UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

SVILUPPO DI UN'INFRASTRUTTURA *PEER-TO-PEER* A SUPPORTO DI UN SISTEMA PER LA CONDIVISIONE DI INFORMAZIONI

Relatore

Chiar.mo Prof. Ing. A. POGGI

Correlatore

Dott. Ing. M. MARI

Tesi di Laurea Specialistica di

MICHELE LONGARI

Ai miei genitori e a chi ha

sempre creduto in me.

Ringraziamenti

Premetto che i ringraziamenti e quindi le persone citate in questa sezione non sono elencate seguendo un ordine particole ma ho semplicemente scritto d'istinto, senza pensarci troppo, e per questo mi scuso già in partenza con tutte le persone che involontariamente potrei avere dimenticato.

Prima di tutto vorrei ringraziare il prof. Agostino Poggi e l'ing. Marco Mari, in qualità rispettivamente di relatore e correlatore della mia tesi, per avermi saputo guidare ed indirizzare in questa attività, evitandomi inutili e fuorvianti "sbandate", nonché per avermi proposto questo progetto che si è rivelato molto interessante e didattico, sia a livello di programmazione che di progettazione di sistemi.
E' d'obbligo anche ringraziare tutto lo staff dei laboratori AOT, per la serietà e disponibilità dimostrati durante tutto il mio periodo di studio.

Poi naturalmente desidero dire grazie a tutta la mia famiglia, ai miei genitori, ai miei nonni, zii, cugini e cagnolini (Carly).
In particolare volevo ovviamente ringraziare mia madre Simonetta e mio padre Ernesto, i miei mecenati, per avermi dato gli stimoli ed il sostegno necessari prima per iniziare questa "avventura" accademica con la laurea triennale e poi per continuarla con quella specialistica.

Inoltre ringrazio veramente di cuore anche tutti i miei compagni di corso, alcuni datati nel tempo, altri conosciuti da poco durante la laurea specialistica. Ciascuno di loro è stato di grande importanza nel darmi qualche dritta per un

esame, un aiuto in un progetto o anche solo accompagnandomi nel peso dello studio.
Come non ringraziare con un tocco di nostalgia Mirko “aeroplano” Boggian e Lorenzo “facciamo-una-cena” Mignani: anche se vi ricorderò di più per i *record* a TrackMania e per le abbuffate a pranzo, piuttosto che per i bei voti ed i pomeriggi di studio, sappiate che vi considero dei geni (incompresi ma pur sempre geni).
Fortunatamente lungo il mio cammino ho conosciuto anche persone, sempre simpatiche e con tanta voglia di scherzare, ma decisamente più serie, che mi hanno aiutato ognuno nel suo specialissimo modo. Sto parlando di tutti i miei altri carissimi compagni di Ingegneria informatica che ringrazio di cuore.
Tra questi in particolare voglio ricordare i più storici e fondamentali durante i miei anni di studi: la Giusy, la Leo, il Lore, Luca, il Matte ed il Rob.

E per ultimi, ma mai ultimi, ringrazio tutti i miei amici di San Secondo, per i momenti di svago e festa che ho condiviso con loro, è grazie anche a quei momenti che sono riuscito a ricaricarmi dopo i periodi pesanti di stress.
Naturalmente inizio col ringraziare l’Elena, che innegabilmente ha condiviso con me più degli altri i miei momenti “si” ed i miei momenti “no”, sopportandomi pazientemente e standomi vicino nei momenti in cui sembravo aver perso la concentrazione e la voglia di studiare. Grazie al suo aiuto ed a quello dei miei genitori (direi quasi in stereo-fonia) sono riuscito a tenere duro fino alla fine raggiungendo questo traguardo molto importante.
E finisco col salutare e ringraziare tutti gli altri “sgùndèn”, che nei più svariati modi si sono rivelati importanti in questi anni, eccoli: l’Anna, l’A.S.D. San Secondo 1917 (…alè alè alè sàn sgònd alè…), il Bigo, Biòla, i ragazzi del “calcetto al martedì”, la Cami, David, Dj Patto, Dj Pippo, Ettore, la Fiorella, Flebo, Gabri, il Galla, il Giava (un grandissimo cassiere), il Giò, i lavoratori delle feste in Rocca, Leloni, Lori, Max, i mangiatori delle “Milkate”, la Nanà, il Pavo, il Pesso, Pol, il Rodo, il mio erede calcistico e musicale Sange, il Sime, la Sté, la Susy, i ragazzi del Torino Club 2005, gli assidui frequentatori del “TULLOng’s Blog”, la Vez, i 300 di “villa Luigina”, Zaumy e… chi più ne ha più ne metta!

Indice

Indice delle Figure……..……………………………..…………………v

Indice dei Listati………..………………………...………..………...vii

Prefazione…………………………………………………...…………viii

1 Introduzione…….…………………………………………………..10

1.1 Sistemi *peer-to-peer*……………….……………………………10

1.2 Classificazione dei sistemi P2P………..…………………...………...14

1.2.1 Sistemi strutturati…………………………………………..14

1.2.2 Sistemi non-strutturati……………………...………………15

1.3 Interoperabilità tra sistemi P2P…….……….……………………..16

1.4 Sistemi multi-agente: MAS………………..……………………...17

1.4.1 Integrazione dei MAS con il Web……………………….....20

2 Strumenti e Tecnologie Utilizzate………...……...……………………............22

2.1 Astrazione dell'architettura: il *framework* Spring……………..……..22

2.2 Supporto MAS: JADE …………..……………….………………….....25

2.3 Supporto P2P: JXTA………………………………………………27

2.3.1 Protocolli………………………………………………...……29

2.3.2 *Advertisement*………………………………………………31

2.3.3 *Peer*………………………………………………...…………31

2.3.4 *Pipe*………………………………………………….………..32

2.4 RAIS: *Remote Assistant for Information Sharing*……….…………...33

3 Il *framework* Spring………..……………….…………………………………...41

3.1 Caratteristiche principali……………………………………..……41
3.2 *Dependency Injection*………………..……………………………43
3.3 *Inversion of Control Container*………………..……………..…..44
3.4 *BeanFactory* e *ApplicationContext*……………………………….45
3.4.1 *Beans* e *BeanFactory*……………………………………....46
3.4.2 Proprietà, collaboratori e controllo delle dipendenze……...49

4 L'integrazione JADE-JXTA-RAIS…….…….…….………………….…..51
4.1 Progetti di partenza: *jade-jxta* e RAIS...….........…………..……...51
4.2 Modifiche apportate………………...…………………………..54
4.2.1 Modifiche a *jade-jxta*…………………………..……....54
4.2.2 Modifiche a RAIS……………………………………....59
4.2.2.1 Modifiche al *DesktopSearchAgent*........................60
4.2.2.2 Modifiche al *LocalSearchAgent*............................63
4.2.2.3 Modifiche ai *Behaviours*.......................................66
4.2.2.4 Il *TimerBehaviour*...73
4.3 Risultati ottenuti…………………………...………………………75

5 L'architettura RAIS………………………………………...…………...79
5.1 Architettura astratta per sistemi *peer-to-peer*……………………..79
5.1.1 Sotto-sistemi funzionali……………………………….....83
5.1.2 Implementazione JADE……………………………………98
5.2 Realizzazione dell'architettura RAIS..………………………………99
5.2.1 Modifiche all'architettura di partenza..……………………...100
5.2.1.1 La classe *JadePlatformConfig*…………………102
5.2.1.2 La classe *JadeUserPeer*………………………...103
5.3 Risultati ottenuti...……………………………………………….106

Conclusioni……………………………………...……………………...109
Bibliografia……………………………………...……………………..111

Indice delle Figure

1.1 Ciclo di vita di un agente informatico..……………………..……………18

1.2 Schema modulare di un MAS……………………………………………19

2.1 Moduli di Spring…..……………………………………..……………24

2.2 Scenari di utilizzo di Spring……………………..……………............25

2.3 Architettura JADE…..……………………………………..……………..26

2.4 Schema base di un sistema P2P……………………...………………..28

2.5 Stack dei protocolli JXTA…………………………...………………...30

2.6 L'architettura multi-agente RAIS……………………...……………...34

2.7 La *Graphic User Interface* di RAIS……………………………………...36

3.1 *Enterprise JavaBean*: il *wrapping* di POJO già implementati…………...43

4.1 Esempio di una rete di *overlay*…………………………………………...52

4.2 Processo comunicativo base di JXTA…………………………….……...53

4.3 Diagramma delle interazioni tra gli agenti RAIS………………………...67

4.4 La nuova piattaforma JADE-JXTA-RAIS…………………..…………...76

4.5 *Testing* di RAIS senza LSA locale…………………………………….....77

4.6 *Testing* di RAIS con LSA locale…….……………………………………...78

5.1 Diagramma delle classi del *package it.unipr.aot.p2p.platform*....…........84

5.2 Diagramma delle classi del *package it.unipr.aot.p2p.id*…………………86

5.3 Diagramma delle classi del *package it.unipr.aot.p2p.peer*………………88

5.4 Diagramma delle classi del *package it.unipr.aot.p2p.channel*…………..90

5.5 Diagramma delle classi del *package it.unipr.aot.p2p.peergroup*………..93
5.6 Diagramma delle classi del *package it.unipr.aot.p2p.message*…………95
5.7 Diagramma delle classi del *package it.unipr.aot.p2p.discovery*..………..97
5.8 Il nuovo *startup* del sistema RAIS…………………………….…..……107

Indice dei Listati

3.1 Implementazione di *XmlBeanFactory*……………………………………46

3.2 Composizione del file XML per l'*XmlBeanFactory* …………………….47

4.1 Il metodo *searchJxtaDF()*…………….…………………………………56

4.2 I nuovi controlli per il metodo *handleSearch()* del JxtaDF……………...58

4.3 I quattro nuovi metodi della GUI nel *DesktopSearchAgent*………...……62

4.4 La registrazione del *LocalSearchAgent* sul JxtaDF……………………...65

4.5 La nuova gestione eventi nel *ReceiveResultsBehaviour*………………...69

4.6 Arrivo di un *mt* e suo processo nel *SetNumResultsBehaviour*…….……..71

4.7 Gestione del *TimerBehaviour* nel *SetNumResultsBehaviour*…………....72

4.8 Il nucleo operativo del *TimerBehaviour*………………………………....74

5.1 La nuova struttura di RAIS.xml……………………………………….. ..101

5.2 La creazione del profilo JADE nella classe *JadePlatformConfig*...…….103

5.3 Creazione dell'entità *peer* nella classe *JadeUserPeer*.………………...105

Prefazione

Negli ultimi anni è sotto gli occhi di tutti il ruolo di primaria importanza assunto da Internet e dalle tecnologie legate al *Web* per quanto riguarda servizi indirizzati ad aziende e privati. Tra questi servizi vi è sicuramente il *file-sharing*, che rende possibile la condivisione di materiale elettronico tra due o più utenti utilizzando reti di tipo *peer-to-peer*.

Generalmente per *peer-to-peer* (o P2P) si intende una rete di computer o qualsiasi rete informatica che non possiede *client* o *server* fissi, ma un numero di nodi equivalenti (*peer*, appunto) che fungono sia da *client* che da *server* verso altri nodi della rete.

Questo modello di rete è l'antitesi dell'architettura *client-server*. Mediante questa configurazione qualsiasi nodo è in grado di avviare o completare una transazione. Detto questo si può affermare con sicurezza che i sistemi *peer-to-peer* presentano alcuni sostanziali vantaggi rispetto ai sistemi tradizionali *client-server*: una maggiore scalabilità, una buona tolleranza ai guasti (*fault tolerance*), efficienza nello sfruttare le risorse di rete inutilizzate, maggiori performance e minori costi di amministrazione e gestione. Tuttavia l'adozione del modello *peer-to-peer* comporta anche alcuni svantaggi sostanziali, in particolare per quanto riguarda l'aspetto della sicurezza del sistema e l'eterogeneità dei *device hardware* che costituiscono i nodi della rete *peer-to-peer*.

Quindi in poche parole un sistema *peer-to-peer* può essere visto architetturalmente come un sistema distribuito nel quale ogni nodo possiede le

stesse capacità e responsabilità ed è anche in grado di instaurare una comunicazione.

Le principali caratteristiche dei sistemi *peer-to-peer* sono la capacità di auto-organizzarsi, il controllo completamente decentralizzato e la potenziale simmetria delle comunicazioni. Tutte queste peculiarità sono messe a servizio di quello che è lo scopo principale dei sistemi *peer-to-peer*: la condivisione di risorse e servizi (appunto il *file sharing*).

Il lavoro svolto in questo progetto di tesi pone le proprie fondamenta appunto nei sistemi *peer-to-peer*, ma vi sono anche altri protagonisti fondamentali: i sistemi multi-agente.
Infatti, da qualche anno a questa parte, i sistemi multi-agente (*Multi Agents Systems*, MAS), stanno ottenendo popolarità in diversi settori, fornendo assistenza all'utente in svariati ambienti di sviluppo. In parole povere, la nascita e la crescita di sistemi di questo tipo, si basa sulla cooperazione e sul "dialogo intelligente" degli agenti, entità autonome capaci di muoversi in rete, reperire informazioni e di interfacciarsi fra di loro.
Cosa sia in realtà un agente e come esso differisca da un normale programma è stato per lunghi anni argomento di dibattito. Sempre più spesso vengono proposte applicazioni dove gli agenti sono definiti come programmi che assistono gli utenti e agiscono per conto di essi. Dal punto di vista dell'utilizzatore finale del prodotto, è possibile definire un agente come un programma che lo aiuta nelle sue attività, permettendo alle persone di delegare parte del proprio lavoro.

Dall'unione di questi due tipi di tecnologie trae origine il sistema sviluppato in questo progetto; infatti abbinando le caratteristiche fondamentali del P2P e dei MAS, è stato possibile creare un architettura completa, in grado di sviluppare una condivisione "intelligente" di documenti e *file*.
Cosa si intende di preciso con l'aggettivo "intelligente", come ciò sia stato raggiunto e quali scenari si prospettano per il futuro di questo sistema, sarà spiegato dettagliatamente nei capitoli successivi.

Capitolo 1

Introduzione

Questo capitolo descrive in generale i due principali sistemi (peer-to-peer e MAS) utilizzati lungo il corso di questo progetto, e le principali problematiche legate ad essi.

1.1 Sistemi *peer-to-peer*

Generalmente per *peer-to-peer* (o P2P) si intende una rete di computer o qualsiasi rete informatica che non possiede nodi gerarchizzati come *client* o *server* fissi, ma un numero di *nodi equivalenti* (pari, in inglese *peer* appunto) che fungono sia da *client* che da *server* verso altri nodi della rete.

Questo modello di rete è l'antitesi dell'architettura *client-server*. Mediante questa configurazione qualsiasi nodo è in grado di avviare o completare una transazione. I nodi equivalenti possono differire nella configurazione locale, nella velocità di elaborazione, nell'ampiezza di banda e nella quantità di dati memorizzati. Il modello *client-server* è ampiamente utilizzato e diffuso, ma presenta alcuni svantaggi che lo rendono difficilmente utilizzabile in alcuni casi:

1. scalabilità: il modello *client-server* è poco scalabile, infatti per aumentare le prestazioni è necessario aumentare la potenza del server;
2. *fault tolerance*: poiché il controllo è centralizzato nel *server*, esso costituisce l'unico punto di fallimento del sistema. Quindi il sistema è sensibile al fallimento del server;
3. sfrutta in modo non bilanciato le risorse di rete;
4. richiede capacità di amministrazione.

I sistemi peer-to-peer risolvono alcuni dei problemi del modello *client-server*:

1. scalabilità: permettono di gestire una grande quantità di utenti, ovvero di richieste di risorse e di servizi;
2. *fault tolerance:* possono continuare ad offrire il servizio anche in presenza di guasti;
3. sfruttano in modo efficiente le risorse di rete inutilizzate;
4. ottime *performance*;
5. richiedono basse capacità di amministrazione, poiché sono in grado di auto-organizzarsi.

Come è già stato accennato sopra, i sistemi P2P presentano numerosi vantaggi ma anche alcuni svantaggi, soprattutto relativamente al tipo di ambiente in cui si decide di installare questo tipo di rete.
Sicuramente uno degli innumerevoli vantaggi è che non si deve acquistare un *server* con potenzialità elevate e quindi non se ne deve sostenere il costo, però dall'altro lato c'è da calcolare che ogni computer (*peer*) deve avere ed assicurare i requisiti tecnici per sostenere la richiesta computazionale sia del singolo utente, sia degli altri utenti che desiderano accedere remotamente alle sue risorse.
Un altro aspetto fondamentale di cui tenere sicuramente conto, è che ogni utente condivide localmente le proprie risorse ed è amministratore del proprio *client-server*. Questo da un lato può essere positivo per una questione di "indipendenza", ma dall'altro richiede competenze di medio-alto livello ad ogni utente, soprattutto per quel che concerne la protezione del sistema.

Non tutte le reti *peer-to-peer* funzionano allo stesso modo o presentano architetture uguali.
Infatti alcune reti e canali, usano il modello *client-server* per alcuni compiti (per esempio la ricerca) e il modello *peer-to-peer* per tutti gli altri. Proprio questa doppia presenza di modelli, fa sì che tali reti siano definite "ibride". Altre reti invece, vengono definite come il vero modello di rete *peer-to-peer* in quanto utilizzano una struttura *peer-to-peer* per tutti i tipi di transazione, e per questo motivo vengono definite "pure".

Quando il termine *peer-to-peer* venne utilizzato per descrivere la prima rete di questo tipo, implicava che la condivisione dei *file*, resa possibile da questo protocollo, fosse la cosa più importante: in realtà la grande conquista di questa architettura fu quella di mettere tutti i computer collegati sullo stesso piano, ed il protocollo "*peer*" era il modo giusto per realizzarlo.

La maggioranza dei programmi *peer-to-peer* garantisce un insieme di funzionalità minime, che comprende:

- supporto multi-piattaforma, *multi-server*, multicanale: il programma è compatibile con tutti i sistemi operativi, *server* e dispositivi *hardware* (PC *laptop*, portatili, palmari, cellulari);
- supporto protocollo IPv6;
- *download* dello stesso *file* da più reti contemporaneamente;
- offuscamento dell'ID di rete;
- offuscamento del protocollo P2P;
- supporto *proxy*;
- supporto crittografia SSL;
- gestione da remoto, sia da PC/*notebook* che da cellulari e palmari.

Utilizzi più innovativi prevedono l'utilizzo delle reti *peer-to-peer* per la diffusione di elevati flussi di dati generati in tempo reale come per esempio programmi televisivi o film. Questi programmi si basano sull'utilizzo delle banda di trasmissione di cui dispongono i singoli utenti e la banda viene utilizzata per

trasmettere agli altri fruitori il flusso dati. Questa tipologia di programmi in linea di principio non richiede *server* dotati di elevate prestazioni, dato che il *server* fornisce i flussi video a un numero molto limitato di utenti, che a loro volta li ridistribuiscono ad altri utenti. Questo metodo di diffusione permette in teoria la trasmissione in tempo reale di contenuti video, ma richiede che i singoli utenti siano dotati di connessioni ad elevata banda sia in ricezione che in trasmissione, altrimenti si arriverebbe rapidamente a una saturazione della banda fornita dal *server*. È da notare che sebbene in questi contesti spesso si parli di *server*, la tecnologia rimane comunque una tecnologia *peer-to-peer*, dato che il *server* funge da indice, sincronizza i vari utilizzatori che condividono la banda e fornisce il flusso di dati iniziale che poi gli utenti condividono. Quindi sebbene tecnicamente sia un *server* in pratica, dopo aver fornito il flusso dati iniziale ed aver messo in comunicazione i vari utenti, il *server* si disinteressa della comunicazione che diventa totalmente *peer-to-peer*.

Tecnicamente, le applicazione *peer-to-peer* dovrebbero implementare solo protocolli di *peering* che non riconoscono il concetto di "*server*" e di "*client*". Tali applicazioni o reti "pure" sono in realtà molto rare. Molte reti e applicazioni che si descrivono come *peer-to-peer* fanno però affidamento anche su alcuni elementi "*non-peer*", come per esempio il DNS. Inoltre, applicazioni globali utilizzano spesso protocolli multipli che fungono simultaneamente da *client*, da *server* e da *peer*. Reti "*peers*" completamente decentralizzate sono state utilizzate per molti anni, per esempio USENET (1979) e FidoNet (1984).

Ruolo importante in questo campo fu assunto dalla *Sun Microsystems* che aggiunse degli oggetti in linguaggio Java per velocizzare lo sviluppo delle applicazioni P2P a partire dal 1990. In questo modo i programmatori poterono realizzare delle piccole applicazioni *chat* in *real time*, prima che divenisse popolare la rete di *Instant Messaging*.

Questi sforzi culminarono con l'attuale progetto JXTA.

1.2 Classificazione dei sistemi P2P

I sistemi *peer-to-peer* possono essere classificati in base a diversi aspetti. In primo luogo è possibile distinguere diverse tipologie di sistemi *peer-to-peer* in base allo scopo funzionale dei sistemi stessi (*file-sharing*, *backup storage*, *collaborative networks*, *instant messaging*, …).
Una seconda importante classificazione dei sistemi *peer-to-peer*, che è già stata introdotta nel paragrafo precedente, può essere operata in base alle caratteristiche dei nodi che compongono il sistema: si possono distinguere sistemi *peer-to-peer* "puri" e sistemi "ibridi".
Un sistema *peer-to-peer* "puro" è un sistema nel quale un qualsiasi nodo della rete può essere eliminato, senza che questo comporti una significativa degradazione delle performance dell'intero sistema.
In un sistema *peer-to-peer* "ibrido", invece, sono presenti alcuni nodi privilegiati (detti anche *SuperPeer*), i quali sono necessari per l'implementazione di uno o più servizi e possono operare un controllo centralizzato.
Infine una ultima classificazione dei sistemi *peer-to-peer* può essere operata in base alla tipologia di organizzazione delle risorse. In base a questa classificazione i sistemi si dividono in due categorie: sistemi non strutturati e sistemi strutturati.
Ogni sistema *peer-to-peer* definisce una rete virtuale, detta *overlay network*, costruita sopra la rete fisica e costituita da tutti i *peers* del sistema. Due peers direttamente collegati nella *overlay network* possono non esserlo a livello fisico.
La classificazione dei sistemi *peer-to-peer* in sistemi non strutturati e strutturati si basa sulla topologia della *overlay network*: nei sistemi non strutturati la topologia della rete è arbitraria, mentre nei sistemi strutturati essa è stabilita da precise regole.

1.2.1 Sistemi strutturati

I sistemi *peer-to-peer* strutturati definiscono una *overlay network* con precise caratteristiche, tali da facilitare e rendere efficiente il processo di ricerca delle risorse: tramite l'utilizzo di un opportuno protocollo, qualsiasi nodo è in grado di

instradare efficientemente la richiesta di una risorsa, in modo che essa possa giungere ai *peers* che la possiedono, anche se la risorsa è molto rara.

Dunque nei sistemi strutturati il processo di ricerca di una risorsa fornisce garanzia di successo ed è inoltre molto efficiente, poiché genera un basso traffico di segnalazione e non influisce sulle prestazioni complessive della rete, garantendo in questo modo la scalabilità.

Il principale svantaggio dei sistemi strutturati risiede nella topologia della *overlay network*: infatti il protocollo di ricerca delle risorse impone che l'organizzazione rete virtuale segua delle precise regole. Questa caratteristica comporta che il processo di aggiunta o rimozione di un nodo della rete sia una operazione piuttosto costosa.

1.2.2 Sistemi non-strutturati

Nei sistemi *peer-to-peer* non strutturati i collegamenti stabiliti tra i nodi della rete virtuale sono del tutto arbitrari: ciascun *peer*, alla connessione, stabilirà un certo numero di *link* in modo arbitrario oppure copiando i collegamenti di un altro *peer*. Nel tempo, ciascun *peer* potrà creare ulteriori *link* con altri *peers*.

In uno scenario di questo tipo, l'organizzazione della rete segue principi molto semplici e l'aggiunta o rimozione di un *peer* è una operazione poco costosa. Al contrario, il processo di ricerca di una risorsa è reso difficoltoso ed inefficiente proprio a causa della mancanza di organizzazione della rete: il meccanismo più utilizzato è senza dubbio il *flooding* (propagazione) delle richieste, in modo che esse raggiungano il maggior numero possibile di *peers*.

L'uso della propagazione dei messaggi di richiesta genera un elevato traffico di segnalazione e, per di più, non fornisce alcuna garanzia di successo: il principale svantaggio dei sistemi non strutturati riguarda, per l'appunto, la non affidabilità del processo di ricerca. Oltre a questo aspetto si deve aggiungere

l'inefficienza dell'intera rete a causa dell'elevato traffico generato dal *flooding* delle *queries*: ciò rende il sistema poco scalabile.

Quello che accade normalmente nei sistemi non strutturati è che il processo di ricerca avrà un'alta probabilità di successo per le risorse "popolari", ovvero le risorse che sono condivise da un elevato numero di *peers*, mentre avrà una scarsa probabilità di successo per le risorse possedute da un ristretto numero di *peers*.

1.3 Interoperabilità tra sistemi P2P

Una delle sfide dei sistemi *peer-to-peer* è quella di assicurare un certo grado di interoperabilità tra le differenti architetture, in modo da aumentare considerevolmente la quantità di risorse condivise e permettere agli utenti di un sistema di poter usufruire dei servizi degli altri sistemi *peer-to-peer*: questi aspetti, in particolare l'aumento delle risorse disponibili, sarebbe molto utile in tutti i sistemi di tipo *file-sharing*.

Il principale ostacolo legato all'interoperabilità dei sistemi è sicuramente di natura tecnologica: a causa della mancanza di veri e propri standard, ciascun sistema *peer-to-peer* utilizza spesso protocolli e tecnologie proprietarie, impedendo dunque ad un *peer* di accedere ed interagire ad una differente rete.

Un primo passo verso l'interoperabilità dei sistemi *peer-to-peer* consiste nell'identificare le caratteristiche comuni ai vari sistemi per arrivare alla definizione di una architettura *peer-to-peer* "astratta", la quale si adatti sufficientemente bene a ciascun sistema che si desidera integrare.
Il secondo passo consiste nello stabilire un opportuno set di protocolli comuni per i vari aspetti dei sistemi *peer-to-peer*, come per esempio le comunicazioni tra i *peers*, la ricerca e l'accesso alle risorse e la gestione della rete virtuale. I protocolli definiti potranno poi essere implementati tramite le tecnologie utilizzate dagli specifici sistemi *peer-to-peer*.

In terzo ed ultimo importante passo per raggiungere l'interoperabilità, è definire un comune linguaggio per la rappresentazione dei messaggi utilizzati nella comunicazione tra i vari nodi della rete.
La scelta privilegiata ricade su standard assai consolidati e diffusi, e soprattutto neutrali, ovvero non vincolati ad uno specifico linguaggio di implementazione, come ad esempio XML e SQL.

