;
            mgmtService = (ServiceMgmtIf) servizio;

        } catch(InstantiationException ie) {

        } catch(IllegalAccessException iae) {

        }

        // gets the proxy to export
        try {

            proxy = exporting();

        } catch(RemoteException re) {

        } catch(ConfigurationException ce) {

        }

    }

    /** Used to permit to a Subclass
     *  to customize the Proxy to be
     *  exported in the LookupServices.
     * This method is Abstract and must
     *  be implemented by all subclasses.
     * @throws RemoteException
     * @throws ConfigurationException Configuration
     *      file does not exists or it is corrupted
     * @return the proxy to register on LookupService
     */
    protected abstract RegistryElementIf exporting()
                                        throws RemoteException,
                                               ConfigurationException;

    /** This method returns a "pure" RMI proxy
     *  that can be used by the client in place
     *  of the Backend service
     * @param CFG_FILE Configuration filename
     * @throws RemoteException
     * @throws ConfigurationException Configuration
     *      file does not exists or it is corrupted
     * @return a pure RMI Proxy
     */
    protected RemoteRegistryElementIf getRMIProxy(String CFG_FILE)
                                          throws RemoteException,
                                                 ConfigurationException {

        RemoteRegistryElementIf RMIproxy;

        // it Read Configuration file
        String[] configArgs = new String[]{ CFG_FILE };

        // gets the configuration
        Configuration conf = ConfigurationProvider.getInstance(
                                    configArgs);

        // build an exporter
        Exporter exp = (Exporter) conf.getEntry("Service", "exporter",
                                                Exporter.class);


        // generate a RMI proxy
        RMIproxy = (RemoteRegistryElementIf) exp.export(serv);

        return RMIproxy;

    }

    /** This method indicate to a service where
     *  to search for LookupService.
     * @param MulticastGroups The groups where
     *  the service will search the LookupService
     * @throws ServiceRegistrationException this
     *  exception will be thrown if an error occours
     *  during the search.
     */
    public void setMulticastGroups(String[] MulticastGroups)
                           throws ServiceRegistrationException {
```

```
        mgmtService.setLookup(proxy, MulticastGroups);
    }

    /** This method indicate to a service where
     *  to search for LookupService.
     * @param UnicastAddresses The addresses where
     *  the service will search the LookupService
     * @throws ServiceRegistrationException this
     *  exception will be thrown if an error
     *  occours during the search.
     */
    public void addUnicastLookup(String[] UnicastAddresses)
                        throws ServiceRegistrationException {

        // adds an unicast link
        // to LookupService
        if(UnicastAddresses != null)

            for(int i = 0; i < UnicastAddresses.length; i++)
                mgmtService.addLookup(proxy, UnicastAddresses[i]);

    }

    /** This method indicate to a service
     *  where to search for LookupService.
     * @param UnicastAddress The address
     *      where the service will search
     *      the LookupService
     * @param port The port where the
     *      service will search the
     *      LookupService
     * @throws ServiceRegistrationException
     *      this exception will be thrown
     *      if an error occours during the search.
     */
    public void addUnicastLookup(String UnicastAddress, int port)
                        throws ServiceRegistrationException {

        if(UnicastAddress != null) {

            mgmtService.addLookup(proxy, UnicastAddress, port);

        }

    }

    /** Service Register start */
    public void run() {

        keepAlive = new Object();
        synchronized(keepAlive) {

            try {

                keepAlive.wait();

            } catch(InterruptedException ie) {

                System.out.println("Serious Error");

            }

        }

    }

    public void endService() {
        synchronized(keepAlive) {

            keepAlive.notify();

        }

        this.interrupt();

    }

}
```

C.9 jjpf.service.core.RemoteRegistryElementIf.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.core;

import java.rmi.Remote;

import jjpf.common.RegistryElementIf;


/**
 *
 * @author  Patrizio Dazzi
 */
public interface RemoteRegistryElementIf extends RegistryElementIf,
                                                 Remote {

}
```

C.10 jjpf.service.core.RemoteServiceIf.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.core;

import java.rmi.Remote;

import jjpf.common.ServiceIf;


/** Remote interface.
 *
 * @author Patrizio Dazzi
 *
 */
public interface RemoteServiceIf extends RemoteRegistryElementIf,
                                         ServiceIf {

}
```

C.11 jjpf.service.core.ServiceMgmtIf.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.core;

import jjpf.common.RegistryElementIf;
import jjpf.common.ServiceIf;

import jjpf.common.exception.ServiceRegistrationException;


/**
 * This is the service management
 *  interface
 * @author Patrizio Dazzi
 */
public interface ServiceMgmtIf {

    /** sets the multicast groups where
     *  search for LookupServices
     * @param proxy the proxy to register
     *       in the lookupServices eventually
     *       found
     * @param groups multicast groups where
     *       search for a lookupService
     * @throws ServiceRegistrationException
     *       if an error occours during registration,
     *       this exception will be thrown
     */
    void setLookup(RegistryElementIf proxy, String[] groups)
            throws ServiceRegistrationException;

    /** adds an unicast address where search
     *  for LookupServices
     * @param proxy the proxy to register in
     *  the lookupService eventually found
     * @param hostname hostname where search
     *  for a lookupService
     * @throws ServiceRegistrationException if
     *  an error occours during registration,
     *  this exception will be thrown
     */
    void addLookup(RegistryElementIf proxy, String hostname)
            throws ServiceRegistrationException;

    /** adds an unicast address where search
     *  for LookupServices
     * @param proxy the proxy to register in
     *  the lookupService eventually found
     * @param hostname hostname where search
     *  for a lookupService
     * @param porta port where search for a
     *  lookupService
     * @throws ServiceRegistrationException
     *  if an error occours during registration,
     *  this exception will be thrown
     */
    void addLookup(RegistryElementIf proxy, String hostname,
                   int porta) throws ServiceRegistrationException;

}
```

C.12 jjpf.service.core.ServiceRegistrationManager.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.core;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;

// Input - Output
import java.io.IOException;

// Java net classes
import java.net.MalformedURLException;

import java.rmi.RemoteException;

// Utils
import java.util.Iterator;
import java.util.Vector;

import jjpf.common.RegistryElementIf;

// JJPF Common
import jjpf.common.ServiceIf;
import jjpf.common.SharedObjectIf;

import jjpf.common.exception.ServiceAlreadyInUseException;
import jjpf.common.exception.ServiceRegistrationException;

// JJPF Service core
import jjpf.service.core.RemoteServiceIf;
import jjpf.service.core.ServiceMgmtIf;

// Jini Configuration
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;

// Jini Discovery
import net.jini.core.discovery.LookupLocator;

// Jini Leasing
import net.jini.core.lease.Lease;
import net.jini.core.lease.UnknownLeaseException;
import net.jini.core.lookup.ServiceID;

// Jini Lookup
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceTemplate;

import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.LookupDiscovery;

import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
```

```
/** This Class aids the development of Service,
 * it is helpful for Service registrations
 * and unregistration;
 * @author Patrizio Dazzi
 */
public abstract class ServiceRegistrationManager
    implements ServiceMgmtIf, DiscoveryListener, LeaseListener {

    /** Section to read in the
     *  Configuration file */
    public static final String MODULE_NAME = "CoreService";

    /** Jeri Configuration File */
    public static final String CONFIG_FILE = "jjpf/service/core/jjpf.service.core.config";

    /** Configuration Object */
    protected Configuration config = null;

    /** ServiceID file */
    protected File serviceIDFile = null;

    /** object used for
     *  synchronization */
    protected Object alreadyInUse = null;

    /** true if the service
     *  is already in use,
     *  false otherwise */
    protected boolean alreadyInUse_bool = false;

    /** Contains <CODE>Registration</CODE>
     *  objects */
    protected Vector registrations = null;

    /** Contains <CODE>Registrar</CODE>
     *  objects */
    protected Vector registrars = null;

    /** Service Item passed to LookupService
     *  for Service Description and
     *  Registration */
    protected ServiceItem sItem = null;

    /** <CODE>LookupDiscovery</CODE> object,
     *  it is used to interact with
     *  LookupService */
    protected LookupDiscovery ld = null;

    /** Provides registration management
     *  of a set of lease associated
     *  with remote entities.
     */
    protected LeaseRenewalManager leaseManager;

    /** Creates a new instance of
     *  ServiceRegistrationManager,
     *  initialize class fields,
     * read the configuration file,
     *  starts discoveryListening
     *  and Leasing Management
     * @throws IOException Thrown
     *   when some errors occours
     *   during <CODE>LookupDiscovery</CODE>
     */
    public ServiceRegistrationManager() throws IOException {

        // Synchronization object
        alreadyInUse = new Object();


        // Registrars Vector
        registrars = new Vector();


        // Registrations Vector
        registrations = new Vector();


        // Lease Manager
        leaseManager = new LeaseRenewalManager();

        // try to read the CoreService Module
```

```
        // in the configuration file
        try {
            config = ConfigurationProvider.getInstance(
                            new String[]{ CONFIG_FILE });
            serviceIDFile = (File) config.getEntry("CoreService",
                                                   "ServiceIDFile",
                                                   File.class, null);
        } catch(ConfigurationException ce) {
            // Configuration file does not exist
            throw new IOException("Configuration File not found");
        }