1.4 Sistemi multi-agente: MAS

Da qualche anno a questa parte i sistemi multi-agente, stanno ottenendo popolarità in diversi settori, fornendo assistenza all'utente in svariati ambienti di sviluppo.
In parole povere, la nascita e la crescita di sistemi di questo tipo, si basa sulla cooperazione e sul "dialogo intelligente" degli agenti, entità autonome capaci di muoversi in rete, reperire informazioni e di interfacciarsi fra di loro.
Prima di procedere è necessario però introdurre brevemente il concetto di agente software, essendo questo l'elemento principale su cui si basa questa sezione.

Cosa sia in realtà un agente e come esso differisca da un normale programma è stato per lunghi anni argomento di dibattito. Sempre più spesso vengono proposte applicazioni dove gli agenti sono definiti come programmi che assistono gli utenti e agiscono per conto di questi.
Dal punto di vista di un utente finale è possibile definire un agente come un programma che lo aiuta nelle sue attività, permettendo alle persone di delegare parte del proprio lavoro. Anche se questa definizione potrebbe essere sostanzialmente corretta, tuttavia non tocca la questione di fondo. Gli agenti si presentano sotto svariate forme ed in molte configurazioni. Si possono trovare in sistemi operativi per computer, software di rete, basi di dati. Essi hanno l'abilità di interagire con l'ambiente di esecuzione e di funzionare in maniera asincrona ed autonoma su di esso. L'agente semplicemente agisce in maniera continua per raggiungere i propri obbiettivi. Nel diagramma di flusso sottostante, è semplicemente descritto un ciclo di vita di un generico agente, dalla sua

inizializzazione sino al raggiungimento della completezza del compito che doveva svolgere:

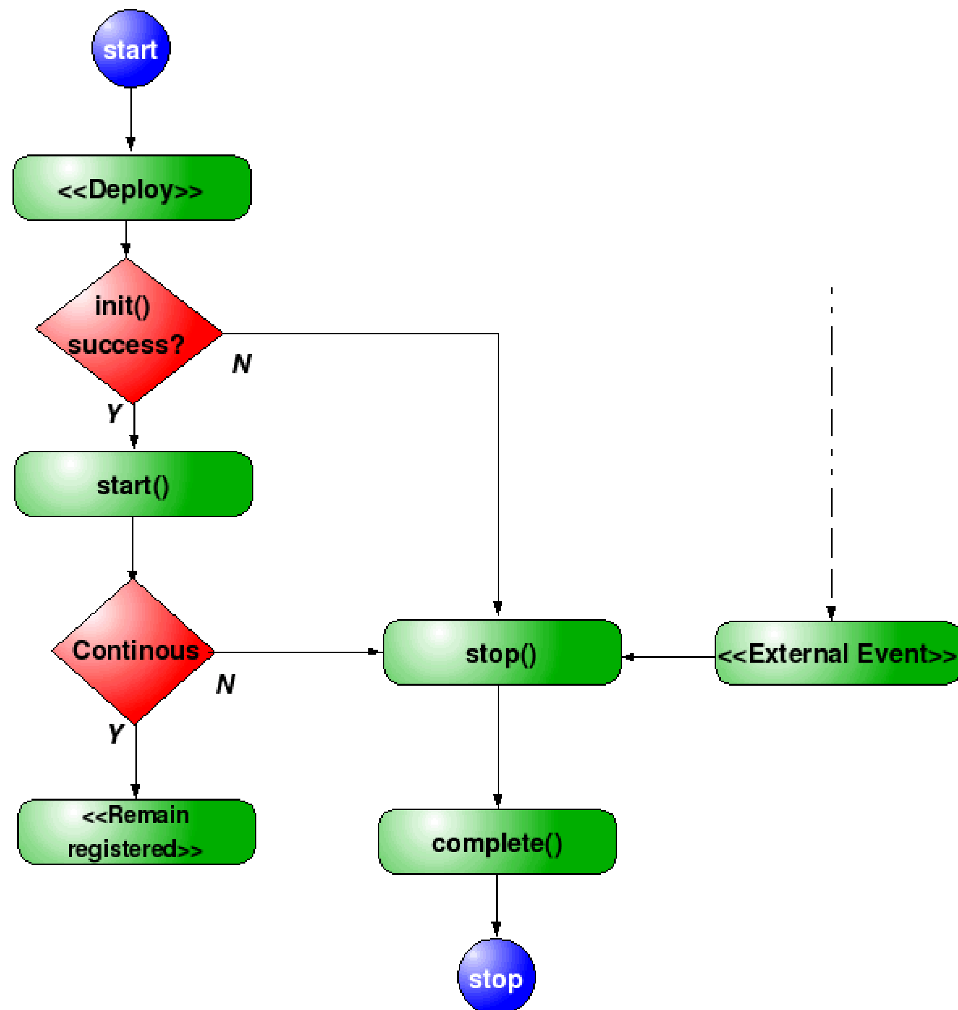

Figura 1.1 Ciclo di vita di un agente informatico.

I MAS, a livello progettuale, sono costituiti dagli agenti e dall'ambiente in cui essi vivono, di cui devono essere specificate le caratteristiche spaziali e temporali. Lo spazio è discretizzato in celle, ognuna delle quali è caratterizzata da una serie di variabili, i cui valori costituiscono gli stimoli che ogni agente percepisce e utilizza per decidere come comportarsi. La dimensione temporale è scandita da un "orologio interno" che consente ai singoli agenti di coordinare le proprie azioni in maniera sequenzialmente corretta. L'interazione infatti è una delle più importanti caratteristiche degli agenti, poiché consente loro di condividere informazioni ed

eseguire attività per perseguire gli obiettivi. Per realizzarla occorre amministrare ed aggiornare i servizi degli agenti e l'informazione.

Per risolvere questi problemi, è stata introdotta la nozione di agenti intermedi (*middle-agents*). Essi sono entità alle quali gli altri agenti non possono né richiedere né proporre, ma la loro funzione è quella di semplici intermediari. Il vantaggio dei *middle-agents* è che permettono ad un MAS di gestire efficacemente il problema della mobilità degli agenti.

Nella figura seguente è rappresentato uno schema modulare di un sistema multi-agente:

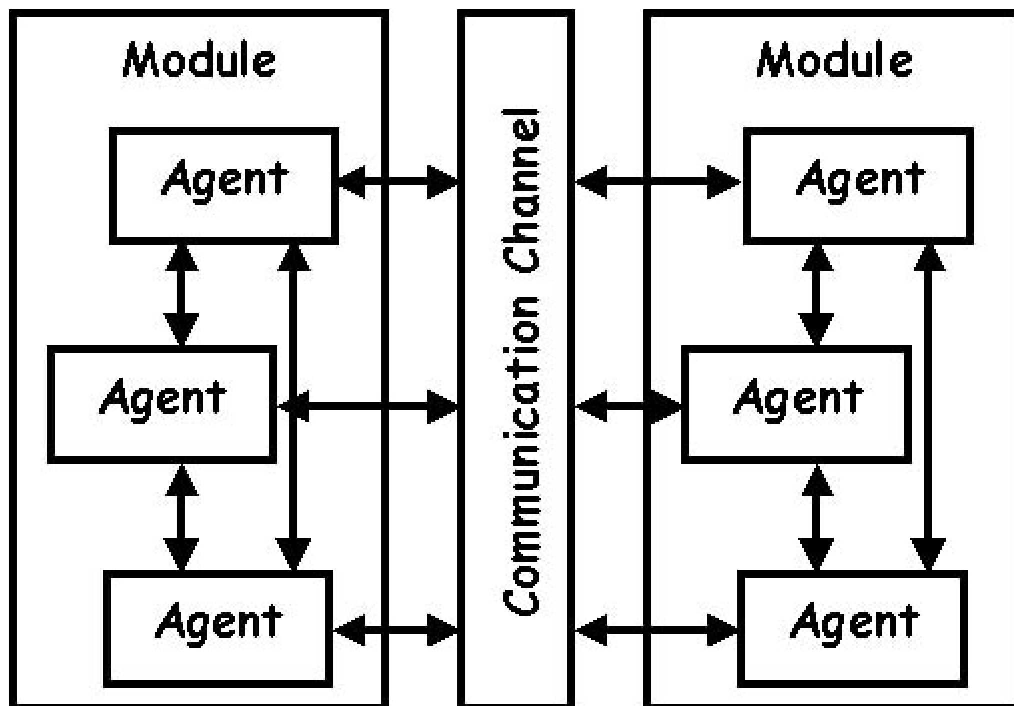

Figura 1.2 Schema modulare di un MAS.

Come si osserva dalla figura, un MAS è molto scalabile ed adattabile in base alle specifiche di progetto.

Questo perché ad alcuni agenti può essere dato il compito di fare solo da intermediari tra entità dello stesso modulo, o tra agenti di moduli diversi; ad altri può essere affidato l’invio sicuro dei dati attraverso il canale di comunicazione; altri ancora possono essere preposti al controllo delle operazioni e delle cooperazioni tra gli agenti stessi del sistema.

1.4.1 Integrazione dei MAS con il Web

Una delle più importanti applicazioni per un sistema multi-agente è la creazione di entità che possano aiutare gli utenti nelle loro attività, ad esempio, sostituendoli nella gestione o nella ricerca di informazioni.
Le attività eseguite dagli agenti personali e dagli altri agenti che devono comunicare con gli utenti del sistema necessitano di un meccanismo di interazione fra l'utente e l'agente che sia affidabile e al tempo stesso di facile implementazione.
Ovviamente, il modo più semplice ed intuitivo per poter interagire con una qualsiasi entità software (nel nostro caso l'agente personale) è attraverso un'interfaccia web.

Da qualche anno a questa parte, molti sviluppatori software stanno utilizzando il Web come un ambiente applicativo indipendente dalla piattaforma utilizzata, al fine di ottenere applicazioni eterogenee e facilmente "assimilabili" dai vari sistemi. Infatti la crescente necessità di gestire sistemi distribuiti ha fatto sorgere l'esigenza di applicare un nuovo approccio che permetta di amministrare ed utilizzare tutte le informazioni acquisibili, ovunque esse si trovino, in modo dinamico e soprattutto efficace.
Sono così nate e si sono sviluppate le tecniche di programmazione distribuita, favorite anche dall'affermazione dei linguaggi orientati agli oggetti che promuovono la programmazione *multi-thread.*

Gli agenti software rappresentano una tecnica innovativa che permette, tra le altre cose, di automatizzare e rendere più efficace la fornitura di un servizio. Questo avviene perché è il sistema ad agenti che si fa carico di tutte quelle operazioni necessarie allo svolgimento di un determinato compito ed è potenzialmente in grado di fornire servizi più evoluti in confronto a quelli realizzati con tecnologie software più tradizionali.
Rispetto ai sistemi abituali nei quali l'intelligenza è centralizzata, un'architettura distribuita, nella quale le capacità elaborativa e decisionale sono condivise e distribuite fra più entità (agenti), può offrire molti vantaggi.

Per un simile sistema si possono ipotizzare molti possibili campi applicativi, potenzialmente tutti quelli in cui si debbano affrontare problemi che coinvolgano strutture variamente complesse o criteri decisionali molto articolati.
Lo sviluppo di tecnologie come quelle sopra descritte, richiede di considerare l'integrazione di strumenti di comunicazione con strumenti avanzati che diano supporto alla gestione della complessità derivante dalla crescita dell'utilizzo di Internet e della tecnologia web.

Questo spiega come mai, per la progettazione, realizzazione e successiva "manutenzione" di sistemi web tecnologicamente avanzati, è risultato via via necessario, col passare degli anni, arrivare all'integrazione ottimale dei vantaggi forniti dai sistemi multi-agente con le innumerevoli funzionalità comunicative fornite dal Web.

Capitolo 2

Strumenti e Tecnologie Utilizzate

Questo capitolo tratta degli strumenti e delle tecnologie che sono state sfruttate lungo il corso della tesi. In particolare si parlerà del framework applicativo Spring che ha permesso di ottenere l'astrazione progettuale; della piattaforma JADE, utilizzata per la progettazione di sistemi multi-agente; dell'architettura JXTA, che ha fornito le basi per lo sviluppo del P2P ed infine di RAIS, un sistema per la condivisione di documenti, progettato nei laboratori AOT dell'Università degli Studi di Parma.

2.1 Astrazione dell'architettura: il *framework* Spring

Molti progetti Java hanno strutture che normalmente coinvolgono vari *framework* per potersi avvalere delle loro caratteristiche.
Questa mole di informazioni, ha necessità di essere trattata

adeguatamente prima di essere aggregata in maniera consona all'utente che deve poter accedere alle informazioni che gli sono necessarie.
Quindi un portale, ad esempio, ancora prima di iniziare il lavoro di elaborazione, avrà bisogno localmente, dei servizi offerti da vari *framework*. Si pensi al recupero delle credenziali dell'utente o delle sue preferenze nella visualizzazione delle informazioni, queste poche operazioni hanno bisogno dei dati recuperati da un *database* e del supporto dei servizi di presentazione, e di logica applicativa che permettono di fornire all' utente finale ciò che ha richiesto. Dal punto di vista architetturale, è necessario usufruire di un *framework* che fornisca dei servizi (accesso ai *database*, *template service* e altri), al quale eventualmente aggiungerne di propri. Naturalmente è necessario anche qualcuno che aggreghi questi servizi e fornisca un unico punto di accesso dal quale consultarli, questa aggregazione può produrre codice molto complesso.

Questo problema è comune anche ad altre tipologie applicative. Essendo l' integrazione dei servizi, un problema ricorrente, si è trovata una nuova soluzione progettuale al problema, mediante un *pattern*, chiamato *Inversion of Control* (IoC) o anche *Dependency Injection*, in antitesi con quello più classico del *Service Locator*. L' IoC permette di descrivere come gli oggetti devono essere valorizzati e con quali altri oggetti hanno delle dipendenze; è il *container* che è il responsabile del collegamento fra questi oggetti, è il *container* che "inietta" le dipendenze tra gli oggetti, da cui il nome di *Inversion of Control*, in questo modo non è necessaria una classe che faccia dei "*look-up*" per trovare i servizi, ma sono i servizi che vengono resi disponibili in un *container*.
Esistono tre tipi di implementazione dell' *Inversion of Control*:

- i servizi implementano un'interfaccia dedicata (***Avalon***);
- ***setter based***: le dipendenze vengono assegnate settando dei valori a dei JavaBeans (Spring e HiveMind);
- ***constructor based***: le dipendenze vengono assegnate, passando degli argomenti ai costruttori (PicoContainer).

Ora vediamo nello specifico il *framework* che ci interessa analizzare: Spring. Spring è un *framework* J2EE "leggero", in opposizione ad approcci cosiddetti

"pesanti" di altre tecnologie J2EE. Non costringe a sposare integralmente le funzionalità che offre, essendo costruito in maniera modulare (i moduli sono contenuti in librerie separate), questo consente di continuare ad utilizzare altri *tool* o altri *framework*. Fornisce delle soluzioni molto eleganti e molto semplici dal punto di vista della scrittura del codice. Andiamo a vedere i moduli di cui è composto:

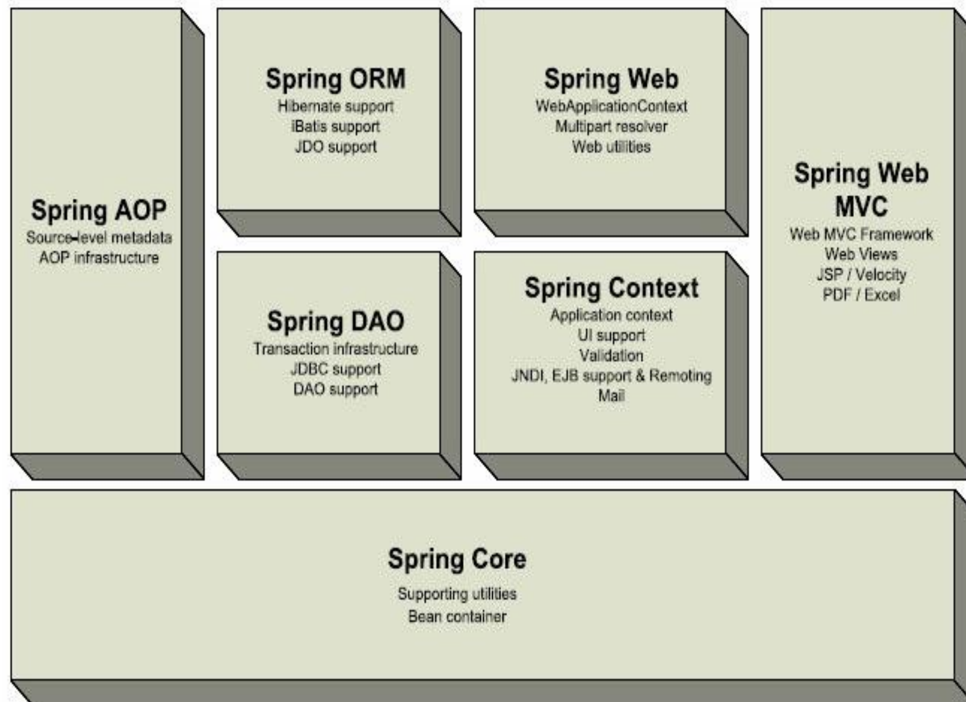

Figura 2.1 Moduli di Spring.

Il *package* Core contiene le parti fondamentali per realizzare l' IoC. L' IoC viene realizzato per mezzo della classe BeanFactory che usando il *pattern factory* rimuove la necessità di *singleton* e permette di disaccoppiare la configurazione e le dipendenze. Il *package* Context fornisce l'accesso ai *bean*, fornisce il supporto per i *resources-bundle*, la propagazione degli eventi, il caricamento delle risorse e la creazione trasparente dei contesti. Il *package* DAO (Data Access Object) fornisce uno strato di astrazione per JDBC (Java DataBase Connector). Fornisce una modalità dichiarativa per la gestione delle transazioni (*transaction management*). Il package ORM fornisce l' integrazione con i più diffusi *object-relational mapping* (JDO, Hibernate e iBatis), sfruttando le caratteristiche di

transazionalità definite precedentemente. AOP fornisce una implementazione aderente alle specifiche AOP-Alliance dell'Aspect Programming. Infine il *package* WEB fornisce l 'integrazione con le caratteristiche orientate al Web, come inizializzazione di contesti usando Servlet Listener, e la creazione di contesti applicativi per applicazioni *web*. Il *package* WEB MVC fornisce una implementazione ModelViewController per applicazioni *web*, fornendo inoltre tutte le altre caratteristiche di Spring. L'uso di tutti o di una sola parte dei moduli di Spring consente il suo utilizzo in diversi scenari:

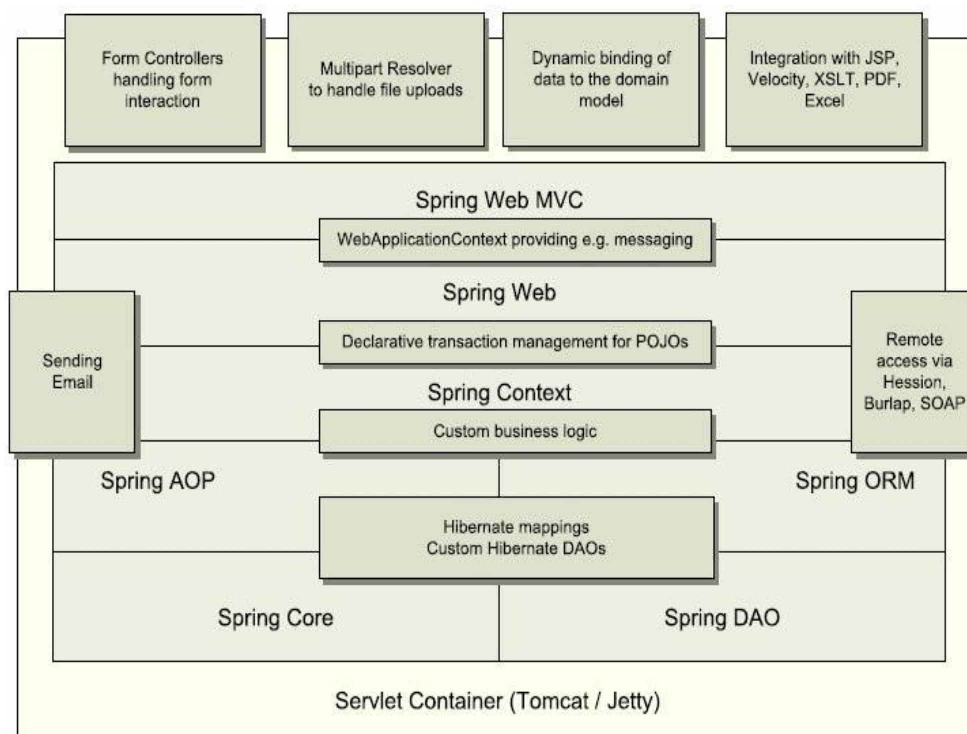

Figura 2.2 Scenari di utilizzo di Spring.

Il *framework* Spring verrà affrontato nel dettaglio nel Capitolo 3, dove verranno presentati tutti i suoi componenti più importanti e le sue funzionalità più avanzate.

2.2 Supporto MAS: JADE

JADE (*Java Agent Development Framework*) è un *framework* software per la realizzazione di sistemi multi-agente, sviluppato da Telecom Italia e distribuito

con licenza *open source*. JADE è sviluppato interamente in linguaggio Java ed è conforme alle specifiche FIPA per lo sviluppo di sistemi ad agenti.

La componente principale del *framework* JADE è costituita da un *runtime environment*, all'interno nel quale sono creati ed eseguiti uno o più agenti.
Ogni istanza di JADE, ed in particolare del componente *runtime environment*, è detta *container*. Una piattaforma JADE (*platform*) è composta da uno o più *containers*: all'interno della piattaforma deve sempre esistere ed essere attivo un *container* speciale, detto *main container*, mentre tutti gli altri *containers*, ciascuno dei quali è detto *agent container*, devono connettersi al *main container* e registrarsi con esso.

Dunque una piattaforma JADE è sempre composta da uno ed un solo *main container*, il quale deve essere creato prima di tutti gli altri *containers*, e da zero o più *agent containers*.

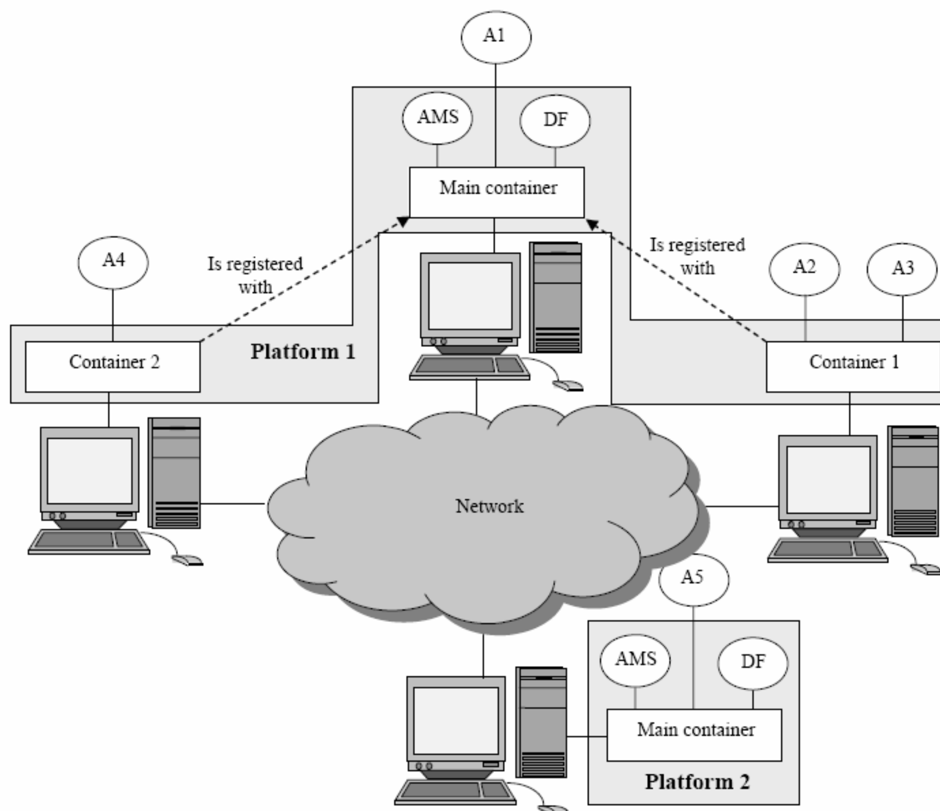

Figura 2.3 Architettura JADE.

Ogni agente JADE è univocamente identificato dal proprio nome, detto AID (*Agent Identifier*), all'interno della piattaforma e può comunicare con gli altri agenti attraverso lo scambio di messaggi.

La comunicazione tra agenti avviene in modo trasparente ed indipendente dalla localizzazione degli agenti stessi: mittente e destinatario possono appartenere allo stesso *container*, a differenti *containers* all'interno della stessa piattaforma oppure anche a piattaforme diverse.

2.3 Supporto P2P: JXTA

La tecnologia JXTA è costituita da una *suite* di protocolli aperti e basati sullo standard XML, disegnati per lo sviluppo di applicazioni *peer-to-peer*. JXTA permette ad ogni dispositivo connesso sulla rete, dai cellulari ai PDA, dai PC ai *server*, di comunicare e collaborare come *peer*.
I protocolli di JXTA sono indipendenti sia dai protocolli di trasporto dati che dal linguaggio di programmazione utilizzato poiché si basano su meta dati (dati che descrivono altri dati). Esistono infatti implementazioni in C/C++, Perl, Phyton e naturalmente Java. Da quando la rete ha incominciato a crescere a causa dell'aumento dei dispositivi in grado di connettersi con un protocollo qualsiasi, dal Bluetooth al TCP/IP, la filosofia P2P è divenuta popolare. Sebbene spesso il termine P2P sia legato a concetti spiacevoli come la pirateria e la violazione del diritto d'autore, JXTA rappresenta un'evoluzione tecnologica notevole che non è di esclusivo appannaggio della condivisione dei file, ma può abbracciare numerosi servizi permettendo, per esempio, la creazione di gruppi di lavoro e servizi esclusivi. Lo scopo principale di JXTA quindi è provvedere alle funzionalità base dei sistemi P2P.
In aggiunta, però JXTA sviluppa concetti come:

- *interoperabilità:* la possibilità per ogni peer di fornire servizi diversificati e poterli cercare;

- *indipendenza dalla piattaforma:* JXTA è progettato per essere indipendente dal linguaggio di programmazione, dai protocolli di trasporto e piattaforma di sviluppo;
- *ubiquità:* JXTA è stato realizzato per essere accessibile da qualsiasi dispositivo digitale (cellulari, PDA, *smartphone*) e non solo dai PC.