        // LookupService Discovery & Listening
        ld = new LookupDiscovery(LookupDiscovery.NO_GROUPS);
        ld.addDiscoveryListener(this);
    }
    /** Sets the Multicast groups to join in
     * @param proxy Used to permit the connection
     *      from a client to this Service
     * @param groups groups (String[]) to join in
     * @throws ServiceRegistrationException An
     *          exception is thrown during the
     *          Service Registration
     */
    public final void setLookup(RegistryElementIf proxy,
                                String[] groups)
                        throws ServiceRegistrationException {
        if(sItem == null)
            sItem = createServiceItem(proxy);
        try {

            // Sets new multicast groups
            ld.setGroups(groups);
        } catch(IOException ioe) {
            throw new ServiceRegistrationException(
                    "Multicast request protocol fail: IOException",
                    ioe);
        } catch(IllegalStateException ise) {
            throw new ServiceRegistrationException(
                    "Method terminate has been called: IllegalStateException",
                    ise);
        } catch(UnsupportedOperationException uoe) {
            throw new ServiceRegistrationException(
                    "There is no groups to add: UnsupportedOperationException",
                    uoe);
        }
    }
    /** Adds a Unicast LookupService Address
     * @param proxy Used to permit the connection
     *      from a client to this Service
     * @param hostname LookupService hostname
     * @throws ServiceRegistrationException
     *      An exception is thrown during the
     *      Service Registration
     */
    public final void addLookup(RegistryElementIf proxy,
                                String hostname)
                        throws ServiceRegistrationException {
        ServiceRegistrar registrar = null;
        LookupLocator    ll = null;
        if(sItem == null)
```

```
            sItem = createServiceItem(proxy);

        try {

            // connecting to the lookup service
            ll = new LookupLocator(hostname);

        } catch(MalformedURLException mue) {

            throw new ServiceRegistrationException(
                    "Connecting Lookup Service: MalformedURLException",
                    mue);

        }

        try {

            // obtaining registrar
            registrar = ll.getRegistrar();

        } catch(IOException ioe) {

            throw new ServiceRegistrationException(
                    "Obtaining Registrar: IOException", ioe);

        } catch(ClassNotFoundException cnfe) {

            throw new ServiceRegistrationException(
                    "Deserialization error: ClassNotFoundException",
                    cnfe);

        }


        // Adds the new registrar to registrars Vector
        registrars.add(registrar);


        // Registration Activation
        activateRegistration(registrar);

    }

    /** Adds a Unicast LookupService Address
     *  and Port
     * @param proxy Used to permit the connection
     *          from a client to this Service
     * @param hostname LookupService hostname
     * @param porta LookupService Server Port
     * @throws ServiceRegistrationException
     *      An exception is thrown during the
     *      Service Registration
     */
    public final void addLookup(RegistryElementIf proxy,
                                String hostname, int porta)
                        throws ServiceRegistrationException {

        ServiceRegistrar registrar = null;
        LookupLocator    ll = null;

        if(sItem == null)
            sItem = createServiceItem(proxy);

        ll = new LookupLocator(hostname, porta);

        try {

            // obtaining registrar
            registrar = ll.getRegistrar();

        } catch(IOException ioe) {

            throw new ServiceRegistrationException(
                    "Obtaining Registrar: IOException", ioe);

        } catch(ClassNotFoundException cnfe) {

            throw new ServiceRegistrationException(
                    "Deserialization error: ClassNotFoundException",
                    cnfe);
```

```
        }

        // Adds the new registrar
        //  to registrars Vector
        registrars.add(registrar);

        // Registration Activation
        activateRegistration(registrar);
    }

    /** Called when a LookupService
     *  is found
     * @param evt discovery event
     */
    public final void discovered(DiscoveryEvent evt) {

        ServiceRegistrar registrar = null;

        for(int i = 0; i < evt.getRegistrars().length; i++) {

            registrar = evt.getRegistrars()[i];
            synchronized(registrars) {

                // Adds the new registrar
                //  to registrars Vector
                registrars.add(registrar);

                // it notify to subclass that
                //  a regisrar has been added
                addedRegistrar(registrar);

            }

            // Registration Activation
            activateRegistration(registrar);

        }
    }

    /** Called when a LookupService
     *  shuts down
     * @param evt the event
     */
    public final void discarded(DiscoveryEvent evt) {

        ServiceRegistrar registrar = null;

        for(int i = 0; i < evt.getRegistrars().length; i++) {

            registrar = evt.getRegistrars()[i];
            synchronized(registrars) {

                // Removes registrar from
                // registrars vector
                registrars.remove(registrar);

                // it notify to subclass that a
                // regisrar has been removed
                removedRegistrar(registrar);

            }

        }
    }

    /** */
    public final void notify(LeaseRenewalEvent e) {

    }

    /** This method must be implemented by all
     *  subclasses,it is called when a registrar
     * object is added to <CODE>Registrars</CODE>
     * @param srvReg the added registrars
```

```
 */
protected abstract void addedRegistrar(ServiceRegistrar srvReg);

/** This method must be implemented by all
 *  subclasses, it is called when a registrar
 * object is removed from <CODE>Registrars</CODE>
 * @param srvReg the removed registrar
 */
protected abstract void removedRegistrar(ServiceRegistrar srvReg);

/** Activate registration, if the Service is
 *  in use it only saves the new registrar
 *  otherwise, it register itself on the just
 *  found LookupService
 * @param registrar found LookupService
 */
private void activateRegistration(ServiceRegistrar registrar) {

    ServiceRegistration registration = null;
    synchronized(alreadyInUse) {

        // If the worker is in use, it only save the
        // register in registers Vector, otherwise
        // it register itself in the discovered
        // LookupService
        if(!alreadyInUse_bool) {

            try {


                // registration
                registration = registrar.register(sItem,
                                                  Lease.FOREVER);

                // If it has already got a ServiceID,
                // it use that one for the new registration
                if(sItem.serviceID == null)
                    consolidateServiceID(
                            registration.getServiceID());


                // Adds new registration in the
                // registrations vector
                registrations.add(registration);

            } catch(RemoteException re) {


                // it removes the register where an
                // exception is occured
                registrars.remove(registrar);

                return;

            }


            // Update lease manager lookup service pool,
            // adding the last found
            leaseManager.renewUntil(registration.getLease(),
                                    Lease.FOREVER,
                                    ((registration.getLease()
                                                  .getExpiration() -
                                                  System.currentTimeMillis()) / 2),
                                    this);

        }

    }

}

/** Create a <CODE>ServiceItem</CODE> from
 *  <CODE>regElem</CODE> searching for an
 *  eventually already present ServiceID.
 * @param regElem Service from which this
 *          method creates ServiceItem
 * @return ServiceItem to register in a
 *          LookupService
 */
protected ServiceItem createServiceItem(RegistryElementIf regElem) {
```

```
        ServiceID       serviceID = null;
        DataInputStream din = null;

        if(serviceIDFile != null) {

            try {


                // it search for an existing ServiceID
                din = new DataInputStream(
                              new FileInputStream(serviceIDFile));
                serviceID = new ServiceID(din);
                din.close();

            } catch(FileNotFoundException fnfe) {

                // file not found, it register itself
                // asking for a new ServiceID
                // to LookupService
                return new ServiceItem(null, regElem, null);

            } catch(IOException fnfe) {

                // file access error, it register itself
                // asking for a new ServiceID
                // to LookupService
                return new ServiceItem(null, regElem, null);

            }

            // using existing ServiceID
            return new ServiceItem(serviceID, regElem, null);

        } else {

            // it register itself asking for a new
            // ServiceID to LookupService
            return new ServiceItem(null, regElem, null);

        }

    }

    /** Saves in a file the ServiceID obtained
     * for consistency purpose
     * @param serviceID the ServiceID to save
     */
    protected void consolidateServiceID(ServiceID serviceID) {


        // reads & saves obtained serviceID
        sItem.serviceID = serviceID;

        if(serviceIDFile != null) {

            // saves serviceID in a file
            DataOutputStream dou = null;

            try {


                // it creates a file
                dou = new DataOutputStream(
                              new FileOutputStream(serviceIDFile));
                serviceID.writeBytes(dou);
                dou.close();

            } catch(FileNotFoundException fnfe) {

                // file not found
            } catch(IOException fnfe) {

                // file access error
            }

        }

    }

}
```

C.13 jjpf.service.core.SharedObjectObserver.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.core;

import java.rmi.RemoteException;

import jjpf.common.SharedObjectIf;

//Jini Config
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;

import net.jini.core.event.EventRegistration;

// Jini Events
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.UnknownEventException;
import net.jini.core.lease.Lease;
import net.jini.core.lookup.ServiceEvent;
import net.jini.core.lookup.ServiceID;

// Jini Lookup
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;

import net.jini.export.Exporter;

// Jini Lease
import net.jini.lease.LeaseRenewalManager;


/** This Class is developed to readly
 * inform a Service when a SharedObject
 * is found
 * @author Patrizio Dazzi
 */
public class SharedObjectObserver implements RemoteEventListener {

    /** observer's Lease Manager */
    protected static LeaseRenewalManager leaseManager =
            new LeaseRenewalManager();

    /* Local variables used to save
     * Class Constructor parameters
     */
    private ServiceRegistrar registrar    = null;
    private Configuration    config       = null;
    private ServiceID        serviceID    = null;
    private SharedObjectIf   sharedObject = null;

    /** Observer Registration Handler */
    public EventRegistration reg = null;

    /**  specify what kind of transitions
     * this Observer is looking for */
    protected final int transitions = ServiceRegistrar.TRANSITION_NOMATCH_MATCH;

    /** Creates a new instance of
     * SharedObjectObserver, it register
     * an observer on a LookupService
     * @param rgistr the LookupService
```

```
     *          Proxy handler
     * @param srvID the ServiceID we
     *          are looking for
     * @param sharedObj the object where
     *          the (eventually found)
     *          sharedObject will be saved
     * @throws RemoteException If an error
     *          occours, a RemoteException
     *          will be thrown
     */
    public SharedObjectObserver(ServiceRegistrar rgistr,
                                ServiceID srvID,
                                SharedObjectIf sharedObj)
                        throws RemoteException {

        // Proxy Exporter
        Exporter exporter = null;


        // Service Registrar
        registrar = rgistr;


        // ServiceID
        serviceID = srvID;


        // Shared Object
        sharedObject = sharedObj;

        try {


            // Creates a new configuration
            // object from the configuration
            // file
            config = ConfigurationProvider.getInstance(
                            new String[]{
                ServiceRegistrationManager.CONFIG_FILE
            });


            // Creates an customized Exporter
            exporter = (Exporter) config.getEntry(
                                ServiceRegistrationManager.MODULE_NAME,
                                "SharedObjectObserverExporter",
                                Exporter.class, null);

        } catch(ConfigurationException ce) {

            // Configuration Error is found
            // at runtime
            throw new RuntimeException(
                    "SharedObjectObserver configuration file not found",
                    ce);

        }

        // Exports a SharedObjectObserver
        // object
        RemoteEventListener proxy = (RemoteEventListener) exporter.export(
                                            this);

        // Creates a searching template
        ServiceTemplate templ = new ServiceTemplate(srvID, null, null);


        // it ask for a notify to LookupService
        // if a new SharedObject is found
        reg = registrar.notify(templ, transitions, proxy, null,
                                Lease.ANY);


        // Lease management
        leaseManager.renewUntil(reg.getLease(), Lease.FOREVER,
                                ((reg.getLease().getExpiration() - System.currentTimeMillis()) / 2),
                                null);

    }

    /**
```

```
     * A transition event is occurred
     * @param evt transition event
     */
    public void notify(RemoteEvent evt) {

        ServiceEvent sevt = (ServiceEvent) evt;

        // Gets transitions type
        int transition = sevt.getTransition();

        // tests the type of transition
        if((transition == ServiceRegistrar.TRANSITION_NOMATCH_MATCH) &&
           (sevt.getServiceItem() != null) &&
           (sharedObject == null)) {
            synchronized(serviceID) {

                // Gets SharedObject
                sharedObject = (SharedObjectIf) sevt.getServiceItem().service;

                // wake up the waiting thread
                serviceID.notify();

            }

        }

    }
}
```

C.14 jjpf.service.hybrid.HybridService.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.hybrid;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import java.rmi.RemoteException;

import java.util.Vector;

import jjpf.common.ProcessoIf;
import jjpf.common.SharedObjectIf;

import jjpf.service.core.AbstractLockableService;

import net.jini.core.lookup.ServiceID;


/*
 * @author  Patrizio Dazzi
 */

/** Hybrid Service */
public class HybridService extends AbstractLockableService {

    private ProcessoIf               proc        = null;
    private HybridServiceClassLoader srvCL       = null;
```

```
private Class                   classToLoad = null;
private Vector                  pif         = null;
private ByteArrayInputStream    byteArrayIS = null;

/** Creates a new instance of
 *  ServiceMinimale
 * @throws IOException
 */
public HybridService() throws IOException {

}

/** this method is used in order to
 *  start the execution of client
 *  code on the service machine.
 * @throws RemoteException thrown
 *      when an error occours on
 *      service backend
 */
public void run() throws RemoteException {

    proc.run();

}

/** this method is used in order to
 *  transfer the client code on the
 *  service machine.
 * @param o the client code
 * @throws RemoteException thrown when
 *  an error occours on service backend
 */
public void setCode(Object o) throws RemoteException {

    try {


        // copy client class vector
        // into inner class vector
        pif = (Vector) o;


        // copy client class vector
        // into hybrid classloader
        // class vector
        HybridServiceRegister.hsc.setClassVector(pif);


        // load by name the first client
        // class from the hybrid classloader
        classToLoad = HybridServiceRegister.hsc.loadClass(
                            ((WorkerClasses) pif.elementAt(0)).getClassName());


        // creates new istance of the
        // client first class
        proc = (ProcessoIf) classToLoad.newInstance();

    } catch(InstantiationException ie) {

        throw new RemoteException(
                "Instantiating Client Objects: InstantiationException",
                ie);

    } catch(IllegalAccessException iae) {

        throw new RemoteException(
                "Instantiating Client Objects: IllegalAccessException",
                iae);

    } catch(ClassNotFoundException cnfe) {

        throw new RemoteException(
                "Instantiating Client Objects: ClassNotFoundException",
                cnfe);

    }

}

/** this method is used in order to get
 *  the result of execution from the
 *  service machine.
 * @throws RemoteException thrown when
 *          an error occours on service
```

```
     *          backend
     * @return the client code execution
     *          result
     */
    public Object getData() throws RemoteException {

        return proc.getData();

    }

    /** this method is used in order to trnasfer
     *  the input data for the execution into the
     *  service machine.
     * @param o input data
     * @throws RemoteException thrown when an
     *  error occours on service backend
     */
    public void setData(Object o) throws RemoteException {

        proc.setData(o);

    }

    /** invokes in sequence:
    *
    * - setData(obj);
    * - run();
    * - getData();
    * @param obj the input data
    * @throws RemoteException thrown if an
    *       error occours on service backend
    * @return the result of the execution of
    *       client code on the service backend
    */
    public Object exec(Object obj) throws RemoteException {

        setData(obj);
        proc.run();

        return proc.getData();

    }

    /** Set in the backend object the SharedObject
     *  ServiceID. It will be used by the service to
     *  find the SharedObject.
     * @param sID the SharedObject ID
     * @throws RemoteException thrown if an error
     *       occours during Shared Object searching
     */
    public void setSharedService(ServiceID sID) throws RemoteException {

        if(sID != null) {

            // Obtains Shared Object
            SharedObjectIf shObj = (SharedObjectIf) searchInKnownRegistrar(
                                            sID);

            // If Shared Object is found it saves
            // them otherwise a Remote Exception will
            // be thrown
            if(shObj != null)
                proc.setSharedObject(shObj);
            else
                throw new RemoteException("Searching shared object");

        }

    }
}
```

C.15 jjpf.service.hybrid.HybridServiceClassLoader.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
```

```
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.hybrid;

import java.util.Iterator;
import java.util.Vector;


/** Class used to load client code
 * @author Patrizio Dazzi
 */
public class HybridServiceClassLoader extends ClassLoader {

    private Vector classesVector = null;

    /** Creates a new instance of
     *  ServiceClassLoader */
    public HybridServiceClassLoader() {

    }

    /** fill the classes Vector with
     *  client classes
     * @param clVec the class Vector
     *  from which the service will
     *  load the client classes.
     */
    public void setClassVector(Vector clVec) {

        classesVector = clVec;

    }

    /** method used by Java runtime for
     *  the classloading
     * @param name class name
     * @param resolve boolean value used
     *       by runtime to recursive resolving
     * @throws ClassNotFoundException thrown
     *       if no class has been found
     * @return the class object
     */
    protected synchronized Class loadClass(String name,
                                           boolean resolve)
                                    throws ClassNotFoundException {

        byte[] classBytes = null;
        Class  classClass = null;


        // it search the class in the loading cache
        classClass = findLoadedClass(name);

        if(classClass == null) {

            // file system class searching
            try {

                classClass = findSystemClass(name);

            } catch(ClassNotFoundException cnfe) {


                // class not found
                classClass = null;

            }

        }

        // search the class in the
        // client class Vector
        if(classClass == null) {
```

```
            try {
                classBytes = loadClassFromVector(name);
            } catch(ClassNotFoundException cnfe) {
                throw (ClassNotFoundException) new ClassNotFoundException(
                        "No such class " + name +
                        " : ClassNotFoundException", cnfe);
            }
            // class building
            try {
                classClass = defineClass(name, classBytes, 0,
                                         classBytes.length);
            } catch(ClassFormatError cfe) {
                throw (ClassFormatError) new ClassFormatError(
                        "Defining Class " + name +
                        " : ClassFormatError : " + cfe);
            }
            if(classClass == null)
                throw new ClassFormatError(name);
        }
        if(resolve)
            resolveClass(classClass);
        return classClass;
    }
    private byte[] loadClassFromVector(String clName)
                                throws ClassNotFoundException {
        if(classesVector != null) {
            Iterator iter = classesVector.iterator();
            while(iter.hasNext()) {
                WorkerClasses wc = (WorkerClasses) iter.next();
                if(wc.getClassName().compareTo(clName) == 0) {
                    return wc.getByteArray();
                }
            }
        }
        throw new ClassNotFoundException(
                "Not found specified class in the Client-Passed Class Vector");
    }
}
```