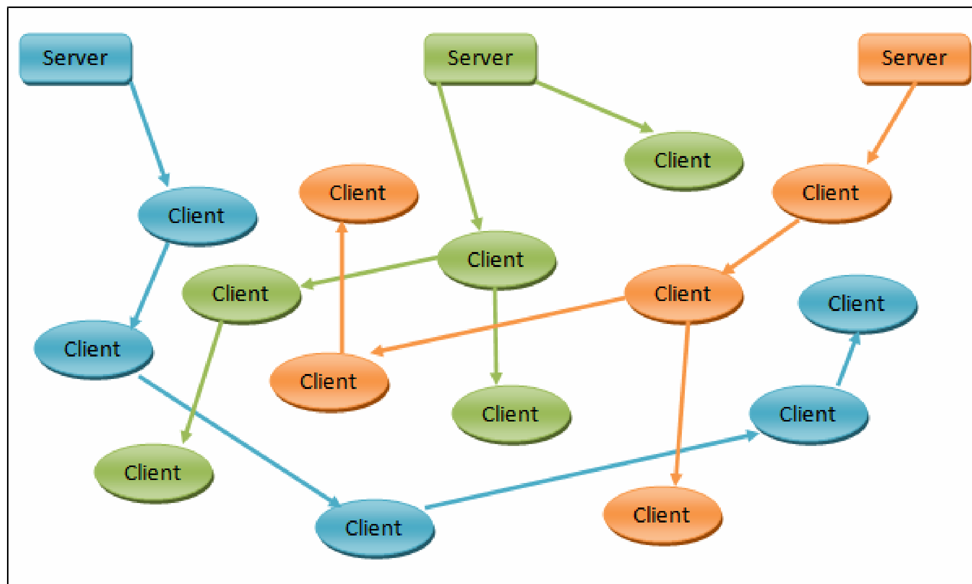

Figura 2.4 Schema base di un sistema P2P.

Con l'utilizzo di JXTA è possibile realizzare applicazioni che permettono di:

- trovare con ricerche dinamiche gli altri *peer* sulla rete, anche se questi sono protetti da *firewall* o NAT;
- condividere senza difficoltà informazioni e documenti;
- creare gruppi dinamici e fornire servizi;
- monitorare l'attività dei peer in remoto;
- comunicare in modo sicuro con gli altri peer della rete.

Le specifiche JXTA definiscono una *overlay network* virtuale, nella quale ogni nodo (*peer*) può interagire con gli altri: l'interoperabilità permette la comunicazione tra i *peer* indipendentemente dalla rete fisica sottostante e dai protocolli di rete utilizzati. Il progetto JXTA (*juxtapose*) è sviluppato da SUN a partire dal 2001, e distribuito con licenza *open source*.

2.3.1 Protocolli

JXTA definisce una serie di formati di messaggi XML o protocolli per la comunicazione tra i *peer*. I peer usano questi protocolli per effettuare ricerche, informare e trovare risorse di rete, e comunicare e instradare i messaggi.

I protocolli JXTA inoltre definiscono uno standard nell'interazione tra i *peer*, fissando il modo in cui ciascun *peer* realizza le funzionalità fondamentali: *discovery* degli altri *peer*, organizzazione in gruppi di *peer*, pubblicizzazione e *discovery* delle risorse di rete, comunicazione con altri nodi, monitor di altri *peer*.

Le specifiche JXTA definiscono un set di sei protocolli:

- ***Peer Discovery Protocol:*** utilizzato dai *peer* per pubblicare e ricercare le risorse;
- ***Peer Information Protocol:*** utilizzato per ottenere informazioni sullo stato degli altri peer;
- ***Peer Resolver Protocol:*** consente ai peer di mandare una richiesta generica ad uno o più peer e ricevere una risposta;
- ***Pipe Binding Protocol:*** utilizzato per stabilire canali virtuali, o pipe, tra due o più peer;
- ***Endpoint Routing Protocol:*** utilizzato per trovare i percorsi verso gli altri peer;
- ***Rendezvous Protocol:*** stabilisce il meccanismo tramite il quale i peer possono utilizzare o fornire loro stessi il servizio di propagazione.

Ecco nel dettaglio come interagiscono tra di loro i diversi protocolli:

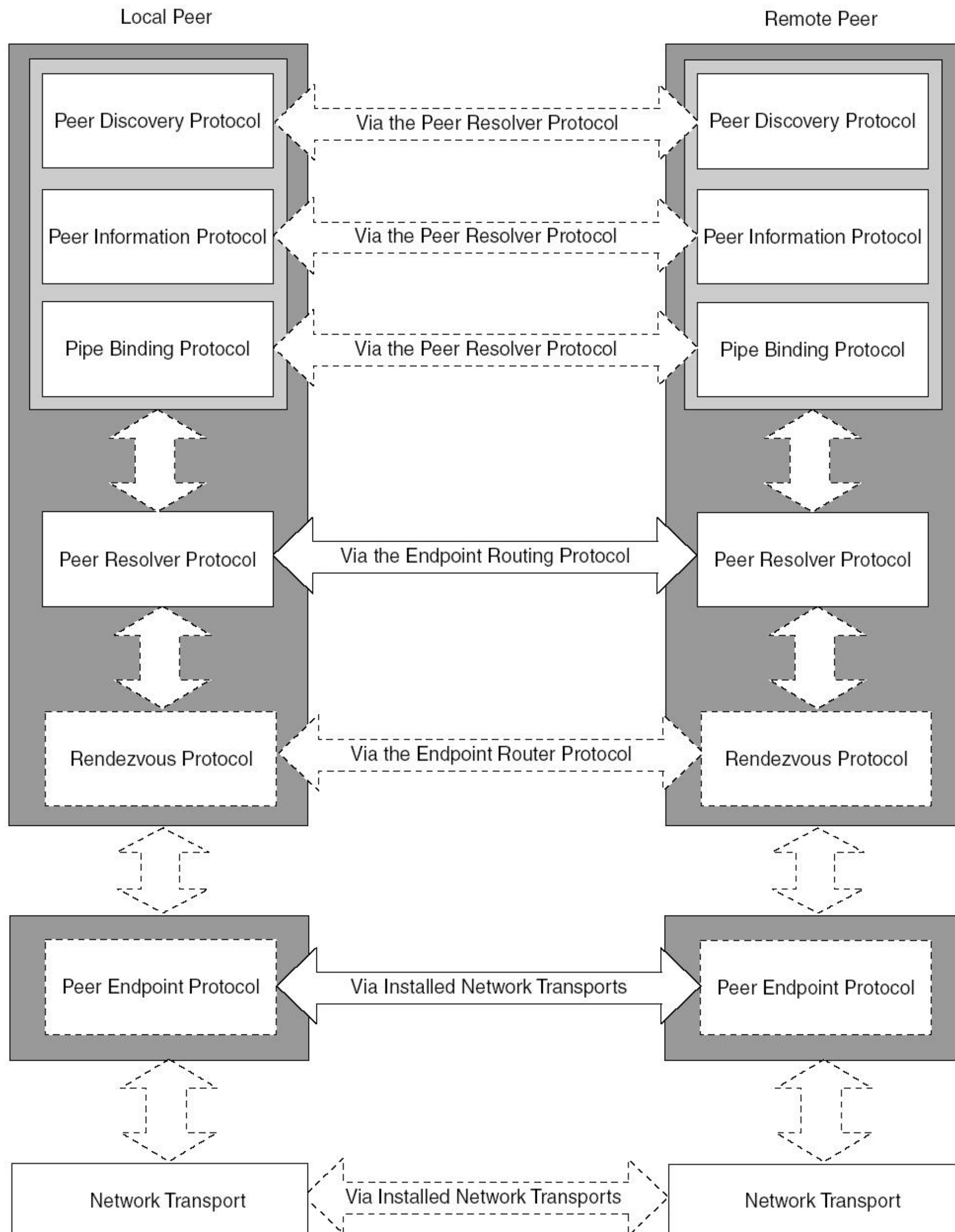

Figura 2.5 Stack dei protocolli JXTA.

I protocolli JXTA sono stati progettati e sviluppati per essere indipendenti tra loro e per introdurre un ridotto *overhead*, in modo da poter essere realizzati anche su *device* con bassa capacità computazionale. Si deve infine notare che un *peer* JXTA non è vincolato ad implementare tutti i protocolli e può realizzare solo i protocolli di cui necessità: questo aspetto fornisce ulteriore scalabilità alle specifiche, permettendo ai *device* con bassa capacità computazionale di implementare solo un *set* minimo di protocolli.

2.3.2 *Advertisement*

I protocolli JXTA si basano sull'utilizzo dell'XML per ottenere informazioni auto-descrittive.
Questo se da un lato permette una notevole interoperabilità ed un utilizzo generale di questa tecnologia, dall'altro appare costoso in termini di numero di messaggi scambiati e occupazione di banda. *L'overhead* prodotto infatti è notevole, come abbiamo potuto sperimentare nel progetto sviluppato. Ciascun *advertisement* (annuncio) è pubblicato con un ciclo di vita che specifica la disponibilità delle risorse associate.

I protocolli JXTA definiscono varie tipologie di annunci:

- ***Peer Advertisement:*** descrivono il *peer* stesso;
- ***Peer Group Advertisement***;
- ***Pipe Advertisement:*** descrivono i canali di comunicazione (*pipe*) che permettono l'effettivo scambio di informazioni;
- ***Module Class Advertisement:*** lo scopo principale è documentare l'esistenza di un modulo che fornirà un servizio;
- ***Module Spec Advertisement:*** lo scopo primario è quello di fornire riferimenti alla documentazione necessaria per consentire la creazione di implementazioni conformi alla specifica di modulo;
- ***Module Impl Advertisement:*** definisce un'implementazione di una specifica di modulo;
- ***Rendezvous Advertisement:*** descrivono un *peer* che agisce come *rendezvous* per un dato gruppo;
- ***Peer Info Advertisement:*** portano con sé informazioni sullo stato di *peer* e sulle sue statistiche.

2.3.3 *Peer*

La rete JXTA è una rete adattativa composta da *peer* connessi. Le connessioni sono temporanee e l'instradamento tra *peer* è non-deterministico.

Un *peer* JXTA è costituito da un'entità che implementa almeno uno dei protocolli definiti dalle specifiche.
I *peer* JXTA possono essere suddivisi in quattro principali categorie:

- ***Peer minimo:*** può mandare e ricevere messaggi ma non può conservare messaggi o instradare quelli di altri *peer*. Sono essenzialmente i *peer* con risorse limitate;
- ***Peer pienamente funzionale:*** può mandare e ricevere messaggi e solitamente mantiene in memoria gli *advertisement* tuttavia non inoltra richieste da parte di altri *peer*;
- ***Peer rendezvous:*** può mandare e ricevere messaggi, mantiene in memoria gli *advertisement* e inoltra richieste da parte di altri *peer*;
- ***Peer relay:*** fornisce un meccanismo *client/server* che permette la comunicazione con *peer* inaccessibili poiché dietro NAT/*firewall*.

2.3.4 *Pipe*

Le specifiche JXTA supportano la comunicazione tra i *peer* attraverso canali virtuali di comunicazione, detti *pipe*.
Una *pipe* consente ad un *peer* di inviare un messaggio ad uno o più *peer* destinatari: essa rappresenta un canale logico di comunicazione asincrono, non affidabile (con l'eccezione della pipe di tipo *unicast secure*) e può essere utilizzata per inviare messaggi contenenti qualsiasi tipologia di dati.
Una *pipe* JXTA può possedere *endpoint* di due tipologie differenti: una *input pipe* che riceve messaggi ed una *output pipe* che invia messaggi. Ciascun endpoint della *pipe* è collegato dinamicamente ad un *peer endpoint*: un *peer endpoint* corrisponde ad una specifica interfaccia di rete di un *peer*, la quale può essere utilizzata per inviare o ricevere messaggi. Gli *endpoint* di una *pipe* JXTA possono essere connessi a differenti *peer endpoint* in diversi istanti di tempo, oppure non essere connessi del tutto: l'unico vincolo imposto dalle specifiche è che la comunicazione abbia luogo all'interno di uno stesso *peer group*, ovvero *input* e *output pipe* devono appartenere allo stesso *peer group*.

JXTA fornisce tre differenti modelli di comunicazione, corrispondenti a tre diverse tipologie di pipe:

- ***Unicast pipe***: una *pipe* di tipo *point-to-point*, la quale connette esattamente due *pipe endpoint*, una *input pipe* di un *peer* riceve i messaggi inviati dall'*output pipe* di un altro *peer*;
- ***Propagate pipe***: una *pipe* di tipo *one-to-many*, la quale connette una *output pipe* a più *input pipe*. Ogni messaggio inviato da un *peer* tramite la *output pipe* è propagato e ricevuto da ciascuna *input pipe*;
- ***Unicast Secure pipe***: una *pipe* di tipo *point-to-point* sicura, la quale connette esattamente due *pipe endpoint*, una *input pipe* di un *peer* riceve i messaggi inviati dall'*output pipe* di un altro *peer*. Fornisce meccanismi di sicurezza e realizza una comunicazione affidabile.

2.4 RAIS: Remote Assistant for Information Sharing

Il sistema RAIS può essere descritto come un sistema multi-agente per la condivisione di informazioni in reti P2P.
Esso offre una potenza di ricerca simile ai motori di ricerca del Web, evitando però la necessità di pubblicare le informazioni, e quindi preservando la confidenzialità delle informazioni nel gruppo di condivisione scelto.

La prima domanda da porsi è ovviamente il perché sia stato deciso di progettare e realizzare un sistema di questo tipo: molto efficiente, ma anche molto complicato da costruire. La necessità di avere un sistema di questo tipo e data dal fatto che la capienza degli *hard-disk* continua ad aumentare, venendo riempita di *file* dai formati diversi. Spesso i file più importanti si perdono in questa vastità di spazio, poiché i tradizionali strumenti di ricerca su disco spesso non sono perfettamente efficienti.

Per sostenere queste mancanze sono stati inventati ed introdotti lungo il corso degli anni dei nuovi strumenti di ricerca caratterizzati da livelli di performance mai visti prima: i *Desktop Search* e gli indici.
Come oramai è stato già più volte sottolineato, oggigiorno per condividere contenuti senza per forza metterli in rete si usa il P2P. Quello che invece non si è ancora specificato è che i sistemi P2P sono molto efficienti per contenuti multimediali perché questi *files* sono facilmente categorizzabili in base al nome del *file* stesso.
Per condividere invece *file* di contenuti più importanti servirebbe che la ricerca fosse effettuata proprio sul contenuto, così come fanno i *Desktop Search*.

Ecco quindi l'idea di mettere in piedi un sistema come RAIS, ovvero un sistema che accoppi alle caratteristiche di condivisione dei sistemi P2P, quelle di ricerca del Web e dei *Desktop Search*. Nello specifico RAIS è un sistema multi-agente P2P composto da diverse piattaforme ad agenti connesse tramite Internet.

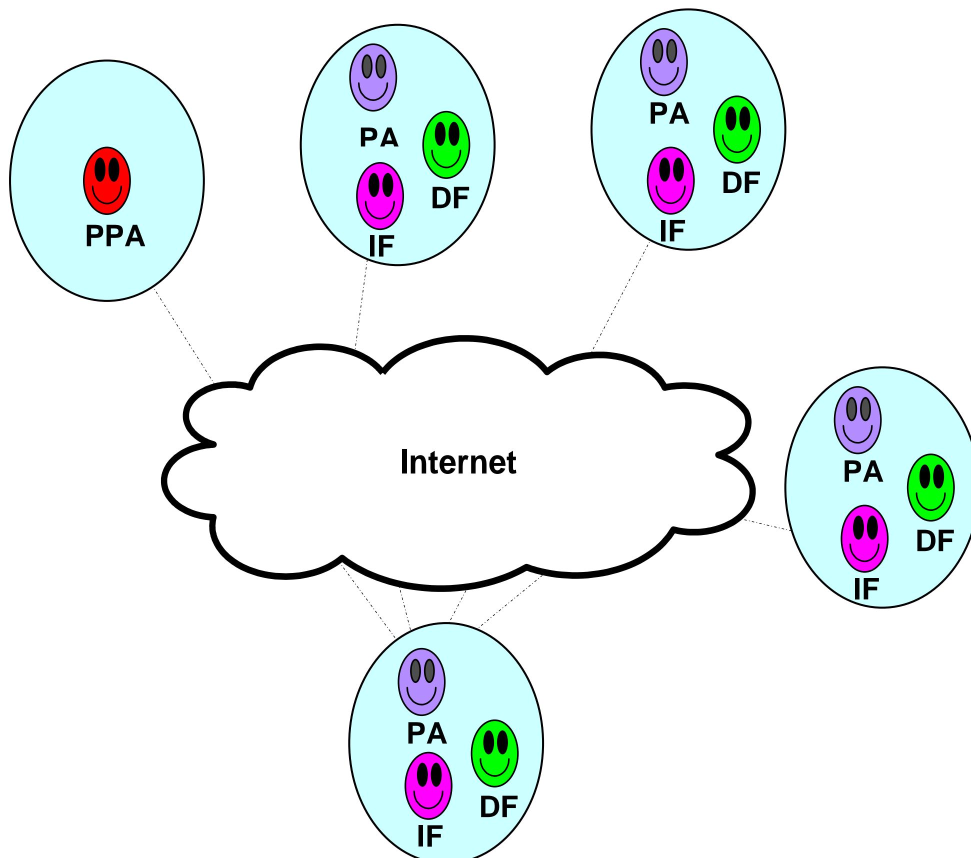

Figura 2.6 L'architettura multi-agente RAIS.

Come si può osservare dalla figura riportata qui sopra, ogni piattaforma agisce come un *peer* ed è composta da tre agenti: un *Personal Assistant*, un *Information Finder* ed un *Directory Facilitator*. In più un altro agente, il *Personal Proxy Assistant*, permette all'utente di accedere in remoto alla sua piattaforma.

Vediamo ora nel dettaglio i compiti riservati ai vari agenti in funzione. Il PA, ha un compito tanto semplice quanto importante, infatti è l'agente che permette l'interazione tra RAIS e l'utente in ogni sua forma.
L'IF invece, è un agente che cerca informazioni nei *repository* del computer nel quale risiede, e fornisce i risultati sia al suo utente che a tutti gli altri.
Il PPA rappresenta un punto d'accesso al sistema per tutti gli utenti che non stanno lavorando sui loro computer ma su periferiche esterne (come le chiavette USB ad esempio). Infatti, se non si deve per forza avere un accesso completo, ma si desidera solo accedere remotamente a *files* e documenti è stato creato per RAIS la possibilità di un accesso remoto appunto tramite un PPA installato (insieme al *core* di RAIS, alla chiave di autenticazione e al PA stesso) su una periferica rimovibile. Il PPA agisce come un semplice *proxy* del PA remoto.
Il DF infine è il vero e proprio responsabile della registrazione della piattaforma ad agenti sulla rete RAIS. Inoltre esso deve informare gli agenti circa gli indirizzi di tutti i *peers* (agenti) che risiedono in altre piattaforme, ma sempre all'interno della rete RAIS (come per esempio tutti gli IF).

Alcune informazioni contenute nei vari *repository* non possono essere accessibili a tutti gli utenti per motivi di *privacy* o ancora più importante di sicurezza. Per questo RAIS tiene conto di: identità, ruolo e attributi di quel particolare utente. L'utente può abilitare "manualmente" gli utenti all'accesso alle proprie informazioni oppure può abilitarli tramite una delega certificata in base alla reputazione dell'utente nella comunità di RAIS (utile per esempio per condividere documenti tra membri di progetti *Open Source*).

RAIS usa varie tecnologie per supportare i propri servizi:

- JADE: per la piattaforma ad agenti;

- JXTA: per l'infrastruttura P2P;
- *Google Desktop Search* o *Lucene Desktop Search*: per la ricerca dei *files*.

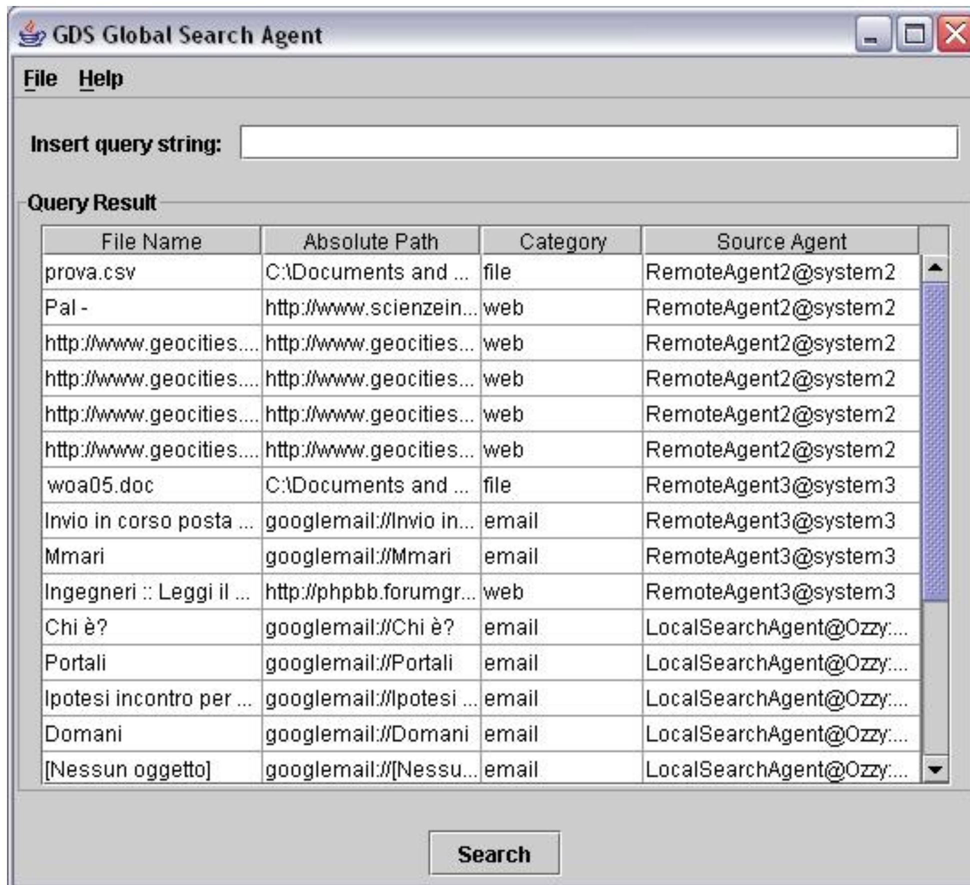

Figura 2.7 La *Graphic User Interface* di RAIS.

Quindi RAIS può essere considerato di più che solo un progetto di ricerca che abbina le caratteristiche dei motori di ricerca *Web* a quelle dei sistemi P2P per la condivisione dell'informazione.

Prima di tutto RAIS assicura la sicurezza, e un accesso garantito, controllato e dinamico. Inoltre l'implementazione basata sugli agenti, semplifica la realizzazione di tre delle caratteristiche principali del sistema:

- il filtraggio delle informazioni che arrivano da più utenti, sulla base della precedente esperienza dell'utente locale;
- l'immissione di nuove informazioni che possono essere d'interesse per un utente;

- la delega dell'autorizzazione in base alla reputazione di un utente nella comunità.

Un primo prototipo di RAIS (ovvero quello usato come punto di partenza per questo lavoro di tesi) è già stato pubblicato e testato installandolo in vari laboratori ed uffici del dipartimento con risultati di successo per tutte le caratteristiche del sistema.

L'applicazione, compilata in linguaggio Java e compatibile sia con sistemi operativi Windows che Unix, è composta da due agenti in ambiente JADE: il *Desktop Search Agent* ed il *Local Search Agent*. Il primo fornisce all'utente un'interfaccia di ricerca nella rete e dialoga con il secondo, il quale si occupa di invocare il sottosistema *Desktop Search* installato per recuperare i risultati.

In conformità alla concezione delle reti *peer-to-peer*, su ciascun nodo della rete è previsto che risiedano entrambi questi agenti, in modo che ciascun utente possa al tempo stesso svolgere ricerche e mettere a disposizione degli altri i propri contenuti. Detto ciò, resta comunque possibile attivare in maniera indipendente uno o l'altro dei due componenti.

Le funzioni *Desktop Search* sono fornite agli agenti tramite un'interfaccia generica, svincolata dalla specifica implementazione, il che rende possibile l'utilizzo di un qualsiasi sistema *Desktop Search* previa costruzione di opportune API di collegamento in linguaggio Java.
Attualmente è incluso il supporto per *Google Desktop* (l'applicativo *leader* nel settore) oltre che per *LuceneDS* (un progetto *Open Source* presentato da *Apache*).
Ciascun nodo può scegliere un sistema *Desktop Search* diverso, pur mantenendo l'interoperabilità con gli altri *peer*.

Il sistema di ricerca prevede inoltre un algoritmo di ordinamento dei risultati ideato in maniera specifica per reti *peer-to-peer*, che si colloca ad un livello superiore rispetto alla funzione di ordinamento offerta dal sistema *Desktop Search*

locale di ogni nodo, e consente una classificazione dei risultati globale ed assoluta.
Questo algoritmo di ordinamento, frutto di un progetto nell'ambito del Laboratorio AOT del Dipartimento (lo stesso di questo lavoro), necessita però di dati statistici sulle risorse indicizzate che non è possibile ottenere mediante le API *Google Desktop*, e di conseguenza è disponibile esclusivamente nei nodi che utilizzano *LuceneDS* come sistema *Desktop Search*.

Il *software* è stato realizzato con l'apporto di svariate librerie *Open Source*, tra cui si ricorda in particolare GDAPI20, che offre API Java per Google Desktop, oltre naturalmente alla piattaforma JADE per la gestione degli agenti.

LuceneDS è incluso sotto forma di archivio JAR indipendente. L'infrastruttura generale prevede un sistema di *logging* completo basato su *Log4J* per facilitare i test in fase di sviluppo e anche per una gestione flessibile dei messaggi mostrati all'utente. È inoltre presente un file di configurazione in formato XML che contiene le impostazioni dell'applicazione, per la gestione del quale sono state riutilizzate le classi generiche sviluppate per *LuceneDS*.
Tutte le opzioni di configurazione vengono lette all'avvio dell'applicazione. Il file di configurazione è condiviso da entrambi gli agenti, e contiene opzioni relative sia ad uno che all'altro. La classe *Config* si occupa di caricare le informazioni di configurazione e di renderle disponibili per tutte le altre classi.
Di seguito si elencano le opzioni disponibili e i nomi dei *tag* relativi:

- *sorting (2step, native)*: seleziona l'algoritmo di ordinamento, a 2 passi (ottimizzato per reti *peer-to-peer*) oppure nativo del sistema *Desktop Search* utilizzato;
- *calcHashSum (true, false)*: indica se calcolare il *checksum* MD5 per ogni risultato. Questa operazione viene eventualmente svolta dal *Local Search Agent* prima di inviare i risultati all'utente che li ha richiesti. Quando questa opzione è attivata il sistema provvede automaticamente al calcolo dei *checksum*, ma solo nel caso in cui il servizio *Desktop Search* non li

abbia già forniti. Si tratta infatti di un'operazione gravosa che introduce rallentamenti ed è consigliabile evitarla quando possibile;

- *maxResults*: specifica il numero massimo di risultati richiesti. Nel caso in cui siano disponibili in rete più risultati, quelli in eccesso vengono scartati. Questa opzione è particolarmente significativa nel caso dell'algoritmo di ordinamento a 2 passi;
- *searchService*: specifica il servizio di *Desktop Search* da utilizzare, sotto forma di nome della classe relativa (completo di *package*);
- *sharedDirs*: elenco delle cartelle condivise, ovvero quelle che si vuole rendere accessibili agli altri *peer*. I risultati delle ricerche vengono infatti filtrati prima dell'invio in modo da restituire al *peer* richiedente soltanto quelli appartenenti alle cartelle condivise. È possibile specificare anche se includere i percorsi condivisi in rete, che vengono determinati automaticamente (solo su sistemi Windows). Da notare che questo elenco è distinto da quello delle cartelle indicizzate dal servizio *Desktop Search*, per la massima flessibilità.

Inoltre viene fornito anche il file di configurazione per il servizio *Desktop Search LuceneDS*, che contiene le opzioni specifiche per questo tipo di servizio di ricerca. Questo file naturalmente è necessario soltanto nel caso in cui si utilizzi *LuceneDS*: altri servizi (ad esempio *Google Desktop*) potrebbero richiedere eventualmente altri sistemi di configurazione opportuni.

Di seguito si presenta una panoramica dei diversi *package* in cui sono suddivise le classi dell'applicazione.
Il prefisso comune a tutti i package è *it.unipr.aotlab.jade*, e viene omesso nel seguente elenco:

- *dsa*: contiene le due classi che rappresentano gli agenti, oltre ai vari *behaviour* definiti per gli agenti stessi;
- *dsa.search*: contiene le interfacce astratte che definiscono il generico servizio di *Desktop Search*, e i formati dei messaggi che contengono i risultati;

  - *dsa.search.gds*: implementazione del servizio di *Desktop Search* tramite *Google Desktop*;
  - *dsa.search.lds*: implementazione del servizio di *Desktop Search* tramite *LuceneDS*;
- *dsa.gui*: definisce l'interfaccia grafica di ricerca offerta dall'agente *Desktop Search Agent*;
- *dsa.util*: contiene classi di varia utilità, per il *parsing* XML e la creazione di *checksum* MD5.

Per quanto riguarda i futuri sviluppi di questo sistema si può dire che principalmente sono due:

- gestione di nuovi tipi di informazioni (*Database*);
- aggiunta di nuove tecniche di ricerca (*Sematic Web*).

Capitolo 3

Il *framework* Spring

Di seguito verrà introdotto e descritto Spring, framework potente ed efficiente, per la creazione di interfacce astratte. La trattazione sarà generalizzata ai casi applicativi più utili, mentre le specifiche dell'utilizzo di Spring nel progetto saranno affrontate nei capitoli successivi.

3.1 Caratteristiche principali

Spring è un framework *Open Source* per lo sviluppo di applicazioni web su piattaforma Java. La prima versione venne scritta da Rod Johnson e rilasciata con la pubblicazione del proprio libro "*Expert One-on-One Java EE Design and Development*" (Wrox Press, Ottobre 2002).
All'inizio il framework venne rilasciato sotto Licenza *Apache* nel giugno 2003. Il primo rilascio importante è stato l'1.0 del marzo 2004, seguito da due successivi rilasci importanti nel settembre 2004 e nel marzo 2005.

Spring è stato largamente riconosciuto all'interno della comunità Java quale valida alternativa al modello basato su *Enterprise JavaBeans* (EJB). Rispetto a quest'ultimo, il *framework* Spring lascia una maggiore libertà al

programmatore fornendo allo stesso tempo un'ampia e ben documentata gamma di soluzioni semplici adatte alle problematiche più comuni.

Sebbene le peculiarità basilari di Spring possano essere adottate in qualsiasi applicazione Java, esistono numerose estensioni per la costruzione di applicazioni *web-based* costruite sul modello della piattaforma J2EE. Questo ha permesso a Spring di raccogliere numerosi consensi e di essere riconosciuto anche da importanti *vendor* commerciali quale *framework* di importanza strategica.
La *Java2 Enterprise Edition* è, come è noto, *la piattaforma* definita da Sun per la realizzazione di *applicazioni Java* di classe Enterprise. I suoi punti cardine sono: uno strato Web, composto da JSP e Servlet, per il *front-end* e uno strato EJB, di *back-office*, che contiene i cosiddetti "*business object*".
In teoria, separare la logica di business dalla presentazione, è una buona idea. L'avere un contenitore**,** che fornisce dei servizi come le transazioni*,* e ne gestisce il ciclo di vita, invece è una ottima idea.
Gli EJB soffrono però di una grande complessità. Sono difficili da capire, e pesanti da implementare. Nella pratica, molti progetti hanno avuto problemi a causa di un uso scorretto degli EJB.

Nel suo libro, Johnson ha cercato di spiegare come sviluppare applicazioni utilizzando correttamente il J2EE (EJB compresi), per poi arrivare a proporre lo sviluppo J2EE ma senza utilizzare gli EJB, o meglio, più correttamente, senza forzatamente incapsulare la *business logic* negli EJB.
Gli EJB rimangono utili, ma sono per certi scopi. Invece, per i *Business Object*, Johnson, riprendendo una scuola di pensiero diffusa, propone di usare dei POJO. POJO significa *Plain Old Java Object* (Vecchio Semplice Oggetto Java).
I POJO però sono troppo semplici, e per essere gestiti correttamente, hanno bisogno di inizializzazione, gestione dipendenze e altri aspetti.
Ecco che per gestire i POJO, Johnson finalmente propone il *framework* Spring.

In pratica Spring è un *framework* che permette di sostituire il contenitore di EJB J2EE, e di fornire a dei POJO servizi similari a quelli offerti dal J2EE, ma

senza EJB. In realtà, ci sono molte altre idee innovative in Spring , come per esempio l'Inversione del Controllo e l'uso intensivo di *Unit Test*.

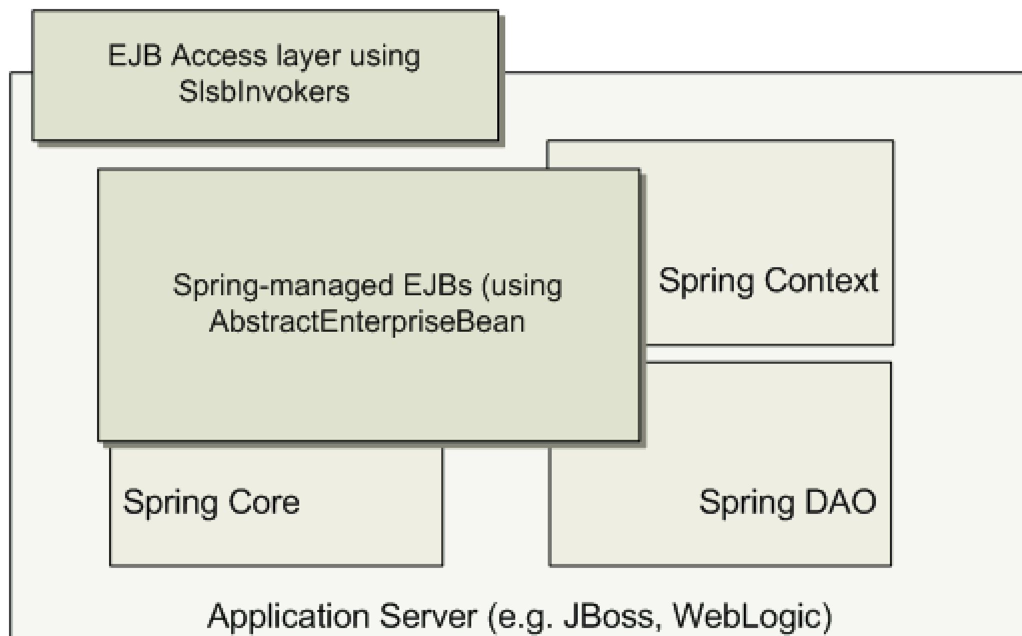

Figura 3.1 *Enterprise JavaBean*: il *wrapping* di POJO già implementati.

Tra le numerose funzioni offerte, c'è anche un semplice *framework* MVC, *l'Aspect Oriented Programming*, l'integrazione con una gran quantità di tecnologie (Hibernate per esempio viene usato spesso insieme a Spring). Le funzionalità del *framework* Spring possono essere usate in qualsiasi server J2EE e molte di esse possono essere adattate ad ambienti non gestiti. Un punto di forza di Spring è quello di permettere il riutilizzo degli oggetti *data-access* e business non legati a specifici servizi J2EE. Alcuni oggetti possono essere riutilizzati tra vari ambienti J2EE (Web e EJB), applicazioni *stand-alone*, ambienti di test, e così via senza problemi.

3.2 Dependency Injection

Come già accennato nel precedente capitolo, *Dependency Injection* (DI) è un *pattern* di programmazione nel quale l'implementazione di una classe, detta *dependent*, è realizzata utilizzando i servizi di un'altra classe, detta *dependency*.

Tipicamente la classe *dependency* costituisce una interfaccia, per la quale sono fornite numerose implementazioni (sottoclassi), ciascuna delle quali realizza le funzionalità richieste in modo differente. Nel caso in cui fosse la stessa classe *dependent* a creare una opportuna istanza della classe *dependency*, si verrebbe a creare una forte dipendenza tra la classe *dependent* e l'implementazione scelta per la classe *dependency*, rendendo l'applicazione difficilmente modificabile.

Il *pattern* DI specifica invece che la creazione dell'istanza della classe *dependency* sia effettuata da una classe esterna e che la *dependency* sia quindi iniettata nella classe *dependent*: questo elimina la dipendenza stretta tra la classe *dependent* e l'istanza della classe *dependency*, permettendo facilmente di modificare, anche dinamicamente, l'implementazione scelta.

3.3 Inversion of Control Container

La componente fondamentale del *framework* Spring, come si è potuto capire dalle spiegazioni date in precedenza, è la funzionalità di *Inversion of Control* (IoC), realizzata attraverso un componente detto *Inversion of Control container* (IoC *container*). Un *container* IoC permette di configurare e gestire il ciclo di vita degli oggetti Java che compongono l'applicazione: le interfacce BeanFactory e ApplicationContext realizzano tali funzionalità. Il *framework* Spring definisce come *beans* tutti gli oggetti che sono configurati e gestiti tramite un IoC *container* di Spring.

Il principale vantaggio del *framework* Spring risiede nel fatto che tutti i *beans* che compongono la propria applicazione e le dipendenze tra di essi possono essere configurati tramite opportuni meta dati (*configuration meta data*), spesso espressi sotto forma di file XML. Infatti tramite la classe XMLBeanFactory è possibile configurare completamente la propria applicazione attraverso un *file* XML, il quale contiene le definizioni dei *beans* e la descrizione delle loro dipendenze: sarà compito del *container* istanziare gli opportuni oggetti Java che rappresentano i *beans*, configurarli e gestire il loro ciclo di vita. Ad ogni *bean* di Spring è inoltre associato uno *scope*: la definizione di un *bean* costituisce infatti solo una ricetta

(*recipe*) per la creazione di istanze della classe associata alla definizione, mentre lo *scope* controlla il modo in cui le istanze sono create.
Il *framework* definisce varie tipologie di *scopes*, tuttavia le più importanti sono gli *scopes singleton* e *prototype*: associando ad un *bean* lo *scope singleton*, si assicura che ne esisterà al più una sola istanza all'interno del *container* IoC, mentre tramite lo *scope prototype* il *container* provvederà a creare una nuova istanza del *bean* ad ogni richiesta.

3.4 *BeanFactory* e *ApplicationContext*

Analizziamo più dettagliatamente quali sono le classi che implementano l'IoC. Due dei package più fondamentali ed importanti in Spring sono: *org.springframework.beans* e *org.springframework.context*.
Il codice presente in questi package, fornisce le basi per l'IoC di Spring (o l'iniezione delle dipendenze che dir si voglia).

In particolare il *BeanFactory* fornisce un meccanismo di configurazione avanzato, capace di amministrare *beans* (oggetti remoti) di ogni natura.

L'*ApplicationContext*, che è nello specifico una sotto-classe di *BeanFactory*, aggiunge altre funzionalità importanti, come una più facile integrazione con le caratteristiche AOP di Spring, risorse per l'internazionalizzazione dei messaggi e un contesto applicativo specifico, come il *WebApplicationContext*, sul quale poter basare il proprio lavoro.

In breve, il *BeanFactory* offre la struttura di configurazione e le funzionalità di base, mentre *l'ApplicationContext* potenzia e migliora sensibilmente quest'ultimo.

In generale, un *ApplicationContext* è un insieme completo di *BeanFactory*, e si dovrebbe considerare che ogni descrizione delle capacità di un *BeanFactory* si possono riferire parallelamente ad un *ApplicationContext*.

I programmatori Java, sono incerti se in certe situazioni sia meglio avvalersi di un *BeanFactory* o di un *ApplicationContext*. Normalmente quando si sta costruendo la maggior parte delle applicazioni un ambiente J2EE, la scelta più flessibile e funzionale è usare l'*ApplicationContext*, poiché offre tutte le caratteristiche del *BeanFactory* e ne aggiunge alcune in più. Invece uno scenario in cui è consigliabile usare il *BeanFactory* è quando la più grande preoccupazione è l'utilizzo di memoria (come in un *applet* dove conti ogni ultimo *kilobyte*), e non si ha bisogno di tutte le caratteristiche dell'*ApplicationContext*.

Ora occupiamoci del primo dei due package fondamentali appena introdotti: il *BeanFactory*.

3.4.1 *Beans* e *BeanFactory*

Il *BeanFactory* è il *container* attuale che istanzia, configura, e amministra i cosiddetti *beans* (oggetti di tipo remoto).
I beans collaborano tipicamente tra di loro, e così hanno una forte relazione che li lega. Queste relazioni sono dichiarate nei dati di configurazione usati dal *BeanFactory* (anche se alcune dipendenze non possono essere visibili come dati di configurazione, ma piuttosto come una serie di interazioni programmatiche tra oggetti a tempo di esecuzione).

Un *BeanFactory* è rappresentato dall'interfaccia: *org.springframework.beans.BeanFactory*. Una sua implementazione è l'*XmlBeanFactory*, ecco cosa serve per usarla nel modo più semplice:

```
InputStream is = new FileInputStream("beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(is);
```

Listato 3.1 Implementazione di *XmlBeanFactory*.

Il file XML che usa l'*XmlBeanFactory* contiene la definizione dei *bean* da gestire:

```
<?xml version=”1.0” encoding=”UTF-8”?>

<!DOCTYPE beans PUBLIC “-//SPRING//DTD BEAN//EN”
http://www.springframework.com/dtd/spring-beans.dtd>

<beans>

<bean id=”…” class=”…”>
…
</bean>

<bean id=”…” class=”…”>
…
</bean>

…

</beans>
```

Listato 3.2 Composizione del file XML per l’*XmlBeanFactory*.

Grazie alla grandissima duttilità del *framework*, possiamo avere vari approcci per l'IoC, esaminiamo nel dettaglio quelli che Spring può usare.

Il primo metodo che può usare Spring è quello *setter-based*. Qui l' iniezione delle dipendenze è realizzata chiamando i metodi *set* sui *bean*, dopo averli istanziati con un costruttore senza argomenti o una *factory*.

Spring usa però principalmente questo approccio: il *constructor-based*. Con questo metodo invece, l'iniezione delle dipendenze è realizzata invocando un costruttore con un numero di parametri. Ciascuno di questi rappresenta un collaboratore o una proprietà.

In aggiunta, si può specificare o il metodo di creazione o il costruttore. Spring supporta anche questo approccio, per integrare eventuali bean pre-esistenti che sono forniti di soli costruttori e non hanno metodi *set*.

Perciò a seconda dell' approccio scelto vengono valorizzate le definizioni dei *bean* nel metodo opportuno.

Quando si crea un *bean* che usa l'approccio basato sul costruttore, tutte le classi normali sono utilizzabili da Spring e compatibili con esso. Ovvero, l'atto di creazione della classe, non ha bisogno di perfezionare delle specifiche interfacce.
Nonostante ciò, in base alla scelta fatta di IoC, si potrebbe aver bisogno di un costruttore predefinito di *default* (vuoto).
Invece, quando si definisce un *bean* che sarà creato usando il metodo statico, oltre all'attributo che specifica la classe contenente il metodo statico, è necessario un altro attributo chiamato *factory-method*, per specificare il nome del metodo vero e proprio.
Spring si aspetta di essere in grado di chiamare questo metodo e di riavere indietro un oggetto che da quel punto in avanti può essere maneggiato come se fosse stato creato normalmente via costruttore.

Piuttosto simile all'utilizzo del metodo di costruzione statico, è l'uso del metodo via istanza (non-statico), dove un costruttore di un *bean* già esistente è invocato per creare il nuovo. Per usare questo meccanismo l'attributo di classe deve essere lasciato vuoto, e l'attributo *factory-bean* deve specificare il nome di un *bean* che contenga il costruttore che si desidera sfruttare.
Una volta costruito il nostro *bean* in base alle specifiche necessarie, è d'obbligo definire i suoi "identificatori", che devono essere nominati in modo tale da definire univocamente un bean nell'ambiente in cui è stato creato.

L'attributo *id* permette di specificare un *id*, ed essendo specificato nell'XML DTD (documento di definizione), come un vero e proprio elemento XML di attributo ID, il *parser* è in grado di fare della validazione addizionale nel

momento in cui altri elementi puntino a quest'ultimo. Questo è il modo preferibile di specificare un *id* per un *bean*. Nonostante ciò, le specifiche XML limitano i caratteri che possono essere utilizzati negli ID XML.
Questo non è di solito realmente un problema, ma nel caso in cui si ha bisogno di usare uno di questi caratteri, o si vogliono dichiarare degli pseudonimi per quel determinato *bean*, si devono specificare i diversi *id*, attraverso l'attributo *name*, separati opportunamente da una virgola o da un punto e virgola.

Tutti i *bean*, una volta costruiti ed identificati univocamente nel contesto di sviluppo, possono essere implementati in due modi: *singleton* o *non-singleton*.
Quando un bean è *singleton*, sarà maneggiata solamente un'istanza condivisa di quest'ultimo, e tutte le richieste per *bean* con un *id* compatibile con quella specifica definizione, utilizzeranno l'istanza globale. Al contrario, un *bean non-singleton*, dà luogo alla creazione di una nuova istanza, ogni volta che viene richiesta una specifica operazione per quel *bean*, o per i suoi pseudonimi.
Questo è ideale in situazioni dove ogni utente ha bisogno di un oggetto indipendente da utilizzare, senza necessità di condividere alcuna informazione col resto del sistema.

3.4.2 Proprietà, collaboratori e controllo delle dipendenze

L' Inversione di Controllo (IoC) è stata definita nei paragrafi precedenti, anche come Iniezione di Dipendenze.

Il principio di base è che i *bean* definiscono le loro dipendenze (cioè gli altri oggetti con cui lavorano) solamente attraverso gli argomenti del costruttore, o quelli di un *factory-method*, o attraverso delle proprietà che sono presenti nell'istanza dell'oggetto una volta creato. Dopo di che, è il compito del *container* quello di iniettare per davvero le dipendenze quando crea l'istanza del bean.
Questo fondamentalmente è l'inverso (da esso il nome di Inversione di Controllo) di quello che succede quando è lo stesso *bean* a localizzare ed implementare l'iniezione delle proprie dipendenze.

Senza dilungarsi troppo sui vantaggi dell'Iniezione di Dipendenze, diviene evidente che il codice trova molti vantaggi sia a livello di chiarezza che di semplicità quando i *beans* non devono controllare direttamente le proprie dipendenze.
Questo è dovuto al fatto che in questa maniera i *beans*, non necessitano di nessuna informazione addizionale da gestire, ma delegano tutte le responsabilità di gestione al proprio *container.*

Un'altra considerazione da fare riguardo i collaboratori di un *bean* e il controllo fatto sulle sue dipendenze è che, per la maggior parte delle applicazioni, tutti o quasi i *beans* nel *container* saranno *singleton*.
Quindi, quando un *bean* di questo tipo necessita di collaborare o di utilizzare un altro *bean singleton*, o quando un *non-singleton* ha bisogno di accedere alle proprietà di un altro *non-singleton*, l'approccio tipico e comune per risolvere e amministrare queste dipendenze, è quello di definire un *bean* come proprietà di un altro.
Questo è un modo corretto e funzionale di risolvere il problema delle dipendenze e del loro controllo, purtroppo presenta alcune difficoltà.
La più lampante e difficile da arginare, è la gestione di *beans* con *"life-cycle"* differenti, infatti se un *bean* viene settato come proprietà di un altro, ma non è temporalmente contemporaneo ad esso, genererà innumerevoli problemi a livello di compatibilità e visibilità.
Una possibile soluzione di questo problema è fare a meno dell'IoC, o comunque di non utilizzarne tutte le funzionalità.
Così facendo il *bean* che deve utilizzare le proprietà di un altro, ha il pieno controllo delle proprie dipendenze, e quindi può accedere alle informazioni che necessita in qualsiasi momento, senza problemi di nessun tipo a livello temporale.
Naturalmente questo significa porre gravi limitazione alle funzionalità avanzate che offre Spring.

Capitolo 4

L'integrazione JADE-JXTA-RAIS

Questo capitolo dapprima descriverà brevemente i due principali progetti che hanno permesso la realizzazione di questo lavoro; in seguito spiegherà come è stata portata a termine la prima parte del progetto di tesi, ovvero l'integrazione di jade-jxta con il sistema P2P ad agenti RAIS, al fine di ottenere un sistema P2P puro.

4.1 Progetti di partenza: *jade-jxta* e RAIS

Andiamo ora a parlare delle due principali fondamenta di questo progetto di tesi: ***jade-jxta*** e **RAIS**. Entrambe le architetture sono riconducibili a progetti realizzati da studenti presso l'*Università degli Studi di Parma*, sempre nell'ambito di ricerca dei laboratori AOT.

Il primo progetto (***jade-jxta***, Agosti-Scalise) partendo appunto dai sorgenti di JADE e di JXTA, ha portato ad ottenere un'integrazione immediata e trasparente della piattaforma JADE all'interno di una rete P2P JXTA. Questo è stato possibile

sia tramite alcune modifiche alle architetture di partenza, sia tramite veri e proprio moduli e *package* creati “*ad-hoc*” ed aggiunti ai sorgenti originali.
L’idea di questo progetto è nata in seguito all’osservazione attenta di alcuni sistemi distribuiti ad agenti. Essi esibivano tutti i pregi degli agenti *software* (autonomia, socievolezza, pro-attività, intelligenza, …) però, interagendo tramite piattaforme confederate e non tramite una vera e propria rete *peer-to-peer*, essi andavano incontro a numerosi problemi comunicativi. Infatti la connessione tra piattaforme remote non sempre è semplice, spesso si possono incontrare ostacoli come il NAT o il DHCP, che rendono difficoltoso lo scambio di informazioni tra due o più piattaforme confederate. Ovviamente con l’utilizzo di reti di tipo *peer-to-peer* questi problemi non si incontrano, poiché essi ricorrono all’utilizzo della rete di *overlay* che rende trasparenti le comunicazioni tra *peer* remoti.

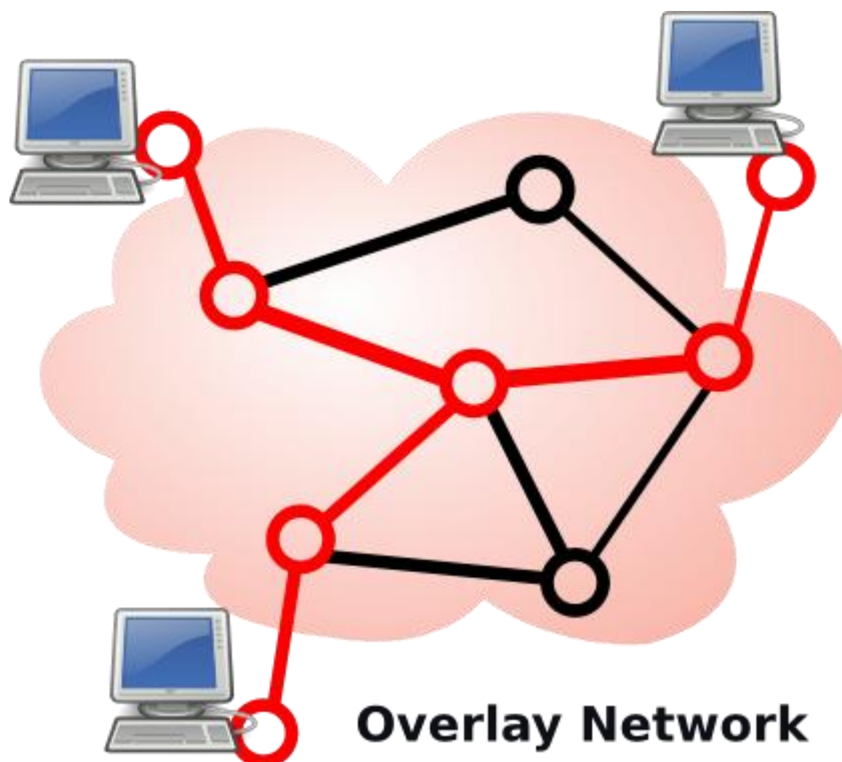

Figura 4.1 Esempio di una rete di *overlay*.

Nasce da qui quindi la necessità di fornire le piattaforme JADE di un sub-strato comunicativo di tipo *peer-to-peer* (JXTA), in modo da mantenere i privilegi dati dall’utilizzo degli agenti abbinati alla trasparenza comunicativa portata dalle reti di tipo *peer-to-peer*.