C.16 jjpf.service.hybrid.HybridServiceProxy.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.hybrid;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

import java.rmi.Remote;
import java.rmi.RemoteException;

import java.util.Vector;

import jjpf.common.ServiceIf;

import jjpf.service.core.RemoteServiceIf;


/** Hybrid service proxy
 * @author Patrizio Dazzi
 */
public class HybridServiceProxy implements Serializable, ServiceIf {

    private RemoteServiceIf RMIproxy;

    /** Creates a new instance of ServiceProxy
     * @param RMIpxy the RMI proxy
     */
    public HybridServiceProxy(Remote RMIpxy) {

        RMIproxy = (RemoteServiceIf) RMIpxy;

    }

    /** this method is used, on the proxy, in order to
     *  get the result of execution from the service
     *  machine.
     * @throws RemoteException thrown if an error occours
     *  during result data transfer
     * @return the result of client code execution
     */
    public Object getData() throws RemoteException {

        return RMIproxy.getData();

    }

    /** A client that invoke this method lock the service
     *  for its purpose
     * @throws RemoteException thrown if an error occours
     *          during service locking
     */
    public void lock() throws RemoteException {

        RMIproxy.lock();

    }

    /** this method, invoked on the proxy, starts
     *  the execution of client code on the
     *  service
     * @throws RemoteException thrown if an error
     *      occours during client code execution
     */
    public void run() throws RemoteException {

        RMIproxy.run();

    }

    /** this method is used, on the proxy, in order
     *  to transfer the client code on the service
     *  machine.
     * @param obj the client code
     * @throws RemoteException thrown if an error
     *      occours during client code transfer
```

```
 */
public void setCode(Object obj) throws RemoteException {

    Class[]     codiceDaEseguire = (Class[]) obj;

    InputStream is         = null;
    byte[]      classByte  = null;
    Vector      vecClasses = new Vector();

    // Client class Vector building
    for(int i = 0; i < codiceDaEseguire.length; i++) {

        classByte = null;
        is        = loadClassFromFile(codiceDaEseguire[i].getName()
                                                          .trim());

        try {

            classByte = new byte[is.available()];
            is.read(classByte);
            is.close();

        } catch(IOException ioe) {

            throw new RemoteException(
                    "Obtaining bytecode: IOException", ioe);

        }

        vecClasses.add(
                new WorkerClasses(codiceDaEseguire[i].getName()
                                                     .trim(),
                                  classByte));

    }


    // uses the rmi proxy to transfer class
    // Vector from the Client Host to the
    // Service Host
    RMIproxy.setCode(vecClasses);

}

/** this method is used, on the proxy, in order
 *  to trnasfer the input data for the execution
 *  into the service machine.
 * @param obj the input data
 * @throws RemoteException thrown if an error
 *              occours during data transfer
 */
public void setData(Object obj) throws RemoteException {

    try {

        RMIproxy.setData(obj);

    } catch(IOException ioe) {

        throw new RemoteException("During SetData: IOException",
                                  ioe);

    }

}

/** A client that invoke this method unlock
 *  the service
 * @throws RemoteException thrown if an error
 *      occours during service unlocking
 */
public void unlock() throws RemoteException {

    RMIproxy.unlock();

}

private InputStream loadClassFromFile(String clName) {

    String fileSeparator = System.getProperty("file.separator");
    clName = clName.replace('.', fileSeparator.charAt(0));
    clName += ".class";

    return ClassLoader.getSystemResourceAsStream(clName);
```

```
    }

    /** Using this proxy method, a client sets in
     *  the backend object the SharedObject ServiceID.
     *  It will be used by the service to find the
     *  SharedObject.
     * @param sID the SharedObject ID
     * @throws RemoteException thrown if an error
     *       occours during Shared Object searching
     */
    public void setSharedService(net.jini.core.lookup.ServiceID sID)
                        throws RemoteException {

        RMIproxy.setSharedService(sID);

    }

    /** calls exec method on service backend
     * @param obj input data
     * @throws RemoteException thrown if an error
     *           occours on service backend
     * @return thrown client code execution result
     */
    public Object exec(Object obj) throws RemoteException {

        return RMIproxy.exec(obj);

    }
}
```

C.17 jjpf.service.hybrid.HybridServiceRegister.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.hybrid;

import java.rmi.RemoteException;

import jjpf.common.RegistryElementIf;

import jjpf.service.core.AbstractServiceRegister;

import net.jini.config.ConfigurationException;


/** Class used to register
 *  hybrid services
 * @author Patrizio Dazzi
 */
public class HybridServiceRegister extends AbstractServiceRegister {

    /** the classloader object
     *  used during JERI
     *  initialization
     */
    public static HybridServiceClassLoader hsc      =
            new HybridServiceClassLoader();
    private static String                  CFG_FILE =
            "jjpf/service/hybrid/jjpf.service.hybrid.config";
```

```
    /** Creates a new instance of
     *  HybridServiceRegister
     * @param classe the service
     *          class to register
     */
    public HybridServiceRegister(Class classe) {
        super(classe, CFG_FILE);

    }

    /** method used to export an hybrid
     *  service proxy
     * @throws RemoteException an error
     *      occours during class exporting
     * @throws ConfigurationException
     *      Configuration File not found
     * @return the exported proxy
     */
    protected RegistryElementIf exporting() throws RemoteException,
                                                   ConfigurationException {

        HybridServiceProxy srvProxy;


        // Creates hybrid service
        srvProxy = new HybridServiceProxy(getRMIProxy(CFG_FILE));

        return srvProxy;

    }
}
```

C.18 jjpf.service.hybrid.WorkerClasses.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.hybrid;

import java.io.Serializable;


/** Class used as container
 *  for the classes to transfer
 * @author Patrizio Dazzi
 */
public class WorkerClasses implements Serializable {

    private String className = null;
    private byte[] byteArray = null;

    /** Creates a new instance of
     *  WorkerClasses
     * @param clName tranferred
     *              class name
     * @param clByte transferred
     *              class bytes
     */
    public WorkerClasses(String clName, byte[] clByte) {
```

```
        className = clName;
        byteArray = clByte;

    }

    /** method used to get the
     *  class name
     * @return the class name
     */
    public String getClassName() {

        return className;

    }

    /** method used to get class
     *  bytes
     * @return the class byte
     */
    public byte[] getByteArray() {

        return byteArray;

    }

}
```

C.19 jjpf.service.rmi.RMIService.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.rmi;

import java.io.IOException;

import java.rmi.RemoteException;

import jjpf.common.ProcessoIf;
import jjpf.common.SharedObjectIf;

import jjpf.service.core.AbstractLockableService;

import net.jini.core.lookup.ServiceID;


/** The main JJPF service.
 * @author Patrizio Dazzi
 */
public class RMIService extends AbstractLockableService {

    private Class[]    classArray = null;
    private ProcessoIf proc = null;

    /** Creates a new instance of RMIService
     * @throws IOException
     */
    public RMIService() throws IOException {

    }

    /** method used to get the result
     *  of the execution
     * @throws RemoteException thrown
```

```
     *  if a error occours on backend service
     * @return execution result
     */
    public Object getData() throws RemoteException {

        return proc.getData();

    }

    /** execute the program setted by
     *  <CODE>setCode</CODE>
     * @throws RemoteException thrown
     *  if a error occours on backend service
     */
    public void run() throws RemoteException {

        proc.run();

    }

    /** sets the code that the inner object
     *  will execute
     * @param pif code to be executed
     *       in the service
     * @throws RemoteException thrown
     *       if a error occours on backend
     *       service
     */
    public void setCode(Object pif) throws RemoteException {

        classArray = (Class[]) pif;

        if(classArray.length > 0) {

            try {


                // creates new inner object instance
                proc = (ProcessoIf) classArray[0].newInstance();

            } catch(InstantiationException ie) {

                throw new RemoteException(
                        "Instantiating Client Objects: InstantiationException",
                        ie);

            } catch(IllegalAccessException iae) {

                throw new RemoteException(
                        "Instantiating Client Objects: IllegalAccessException",
                        iae);

            }

        }

    }

    /** Sets inner ProcessoIf object
     *  input data
     * @param obj input data
     * @throws RemoteException errors
     *    occured durig data transfer
     */
    public void setData(Object obj) throws RemoteException {

        proc.setData(obj);

    }

    /** Set in the backend object the
     *  SharedObject ServiceID.
     *  It will be used by the service
     *  to find the SharedObject.
     * @param sID the SharedObject ID
     * @throws RemoteException thrown
     *      if a error occours on backend
     *      service
     */
    public void setSharedService(ServiceID sID) throws RemoteException {

        if(sID != null) {

            // Obtains the Shared Object
            SharedObjectIf shObj = (SharedObjectIf) searchInKnownRegistrar(
```

```
                                        sID);
            if(shObj != null)
                // copy the Shared Object
                // in the inner ProcessoIf object
                proc.setSharedObject(shObj);
            else
                throw new RemoteException("Searching shared object");
        }
    }
    /** invokes in sequence:
     *
     * - setData(obj);
     * - run();
     * - getData();
     * @param obj execution input data
     * @return execution result
     */
    public Object exec(Object obj) {
        proc.setData(obj);
        proc.run();
        return proc.getData();
    }
}
```

C.20 jjpf.service.rmi.RMIServiceRegister.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.service.rmi;

import java.rmi.RemoteException;

import jjpf.common.RegistryElementIf;

import jjpf.service.core.AbstractServiceRegister;

import net.jini.config.ConfigurationException;


/** RMIService Register.
 * This class register an instance
 * of RMIService class in a LookupService
 * @author Patrizio Dazzi
 */
public class RMIServiceRegister extends AbstractServiceRegister {

    private static String CONFIG_FILE = "jjpf/service/rmi/jjpf.service.rmi.config";

    /**
     * Creates a new instance of
     *   RMIServiceRegister
     * @param classe class of the
```

```
     *        service to register
     */
    public RMIServiceRegister(Class classe) {
        super(classe, CONFIG_FILE);

    }

    /**
     * @throws RemoteException
     * @throws ConfigurationException
     *           errors occurred during
     *           configuration file
     *           reading
     * @return the proxy to register
     */
    protected RegistryElementIf exporting() throws RemoteException,
                                                   ConfigurationException {

        return getRMIProxy(CONFIG_FILE);

    }
}
```

C.21 jjpf.shared.RemoteSharedObjectIf.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.shared;

import java.rmi.Remote;

import jjpf.common.SharedObjectIf;

import jjpf.service.core.RemoteRegistryElementIf;


/**
 *
 * @author  Patrizio Dazzi
 */
public interface RemoteSharedObjectIf
    extends RemoteRegistryElementIf, SharedObjectIf {

}
```

C.22 jjpf.shared.SharedObject.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
```

```
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.shared;

import java.io.IOException;

import jjpf.service.core.ServiceRegistrationManager;

import jjpf.shared.RemoteSharedObjectIf;

import net.jini.core.lookup.ServiceRegistrar;


/**
 * This class simplifies the development
 *  of SharedObjects
 * @author  Patrizio Dazzi
 */
public abstract class SharedObject
    extends ServiceRegistrationManager implements RemoteSharedObjectIf {

    /** Creates new istance of SharedObject
     * @throws IOException
     */
    public SharedObject() throws IOException {

    }

    /** Gets data from SharedObject
     * @param index index of the data to get
     * @throws RemoteException
     * @return a data object
     */
    public abstract Object get(int index)
                      throws java.rmi.RemoteException;

    /** Sets data into a SharedObject
     * @param index index of data object
     * @param obj the data object
     * @throws RemoteException
     */
    public abstract void set(int index, Object obj)
                     throws java.rmi.RemoteException;

    /** implements a void addedRegistrar to
     *  semplfy the develpoment of SharedObject
     * @param srvReg lookupService handler
     */
    protected void addedRegistrar(ServiceRegistrar srvReg) {

    }

    /** implements a void removedRegistrar to
     *  semplfy the develpoment of SharedObject
     * @param srvReg lookupService handler
     */
    protected void removedRegistrar(ServiceRegistrar srvReg) {

    }

}
```

C.23 jjpf.shared.SharedObjectRegister.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
```

```
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.shared;

import java.io.IOException;

import java.rmi.RemoteException;

import jjpf.common.RegistryElementIf;

import jjpf.service.core.AbstractServiceRegister;

// Jini Configuration
import net.jini.config.ConfigurationException;


/**
 * Register an istance of SharedObject
 *  class in a LookupService
 * @author  Patrizio Dazzi
 */
public class SharedObjectRegister extends AbstractServiceRegister {

    private static String CONFIG_FILE = "jjpf/shared/jjpf.shared.config";

    /**
     *  Creates a new instance of
     *  SharedServiceRegister.
     *  @param classe class of the
     *          shared object to register
     */
    public SharedObjectRegister(Class classe) throws IOException {
        super(classe, CONFIG_FILE);

    }

    /**
    * @throws RemoteException
    * @throws ConfigurationException errors
    *         occurred during configuration
    *         file reading
    * @return the proxy to register
    */
    protected RegistryElementIf exporting() throws RemoteException,
                                                   ConfigurationException {

        return getRMIProxy(CONFIG_FILE);

    }

}
```

C.24 jjpf.client.AbstractClient.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.client;

import com.sun.jini.tool.ClassServer;

import java.io.IOException;

// RMI import
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;

import java.util.Collection;

// Utils
import java.util.Hashtable;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Vector;

import jjpf.common.ProcessoIf;
import jjpf.common.ServiceIf;
import jjpf.common.SharedObjectIf;

import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationNotFoundException;

// Configuration
import net.jini.config.ConfigurationProvider;

import net.jini.core.lease.UnknownLeaseException;
import net.jini.core.lookup.ServiceID;

// Service Management
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;

import net.jini.discovery.DiscoveryEvent;

// Discovery
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.LookupDiscovery;


/**
 *
 * @author  Patrizio Dazzi
 */
public abstract class AbstractClient extends Thread implements DiscoveryListener
{
    /**
     *  Conveniece constant used to request that
     *  attempts be made to discover all lookup
     *  services that are within range, and which
     *  belong to any group.
     */
    public static final String[] ALL_GROUPS = LookupDiscovery.ALL_GROUPS;

    /**
     *  Convenience constant used to request
     *  that discovery by group membership
     *  be halted or not started.
     */
    public static final String[] NO_GROUPS = LookupDiscovery.NO_GROUPS;

    /** true if it wants to log */
    public static final boolean LOG = true;

    /** configuration file path */
    private static final String CONFIG_FILE = "jjpf/client/jjpf.client.config";

    /** this hashtable contains workers ServiceID */
    protected Hashtable workersHash = null;

    /** sharedobject ServiceID */
```

```
protected ServiceID sharedObjectID = null;

/** Holds value of property MAXWORKER. */
private int MAXWORKER = Integer.MAX_VALUE;
private Vector serviceObservers              = null;
private LinkedList sharedObjectObservers     = null;
private Object monitor;
private Configuration config                 = null;
private ServiceTemplate serviceTemplate      = null;
private ServiceTemplate sharedObjectTemplate = null;
private LookupDiscovery ld                   = null;
private boolean firstNotNull                 = true;

/** shared object classtype */
Class sharedObjClass = null;

/** Creates a new instance of AbstractClient
 * @param clArray
 * @param sharedObject
 * @throws IOException
 * @throws ConfigurationException
 */
public AbstractClient(Class[] clArray, Class sharedObject)
               throws IOException, ConfigurationException
{
    // if the first class contained in clArray does
    // not implement the ProcessoIf interface it will
    // launch an exception
    if(!(ProcessoIf.class.isAssignableFrom(clArray[0])))
        throw new ClassCastException();

    if(sharedObject != null) {
        sharedObjClass = sharedObject;

        // if the first class sharedObject does
        // not implement the SharedObjectIf interface
        // it will launch an exception
        if(!(SharedObjectIf.class.isAssignableFrom(sharedObject)))
            throw new ClassCastException();

        // creates an array containing the SharedObject class
        Class[] sharedObjectClass = new Class[]{ SharedObjectIf.class };


        // ServiceTemplate generation
        sharedObjectTemplate = new ServiceTemplate(null, sharedObjectClass,
                                                   null);


        // SharedObject Observer List
        sharedObjectObservers = new LinkedList();
    }


    // worker hash table initialization
    workersHash = new Hashtable();


    // Service observers Vector
    serviceObservers = new Vector();

    // Observer Configuration
    String[] configArgs = new String[]{ CONFIG_FILE };
    config = ConfigurationProvider.getInstance(configArgs);

    // reads ClassServer configuration
    String baseDir         = (String) config.getEntry("Client",
                                                      "classServerBaseDir",
                                                      String.class, null);
    String classServerPort = (String) config.getEntry("Client",
                                                      "classServerPort",
                                                      String.class, null);

    // creates ClassServer
    ClassServer cs = new ClassServer(
                         new String[]{
        "-port", classServerPort, "-dir", baseDir
    }, null);

    // reads security configuration
    String securityPolicy = (String) config.getEntry("Client",
```

```
                                                    "securPolicy",
                                                    String.class, null);
        System.setProperty("java.security.policy", securityPolicy);

        // sets codebase property
        String codebase = (String) config.getEntry("Client", "codebase",
                                                   String.class, null);
        System.setProperty("java.rmi.server.codebase", codebase);


        // starts security manager
        System.setSecurityManager(new RMISecurityManager());

        // creates class array for
        // the template generation
        Class[] classi = new Class[]{ ServiceIf.class };


        // Creates search template
        serviceTemplate = new ServiceTemplate(null, classi, null);


        // starts multicast discovery
        ld = new LookupDiscovery(LookupDiscovery.NO_GROUPS);
        ld.addDiscoveryListener(this);
    }

    /**
     * Sets the groups where client will
     * perform the search
     * @param groups the groups names
     */
    public final void setLookup(String[] groups)
    {
        try {
            ld.setGroups(groups);

            if(LOG)
                log("Settati gruppi Lookup");
        } catch(IOException ioe) {
            System.out.println(
                    "Errore durante la connessione a gruppi del client: " +
                    ioe);
        } catch(IllegalStateException ise) {
            System.out.println(
                    "Errore durante la connessione a gruppi del client: " +
                    ise);
        } catch(UnsupportedOperationException uoe) {
            System.out.println(
                    "Errore durante la connessione a gruppi del client: " +
                    uoe);
        }
    }