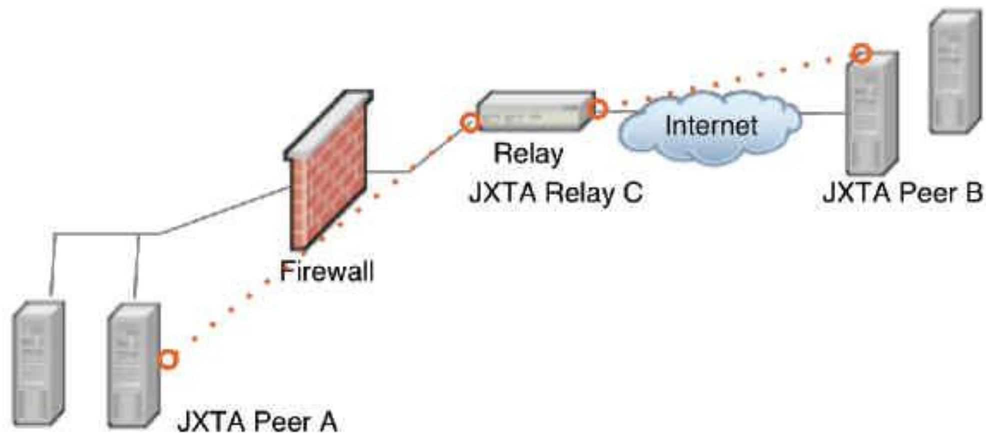

Figura 4.2 Processo comunicativo base di JXTA.

Il secondo progetto (**RAIS**: *Remote Assistant for Information* Sharing, Minari-Simonazzi), di cui si è già ampiamente parlato nel capitolo 2, si è posto come scopo la realizzazione di un sistema P2P per il c*ontent-sharing* (a livello accademico e non), che permettesse una condivisione di risorse in modo veloce ed efficiente. Infatti, per far fronte all’esigenza sempre crescente di categorizzare e rendere immediatamente reperibili i documenti testuali o multimediali all’utente, sono sorte in ambito informatico tecnologie per il c*ontent-sharing*. Con questo termine si indica la condivisione di documenti in reti *peer-to-peer*, abbinata alle capacità di ricerca di un’applicazione *Desktop Search*, che permetta di effettuare una ricerca “intelligente” su migliaia di *file*. L’unico problema che presentava RAIS nel suo primo prototipo era che, per simulare una rete *peer-to-peer*, faceva uso di piattaforme remote confederate, andando incontro ai problemi di comunicazione tipici di cui si è parlato sopra nel dettaglio.

Quindi è facile intuire come mai la prima parte di questo lavoro di tesi sia stata incentrata in un primo momento sull’integrazione dei protocolli comunicativi di JXTA con le piattaforme JADE e poi sulla ricostruzione di RAIS con questa nuova “impalcatura” di rete.

4.2 Modifiche apportate

In questa sezione andremo a studiare nel dettaglio le modifiche che è stato necessario apportare a *jade-jxta* ed a RAIS, per fare in modo che si potesse realizzare un'integrazione precisa e coesa fra i due progetti.

Nella maggior parte dei casi queste modifiche sono state veri e propri moduli aggiuntivi creati "*ad-hoc*" per poter implementare nuove funzionalità, o per permettere a funzionalità già presenti di interfacciarsi con quelle nuove inserite.

Un'altra parte consistente delle modifiche è stata fatta modellando, in base alle nuove esigenze, i vecchi codici sorgente dei due sistemi per fare in modo che l'integrazione potesse avvenire a tutti i livelli: da quello più alto delle interfacce a quello più basso dell'invio di messaggi.

Infine altre modifiche sono state fatte per correggere vecchi errori o imperfezioni che rendevano difficoltose le fasi di *testing* del sistema, ma queste rappresentano un percentuale minima dell'intero lavoro svolto.

Ora andiamo a vedere nel dettaglio le modifiche apportate nell'ordine prima a *jade-jxta* e successivamente a RAIS.

4.2.1 Modifiche a *jade-jxta*

Prima ancora di inserire le funzionalità di *jade-jxta* in RAIS è stato necessario renderle operative e disponibili, sulla macchina sulla quale si stava lavorando. Inizialmente si è dovuta cancellare la cartella *.jxta* contenente le configurazioni passate della rete JXTA, successivamente al primo avvio, nel momento in cui vengono fatti partire JADE e JXTA, sono stati immessi manualmente i *settings* di *username*, *password* e tipo di *peer* (solitamente da impostare come *rendez-vous*).
Sin dall'inizio si è notato che non si riusciva a far partire due *peer* contemporaneamente sullo stesso PC poiché non possono essere lanciate nel medesimo momento due istanze di JADE. Quindi già per effettuare tutte le

primissime fasi di *testing*, è stato d'obbligo lanciare due *peer* su PC remoti differenti. Per fare ciò solitamente sono stati configurati entrambi gli utenti come *rendez-vous peer*, oppure è anche possibile impostarne uno come *rendez-vous* e l'altro che si collega a quest'ultimo tramite la configurazione manuale dell'indirizzo *rendez-vous*: tcp://XXX.XXX.XXX.XXX:9701.

Per far funzionare RAIS con il supporto P2P di *jade-jxta* è bastato aggiungere manualmente i sorgenti e le librerie nel *classpath* specifico di RAIS, e puntualizzare, al momento dell'avvio del sistema, che si doveva utilizzare l'MTP (*Message Transport Protocol*) di *jade-jxta* e non quello di *default* di JADE.
Quindi per il primissimo avvio non si sono incontrate particolari difficoltà. Nonostante ciò, con successivi test, è risultato chiaro che alcune modifiche andavano apportate anche ai sorgenti di *jade-jxta*, in quanto questo progetto poteva fornire servizi di base, ma per alcune funzionalità avanzate di cui necessitava RAIS vi era bisogno di mettere le mani sul nucleo operativo di *jade-jxta* e quindi modificarne il codice.

Quando si parla del nucleo operativo di *jade-jxta* equivale a parlare dell'agente JxtaDF.
Il JxtaDF è un agente JADE che implementa alcuni servizi offerti solitamente dal DF *standard*. Esso si registra presso il DF come un normale servizio di nome "JXTA DF". L'agente JxtaDF mantiene un riferimento al nodo JXTA associato all'MTP utilizzato e lo usa per pubblicare e ricercare *advertisement* codificati secondo lo *standard* FIPA 00096.

```
* Static method to get the AID of the JXTA DF registered to this platform
        *
        * @param sender: the AID of the agent that performs the request
        * @return: the AID of JXTA DF, null if is not registered
        *
        */
      public static AID searchJxtaDF(Agent sender) {
            DFAgentDescription dfd = new DFAgentDescription();
            ServiceDescription sd = new ServiceDescription();
            sd.setName(JxtaDF.SERVICE_NAME);
            sd.setType(JxtaDF.SERVICE_NAME);
            dfd.addServices(sd);
            DFAgentDescription[] result = null;
            try {
                  SearchConstraints sc = new SearchConstraints();
                  sc.setMaxResults(new Long(1));
                  sc.setMaxDepth(new Long(20));
                  result = DFService.search(sender, dfd, sc);
            } catch (FIPAException e) {
                  e.printStackTrace();
            }
            if (result.length > 0)
                  return result[0].getName();
            else {
                  System.out.println("No JxtaDF registered in this DF");
                  return null;
            }
      }
}
```

Listato 4.1 Il metodo *searchJxtaDF()*.

Gli agenti JADE possono effettuare le operazioni canoniche di REGISTER e SEARCH semplicemente richiedendo l’AID del JxtaDF tramite un metodo di tipo *static* messo a disposizione dall’MTP stesso (come si può osservare dal listato 4.1 riportato sopra).

Nello specifico del nostro sistema, le operazioni che deve svolgere il JxtaDF, sono di un’importanza fondamentale poiché permettono agli agenti locali

dei vari *peer* di registrarsi presso il JxtaDF, che a questo punto diventa una sorta di DF pubblico, grazie al quale è possibile fare il *forwarding* delle *query* ai vari utenti che ne han fatto richiesta.
In pratica è proprio il JxtaDF che fa da “collante” tra la piattaforma JADE con i suoi agenti e l’MTP di JXTA con la sua rete di *overlay*.

Le modifiche che è stato necessario apportare al JxtaDF per fare in modo che si integrasse appieno con RAIS, a dire il vero, non sono state di grande entità.
Infatti l’intera struttura già esistente era perfettamente funzionante, ma presentava anomalie operative abbastanza importanti, sia quando non vi era alcun agente registrato sul JxtaDF, sia quando ve ne era più di uno per un determinato *peer*.
Questi errori si verificavano poiché il sistema in principio non era stato progettato per gestire casi operativamente complessi come RAIS. Esso infatti riusciva solo a gestire le interazioni e le ricerca di due semplici agenti di *testing*: un *PublisherAgent* ed un *QuerierAgent*.
Quando non vi erano agenti registrati come servizi (i *PublisherAgent*) o quando ve ne era più di uno registrato per uno stesso nodo della rete JXTA, il sistema iniziava ad entrare in un *loop* di controlli lunghissimo, che finiva col mandare in blocco le funzioni operative di RAIS.
Per questo motivo è stato necessario sistemare il metodo *setup()* del JxtaDF nel punto in cui, una volta ricevuta una richiesta di tipo SEARCH, andava a verificare gli agenti registrati a tale nome.
Quello che è stato fatto, è semplicemente andare a sistemare i controlli che venivano effettuati dal JxtaDF circa il numero di *PublisherAgent* registrati, ed aggiungere altri casi operativi contemplati nell’utilizzo di un sistema come RAIS.
Il listato 4.2 illustra le modifiche apportate al codice del JxtaDF secondo le direttive illustrate sopra:

```
… … …
else if (concept instanceof Search)
{
 Search searchRequest = (Search) concept;
 System.out.println("");
 System.out.println("JxtaDF: New SEARCH request from "
 + ((DFAgentDescription) searchRequest.getDescription()).getName() + ".");
 System.out.println("");

 jade.util.leap.List results =
 handleSearch((DFAgentDescription) searchRequest.getDescription());

 // Case: 1 or more agents found
 if (results.size() > 0)
  {
   ACLMessage response = createReply((Action) content, request);
   … … …
   try
     {
      getContentManager().fillContent(response, ce);
      … … …
     }
     … … …
     send(response);
  }
 // Case: no agents found
 else
    {
     ACLMessage response = createReply((Action) content, request);
     Action act = new Action(getAID(),
     (Action) content);
     ContentElement ce = new Result(act, results);
     try
       {
        getContentManager().fillContent(response, ce);
        … … …
       }
       … … …
       send(response);
… … …
```

Listato 4.2 I nuovi controlli per il metodo *handleSearch()* del JxtaDF.

Dopo avere illustrato i cambiamenti che è stato necessario apportare al sub-strato comunicativo fornito da *jade-jxta*, andiamo ora a vedere, nei prossimi paragrafi, le modifiche apportate all'altro protagonista di questo lavoro: RAIS. I cambiamenti apportati alle classi originarie del sistema in questione, sono stati effettuati quasi esclusivamente per sfruttare al meglio le nuove potenzialità portate dall'uso dell'MTP di JXTA e quindi di una vera e propria rete *peer-to-peer*.

4.2.2 Modifiche a RAIS

Il funzionamento di RAIS nella sua prima stesura era molto semplice e lineare: ogni *peer* era rappresentato da una piattaforma JADE confederata ad altre rappresentanti gli altri *peer*. Questa piattaforma conteneva, oltre agli agenti propri di JADE, un *DesktopSearchAgent* ed un *LocalSearchAgent*. I due agenti si facevano carico della ricerca e della successiva visualizzazione dei *file* contenenti le parole chiave. In poche parole il *DesktopSearchAgent* veniva usato dall'utente per sottoscrivere la *query* e successivamente per inviare tale query a tutti i *LocalSearchAgent* presenti nelle diverse piattaforme remote confederate che davano una risposta in base ai risultati della loro ricerca locale.
E' chiaro che un funzionamento di questo tipo è ottimale, ma troppo centralizzato e diverso dal concetto vero e proprio di *peer-to-peer* poiché l'uso di piattaforme confederate è troppo centralizzato e non sfrutta i vantaggi delle reti di *overlay*.
Per questo motivo è stato necessario effettuare l'integrazione con *jade-jxta*, andando a modificare alcune parti chiave dei vecchi *file* sorgente.

Prima di tutto andiamo ad elencare le fasi di *setup* che il nuovo sistema RAIS deve essere in grado si processare:

1. eseguire il *boot* di JADE col DF *standard* ma usando l'MTP di *jade-jxta*;
2. aggiungere alla piattaforma l'agente JxtaDF;
3. aggiungere alla piattaforma il *DesktopSearchAgent* e il *LocalSearchAgent*;
4. il *LocalSearchAgent* chiede al DF (via MTP) l'indirizzo del JxtaDF;
5. il *LocalSearchAgent* ottiene l'AID del JxtaDF;
6. il *LocalSearchAgent* si registra come servizio sul JxtaDF.

Ovviamente queste fasi devono essere effettuate da ogni *peer* che desideri registrarsi al sistema per la condivisione di informazioni.

Da quanto detto sopra risultano evidenti i primi cambiamenti che vanno apportati:

- il servizio di ricerca deve essere registrato sul JxtaDF e non più sul DF;
- la ricerca di un determinato servizio và fatta sul JxtaDF e non più sul DF.

Un primo problema incontrato per la registrazione del servizio, è stato dato dalle librerie di JAXB. Il problema era dato dal fatto che la JRE 1.6 integra al suo interno JAXB 2.0, mentre il progetto originario utilizza JAXB 2.1, quindi fallisce la fase di *binding*. La soluzione ottimale è stata quella di effettuare un *endorsment*, ovvero una specie di “forzatura” per la VM a caricare la nostra libreria prima di quella sua di *default*.

Una volta sistemate le fasi di *setup* del sistema esso è pronto per processare le richieste. Anche in questo caso però, per rendere perfettamente operativo RAIS con il nuovo sub-strato, è stato necessario sistemare gli attori principali delle interazioni: il *DesktopSearchAgent*, il *LocalSearchAgent* ed i *Behaviours*.

4.2.2.1 Modifiche al *DesktopSearchAgent*

Il *DesktopSearchAgent* è l’agente utilizzato per la ricerca nei *peer*. Esso invia le *query* ai *LocalSearchAgent* e visualizza i successivi risultati ottenuti nella GUI da lui creata. E’ uno dei due diversi agenti che compongono l’applicazione. Esso presenta all’utente un’interfaccia grafica che permette di lanciare ricerche e visualizzare i risultati. Questo agente ha il compito di inviare le *query* a tutti i *peer* della rete su cui sono attivi gli agenti *LocalSearchAgent*, che a loro volta risponderanno inviando i risultati della ricerca. L’agente può anche essere usato semplicemente per svolgere ricerche sulla macchina locale, nel caso non sia disponibile la rete *peer-to-peer*.

Il *DesktopSearchAgent* non impone alcuna dipendenza dal componente per il *Desktop Search* vero e proprio, che viene contattato esclusivamente dall'altro agente che compone l'applicazione; di fatto non è necessario che sulla macchina locale sia installato alcun *software Desktop Search* per eseguire ricerche sulla rete. Tutti i messaggi di richiesta e risposta scambiati tra questo agente ed i *LocalSearchAgent* si basano sull'infrastruttura di comunicazione offerta dalla piattaforma JADE. Il formato dei risultati ricevuti è *standard* e non dipende dal *software Desktop Search* usato da ciascun *peer*.

Questo agente presenta tre *behaviour*, ovvero implementazioni dell'interfaccia *Behaviour* di JADE usata per modellare i compiti svolti dagli agenti. Tutti e tre sono, in particolare, estensioni di *CyclicBehaviour*, ovvero vengono eseguiti a ciclo continuo:

- *ReceiveResultsBehaviour*: attivato alla ricezione dei risultati inviati da un *LocalSearchAgent*, nel caso dell'ordinamento di tipo "nativo". L'arrivo dei risultati viene notificato all'interfaccia grafica, che si occupa di visualizzarli;
- *SetNumResultsBehaviour*: attivato alla ricezione dei dati per il primo passo dell'algoritmo di ordinamento a "due passi". Viene calcolato in base all'algoritmo il numero di risultati che ciascun *peer* dovrà inviare, e questo valore viene spedito come risposta a ciascun *peer*;
- *ReceiveResults2StepBehaviour*: attivato alla ricezione dei risultati nel caso dell'ordinamento a "due passi". Rispetto al caso dell'ordinamento nativo viene anche eseguito il riordino della lista dei risultati.

Oltre a questi vi è un altro *behaviour*, la cui attivazione è comandata dall'utente tramite interfaccia grafica: si tratta di un *SenderBehaviour* che invia la *query* dando così inizio allo scambio di messaggi con i *peer*. È proprio questo messaggio ad indicare la tipologia di ordinamento prescelta ("nativa" o a "due passi"), in base al campo *Ontology*. Il messaggio che rappresenta la *query* è un oggetto di tipo *QueryMessage*, contenente i termini di ricerca in formato testuale e il numero massimo di risultati che si intende ricevere.

```
… … …
// The next 4 functions are necessary to
// change the results message in the GUI
public void getResults(ArrayList<CommonSearchResult> list)
throws Exception {
            counter++;
            if (counter == numPeers) {
                  gui.addResults(list, true);
                  counter = -1;
            } else
                  gui.addResults(list, false);
      }

public void getAndSortResults(ArrayList<CommonSearchResult> list)
throws Exception {
            counter++;
            if (counter == numPeers) {
                  gui.addAndSortResults(list, true);
                  counter = -1;
            } else
                  gui.addAndSortResults(list, false);
      }

public void noResultsFound() {
counter++;
if (counter == numPeers) {
      gui.setMessage("No search results received from every peer!");
      counter = -1;
} else
      gui.setMessage("No search results received from peers reachable
                     at the moment!");
      }

public void needRestart() {
      gui.setMessage("Every peer is unreachable at the moment: try to
                     type a new query or restart the program!");
      }
// ...end GUI function
… … …
```

Listato 4.3 I quattro nuovi metodi della GUI nel *DesktopSearchAgent*.

Le modifiche che è stato necessario apportare a questo agente riguardano per lo più controlli necessari ad una corretta visualizzazione dei risultati. Inoltre sono stati anche migliorati i messaggi e gli eventi d'avviso, per rendere più chiaro e trasparente nei confronti dell'utente il comportamento del sistema. Oltre a delle semplici variabili *flag*, che si occupassero di salvare alcune importanti informazioni storiche riguardo le varie ricerche effettuate (come per esempio il numero di *peer* che hanno risposto, quelli che hanno portato risultati utili, …), le vere novità aggiunte al DSA sono stati quattro metodi che si occupassero delle varie visualizzazioni sulla GUI di tutti i casi possibili di risultati e risposte ricevute o meno. Il listato 4.3 riportato sopra, evidenzia i comportamenti e le funzionalità di questi nuovi metodi.

Andiamo ora a vedere nel dettaglio le modifiche fatte all'altro agente protagonista del sistema: il *LocalSearchAgent*.

4.2.2.2 Modifiche al *LocalSearchAgent*

Il *LocalSearchAgent* è il vero e proprio agente di ricerca. Esso accetta le *query* provenienti dal *DesktopSearchAgent* e usa le funzionalità offerte da un servizio di *Desktop Search* per fornire i risultati. Il *LocalSearchAgent* è l'altro agente dell'applicazione. La sua funzione è quella di servire le *query* inviategli dal *DesktopSearchAgent*, contattando un servizio *Desktop Search* sul computer locale per ottenere i risultati della ricerca e inviarli in risposta. Questo agente funge quindi da intermediario con il sistema *Desktop Search*, ed è necessario per abilitare la condivisione dei documenti locali; nel caso in cui non venga attivato, gli altri *peer* non potranno accedere ai contenuti locali dell'utente, anche se l'utente stesso potrà comunque eseguire ricerche nella rete *peer-to-peer* tramite l'altro agente, il *DesktopSearchAgent*.
L'agente presenta tre *behaviour* di tipo *CyclicBehaviour*:

- *SendResultsBehaviour*: attivato nel caso in cui l'algoritmo di ordinamento richiesto sia quello nativo del sistema *Desktop Search*. Si occupa di ricevere i termini di ricerca, eseguire la ricerca tramite Desktop Search, e rispedire in risposta i risultati;

- *QueryRequest2StepBehaviour*: attivato nel caso in cui l’algoritmo di ordinamento richiesto sia quello a “due passi”. Si occupa di ricevere la *query*, ma invece di restituire direttamente i risultati invia in risposta i dati statistici necessari all’algoritmo. Questo scambio di messaggi è il primo passo dell’algoritmo stesso;
- *SendResults2StepBehaviour*: esegue il secondo passo dell’algoritmo di ordinamento a “due passi”, e restituisce il numero di risultati richiesti per la *query*.

Il comportamento dell’agente quindi è di tipo passivo, in quanto tutti i suoi *behaviour* sono eseguiti in reazione alla ricezione di determinate richieste da parte degli altri *peer*. L’agente non ha controllo né sulla scelta dell’algoritmo di ordinamento, né sul numero massimo di risultati da inviare; può però eseguire un filtraggio preliminare dei risultati per evitare di esporre pubblicamente documenti che l’utente ha deciso di non condividere. A questo scopo viene utilizzata la lista dei percorsi del *file system* da condividere, che è stata specificata in fase di configurazione dell’applicazione.
È opportuno considerare, infatti, che le cartelle indicizzate dal sistema *Desktop Search* locale, e quindi accessibili per la ricerca, non necessariamente corrispondono a quelle che l’utente permette di condividere. I due elenchi infatti sono specificati separatamente, nella configurazione di questa applicazione e in quella dell’applicazione *Desktop Search*. Questo permette la massima versatilità di configurazione: l’utente può limitare l’accesso esterno solo ad alcuni tra i percorsi indicizzati, e i restanti saranno accessibili esclusivamente alle ricerche effettuate in locale sulla stessa macchina. Ovviamente non sarà in nessun caso possibile la ricerca su percorsi non indicizzati dal sistema *Desktop Search*. La comunicazione con il servizio *Desktop Search* avviene tramite interfacce generiche, per renderla il più possibile indipendente dall’implementazione del *Desktop Search*.

Per questo agente le uniche modifiche che è stato necessario effettuare riguardano la registrazione dell’agente sul JxtaDF. Infatti nella vecchia versione

di RAIS i *LocalSearchAgent* si registravano sul DF *standard* della piattaforma JADE. Ora invece è necessario che questi agenti si registrino sul DF della rete JXTA, in modo da essere visibili su tutta la rete di *overlay*.

Il listato 4.4 riporta il pezzo di codice utilizzato per registrare il LocalSearchAgent sul JxtaDF:

```
… … …
// Gets the AID of JXTADF
jxtaDF = MessageTransportProtocol.searchJxtaDF(this);

// Builds desired service description
ServiceDescription sd = new ServiceDescription();
sd.setName("DS-agent");
sd.setType("Desktop-Search-Engine");

// Builds the agent description used to make the query
DFAgentDescription dfd = new DFAgentDescription();
dfd.addServices(sd);
dfd.setName(getAID());

// Register LocalSearchAgent on the yellow pages of the JXTA DF
try {
     DFService.register(this, jxtaDF, dfd);
    } catch (FIPAException e1) {
                                 e1.printStackTrace();
                                }

dirList = Config.getInstance().getSharedDirs();
numStart = 0;

sservice = SearchServiceFactory.createSearchService();

// Add the behaviours to serve queries
addBehaviour(new SendResultsBehaviour(this));
addBehaviour(new QueryRequest2StepBehaviour(this));
addBehaviour(new SendResults2StepBehaviour(this));
… … …
```

Listato 4.4 La registrazione del *LocalSearchAgent* sul JxtaDF.

Dopo aver analizzato i due agenti che compongono l'applicazione, è necessario ora andare a studiare il lavoro eseguito sulle interfacce che modellano i compiti svolti dagli agenti: i *Behaviours*.

4.2.2.3 Modifiche ai *Behaviours*

Come già accennato sopra, le operazioni che gli agenti JADE sono in grado di compiere vengono definite attraverso oggetti di tipo *Behaviour*. I *behaviour* vengono creati estendendo la classe astratta *jade.core.behaviours.Behaviour*. Per fare in modo che un agente sia in grado di eseguire un particolare compito è sufficiente creare un'istanza della sottoclasse di *Behaviour* che implementa il *task* desiderato ed utilizzare il metodo *addBehaviour()* dell'agente. Ogni sotto-classe di *Behaviour* deve implementare i metodi:

- *action()*: cosa il *behaviour* deve fare;
- *done()*: condizione per cui il *behaviour* può considerarsi concluso.

In base a come vengono utilizzati questi metodi, si possono avere tre differenti tipi di *behaviour*:

1. *OneShotBehaviour*: il metodo *action()* viene eseguito un'unica volta. Il metodo *done()* restituisce sempre *true*;
2. *CyclicBehaviour*: non terminano mai completamente. Il metodo *action()* esegue la stessa operazione ogni volta che viene invocato. Il metodo *done()* restituisce sempre *false*;
3. *ComplexBehaviour*: hanno uno stato. Il comportamento del metodo *action()* varia in funzione dello stato. Il metodo *done()* restituisce *true* solo quando è verificata una certa condizione.

La comunicazione tra i due agenti del sistema, *DesktopSearchAgent* e *LocalSearchAgent*, segue protocolli differenti a seconda del tipo di algoritmo di ordinamento selezionato. Nel caso, più semplice, in cui si utilizzi l'ordinamento nativo il protocollo di comunicazione prevede soltanto due messaggi: una *query* che viene inviata dal *DesktopSearchAgent* a ciascun *LocalSearchAgent* della rete, e un elenco di risultati inviato in risposta da ogni *LocalSearchAgent*.

Nel caso dell'algoritmo di ordinamento a due passi (quello più frequentemente utilizzato), la comunicazione avviene come rappresentato nel seguente diagramma, in cui si riportano il campo *Ontology* di ogni messaggio ed i *behaviour* designati per gestire la comunicazione:

Figura 4.3 Diagramma delle interazioni tra gli agenti di RAIS.

Fatta questa necessaria introduzione ai *behaviour* ed a come vengono utilizzati durante le fasi operative del sistema, passiamo ora a vedere le modifiche apportate a questi particolari componenti di RAIS.

I cambiamenti apportati in questa fase sono stati fatti per migliorare la gestione delle interazioni e per controllare situazioni operative non ancora previste nel vecchio prototipo.
Questi miglioramenti si sono dovuti effettuare poiché nel vecchio sistema i *behaviour* interagivano tramite piattaforme remote confederate, mentre ora, utilizzando una vera e propria rete *peer-to-peer*, si è dovuto adottare alcuni accorgimenti per rendere gli scambi di messaggi compatibili con il nuovo MTP utilizzato.

Procedendo per ordine andiamo dapprima ad analizzare i *behaviour* del DSA (*ReceiveResultsBehaviour*, *SetNumResultsBehaviour* e *ReceiveResults2StepBehaviour*) e successivamente quelli dell’LSA (*SendResultsBehaviour*, *QueryRequest2StepBehaviour* e *SendResults2StepBehaviour*).