    /**
     *  starts client execution, exits when all task
     *  has been accomplished
     */
    public void run()
    {
        // suspends itself
        monitor = new Object();
        synchronized(monitor) {
            try {
                monitor.wait();
            } catch(Exception e) {
                System.err.println("Errore nella wait");
            }
        }

        finalization();
    }

    public final void discarded(DiscoveryEvent e)
    {
    }

    public final void discovered(DiscoveryEvent e)
    {
```

```
        ServiceRegistrar[] serviceRegis = e.getRegistrars();
        ServiceMatches     servMatch = null;
        ServiceMatches     sharObj   = null;
        ServiceItem        si        = null;

        if(LOG)
            log("Trovati " + serviceRegis.length + " ServiceRegistrar");

        // it slides all the found objects
        for(int i = 0; i < serviceRegis.length; i++) {
            try {
                // adds an observer
                serviceObservers.add(
                        new ServiceObserver(config, serviceRegis[i], this));

                if(LOG)
                    log("Aggiunti Observers");
            } catch(RemoteException reme) {
                continue;
            } catch(ConfigurationException ce) {
                if(LOG)
                    log("File di configurazione del ServiceObserver errata o mancante");

                throw new RuntimeException(
                        "ConfigurationException creating new ServiceObserver",
                        ce);
            }

            //  Searching for a service that match
            //  the template
            try {
                servMatch = serviceRegis[i].lookup(serviceTemplate, MAXWORKER);

                // if a shared object must be found,
                // it looks for it
                if(sharedObjClass != null)
                    sharObj = serviceRegis[i].lookup(sharedObjectTemplate, 1);
            } catch(RemoteException re) {
                continue;
            }

            if((sharedObjClass != null) && (sharObj != null)) {
                // Saves Shared Object ServiceID
                if((sharedObjectID == null) && (sharObj.totalMatches > 0)) {
                    foundSharedObject(sharObj.items[0].serviceID);
                }
            }

            // Slides services matching search template
            if(servMatch.items != null) {
                if(LOG)
                    log("Il numero di elementi che matchano il ServiceTemplate e' " +
                        servMatch.items.length);

                for(int j = 0; j < servMatch.items.length; j++) {
                    // saves serviceItem
                    si = servMatch.items[j];

                    if(si != null) {
                        if((si.service != null) && (si.serviceID != null)) {
                            workerDiscovered(si);
                        }
                    }
                }
            } //fi(servMatch.items != null)
        } // endfor
    }

    /** All the extending classes must
     *  implements this method to execute some
     *  finalization operation.
     */
    abstract void finalization();

    /** All the extending classes must
     *  implements this method to have a
     *  notify when some workers leaves.
     *  @param serviceItem service item describes
     *  which workers have gone
     */
    abstract void workerLeft(ServiceItem serviceItem);
```

```
    /** All the extending classes must
     *  implements this method to have a
     *  notify when some new workers are found.
     *  @param serviceItem service item describes which workers have been found
     */
    abstract void workerDiscovered(ServiceItem serviceItem);

    /** All the extending classes must
     *  implements this method to have a
     *  notify when a shared object is found.
     *  @param shrObjID shared object found
     */
    abstract void foundSharedObject(ServiceID shrObjID);

    /** All the extending classes must implements
     *  this method to obtain JJPF logs
     * @param s string to be logged
     */
    abstract void log(String s);

    /** Getter for property MAXWORKER.
     * @return Value of property MAXWORKER.
     *
     */
    public int getMAXWORKER()
    {
        return this.MAXWORKER;
    }

    /** Setter for property MAXWORKER.
     * @param MAXWORKER New value of property MAXWORKER.
     *
     */
    public void setMAXWORKER(int MAXWORKER)
    {
        this.MAXWORKER = MAXWORKER;
    }

    /** Stop the discovering of new Services */
    protected void stopServiceDiscovery()
    {
        // removes Jini discovery listener
        ld.removeDiscoveryListener(this);
        ld.terminate();

        // stop service renewal
        Iterator it = serviceObservers.iterator();

        while(it.hasNext()) {
            try {
                ServiceObserver so = (ServiceObserver) it.next();
                so.reg.getLease().cancel();
            } catch(UnknownLeaseException ule) {
                continue;
            } catch(RemoteException re) {
                continue;
            }
        }
        synchronized(monitor) {
            monitor.notify();
        }
    }
}
```

C.25 jjpf.client.BasicClient.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
```

```
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.client;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

import java.util.Collection;
import java.util.LinkedList;

import jjpf.client.TaskHolder;

import jjpf.common.ProcessoIf;

import net.jini.config.ConfigurationException;

import net.jini.core.lookup.ServiceID;
import net.jini.core.lookup.ServiceItem;


/** Basic fully functional Client
 * @author Patrizio Dazzi
 */
public class BasicClient extends jjpf.client.AbstractClient
{
    private Object         threadSynchro      = null;
    private BufferedWriter fileStream         = null;
    private TaskHolder     taskHolder         = null;
    private LinkedList     waitingQueue       = null;
    private Class[]        classesArray       = null;
    private int            numeroThreadAttivi = 0;
    private String         lineSeparator      = (String) java.security.
                                                    AccessController.
                                                    doPrivileged(
                                                        new sun.security.
                                                            action.
                                                            GetPropertyAction(
                                                            "line.separator"));

    /** Creates a new instance of BasicClient
     * @param clArray
     * @param sharedObject
     * @param input
     * @param output
     * @throws IOException
     * @throws ConfigurationException
     */
    public BasicClient(Class[] clArray, Class sharedObject, Collection input,
                       Collection output) throws IOException,
                                                 ConfigurationException
    {
        super(clArray, sharedObject);


        // classArray of the process to export
        classesArray = clArray;


        // Task container
        taskHolder = new TaskHolder(input, output);


        // initialize SharedServiceID Queue
        waitingQueue = new LinkedList();


        // synchronization object
        threadSynchro = new Object();

        // logger initilization
        if(AbstractClient.LOG)
            fileStream = new BufferedWriter(new FileWriter("jDP.log", false));
    }
```

```
/** executed when a worker is left
 * @param serviceItem the ServiceItem
 *        of the left service
 */
void workerLeft(ServiceItem serviceItem)
{
    synchronized(threadSynchro) {
        if(LOG)
            log("Disarruolato il worker con serviceID: " +
                serviceItem.serviceID);

        workersHash.remove(serviceItem.serviceID);
        numeroThreadAttivi--;

        if(AbstractClient.LOG)
            log("Worker Attivi: " + numeroThreadAttivi);

        if((numeroThreadAttivi == 0) && (!taskHolder.hasMoreTask())) {
            if(LOG)
                log("Terminato il Discovery di nuovi Worker");

            System.out.println("Terminato il Discovery di nuovi Worker");

            try {
                fileStream.flush();
            } catch(IOException ioe) {
                System.err.println("During log Flushing" + ioe);
            }

            stopServiceDiscovery();
        }
    }
}

/** executed when a worker is found
 * @param serviceItem the serviceitem
 *          of the new service
 */
void workerDiscovered(ServiceItem serviceItem)
{
    synchronized(threadSynchro) {
        // controls the serviceID of the found
        // service in order to avoid to enlist
        // two times the same worker
        if((!workersHash.containsKey(serviceItem.serviceID)) &&
           (taskHolder.hasMoreTask())) {
            // puts the found service in the
            // worker hashtable
            workersHash.put(serviceItem.serviceID, serviceItem.service);

            // if a user has been specified a sharedobject
            // the program controls if the sharedobject has been
            // found before starting a thread
            if((sharedObjectID != null) || (sharedObjClass == null)) {
                // creates new Thread
                BasicClientThread BTclient = new BasicClientThread(
                                                    classesArray,
                                                    serviceItem,
                                                    taskHolder, this);


                // Sets shared object
                BTclient.setSharedObject(sharedObjectID);

                if(AbstractClient.LOG)
                    log("ServiceID: " + serviceItem.serviceID +
                        " : Arruolato");


                // starts the thread
                BTclient.start();


                // increases thread number
                numeroThreadAttivi++;

                if(AbstractClient.LOG)
                    log("Worker Attivi: " + numeroThreadAttivi);
            } else {
                waitingQueue.add(serviceItem);
            }
        } // fi()
    } // end synchronized(threadSynchro)
```

```
    }

    /** invoked when a sharedobject is found
     * @param shrObjID the shared object found
     */
    void foundSharedObject(ServiceID shrObjID)
    {
        sharedObjectID = shrObjID;

        if(waitingQueue.size() > 0)
            synchronized(threadSynchro) {
                ServiceItem serviceItem = (ServiceItem) waitingQueue.removeLast();

                // Creates a new Thread
                BasicClientThread BTclient = new BasicClientThread(classesArray,
                                                                   serviceItem,
                                                                   taskHolder,
                                                                   this);


                // Sets shared object
                BTclient.setSharedObject(sharedObjectID);

                if(LOG)
                    log("ServiceID: " + serviceItem.serviceID +
                        " : Arruolato");


                // starts the thread
                BTclient.start();


                // increases thread number
                numeroThreadAttivi++;

                if(LOG)
                    log("Worker Attivi: " + numeroThreadAttivi);
            } // synchronized(threadSynchro)
    }

    /** Used to perform finalization operations. */
    void finalization()
    {
        try {
            fileStream.close();
        } catch(IOException ioe) {
            System.err.println("Closing logfile : " + ioe);
        }
    }

    /** logs String content
     * @param s String to log
     */
    void log(String s)
    {
        try {
            StringBuffer sb = new StringBuffer();
            sb.append(lineSeparator);
            sb.append(System.currentTimeMillis());
            sb.append(lineSeparator);
            sb.append(s);
            sb.append(lineSeparator);
            fileStream.write(sb.toString());
        } catch(IOException ioe) {
            System.err.println("During Logging : " + s + " : " + ioe);
        }
    }
}
```