Nel *ReceiveResultsBehaviour*, e quindi parallelamente anche nel *ReceiveResults2StepBehaviour*, l’unica modifica apportata riguarda l’utilizzo di una variabile che permetta di stabilire e tenere in memoria se a quel determinato DSA sia mai arrivata una risposta positiva alla propria *query* da parte di qualche *peer*. Questo permette di conservare importanti informazioni storiche della ricerca, ma anche di far visualizzare all’utente attraverso la GUI gli opportuni messaggi in corso d’opera.
Il listato 4.5 permette di vedere come venga salvato lo stato del sistema e attraverso quali operazioni è possibile avvisare l’utente:

```
… … …
/**
  * Handles incoming search results from agents.
  */
public void action()
{
 ACLMessage msg = myAgent.receive(mt);
 if (msg != null) {
    try {
         ArrayList<CommonSearchResult> results =
         (ArrayList<CommonSearchResult>)msg.getContentObject();
         ((DesktopSearchAgent) myAgent).getResults(results);
         log.info("Received " + results.size() +
                  " results from agent " +
                  msg.getSender().getName() + ".");

 // Set to TRUE value the successful variable
((DesktopSearchAgent) myAgent).successful = true;
        } catch (Exception e) {
 // If there aren't search result,
 // it's shown on the GUI a failure message
 if(!((DesktopSearchAgent) myAgent).successful) ((DesktopSearchAgent)
    myAgent).noResultsFound();

        log.info("No search results found in message from agent " +
                 msg.getSender().getName() + "...");
                              }
                  } else {
                         block();
                        }
}
```

Listato 4.5 La nuova gestione eventi nel *ReceiveResultsBehaviour*.

Più complicate invece sono state le modifiche effettuate sul *SetNumResultsBehaviour*. Questo è dovuto al fatto che questo *behaviour* è il vero e proprio responsabile dell’invio delle *query* e della loro gestione, ovvero di quella che può essere definita come una delle parti più complicate e delicate dell’intero sistema.

Andiamo ora a vedere nel dettaglio le modifiche effettuate.

La prima cosa implementata nel *SetNumResultsBehaviour* è stata la gestione del caso di *failure*. Infatti nel primo prototipo, se nella ricerca precedente non venivano forniti risultati, l'*array docs* rimaneva vuoto e veniva conseguentemente lanciata una semplice eccezione. Ora invece ora invece, anche caso di ricerca fallimentare, si risponde nuovamente all'LSA, in modo che quest'ultimo gestisca anche il caso di *failure* e non solo quello di ricezione corretta dei risultati.
A questo punto le ricerche funzionavano bene, però si notava un comportamento anomalo nelle ricerche successive alla prima. In pratica, dopo la prima ricerca il *behaviour*, si comportava come se avesse già ricevuto da subito tutte le risposte dai *peer*. Questo avveniva poiché alla fine di una ricerca su una determinata *query* non veniva resettata la variabile *results*, e quindi il sistema si comportava come se fossero già arrivati dei risultati e quindi come se la ricerca fosse già conclusa. La correzione di questo problema, tramite il *reset* della variabile *results* alla fine di ogni ricerca, ha permesso di sistemare questa anomalia di funzionamento. Come abbiamo già visto, la prima modifica apportata è stata quella che, in caso di *failure*, mandava il messaggio di fallimento all'agente interessato conseguentemente al lancio di un'eccezione. Dopo svariate prove e test si è capito che così facendo non si faceva una cosa molto importante: incrementare il numero di *peer* che hanno risposto e quindi aspettare una risposta da quelli mancanti. Si è risolto il tutto gestendo separatamente e specularmente (all'arrivo di un *MessageTemplate*) i casi fallimentari e quelli con successo:

I. arriva un *MessageTemplate mt*;

II. vi è stato successo nella ricerca sull'LSA?

   A. aggiungo il *peer* a quelli di successo;

   B. tutti i *peer* hanno risposto?

      1. calcolo la tabella dei risultati e poi invio ai vari *peer* il messaggio che gli spetta.

   C. mancano dei *peer* all'appello?

1. attendo una loro risposta.

III. vi è stato fallimento nella ricerca sull'LSA?

A. aggiungo il *peer* a quelli di fallimento;

B. tutti i *peer* hanno risposto?

1. calcolo la tabella dei risultati e poi invio ai vari *peer* il messaggio che gli spetta.

C. mancano dei *peer* all'appello?

1. attendo una loro risposta.

```
… … …
public void action() {
// Non-blocking receive
ACLMessage msg = myAgent.receive(mt);
      if (msg != null) {
      try {
          FirstStepResult result = (FirstStepResult) msg.getContentObject();
… … …
trueadditionalterms = result.getAdditionalTerms();
truequery = result.getQuery();
truequeryterms = result.getQueryTerms();

if (!msg.getSender().equals(myAgent.getAID()))
 log.info("First step reply from agent " + msg.getSender().getName() + ".");

// Case: successful answer received from the current agent
if (result.getNumDocs() != null) {
… … …
/*
 * Add result to the list of replies so far. THERE IS ONLY ONE RESULTS LIST
 * and it's relative to the current query, so it is supported only one query
 * at a time... The other list ( noresults ) collects the results received
 * from the failure search!
 */
results.put(msg.getSender(), result);
… … …
```

Listato 4.6 Arrivo di un *MessageTemplate* e suo processo nel *SetNumResultsBehaviour*.

Per concludere, l'ultima modifica apportata al *SetNumResultsBehaviour*, è stata necessaria per rendere operativo e compatibile col vecchio sistema, il nuovo componente sviluppato *ad-hoc* in questo lavoro, di cui si parlerà approfonditamente nel prossimo paragrafo: il *TimerBehaviour*. Per integrare questo nuovo *behaviour* nel sistema, è stato necessario fare in modo che il DSA fosse in grado di gestirne le funzionalità. Per questo nel *SetNumResultsBehaviour* è stato aggiunto il seguente codice che processa i messaggi provenienti dal *TimerBehaviour*:

```
… … …
if (result.getQuery().equals("TIMER_ELAPSED"))
{
 /*
  * If we arrive in this piece of code, it means that we have
  * 1 or more peers that don't give response. For this we
  * have to force the completion of the search.
  */

notresponding = (((DesktopSearchAgent) myAgent).numPeers –
                (results.size() + noresults.size()));

log.warn("TIMER_ELAPSED: " + notresponding +
         " response from peers not received!");

result.setAdditionalTerms(trueadditionalterms);
result.setQuery(truequery);
result.setQueryTerms(truequeryterms);

notresponding--;
}
… … …
```

Listato 4.7 Gestione del *TimerBehaviour* nel *SetNumResultsBehaviour*.

Dopo questa breve introduzione, risulta ora necessario spiegare meglio come si comporti e cosa faccia di preciso il nuovo *behaviour* implementato.

4.2.2.4 Il *TimerBehaviour*

Nell’ultima fase di modiche a RAIS è stato progettato ed implementato un *TimerBehaviour*, che permetta di ripristinare il sistema nel caso in cui cada la comunicazione nel bel mezzo di una ricerca. Questo *behaviour* è stato implementato dopo l’osservazione che nel vecchio prototipo di RAIS, qualora ci fossero problemi comunicativi tra le piattaforme, il sistema rimaneva in un’attesa perenne dei contatti da parte dei *peer*, degradando performance e tempi di risposta. Aggiungendo il fatto che in una rete *peer-to-peer* scenari di caduta e chiusura della comunicazione sono molto frequenti (un *peer* che si disconnette dalla rete, *firewall software* che bloccano le comunicazioni in ingresso/uscita, …), è comprensibile come mai si sia ritenuto necessario realizzare questo componente. Il funzionamento del *TimerBehaviour* è altrettanto semplice quanto funzionale.
Quando parte il lancio di una *query* da parte dell’utente, viene fatto partire un conto alla rovesci della durata desiderata. Se il conto alla rovescia termina, significa che la ricerca è ancora in corso, quindi molto probabilmente si stanno attendendo dei risultati da *peer* che non risponderanno mai. Per evitare un blocco del sistema, il *TimerBehaviour* invia un messaggio di *warning* al DSA stesso. All’arrivo di un messaggio dal *timer*, il DSA si comporta come se avesse ricevuto un messaggio NULL, solo che in questo particolare caso viene forzata la conclusione della ricerca. Questo perché se dopo *timeout* secondi (questo è il nome della variabile usata per impostare il tempo d’attesa massimo) il *timer* scatta, vuol dire che abbiamo uno o più *peer* che non hanno ancora risposto (probabile CONNECTION_FALL), allora si forza la conclusione della ricerca, visualizzando i risultati ottenuti sino a quel momento. Qualora tutti i *peer* non rispondessero, viene visualizzato un messaggio di *warning* consigliando di riavviare il sistema, poiché potrebbero essere occorsi dei problemi durante la registrazione dei *peer*, oppure più facilmente una non corretta de-registrazione dall’elenco dei servizi del JxtaDF da parte dell’LSA. Infatti la de-registrazione di un servizio dal JxtaDF non è ancora stata realizzata nella versione corrente di *jade-jxta*, ma si prevede che venga presto aggiunta ai metodi di registrazione e ricerca già implementati.

```
… … …
/**
 * Defines the behaviour that limit at "timeout" seconds the maximum waiting
 * time to receive response from peers.
 *
 * @author Michele Longari
 */
public class TimerBehaviour extends WakerBehaviour {

private static final long serialVersionUID = 1L;
private int searchid;

public TimerBehaviour(Agent a, long timeout, int id) {
      super(a, timeout);
      searchid = id;
}

protected void onWake() {
/*
 * After "timeout" seconds, IF there is a search in progress, AND IF the
 * search in progress and the timer-search are the same AND IF not all
 * the peers give a response: it means that the search is BLOCKED for
 * CONNECTION FALL!
 * For this reason the DSA "alerts" himself, and after aborts the
 * search.
 */
 if (((DesktopSearchAgent) myAgent).counter != -1)
  if (((DesktopSearchAgent) myAgent).idsearch == searchid)
   if (((DesktopSearchAgent) myAgent).counter <
       ((DesktopSearchAgent)myAgent).numPeers)
{
String query = "TIMER_ELAPSED";
String[] terms = query.split(" ");
FirstStepResult fsr = new FirstStepResult(query, terms, 0, null, null, 0);

try {
      ACLMessage TimerMsg = new ACLMessage(ACLMessage.INFORM);
      TimerMsg.setOntology(DesktopSearchAgent.MSG_QUERY_2S);
      TimerMsg.setContentObject(fsr);
      TimerMsg.addReceiver(myAgent.getAID());
      myAgent.send(TimerMsg);
… … …
```

Listato 4.8 Il nucleo operativo del *TimerBehaviour*.

L'unico problema che si è dovuti risolvere durante l'implementazione del *TimerBehaviour* è stato quello che il *timer* avvisava correttamente il DSA, però la sua *query* d'avviso "TIMER_ELAPSED" veniva ritenuta una *query* valida dall'agente e quindi venivano corrotti i risultati finali. La soluzione trovata è stata quella di tenere in memoria la *query* originale, e rimpiazzarla subito dopo l'aver ricevuto il segnale di allerta del *timer*: così facendo veniva correttamente interpretato il messaggio di *warning* del *timer*, evitando però di andare a creare problemi nella visualizzazione dei risultati ottenuti sino a quell'istante.

4.3 Risultati ottenuti

Alla fine di questa prima parte quello che si è ottenuto è stato un sistema funzionante ed operativo.

Per ottenere maggiore chiarezza ed ordine è stato deciso di riunire in un un'unica libreria (*jade-jxta.jar*) tutta la parte progettuale comprendente il nucleo operativo del nuovo sub-strato introdotto. In pratica al progetto RAIS è stata aggiunta come libreria *jade-jxta.jar*, che permette al sistema di effettuare il *boot* utilizzando come supporto di rete i servizi *peer-to-peer* di JXTA e del suo MTP. A livello pratico quest'operazione non ha portato niente di nuovo, però a livello concettuale ha portato un netto miglioramento nella gestione del codice e delle dipendenze.

Una volta effettuato lo *startup* del sistema con gli opportuni parametri che permettessero di far partire sia la rete *peer-to-peer* di JXTA che il MAS, quello che si otteneva era una piattaforma JADE registrata ad uno specifico nodo della rete ed un agente grafico (il DSA) pronto per processare le query dell'utente.
La figura 4.4 rappresenta uno *screenshot* della piattaforma JADE una volta effettuatone il nuovo *boot*. Come è possibile osservare l'indirizzo dei vari agenti corrisponde a quello specifico del nodo della rete JXTA:

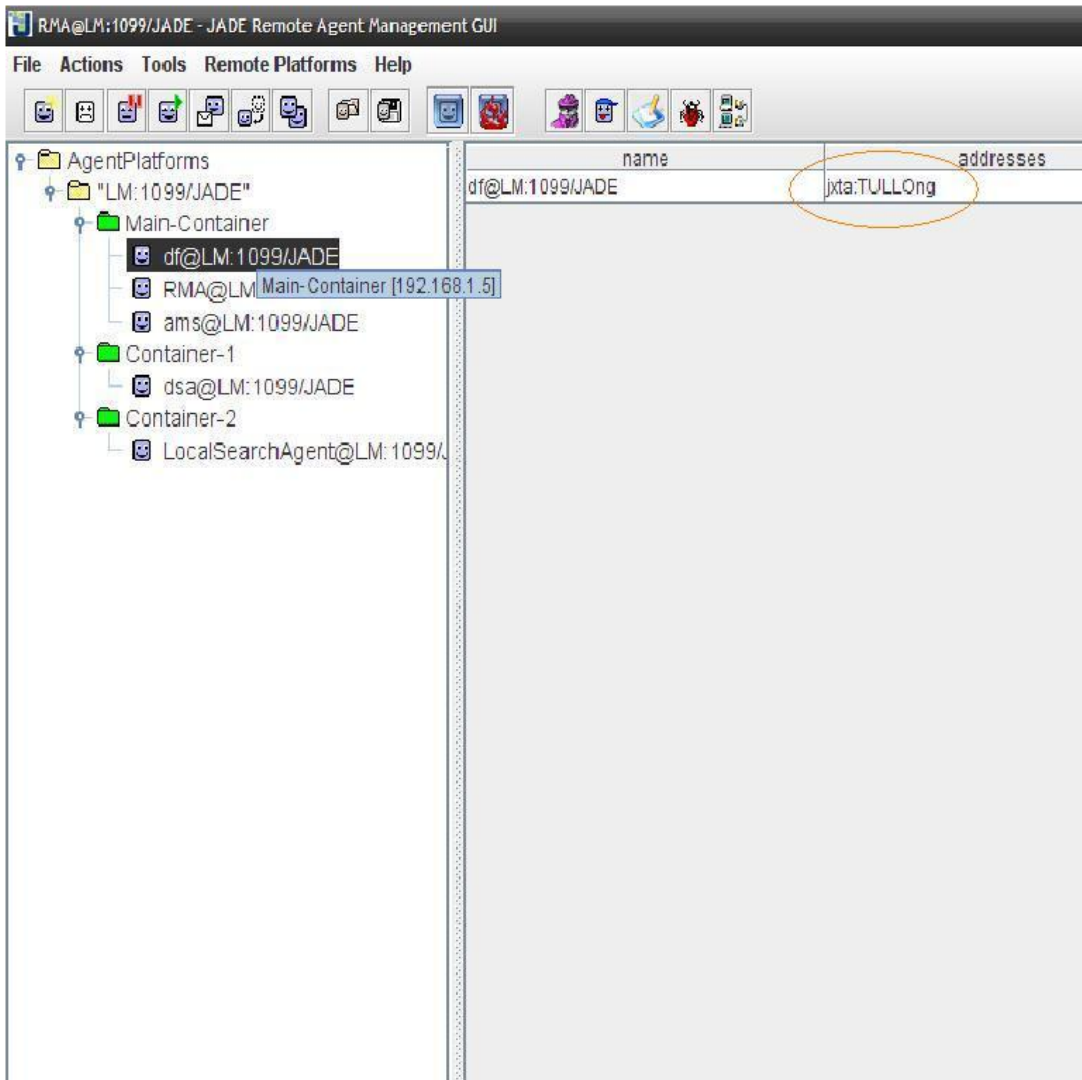

Figura 4.4 La nuova piattaforma JADE-JXTA-RAIS.

Da un punto di vista computazionale si sono fatti vari test usando solo agenti locali, agenti locali e remoti e solo agenti remoti. Tutti test sono andati a buon fine. Come è ovvio le ricerche in locale mostravano dei tempi di risposta nettamente migliori rispetto a quelle in remoto, anche se in condizioni di traffico di rete soddisfacenti RAIS ha mostrato una velocità nel processare le richieste dell'utente più che sufficiente. La selezione di una *query*, l'invio della *query* a tutti gli agenti di ricerca locale presenti sulla rete, la ricerca *Desktop Search* ed infine la risposta al DSA richiedente sono state operazioni che nelle varie fasi di *testing* effettuate hanno richiesto circa dai 7 ai 25 secondi, in base alle macchine usate, al traffico di rete e alla mole di risultati ottenuti.

Nelle figure 4.5 e 4.6 è possibile vedere il funzionamento del sistema in due casi operativi testati: nel primo caso è stato fatta partire una ricerca solo sul *peer* remoto, senza utilizzare l’LSA personale; nel secondo caso invece la ricerca è stata effettuata sia sull’LSA del *peer* remoto che su quello locale del *peer* richiedente:

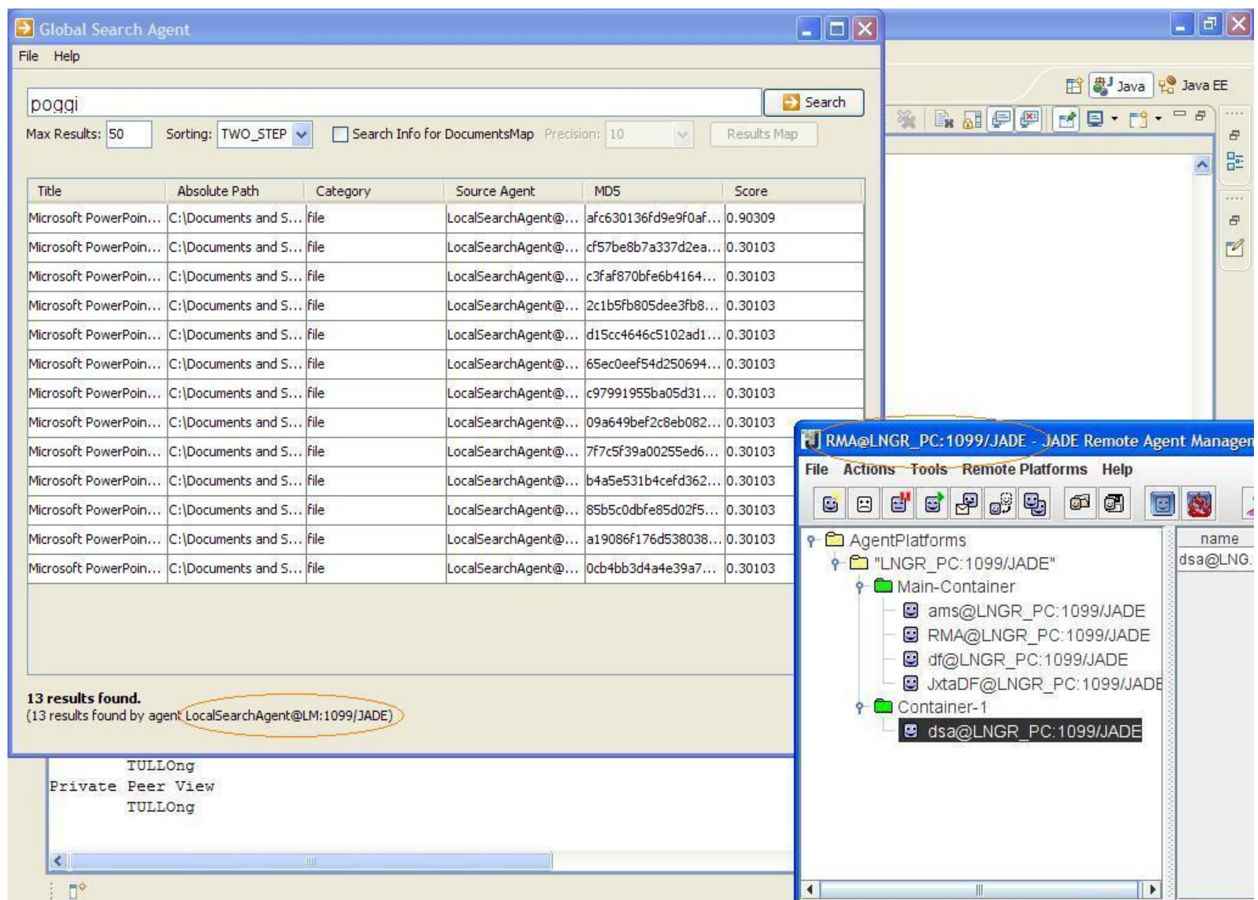

Figura 4.5 *Testing* di RAIS senza LSA locale.

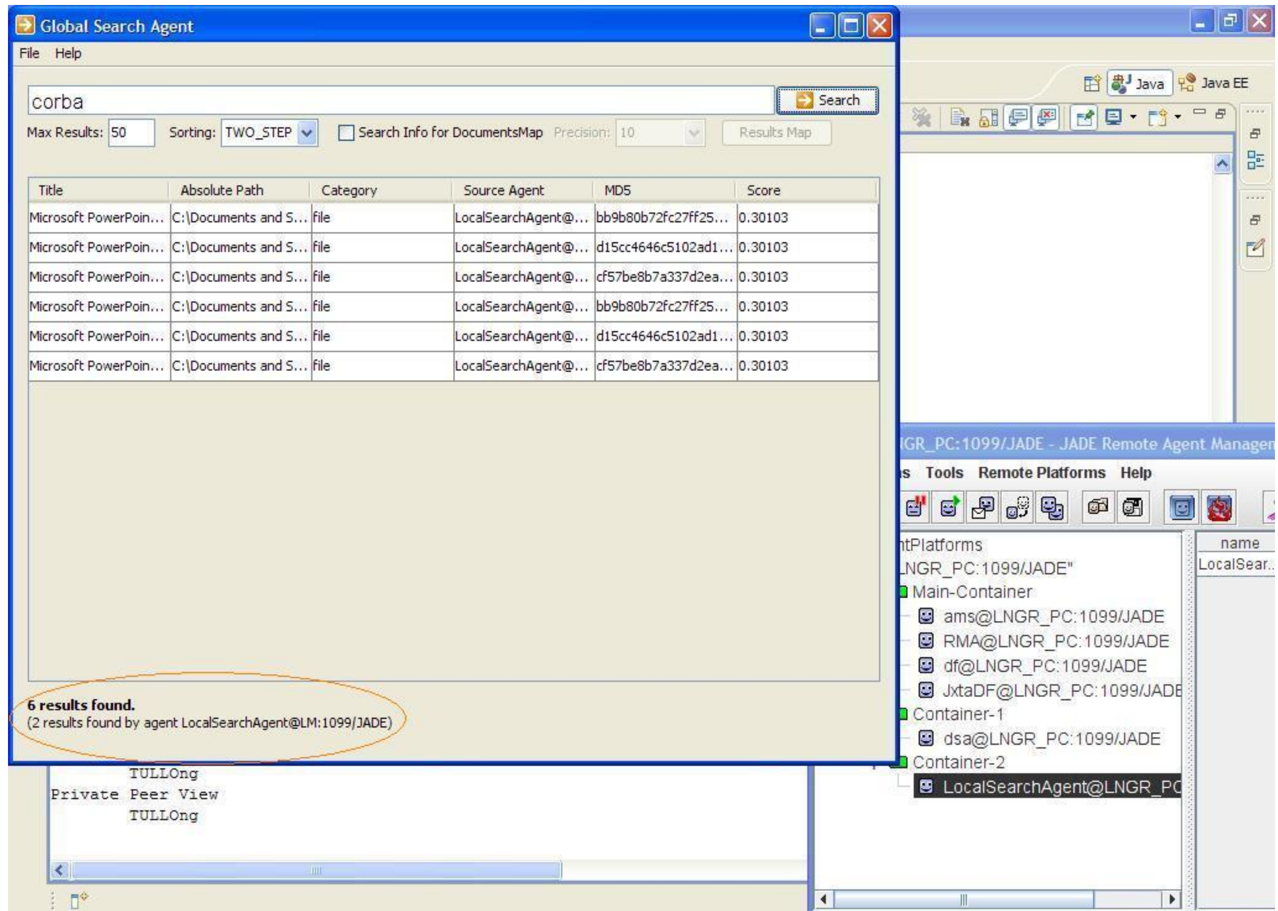

Figura 4.6 Testing di RAIS con LSA locale.

Come si può osservare dalle figure, nel primo caso operativo, sulla piattaforma JADE sono presenti tutti gli agenti eccetto il *LocalSearchAgent* personale (non fatto partire di proposito), quindi la ricerca viene processata dai soli agenti di ricerca presenti sulla rete (in questo caso vi è solo LocalSearchAgent@LM:1099/JADE) che provvederanno a rispondere al DSA che ha fatto la richiesta (ovvero dsa@LNGR_PC:1099/JADE).

Nel secondo caso invece, essendo stato fatto partire anche l'LSA personale dell'utente "ricercatore", la richiesta viene processata sia dall'agente di ricerca locale (che come si può notare ha fornito 4 risultati su 6 totali), sia da quelli remoti presenti sulla rete *peer-to-peer* (che hanno risposto con altri 2 risultati compatibili con la *query*).

Capitolo 5

L'architettura RAIS

Questo capitolo inizierà descrivendo in maniera generale l'architettura astratta per sistemi peer-to-peer dalla quale si è partiti, per poi concentrarsi a spiegare le varie fasi progettuali che hanno permesso la realizzazione della seconda ed ultima parte di questo lavoro: l'architettura RAIS.

5.1 Architettura astratta per sistemi *peer-to-peer*

L'architettura astratta che si utilizzata come fondamenta di questo lavoro, fa parte di un progetto accademico realizzato presso i laboratori AOT dell'Università degli Studi di Parma. In particolare il prototipo da noi utilizzato è stato presentato come Tesi di Laurea Specialistica dal dott. Luca Gatti ("*Sviluppo di una architettura astratta per la realizzazione di applicazioni peer-to-peer"*, dott. L. Gatti, dicembre 2007).

Lo scopo di questo lavoro era stato quello di progettare e successivamente implementare una architettura "astratta" per lo sviluppo di sistemi *peer-to-peer*. L'architettura da sviluppare doveva fornire funzionalità di alto livello che

semplificassero la realizzazione di applicazioni *peer-to-peer* sia in termini di tempo necessario allo sviluppo, sia in termini di complessità del *software* e di conoscenze tecniche necessarie per lo sviluppatore.

L’aspetto più importante di questo progetto riguarda l’astrazione che l’architettura deve fornire: essa deve essere sufficientemente generica e flessibile da rendere possibili diverse implementazioni, le quali possono essere realizzate utilizzando sistemi anche molto diversi tra loro.