C.26 jjpf.client.BasicClientThread.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
```

```
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */
package jjpf.client;

import java.io.IOException;

import java.rmi.RemoteException;

import java.util.NoSuchElementException;
import java.util.logging.Logger;

import jjpf.client.TaskHolder;

import jjpf.common.ServiceIf;

import net.jini.core.lookup.ServiceID;
import net.jini.core.lookup.ServiceItem;


/** This class helps service management at client side.
 * @author Patrizio Dazzi
 */
public class BasicClientThread extends Thread {

    private boolean        workerDied = false;
    private ServiceIf      worker = null;
    private Class[]        classesArray  = null;
    private ServiceItem    serviceItem   = null;
    private TaskHolder     taskHolder    = null;
    private ServiceID      shObj         = null;
    private AbstractClient threadManager = null;

    /** Creates a new instance of BasicClientThread
     * @param clArray class array to send to the
     *            managed worker.
     * @param srvItm worker serviceItem
     * @param taskH TaskHolder object is used to read
     *          tasks to compute. It is used moreover
     *          in order to save
     * @param threadMngr thread manager object.
     *          This is an handle to the client.
     */
    public BasicClientThread(Class[] clArray, ServiceItem srvItm,
                             TaskHolder taskH,
                             AbstractClient threadMngr) {

        classesArray  = clArray;
        serviceItem   = srvItm;
        taskHolder    = taskH;
        threadManager = threadMngr;

        worker = (ServiceIf) srvItm.service;

    }

    /** sets the shared object
     * @param sID Shared Object ServiceID.
     *           This thread sends it to the managed service.
     */
    void setSharedObject(ServiceID sID) {

        shObj = sID;

    }

    /** Starts the thread. */
    public void run() {

        try {

            try {
```

```
// locks the worker
worker.lock();

if(threadManager.LOG)
    threadManager.log("ServiceID: " +
                      serviceItem.serviceID +
                      " : Eseguita Lock");


// loads code on the worker
worker.setCode(classesArray);


// says to the worker the Shared
//  Object ServiceID
worker.setSharedService(shObj);

Object elem = null;

// while there are more task, it reads a
//  new task to send to worker for processing
while((taskHolder.hasMoreTask()) && (!workerDied)) {

    try {


        // reads a new task to process
        elem = taskHolder.nextTask();

        try {

            if(threadManager.LOG)
                threadManager.log("ServiceID: " +
                                  serviceItem.serviceID +
                                  " : Spedisco task");

            taskHolder.commitTask(worker.exec(elem));

            taskHolder.commitTask(worker.getData());

            if(threadManager.LOG)
                threadManager.log("ServiceID: " +
                                  serviceItem.serviceID +
                                  " : Ricevuto risultato");

        } catch(RemoteException re) {

            System.out.println(
                    "Thread terminato in modo non corretto:" + re);


            // if an error occours  it says to the object
            //  to send the data to an other worker
            taskHolder.lostTask(elem);

            if(threadManager.LOG)
                threadManager.log("ServiceID: " +
                                  serviceItem.serviceID +
                                  " : Collegamento interrotto,"+
                                  " Task segnalato come perduto");

            workerDied = true;

            break;

        }

    } catch(NoSuchElementException nsee) {

        if(threadManager.LOG)
            threadManager.log(
                    "Terminati i task da eseguire");

        break;

    }

}


// frees the worker
worker.unlock();

if(threadManager.LOG)
    threadManager.log("ServiceID: " +
```

```
                                    serviceItem.serviceID +
                                    " : UnLockato il Worker");

            } catch(RemoteException re) {

                if(threadManager.LOG)
                    threadManager.log("ServiceID: " +
                                    serviceItem.serviceID +
                                    " : " + re);

                workerDied = true;

            } catch(IOException ioe) {

                if(threadManager.LOG)
                    threadManager.log("ServiceID: " +
                                    serviceItem.serviceID +
                                    " : " + ioe);

                workerDied = true;

            }

        } catch(RuntimeException re) {

            if(threadManager.LOG)
                threadManager.log("ServiceID: " +
                                serviceItem.serviceID + " : " + re);

        }
         finally {


            // removes the ServiceID from Service Hash
            threadManager.workerLeft(serviceItem);

        }

    }
}
```

C.27 jjpf.client.ServiceObserver.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */
package jjpf.client;

import java.rmi.RemoteException;

import java.util.logging.Logger;

import jjpf.common.ServiceIf;
import jjpf.common.SharedObjectIf;

import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;

import net.jini.core.event.EventRegistration;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.UnknownEventException;
import net.jini.core.lease.Lease;
```

```
import net.jini.core.lookup.ServiceEvent;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;

import net.jini.export.Exporter;

import net.jini.lease.LeaseRenewalManager;


/** JJPF service and sharedobject
 *  observer used by a client.
 *  @author Patrizio Dazzi
 */
public class ServiceObserver implements RemoteEventListener {

    /** lease renewal manager */
    protected static LeaseRenewalManager leaseManager =
            new LeaseRenewalManager();

    /** lookup service registrar object */
    protected ServiceRegistrar registrar;

    /** transition bitmask, used to select
     *  what kind of notify it is interested to
     *  receive.
     */
    protected final int transitions = ServiceRegistrar.TRANSITION_NOMATCH_MATCH;

    /** service event registration object */
    public EventRegistration reg = null;

    /** shared object event registration object */
    public EventRegistration regSharedObject = null;
    private AbstractClient   client;

    /** Creates a new instance of ServiceObserver
     * @throws RemoteException
     */
    public ServiceObserver() throws RemoteException {

    }

    /**
     * Creates a new instance of ServiceObserver
     *
     * @param config exporter configuration file
     * @param rgistr lookup service registrar
     * @param clnt client object
     * @throws RemoteException
     * @throws ConfigurationException thrown when
     *         no configuration file is found
     */
    public ServiceObserver(Configuration config,
                           ServiceRegistrar rgistr,
                           AbstractClient clnt) throws RemoteException,
                                                       ConfigurationException {

        client = clnt;


        // Service Registrar
        this.registrar = rgistr;

        // creates an exporter
        Exporter exporter = (Exporter) config.getEntry("Client",
                                                       "serviceObserverExporter",
                                                       Exporter.class,
                                                       null);

        // exports a ServiceObserver
        RemoteEventListener proxy = (RemoteEventListener) exporter.export(
                                             this);

        // creates classes array for
        // template building
        Class[] classArray = new Class[]{ ServiceIf.class/*, SharedObjectIf.class*/
         };

        Class[] sharedObjectClassArray = new Class[]{
            SharedObjectIf.class
        };
```

```
        // Creates shared object search template
        ServiceTemplate templSharedObject = new ServiceTemplate(null,
                                                                 sharedObjectClassArray,
                                                                 null);

        // Creates service search template
        ServiceTemplate templ = new ServiceTemplate(null, classArray,
                                                    null);


        // asks the service lookup for being
        // informed when a new service is discovered
        regSharedObject = registrar.notify(templSharedObject,
                                           transitions, proxy, null,
                                           Lease.ANY);
        reg = registrar.notify(templ, transitions, proxy, null,
                               Lease.ANY);


        // activates Lease manager
        leaseManager.renewUntil(regSharedObject.getLease(),
                                Lease.FOREVER,
                                ((regSharedObject.getLease()
                                                 .getExpiration() -
                                   System.currentTimeMillis()) / 2),
                                null);

        leaseManager.renewUntil(reg.getLease(), Lease.FOREVER,
                                ((reg.getLease().getExpiration() -
                                                 System.currentTimeMillis()) / 2),
                                null);
    }

    /** method called when a new shared object is found
     * @param evt discovery event
     * @throws UnknownEventException thrown when the
     *         event received is unknown
     * @throws RemoteException thrown when a Remote
     *         exception occours
     */
    public void notify(RemoteEvent evt) throws UnknownEventException,
                                               RemoteException {

        ServiceEvent sevt = (ServiceEvent) evt;

        int          transition = sevt.getTransition();

        if((transition == ServiceRegistrar.TRANSITION_NOMATCH_MATCH) &&
           (sevt.getServiceItem() != null)) {

            if(ServiceIf.class.isAssignableFrom(
                       sevt.getServiceItem().service.getClass())) {


                // says to client that a new worker
                // has been found
                client.workerDiscovered(sevt.getServiceItem());

            }

            if(SharedObjectIf.class.isAssignableFrom(
                       sevt.getServiceItem().service.getClass())) {

                // says to client that the sharedobject
                // has been found
                if(client.sharedObjClass != null)
                    client.foundSharedObject(
                            sevt.getServiceItem().serviceID);

            }

        }

    }
}
```

C.28 jjpf.client.TaskHolder.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.client;

import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.NoSuchElementException;


/** class used for the management of the
 *  task to process.
 *  @author Patrizio Dazzi
 */
public class TaskHolder {
    private LinkedList faultTaskList = null;
    private Collection output       = null;
    private Object     synchronizer = null;
    private Iterator   inputIter    = null;

    /** Creates a new instance of TaskHolder
     * @param in tasks to process
     * @param out processed tasks
     */
    public TaskHolder(Collection in, Collection out) {
        inputIter = in.iterator();

        output = out;

        faultTaskList = new LinkedList();

        synchronizer = new Object();
    }

    /** iterance condition
     * @return true if thre are more task to process,
     *         false otherwise.
     */
    boolean hasMoreTask() {
        synchronized(synchronizer) {
            // true if there are more task to execute
            if((faultTaskList.size() > 0) ||
               (inputIter.hasNext()))
                return true;
            else

                return false;
        }
    }

    /** getter used to get a task from the taskholder
     * @throws NoSuchElementException thrown when there
     *          aren't any task to process
     * @return a task to process
     */
    Object nextTask() throws NoSuchElementException {
        synchronized(synchronizer) {
            // if there aren't more task, an
            // exception will be thrown
            if(inputIter.hasNext())
                return inputIter.next();

            else if(faultTaskList.size() > 0)
```

```
                return faultTaskList.removeFirst();

            else
                throw new NoSuchElementException("No more Work to do");
        }
    }

    /** used in order to save a task process result
     * @param task prcocessed task
     */
    void commitTask(Object task) {
        synchronized(synchronizer) {
            output.add(task);
        }
    }

    /** used in order to comunicate to taskholder
     *  that a data must be processed newly
     * @param task task to process newly
     */
    void lostTask(Object task) {
        synchronized(synchronizer) {
            faultTaskList.addLast(task);
        }
    }
}
```

C.29 jjpf.util.ByteArray.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.util;

import java.io.Serializable;


/**
 *
 * @author  Patrizio Dazzi
 */
public class ByteArray implements Serializable {

    private byte[] bytearray;

    /** Creates a new instance of ByteArray */
    public ByteArray(byte[] bytA) {

        bytearray = bytA;

    }

    /**
     *  @return the Byte array
     */
    public byte[] getByteArray() {

        return bytearray;

    }

}
```

C.30 jjpf.util.DependencyResolver.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.util;

import com.sun.jini.tool.ClassDep;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

import java.util.Vector;


/**
 *  This class is used in order to examine
 *  the dependencies of a class and
 *  in order to understand which classes
 *  will try to load once sent in execution
 *  @author  Patrizio Dazzi
 */
public class DependencyResolver {

    private static final String STD_FILE = "exclude.dat";

    /** Creates a new instance of
      * DependencyResolver
      */
    public DependencyResolver() {

    }

    /** returns a class array containing
     *  the classes on which input class depends
     *  input class
     * @param classfile the class to analyze
     * @throws IOException
     * @throws FileNotFoundException
     * @throws ClassNotFoundException
     * @return the class array
     */
    public static Class[] classDep(Class classfile)
                          throws IOException, FileNotFoundException,
                                 ClassNotFoundException {

        return classDep(classfile, STD_FILE);

    }

    /** returns a class array containing the classes
     *  on which input class depends. Subsequently
     *  it removes from that Array the classes
     *  pertaining to the package lists defined in
     *  <CODE>excludeFile</CODE>
     * @param classfile the class to analyze
     * @param excludeFile file containing the package
               lists to remove from dependency array.
     * @throws IOException
     * @throws FileNotFoundException
     * @throws ClassNotFoundException
     * @return the class array
     */
```

```
    public static Class[] classDep(Class classfile,
                                   String excludeFile)
                            throws IOException, FileNotFoundException,
                                   ClassNotFoundException {

        BufferedReader br = new BufferedReader(
                                    new FileReader(excludeFile));
        Vector         excludeVec = new Vector();

        while(br.ready()) {

            excludeVec.add(br.readLine());

        }

        return classDep(classfile, excludeVec);

    }

    /** returns a class array containing the classes
     *  on which input class depends. Subsequently
     *  it removes from that Array the classes pertaining
     *  to the package lists defined in <CODE>strVector</CODE>
     * @param classfile the class to analyze
     * @param strVec Vector containing the package lists
     *          to remove from dependency array.
     * @throws IOException
     * @throws FileNotFoundException
     * @throws ClassNotFoundException
     * @return the class array
     */
    public static Class[] classDep(Class classfile, Vector strVec)
                            throws IOException, FileNotFoundException,
                                   ClassNotFoundException {

        String   classname = classfile.getName();
        ClassDep cd = new ClassDep();

        cd.setClassPath(System.getProperty("java.class.path"));
        cd.addClasses(classname);

        for(int i = 0; i < strVec.size(); i++) {

            cd.addHides((String) strVec.elementAt(i));

        }

        String[] strA = cd.compute();

        Class[]  clsA = new Class[strA.length];

        for(int i = strA.length - 1; i >= 0; i--)
            clsA[strA.length - i - 1] = ClassLoader.getSystemClassLoader()
                                                   .loadClass(strA[i]);

        return clsA;

    }
}
```

C.31 jjpf.util.SimplestFormatter.java

```
/*
 * This file is part of Java & Jini Parallel Framework (JJPF)
 * Copyright (C) 2004 Patrizio Dazzi
 *
 * Java & Jini Parallel Framework is a free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of the
 * License, or (at your option) any later version.
 *
 * Java & Jini Parallel Framework is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
```

```
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

package jjpf.util;

import java.io.*;

import java.text.MessageFormat;

import java.util.Date;


/**
 *
 * @author  Patrizio Dazzi
 */
public class SimplestFormatter extends java.util.logging.Formatter {

    private Date dat;

    // Line separator string.  This is
    // the value of the line.separator
    // property at the moment that the
    // SimpleFormatter was created.
    private String lineSeparator = (String) java.security.AccessController.doPrivileged(
                                        new sun.security.action.GetPropertyAction(
                                                "line.separator"));
    Object[]       arguments = new Object[3];

    /** Creates a new instance of
      * SimplestFormatter
      */
    public SimplestFormatter() {

        dat = new Date();

    }

    public String format(java.util.logging.LogRecord record) {

        StringBuffer sb        = new StringBuffer();
        StringBuffer levelName = new StringBuffer();

        arguments[0] = new Long(record.getMillis() - dat.getTime());

        levelName.append(record.getLevel().getLocalizedName());

        while(levelName.length() < 7)
            levelName.append(' ');

        arguments[1] = levelName;
        arguments[2] = formatMessage(record);

        sb.append(MessageFormat.format(
                        "{0,number,0000000000000} | {1} | {2}",
                        arguments));

        sb.append(lineSeparator);
        sb.append("----------------------------------------------------------------");
        sb.append(lineSeparator);

        return sb.toString();

    }

}
```

Appendice D

File di Configurazione

D.1 jjpf.service.core.jjpf.service.core.config

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import java.io.File;

CoreService{

    ServiceIDFile = new File("ServiceID.id");
    SharedObjectObserverExporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                                         new BasicILFactory());

}
```

D.2 jjpf.service.hybrid.jjpf.service.hybrid.config

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import com.sun.jini.config.ConfigUtil;
import jjpf.service.hybrid.HybridServiceClassLoader;
import jjpf.service.hybrid.HybridServiceRegister;

Service{
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory(null,null,HybridServiceRegister.hsc));

    securPolicy = new String("jjpf/service/hybrid/hybrid.start.policy");

    localhost = ConfigUtil.getHostName();

    classServerPort = "8080";
    classServerBaseDir = ".";
    codebase = ConfigUtil.concat(new String[]{
                                              "http://",
                                              localhost,
                                              ":",
                                              classServerPort,
                                              "/"
                                             }
                                );
}
```

D.3 jjpf.service.rmi.jjpf.service.rmi.config

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import com.sun.jini.config.ConfigUtil;


Service{
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());

    securPolicy = new String("jjpf/service/rmi/rmi.start.policy");

    localhost = ConfigUtil.getHostName();

    classServerPort = "8080";
    classServerBaseDir = ".";
    codebase = ConfigUtil.concat(new String[]{
                                              "http://",
                                              localhost,
                                              ":",
                                              classServerPort,
                                              "/"
                                             }
                                 );
}
```

D.4 jjpf.shared.jjpf.shared.config

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import com.sun.jini.config.ConfigUtil;


Service{
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());

    securPolicy = new String("jjpf/shared/shared.start.policy");

    localhost = ConfigUtil.getHostName();

    classServerPort = "5050";
    classServerBaseDir = ".";
    codebase = ConfigUtil.concat(new String[]{
                                              "http://",
                                              localhost,
                                              ":",
                                              classServerPort,
                                              "/"
                                             }
                                 );
}
```

D.5 jjpf.client.jjpf.client.config

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
import com.sun.jini.config.ConfigUtil;

Client{
    serviceObserverExporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                                    new BasicILFactory());

    securPolicy = new String("jjpf/client/client.start.policy");

    localhost = ConfigUtil.getHostName();


    classServerPort = "7070";
```

```
    classServerBaseDir = ".";


    codebase = ConfigUtil.concat(new String[]{
                                                "http://",
                                                localhost,
                                                ":",
                                                classServerPort,
                                                "/"
                                               }
                                 );

}
```