L’obiettivo principale del progetto del dott. Gatti è stato infatti quello di fornire funzionalità di alto livello per facilitare lo sviluppo di applicazioni *peer-to-peer*, non vincolando tali applicazioni al sistema utilizzato per l’implementazione dell’architettura: ciò consente allo sviluppatore di poter scegliere l’implementazione più opportuna in base ai requisiti ed ai vincoli di progetto, sfruttando inoltre anche la possibilità di testare le prestazioni delle varie implementazioni disponibili.

Un obiettivo secondario invece consisteva nella possibilità di modificare facilmente la scelta dell’implementazione della propria applicazione, permettendo così di fronteggiare i problemi di cui attualmente soffrono alcuni sistemi *peer-to-peer Open Source* come per esempio la lentezza nello sviluppo e nel rilascio di nuove versioni del *software*, *bugs* ed errori del *software* oppure la mancanza di supporto nel caso di abbandono del progetto.

Infine un ultimo aspetto molto importante del progetto del dott. Gatti riguarda la possibilità di utilizzare l’architettura su dispositivi *hardware* eterogenei: ciò pone limitazioni soprattutto nella scelta dei sistemi utilizzabili come implementazione, i quali devono supportare dispositivi *hardware* anche molto diversi tra loro ed inoltre fornire prestazioni sufficientemente buone e scalabili, per adattarsi anche a macchine con ridotta capacità computazionale.

Gli attuali sistemi *peer-to-peer* possiedono caratteristiche anche molto diverse tra loro, ma presentano alcuni aspetti comuni:

1. un sistema *peer-to-peer* opera su di una rete “astratta” (virtuale), detta anche *overlay network*, costituita da nodi, ciascuno dei quali è detto *peer*;
2. deve essere presente un meccanismo di identificazione delle risorse che costituiscono il sistema *peer-to-peer*;
3. su ciascun dispositivo fisico possono essere contemporaneamente attivi anche multipli *peers*;
4. *peers* con caratteristiche comuni possono formare gruppi di peers (*peer groups*), i quali possono prevedere meccanismi di accesso sicuro o aperto;
5. i *peers* possono comunicare tra di loro attraverso lo scambio di messaggi;
6. tipicamente la comunicazione tra i *peers* avviene attraverso canali logici (virtuali), anziché attraverso l’invio esplicito di messaggi ad altri *peers*;
7. la rete *peer-to-peer* fornisce servizi di *discovery*, tramite i quali è possibile conoscere dinamicamente le risorse disponibili nella rete ed in seguito interagire con esse;
8. la rete *peer-to-peer* fornisce servizi di *publishing*, tramite i quali è possibile notificare la rete della presenza di risorse.

Basandosi sulle caratteristiche comuni individuate, nel progetto del dott. Gatti si è scomposta l’architettura in sotto-sistemi, ciascuno dei quali implementa uno specifico servizio.
Si sono dunque individuati i seguenti sotto-sistemi:

1. *Platform*: rappresenta la piattaforma di rete (*network platform*), la quale fornisce tutti i principali servizi di rete del sistema *peer-to-peer*;
2. *Id*: rappresenta il sistema di creazione e gestione degli identificativi delle risorse presenti nella rete *peer-to-peer*;
3. *Peer*: permette la creazione di *peers* e la gestione del loro ciclo di vita;

4. *Channel*: permette la creazione e gestione dei canali logici di comunicazione;
5. *PeergGroup*: permette la creazione e gestione dei *peer groups*;
6. *Message*: permette la creazione e gestione dei messaggi che i *peer* possono scambiarsi attraverso i canali logici;
7. *Discovery*: implementa i servizi di *discovery* delle risorse della rete *peer-to-peer*.

Tutta l'architettura è basata sul pattern *AbstractFactory*: ogni sottosistema definisce una interfaccia *factory* ed alcune interfacce *product*, mentre sarà compito delle implementazioni fornire una *concrete factory* ed i rispettivi *concrete products*.

Per ciascun sottosistema il dott. Gatti ha creato uno specifico *package java*, a partire dal *package* radice *it.unipr.aot.p2p*.
Pertanto sono presenti i seguenti *package*:

1. *it.unipr.aot.p2p.platform*;
2. *it.unipr.aot.p2p.id*;
3. *it.unipr.aot.p2p.peer*;
4. *it.unipr.aot.p2p.channel*;
5. *it.unipr.aot.p2p.peergroup*;
6. *it.unipr.aot.p2p.message*;
7. *it.unipr.aot.p2p.discovery*.

Alla descrizione completa di ogni *package* è dedicata la successiva sezione, la quale ne descrive la struttura e i servizi.

Per quanto riguarda le implementazioni, Gatti ha scelto la convenzione di creare, all'interno di ogni *package*, un *sotto-package* con il nome dell'implementazione. Pertanto, per esempio, al *package it.unipr.aot.p2p.id* corrispondono i *package* di implementazione *it.unipr.aot.p2p.jade* e *it.unipr.aot.p2p.jxta* che contengono rispettivamente le implementazioni basate su JADE e su JXTA.

Nello specifico del progetto realizzato in questa occasione, l'implementazione sulla quale si è andati a lavorare è esclusivamente quella JADE, poiché RAIS a tutti gli effetti nel suo nucleo operativo principale è un MAS. Quindi si può dire che JADE è usato come *framework* operativo, mentre JXTA solo come sub-strato comunicativo.

Dell'implementazione JADE sviluppata dal dott. Gatti, se ne parlerà nel dettaglio nella sezione 5.1.2.

5.1.1 Sotto-sistemi funzionali

In questa sezione analizzeremo e studieremo nel dettaglio i sette diversi sotto-sistemi funzionali realizzati dal dott. Gatti, che presi singolarmente non hanno valenza operativa, ma sfruttati in maniera corale permetto all'intera architettura di funzionare in maniera lineare ed efficiente.
Essi sono come già accennato nella sezione precedente: ***Platform***, ***Id***, ***Peer***, ***Channel***, ***Peer Group***, ***Message*** e ***Discovery***.
Sviluppiamoli uno ad uno.

Il **sotto-sistema *Platform*** costituisce il cuore dell'architettura: tramite esso è possibile istanziare tutte le *factory* concrete degli altri sotto-sistemi, nonché configurare ed attivare la piattaforma di rete dello specifico sistema *peer-to-peer* scelto come implementazione.
La figura 5.1 mostra il diagramma delle classi del *package* *it.unipr.aot.p2p.platform*:

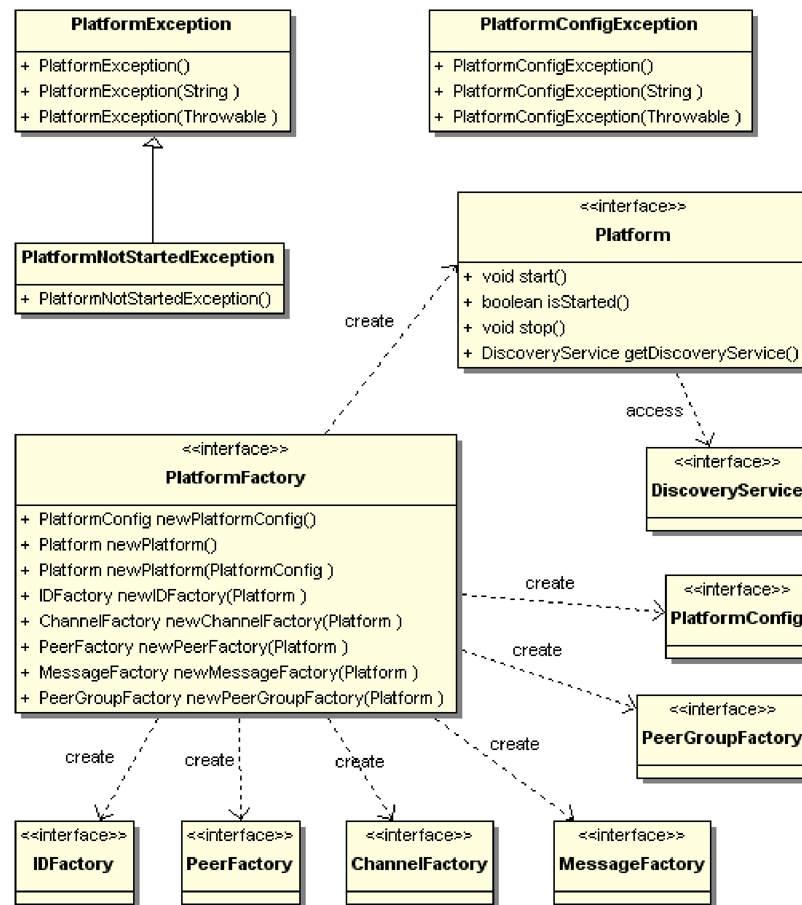

Figura 5.1 Diagramma delle classi del *package it.unipr.aot.p2p.platform*.

La componente centrale del sotto-sistema è l'interfaccia *PlatformFactory*: essa utilizza il pattern *AbstractFactory* per fornire un punto di accesso all'intera architettura. Creando un oggetto di una classe che implementa tale interfaccia è possibile istanziare una opportuna implementazione dell'architettura.

Le altre principali componenti del sotto-sistema sono le interfacce *Platform* e *PlatformConfig*.

L'interfaccia *Platform* rappresenta una *network platform* per il sistema *peer-to-peer*, ovvero un nodo della rete virtuale del sistema. La piattaforma di rete è la componente principale di un sistema *peer-to-peer*: le implementazioni di questa interfaccia consentono tipicamente di creare risorse del sistema (*peers*, canali, *peergroups*, …) e di effettuare la *discovery* di risorse remote. L'unico servizio messo direttamente a disposizione dello sviluppatore dall'interfaccia *Platform* è

l'accesso al servizio di *discovery*, il quale è costituito da una implementazione dell'interfaccia *DiscoveryService*.

L'interfaccia *PlatformConfig* rappresenta invece una generica configurazione per una piattaforma di rete: le implementazioni di tale interfaccia sono specifiche per il sistema *peer-to-peer* utilizzato e consentono di fornire al sistema tutte le informazioni necessarie per effettuare la fase di *bootstrap* della piattaforma.
Attualmente l'interfaccia *PlatformConfig* è una interfaccia vuota.
L'utilizzo tipico del sottosistema Platform è il seguente:

1. creazione di una opportuna istanza che implementa l'interfaccia *PlatformFactory*;
2. tramite l'oggetto creato, istanziazione e configurazione di un oggetto della classe *PlatformConfig*;
3. creazione di una istanza della classe *Platform*, passando come parametro l'oggetto di configurazione della piattaforma. In alternativa è possibile saltare la fase di creazione dell'oggetto di configurazione della piattaforma, richiedendo direttamente la creazione di oggetto di tipo *Platform* con una configurazione di *default*.

In generale il sotto-sistema *Platform* è l'unico per il quale si devono settare parametri specifici per il sistema *peer-to-peer* scelto come implementazione. Sebbene sia possibile inserire *settings* specifici anche in altri contesti, tipicamente ciò non è mai necessario: questo rende possibile modificare l'implementazione dell'architettura scelta semplicemente cambiando poche righe di codice o, ancora meglio, modificando poche righe di un file di configurazione in formato XML.
Tuttavia l'architettura lascia allo sviluppatore la libertà, nel caso in cui sia necessario, di fornire parametri specifici per l'implementazione scelta: ciò naturalmente porta ad una minore riusabilità del codice, e di conseguenza ad un maggiore sforzo necessario alla modifica del sistema *peer-to-peer* scelto come implementazione.
Infine il sottosistema definisce alcune classi di utilità, le quali rappresentano eccezioni:

1. *PlatformException*: rappresenta una generica eccezione generata dalla piattaforma di rete;
2. *PlatformNotStartedException*: rappresenta una eccezione dovuta al tentato utilizzo di un servizio della piattaforma, mentre questa non è stata ancora attivata;
3. *PlatformConfigException*: rappresenta un errore dovuto alla configurazione della piattaforma di rete come, ad esempio, un parametro errato.

Il **sotto-sistema *Id*** implementa tutte le funzionalità di generazione degli identificativi associati alle risorse della rete *peer-to-peer*.
Si deve innanzitutto notare che ogni specifico sistema *peer-to-peer* implementa un proprio formato per tali identificativi, nonché un opportuno meccanismo di generazione degli stessi: per questi motivi la modifica del sistema *peer-to-peer* utilizzato come implementazione può comportare una modifica di tutti gli identificativi generati tramite l’architettura astratta, senza che sia stato modificato alcun parametro relativo ad essi.

Il diagramma delle classi del *package it.unipr.aot.p2p.id* è mostrato in figura 5.2:

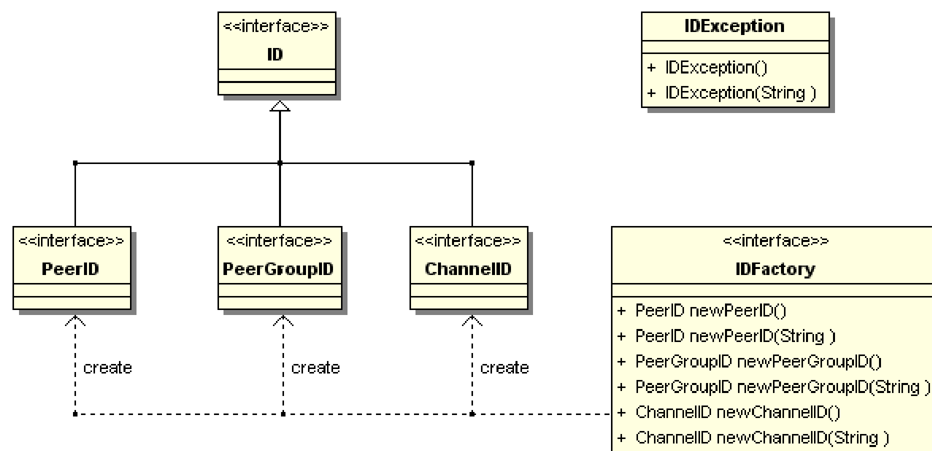

Figura 5.2 Diagramma delle classi del *package it.unipr.aot.p2p.id*.

La componente principale del sotto-sistema è l'interfaccia *IDFactory*, la quale utilizza il *pattern abstract factory* per implementare un meccanismo astratto di generazione di identificativi per le risorse del sistema *peer-to-peer*. L'interfaccia *IDFactory* mette a disposizione metodi per la creazione di identificativi di ogni tipologia di risorsa: identificativi di *peers*, canali e *peer groups*. Per ogni tipo di risorsa, la *factory* mette a disposizione due metodi di creazione di identificativi: un metodo privo di argomenti, il quale genera un identificativo pseudo-casuale, ed un altro metodo che richiede un parametro di tipo stringa e genera un identificativo in modo deterministico in base ad esso.
Come già detto in precedenza, il formato degli identificativi può variare notevolmente in base al sistema *peer-to-peer* utilizzato come implementazione.
Il sotto-sistema *Id* definisce, per ogni tipologia di risorsa, una interfaccia che ne rappresenta l'identificativo: *PeerID* per l'identificativo di un *peer*, *ChannelID* per quello di un canale e *PeerGroupID* per quello di un gruppo di *peers*. Tutte le interfacce sono vuote ed ereditano dall'interfaccia *ID*: anch'essa è un'interfaccia vuota, la quale rappresenta un generico identificativo per una risorsa del sistema *peer-to-peer* astratto.
E' infine presente la classe *IDException*, la quale rappresenta una generica eccezione causata da un identificativo di risorsa del sistema *peer-to-peer*.

Il **sotto-sistema *Peer*** permette la creazione, la configurazione e la gestione del ciclo di vita dei *peers* che popolano il sistema.
La figura 5.3 mostra il diagramma delle classi del *package it.unipr.aot.p2p.peer*:

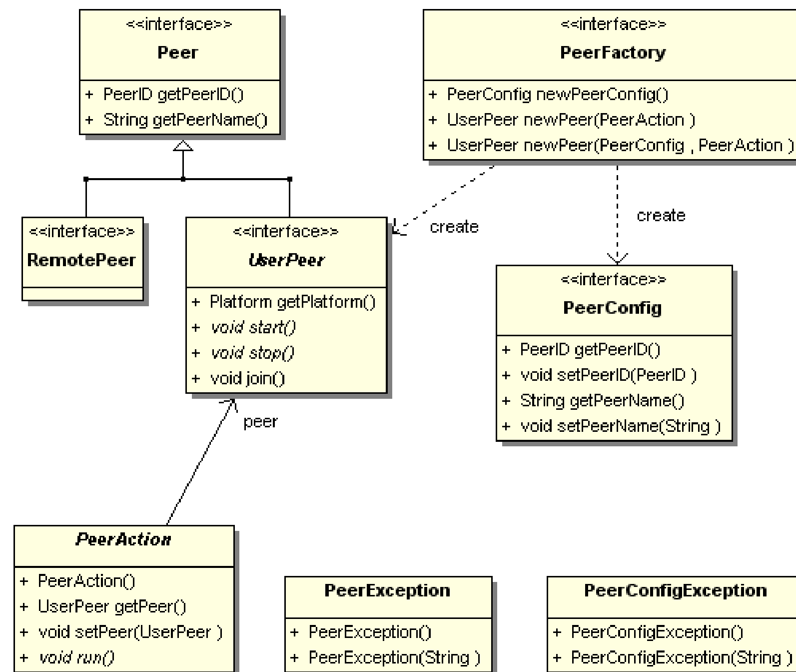

Figura 5.3 Diagramma delle classi del *package it.unipr.aot.p2p.peer*.

La componente principale del sotto-sistema è l’interfaccia *PeerFactory*, la quale utilizza ancora una volta il pattern *AbstractFactory* per permettere la creazione di risorse di tipo *peer*, oltre ad oggetti di configurazione per i *peer* stessi.

L’interfaccia *Peer* rappresenta una generica risorsa di tipo *peer* per il sistema. Le interfacce *UserPeer* e *RemotePeer* ereditano dall’interfaccia *Peer*: esse rappresentano rispettivamente un *peer* a livello utente (*peer* locale) ed un *peer* remoto.

L’interfaccia *PeerConfig* rappresenta una generica configurazione per una risorsa di tipo peer a livello utente. Le implementazioni concrete dell’interfaccia *PeerConfig* possono includere parametri di configurazione specifici per il sistema *peer-to-peer* utilizzato: l’utente può scegliere di attenersi all’interfaccia base per facilitare la ricusabilità del codice, oppure utilizzare l’interfaccia di configurazione specializzata, in modo da poter configurare le risorse *peer* con maggiore precisione.

La classe astratta *PeerAction* rappresenta l’azione da eseguire per un *peer* a livello utente. Nel caso generale di utilizzo, per creare un *peer* l’utente dovrà definire una

propria classe concreta che estende *PeerAction* e definisce il ciclo di esecuzione del *peer* stesso. In alternativa l’utente può riutilizzare una classe già esistente, la quale estende la classe *PeerAction*.

L’utilizzo tipico del sotto-sistema *Peer* è dunque il seguente:

1. creazione di una opportuna istanza che implementa l’interfaccia *PeerFactory*;
2. tramite l’oggetto creato, istanziazione e configurazione di un oggetto che implementa l’interfaccia *PeerConfig*;
3. creazione di una istanza della classe *PeerAction*;
4. creazione di una istanza della classe *UserPeer*, passando come parametro l’oggetto di configurazione del *peer* utente e l’istanza della classe *PeerAction*. In alternativa è possibile saltare la fase di creazione dell’oggetto di configurazione del *peer*, richiedendo direttamente la creazione di un oggetto di tipo *UserPeer* con una configurazione di *default*, fornendo sempre l’oggetto *PeerAction* come parametro.

Infine sono definite due classi che rappresentano eccezioni:

1. *PeerException*: rappresenta una generica eccezione dovuta ad un oggetto di tipo *peer*;
2. *PeerConfigException*: rappresenta un’eccezione dovuta ad un errore nella configurazione di un *peer*.

Il **sotto-sistema *Channel*** permette la creazione, la configurazione e la gestione del ciclo di vita dei canali logici del sistema *peer-to-peer*.

La figura 5.4 mostra il diagramma delle classi del *package* *it.unipr.aot.p2p.channel*:

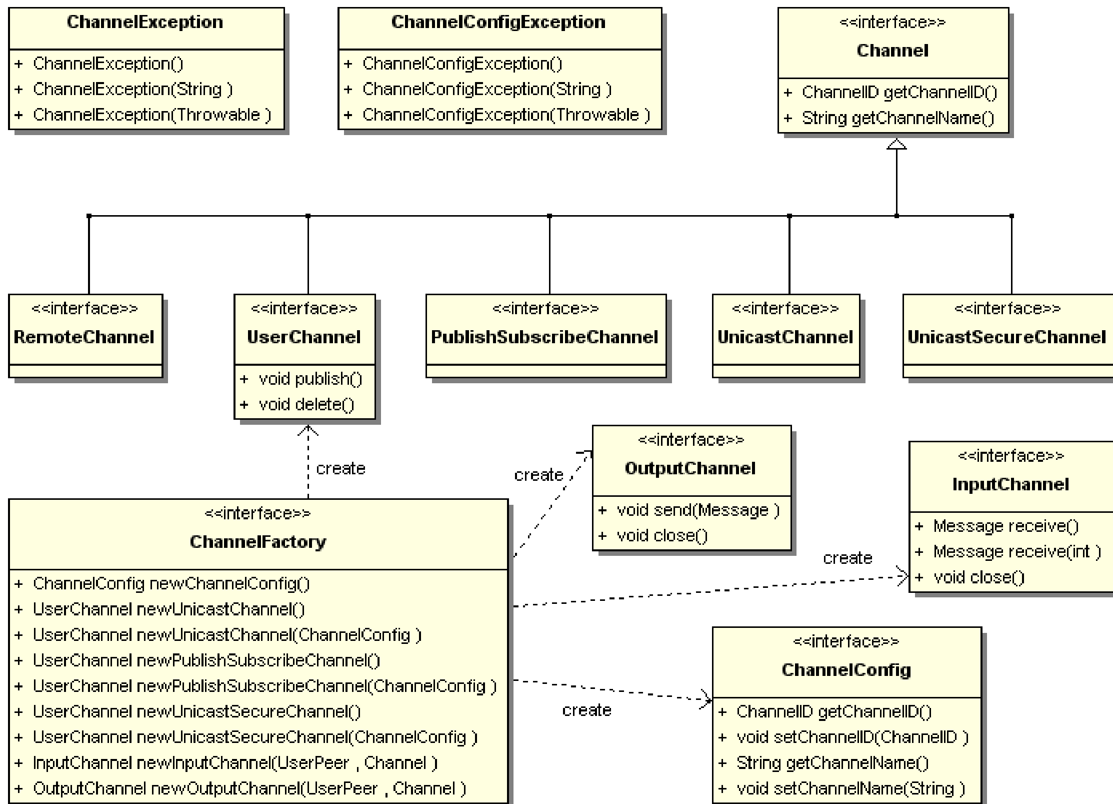

Figura 5.4 Diagramma delle classi del *package it.unipr.aot.p2p.channel*.

La componente principale del sotto-sistema è l’interfaccia *ChannelFactory*, la quale permettere la creazione di risorse di tipo *channel*, ovvero canali virtuali di comunicazione del sistema *peer-to-peer*, oltre ad oggetti di configurazione ed *endpoint* di *input* o di *output* per i canali stessi. L’interfaccia *ChannelFactory* è basata sul *pattern abstract factory*.

L’interfaccia *Channel* rappresenta una generica risorsa di tipo canale per il sistema. Le interfacce *UserChannel* e *RemoteChannel* ereditano dall’interfaccia *Channel* e rappresentano rispettivamente un canale a livello utente e un canale remoto. Una risorsa di tipo *channel* rappresenta soltanto un canale logico, per poter inviare o ricevere messaggi dal canale è necessario associarvi un *endpoint*. Le interfacce *InputChannel* ed *OutputChannel* rappresentano le due diverse tipologie di *endpoint* che è possibile associare ad un canale: una istanza della classe *InputChannel* rappresenta un *input endpoint* per un canale, attraverso il

quale è possibile ricevere messaggi. Una istanza della classe *OutputChannel* rappresenta invece un *output endpoint*, attraverso cui è possibile inviare messaggi sul canale. I canali logici dell'architettura astratta implementano sempre una comunicazione di tipo unidirezionale, nella quale i messaggi fluiscono dall'*output endpoint* che invia il messaggio a tutti gli *input endpoints* associati a tale canale.

Le differenti tipologie di canale sono individuate dal tipo di *endpoint* che è possibile associarvi e dalle caratteristiche della comunicazione:

1. un canale di tipo *unicast* rappresenta un canale punto-punto, ovvero un canale per il quale sono ammessi al più un *endpoint* di *input* ed un *endpoint* di *output*. In un canale di tipo *unicast* non vi è alcuna garanzia di ricezione né di integrità dei messaggi;
2. un canale di tipo *publish-subscribe* implementa il *pattern* di comunicazione *publish-subscribe*, ovvero una tipologia di canale molti a molti, nel quale ogni messaggio inviato da un *output endpoint* è ricevuto da tutti gli *input endpoint* associati a tale canale. Anche per un canale di tipo *publish-subscribe* non vi è alcuna garanzia di ricezione né di integrità dei messaggi;
3. un canale di tipo *unicast secure* rappresenta un canale punto-punto sicuro, nel quale sono ammessi al più un *endpoint* di *input* ed un *endpoint* di *output*. Un canale di tipo *unicast secure* offre garanzia di ricezione, di integrità dei messaggi ed inoltre di confidenzialità della comunicazione.

Per ogni tipologia di canale è stata creata una specifica interfaccia che eredita dall'interfaccia *UserChannel*:

1. *UnicastChannel*: rappresenta un canale di tipo *unicast*;
2. *PublishSubscribeChannel*: rappresenta un canale di tipo *publish-subscribe*;
3. *UnicastSecureChannel*: rappresenta un canale di tipo *unicast secure*.

L'interfaccia *ChannelConfig* rappresenta una configurazione per una risorsa di tipo canale a livello utente: le implementazioni concrete dell'interfaccia possono

includere parametri specifici per il sistema *peer-to-peer* utilizzato come implementazione, attraverso cui è possibile configurare le risorse di tipo *channel* con maggiore precisione, ovviamente a discapito della riusabilità del codice.
L'utilizzo tipico del sotto-sistema *Channel* è dunque il seguente:

1. creazione di una opportuna istanza che implementa l'interfaccia *ChannelFactory*;
2. tramite l'oggetto creato, istanziazione e configurazione di un oggetto che implementa l'interfaccia *ChannelConfig*;
3. creazione di una istanza della classe *UserChannel*, passando come parametro l'oggetto di configurazione del canale utente. In alternativa è possibile saltare la fase di creazione dell'oggetto di configurazione della piattaforma, richiedendo direttamente la creazione di un oggetto di tipo *UserChannel* con una configurazione di *default*;
4. creazione di un *endpoint* (di *input* o di *output*) per il canale.

Infine sono definite due classi che rappresentano eccezioni:

1. *ChannelException*: rappresenta una generica eccezione dovuta ad un errore di una risorsa di tipo canale;
2. *ChannelConfigException*: rappresenta una eccezione causata da un errore nella configurazione di un canale.

Il **sotto-sistema *Peer Group*** permette la creazione e la configurazione dei gruppi di *peers*.
La figura 5.5 mostra il diagramma delle classi del *package* *it.unipr.aot.p2p.peergroup*:

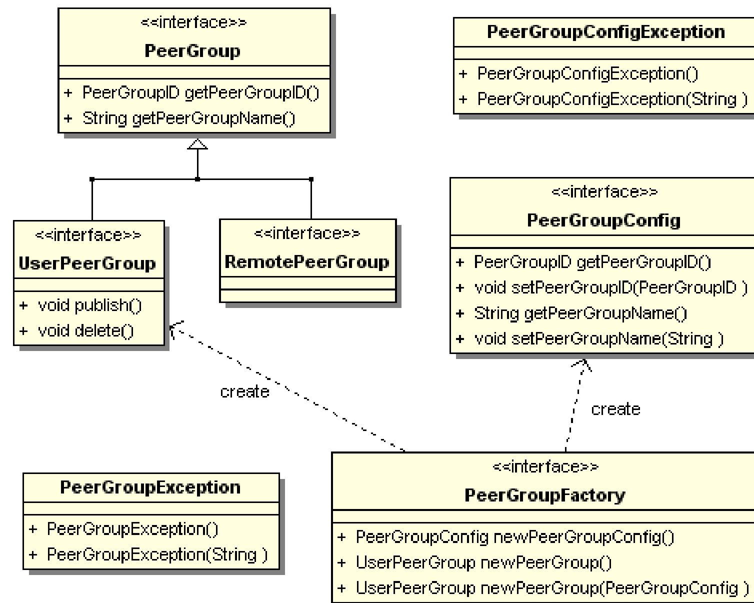

Figura 5.5 Diagramma delle classi del *package it.unipr.aot.p2p.peergroup*.

La componente principale del sottosistema è l'interfaccia *PeerGroupFactory*, la quale si basa sul *pattern AbstractFactory* per permettere la creazione di *peer groups*, oltre ad oggetti di configurazione per i *peer groups* stessi.

L'interfaccia *PeerGroup* rappresenta un generico gruppo di *peer* per il sistema. Le interfacce *UserPeerGroup* e *RemotePeerGroup* ereditano dall'interfaccia *PeerGroup*: esse rappresentano rispettivamente un *peer group* a livello utente (*peer group* locale) ed un *peer group* remoto.

L'interfaccia *PeerGroupConfig* rappresenta una configurazione per un gruppo di *peers* a livello utente. Le implementazioni concrete dell'interfaccia *PeerGroupConfig* possono includere parametri di configurazione specifici per il sistema *peer-to-peer* utilizzato: l'utente può scegliere di attenersi all'interfaccia base per facilitare la riusabilità del codice, oppure utilizzare l'interfaccia di configurazione specializzata, in modo da poter configurare i gruppi di *peers* con maggiore precisione.

L'utilizzo tipico del sotto-sistema *Peer Group* è dunque il seguente:

1. creazione di una opportuna istanza che implementa l'interfaccia *PeerGroupFactory*;

2. tramite l’oggetto creato, istanziazione e configurazione di un oggetto che implementa l’interfaccia *PeerGroupConfig*;
3. creazione di una istanza della classe *UserPeerGroup*, passando come parametro l’oggetto di configurazione del *peer group* utente. In alternativa è possibile saltare la fase di creazione dell’oggetto di configurazione della piattaforma, richiedendo direttamente la creazione di un oggetto di tipo *UserPeerGroup* con una configurazione di *default*.

Infine sono definite due classi che rappresentano eccezioni:

1. *PeerGroupException*: rappresenta una generica eccezione dovuta ad un errore di un oggetto di tipo *peer group*;
2. *PeerGroupConfigException*: rappresenta una eccezione dovuta ad un errore nella configurazione di un *peer group*.

Il **sotto-sistema *Message*** permette la creazione di messaggi, utilizzati per la comunicazione attraverso i canali virtuali del sistema.
La figura 5.6 mostra il diagramma delle classi del *package* *it.unipr.aot.p2p.message*:

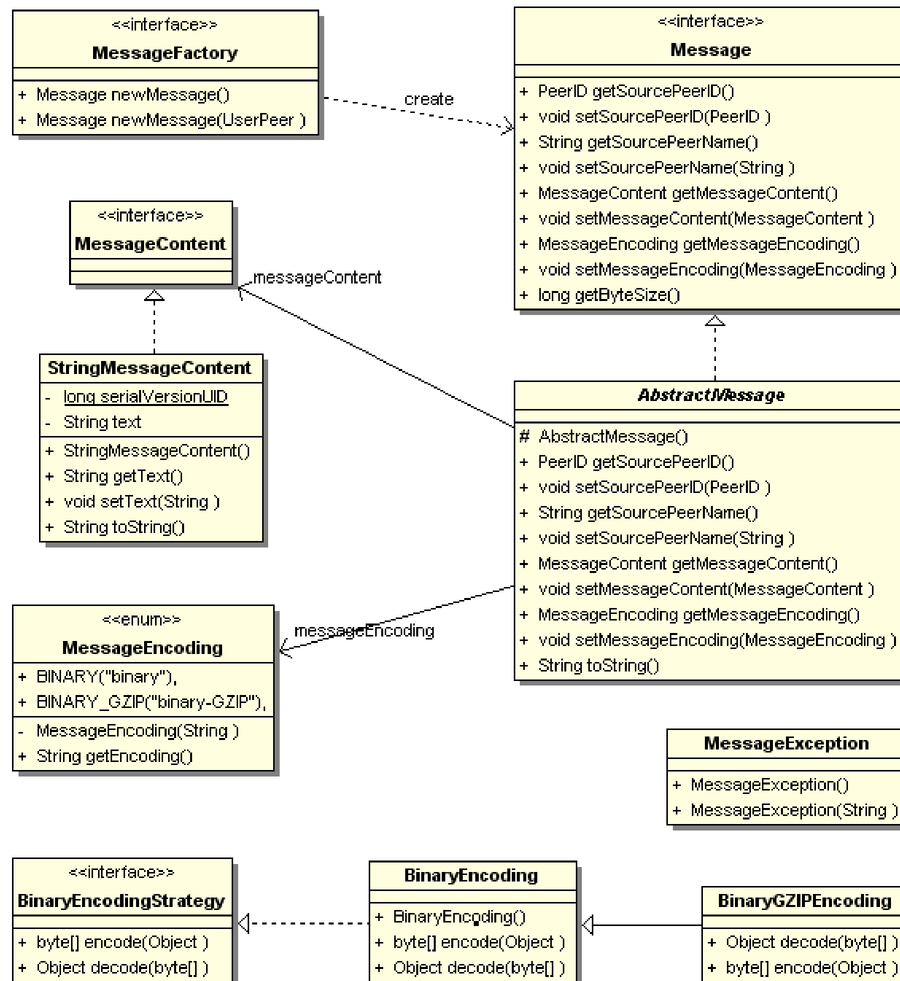

Figura 5.6 Diagramma delle classi del *package it.unipr.aot.p2p.message*.

La componente principale del sotto-sistema è l’interfaccia *MessageFactory*, la quale, basata sul *pattern AbstractFactory*, permette la creazione di nuovi messaggi.

L’interfaccia *Message* rappresenta un generico messaggio usato per la comunicazione attraverso i canali logici del sistema: un messaggio è caratterizzato da un contenuto, definito dall’interfaccia *MessageContent*, e da alcune informazioni di intestazione come l’identificativo e il nome simbolico del *peer* sorgente, il tipo di codifica del contenuto. La classe *StringMessageContent* implementa l’interfaccia *MessageContent* e rappresenta un contenuto di tipo testuale per un messaggio.

La classe di enumerazione *MessageEncoding* rappresenta le possibili tipologie di codifica del contenuto di un messaggio supportate dall’architettura *peer-to-peer* astratta.
Un'altra importante componente del *package message* è la classe *BinaryEncodingStrategy*, la quale si basa sul *pattern Strategy* e definisce una interfaccia che rappresenta una strategia di codifica e decodifica binarie: tale interfaccia è implementata dalle classi *BinaryEncoding* e *BinaryGZIPEncoding*.
La classe *BinaryEncoding* implementa la strategia di codifica e decodifica binaria attraverso una semplice serializzazione degli oggetti *Java*.
La classe *BinaryGZIPEncoding*, invece, implementa l’interfaccia *BinaryEncodingStrategy* effettuando una serializzazione degli oggetti ed una compressione o decompressione dei *bytes* così ottenuti attraverso l’algoritmo GZIP.
Un altro componente del *package* è la classe di utilità *AbstractMessage*, una classe astratta da cui ereditano le classi che implementano l’interfaccia *Message*.
Infine è definita una classe di eccezione *MessageException*, la quale rappresenta un errore dovuto ad un oggetto di tipo messaggio.

Il **sotto-sistema *Discovery*** fornisce una interfaccia per un generico servizio di *discovery* del sistema *peer-to-peer*.
La figura 5.7 mostra il diagramma delle classi del *package it.unipr.aot.p2p.discovery*:

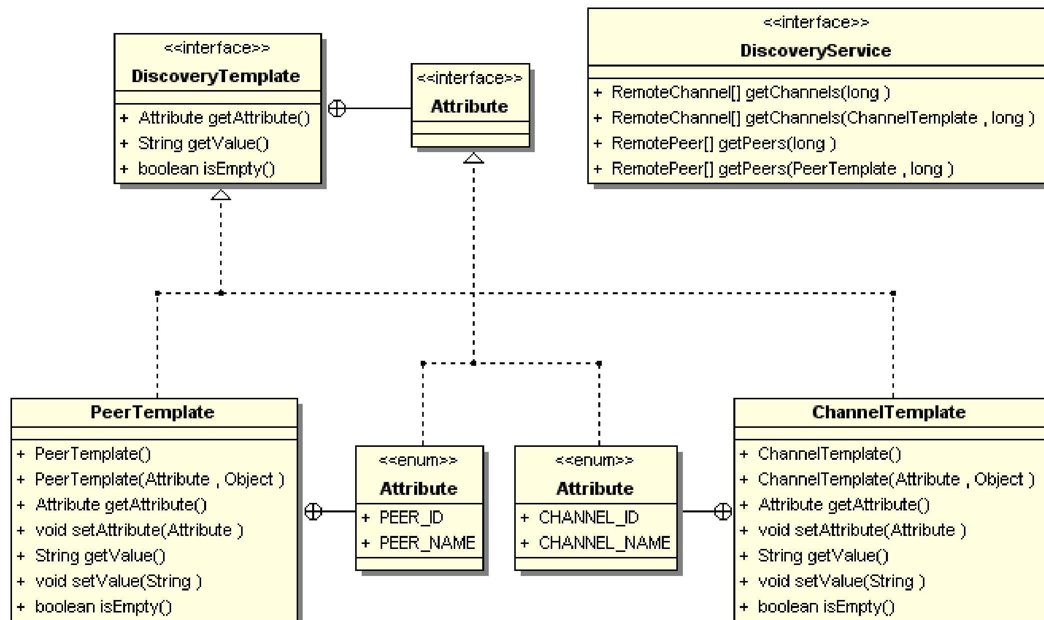

Figura 5.7 Diagramma delle classi del *package it.unipr.aot.p2p.discovery*.

La componente principale del sotto-sistema è l'interfaccia *DiscoveryService*, la quale mette a disposizione i servizi di *discovery* della rete *peer-to-peer*. Per ogni tipologia di risorsa del sistema *peer-to-peer* è presente un diverso servizio di *discovery*. Per ogni servizio può essere specificato un *template* della risorsa, ovvero un insieme di vincoli che le risorse devono soddisfare. Le caratteristiche dei servizi di *discovery* dipendono dalla specifica implementazione: sono possibili servizi affidabili, i quali ritornano sempre come risultato tutte le risorse che soddisfano i vincoli della ricerca, oppure servizi di tipo *best effort*, i quali ritornano un sottoinsieme del risultato.

L'interfaccia *DiscoveryTemplate* rappresenta un generico *template* per il servizio di *discovery*: esso è composto da un attributo e dal corrispondente valore. Il servizio di *discovery* effettuerà una ricerca e ritornerà tutte le risorse trovate che possiedono tale attributo e per le quali il valore dell'attributo stesso corrisponde al valore specificato dal *template*.

Per ogni tipologia di risorsa associata ad un servizio di *discovery* è presente una specifica implementazione dell'interfaccia *DiscoveryTemplate*: la classe *ChannelTemplate* rappresenta un *template* per il servizio di *discovery* delle risorse

di tipo canale, mentre la classe *PeerTemplate* rappresenta un *template* per il servizio di *discovery* dei *peers*.

Dopo aver trattato in maniera esauriente i sotto-sistemi funzionali che compongono l'architettura, andiamo a vedere nello specifico l'implementazione JADE progettata dal dott. Gatti.

5.1.2 Implementazione JADE

Questo paragrafo descrive, per sommi capi, come sia stata realizzata l'implementazione dell'architettura *peer-to-peer* astratta basata sulla tecnologia JADE.

JADE (*Java Agent Development Framework*) è un *framework software* per lo sviluppo di sistemi multi-agente ed è distribuito con licenza *Open Source*. JADE è sviluppato interamente in linguaggio *Java* ed implementa le specifiche FIPA per lo sviluppo di sistemi ad agenti.
Per la realizzazione dell'implementazione dell'architettura astratta si è utilizzata l'ultima versione disponibile del *framework* JADE, giunta alla versione 3.5. La versione 3.5 di JADE introduce alcune importanti novità, come il paradigma di comunicazione basato sui canali virtuali, chiamati *topic*, i quali implementano il *pattern* di comunicazione *publish-subscribe*.

Per ogni *package* dell'architettura astratta è fornita una implementazione basata sulla tecnologia JADE: la convenzione adottata prevede che, per ogni *package*, venga definito un sotto-package di nome "*jade*", nel quale è contenuta la sua implementazione basata sul *framework* JADE.
Secondo questa convenzione, ad esempio, l'implementazione JADE del *package* *it.unipr.aot.p2p.platform* sarà contenuta nel *package* *it.unipr.aot.p2p.platform.jade*.

I sotto-package dell'implementazione JADE, non fanno altro che sviluppare, adattare e specificare meglio come devono comportarsi e cosa

debbano fare i sette sotto-sistemi funzionali descritti esaurientemente nel precedente paragrafo.
Per avere informazione più dettagliate e precise riguardo la realizzazione dell’implementazione JADE, si rimanda a visionare la relazione progettuale redatta dal dott. Gatti per il suo esame di laurea: “*Sviluppo di una architettura astratta per la realizzazione di applicazioni peer-to-peer*”.

Esaurite le necessarie spiegazioni riguardo il sistema dal quale si è partiti, andiamo ora a vedere nel dettaglio come è stata realizzata l’architettura RAIS, e quali manipolazioni all’architettura di partenza sono state effettuate.

5.2 Realizzazione dell’architettura RAIS

La realizzazione della seconda parte del lavoro di tesi, ha richiesto un particolare sforzo di ricerca e di studio. Infatti dal punto di vista materiale, non sono state necessarie molte modifiche all’architettura originaria del dott. Gatti, però per capire come affrontare il problema e come muoversi, si è dovuto prima capire appieno i meccanismi della vecchia architettura e farli girare secondo le nostre nuove necessità. Tenendo conto che il lavoro realizzato dal dott. Gatti era composto da circa 200 classi ed interfacce Java, si può capire che lavoro di studio preliminare sia stato necessario fare su questa architettura per comprenderne anche solo i funzionamenti di base.

Nei prossimi paragrafi andremo a vedere nel dettaglio le modifiche apportate sia a livello di codice sorgente che a livello dei *file* di configurazione anche se, come abbiamo già detto, globalmente non sono state di grande entità.
Infine andremo ad analizzare i risultati finali del progetto, illustrando la nuova infrastruttura così creata ed i suoi possibili miglioramenti e sviluppi futuri.

5.2.1 Modifiche all’architettura di partenza

Dopo uno studio approfondito sull’architettura, la prima operazione che si è deciso di compiere è stata quella di riordinare e riorganizzare i package del progetto secondo le nostre esigenze.
Prima di tutto sono state eliminate tutte le classi riferite all’implementazione JXTA fatta dal dott. Gatti, poiché nel nostro caso specifico era necessario esclusivamente l’implementazione JADE, dato che il nucleo operativo di RAIS è il MAS di cui si è già parlato più e più volte lungo questa trattazione.
In secondo luogo è stato creato un nuovo *root package* chiamato *main*, dove all’interno si trovano le classi necessarie al vero e proprio avvio della nuova infrastruttura, ovvero:

- *MainUtils*: legge il file di configurazione passato come argomento e lo consegna al *P2PBeanFactory*;
- *P2PBeanFactory*: fa partire i *container* di Spring ed i *beans* ad essi associati;
- *RAIS*: il vero e proprio *startup* del sistema grazie alla nuova infrastruttura;
- *RAISAction*: il *setting* iniziale del peer RAIS appena registrato al sistema.

Una seconda operazione che da subito si è resa necessaria è stata la modifica del *file* di configurazione (RAIS.xml) secondo le nuove e specifiche particolarità del sistema che si stava andando a delineare.
Infatti, oltre alle modifiche alle classi incaricate della gestione della piattaforma JADE e del suo *peer* che andremo a vedere nei prossimi paragrafi, è stato modificato anche il codice XML necessario per la prima configurazione del sistema.
In particolare sono state modificate le sezioni appartenenti ai *beans* che verranno poi utilizzati dalle due classi già citate, ovvero:

- piattaforma JADE:
    - *platformConfig*;
    - *platform*;
- *peer*:
    - *peerID*;

- *peerConfig*;
- *peerAction*;
- *peer*.

Il listato 5.1 illustra nello specifico la nuova struttura di RAIS.xml:

```
… … …
<!-- JADE MAIN PLATFORM CONFIG -->
<bean id="platformConfig" factory-bean="platformFactory"
factory method="newMainPlatformConfig">
<property name="agents" value="JxtaDF:com.marsteam.jade.agents.JxtaDF;
LSA:it.unipr.aotlab.jade.dsa.LocalSearchAgent;
DSA:it.unipr.aotlab.jade.dsa.DesktopSearchAgent"/>
<property name="dump" value="true"/>
<property name="MTPs"
value="com.marsteam.jade.mtp.jxta.MessageTransportProtocol"/>
</bean>
<bean id="platform" factory-bean="platformFactory"
factory method="newMainPlatform">
<constructor-arg ref="platformConfig"/>
</bean>
<!-- END PLATFORM SPECIFIC -->
… … …
<!-- PEER -->
<bean id="peerID" factory-bean="idFactory" factory-method="newPeerID">
<constructor-arg value="LocalPeer"/>
</bean>
<bean id="peerConfig" factory-bean="peerFactory"
factory-method="newPeerConfig">
<property name="peerID" ref="peerID"/>
<property name="peerName" value="Michele"/>
</bean>
<bean id="peerAction" class="main.RAISAction">
</bean>
<bean id="peer" factory-bean="peerFactory" factory-method="newPeer">
<constructor-arg ref="peerConfig"/>
<constructor-arg ref="peerAction"/>
</bean>
<!-- END PEER -->
… … …
```

Listato 5.1 La nuova struttura di RAIS.xml.

Dopo avere descritto le modifiche preliminari che si sono apportate all'architettura, andiamo ora a vedere nello specifico i cambiamenti al codice delle due classi incaricate delle gestione della piattaforma JADE e del *peer* ad essa registrato: *JadePlatformConfig* e *JadeUserPeer*.

5.2.1.1 La classe *JadePlatformConfig*

Questa classe appartiene al *package it.unipr.aot.p2p.platform.jade*. Essa ha il compito di implementare la configurazione della piattaforma JADE. In pratica la classe *JadePlatformConfig* acquisisce i *beans* riferiti alla piattaforma, con tutte le loro proprietà e si fa carico di costruire un opportuno oggetto *Profile* (*jade.core.Profile*) per il successivo *boot* di JADE.

Originariamente questa classe era in grado di gestire solo alcuni parametri di configurazione (ovvero le proprietà dei *beans*). Dovendo costruire questa infrastruttura a sostegno di un sistema complesso come RAIS, è stato necessario aggiungere la compatibilità con altre proprietà al fine di rendere più funzionale e preciso il profilo implementato.
Le proprietà acquisite da questa classe per mezzo del *file* di configurazione RAIS.xml sono le seguenti:

- *agents*: definisce quali agenti far partire insieme alla piattaforma;
- *dump*: stabilisce se visualizzare o meno le opzioni scelte durante il *boot*;
- *localHost*: *setting* del *localHost*;
- *localPort*: *setting* della *localPort*;
- *main*: definisce se il *container* in questione è o meno il *main container*;
- *MTPs*: definisce quale MTP utilizzare per la piattaforma JADE;
- *platformID*: *setting* dell'ID univoco della piattaforma.

Nel listato 5.2 è possibile vedere i dettagli implementativi del codice che permette di svolgere le operazioni di creazione del profilo di *boot* nella classe *JadePlatformConfig*:

```
… … …
public abstract class JadePlatformConfig implements PlatformConfig {
private String agents;
private String dump;
private String localHost;
private String localPort;
… … …
public JadePlatformConfig(JadePlatformConfig config) {
	this.agents = config.agents;
	this.dump = config.dump;
	this.localHost = config.localHost;
	this.localPort = config.localPort;
	this.main = config.main;
	this.MTPs = config.MTPs;
	this.platformID = config.platformID;
}
… … …
protected Profile createProfile() {
	// Main container profile
	Profile profile = new ProfileImpl();
	setParameter(profile, Profile.AGENTS, agents);
	setParameter(profile, Profile.DUMP_OPTIONS, dump);
	setParameter(profile, Profile.LOCAL_HOST, localHost);
	setParameter(profile, Profile.LOCAL_PORT, localPort);
	setParameter(profile, Profile.MAIN, main);
	setParameter(profile, Profile.MTPS, MTPs);
	setParameter(profile, Profile.PLATFORM_ID, platformID);
try {
// Get jade kernel services
jade.util.leap.List services = profile.getSpecifiers(Profile.SERVICES);
// Add topic service
Specifier topicService = new Specifier();
… … …
// Save jade kernel services
profile.setSpecifiers(Profile.SERVICES, services);
… … …
	return profile;
}
```

Listato 5.2 La creazione del profilo JADE nella classe *JadePlatformConfig*.

5.2.1.2 La classe *JadeUserPeer*

Questa classe appartiene al *package* *it.unipr.aot.p2p.peer.jade*. Essa implementa l’entità *peer* che andrà ad interfacciarsi prima con la piattaforma JADE e successivamente col sistema RAIS.

JadeUserPeer come classe entra in gioco durante l’avvio della piattaforma e quindi del *peer* registrato su di essa. Tramite le proprietà dei *beans* di Spring presenti sul *file* XML di configurazione, è possibile eseguire i *setting* di tutte le principali caratteristiche necessarie per il corretto funzionamento del sistema.

La classe *JadeUserPeer* è internamente implementata attraverso un agente JADE, ovvero una istanza della classe *jade.core.Agent*.
Per quanto riguarda i dettagli implementativi, la classe *JadeUserPeer* costituisce un *wrapper* per la classe *jade.wrapper.AgentController*: tale classe rappresenta un *controller* per un agente JADE, permettendone la gestione.
All’interno della classe *JadeUserPeer* è definita una *inner class PeerAgent* che estende la classe *jade.core.Agent*: essa rappresenta l’agente JADE che implementa il *peer*.

Il metodo *start()* dell’interfaccia *UserPeer* è implementato invocando internamente il metodo *start()* dell’oggetto di classe *jade.wrapper.AgentController*, il quale provoca l’attivazione e l’esecuzione dell’agente. Similmente il metodo *stop()* invoca rispettivamente il metodo *kill()* dell’oggetto *controller* per terminare l’esecuzione dell’agente. principali caratteristiche necessarie per il corretto funzionamento del sistema.

Detto in parole povere, la classe *JadeUserPeer* si comporta con l’entità *peer* in modo speculare a come la classe *JadePlatformConfig* si comporta con la piattaforma JADE.

Il listato 5.3 riporta uno stralcio dei metodi più importanti per la creazione del *peer* in questa classe dell’infrastruttura:

```
… … …
public class JadeUserPeer extends JadePeer implements UserPeer {
      private JadePlatform platform;
      private PeerAction peerAction;
      private AgentController agentController;
      private PeerAgent agent;
      private boolean agentStarted;
      private Object agentEnd;

      /**
       * Creates a new user peer with the specified peer configuration
       *
       * @param platform: the jade network platform
       * @param peerConfig: the jade peer configuration
       * @param peerAction: the peer action
       * @throws PeerConfigException
       */
public JadeUserPeer(JadePlatform platform,
                    JadePeerConfig peerConfig,
                    PeerAction peerAction) throws PeerConfigException {
      super(peerConfig.toPeerDescription());
      this.platform = platform;
      // Peer action
      this.peerAction = peerAction;
      // Register peer action owner
      peerAction.setPeer(this);
      // Create jade agent
      AgentContainer container = platform.getContainer();
      // Nickname
      String nickname = ((JadePeerID)peerConfig.getPeerID())
                        .getJadePeerID().getLocalName();
      // Agent class
      String className = JadeUserPeer.PeerAgent.class.getName();
      // Agent args
      Object[] args = new Object[1];
      args[0] = this;
      try {
           this.agentController = container.createNewAgent(nickname,
                                                           className,
                                                           args);
… … …
```

Listato 5.3 Creazione dell’entità *peer* nella classe *JadeUserPeer*.

5.3 Risultati ottenuti

Il risultato finale di questa seconda parte del progetto (e quindi dell’intero lavoro di tesi) è stato molto soddisfacente, anche se non sono mancati problemi implementativi e progettuali in corso d’opera.

Dopo avere modificato l’architettura del dott. Gatti secondo le esigenze di RAIS, è stato creato nuovamente un altro *file* JAR (*RAIS-arch.jar*).
Questo *file* è composto nel suo nucleo, dai sorgenti dell’architettura astratta implementata. Inoltre, nelle librerie è presente anche il riferimento a *jade-jxta.jar*, utilizzato come supporto comunicativo del sistema. Così facendo, è stato creato un unico file JAR, contenente al suo interno l’intera infrastruttura *peer-to-peer* realizzata. Aggiungendo questa libreria a RAIS, gli si fornisce la vera e propria struttura *software* sulla quale può lavorare ed operare.

Per mezzo dell’utilizzo del *framework* Spring, non solo si è riusciti ad implementare una configurazione del sistema semplice, versatile e potenzialmente avanzata, ma si è anche potuto creare un semplice *file* eseguibile (*RAIS.bat*) che permetta di fare lo *startup* di RAIS in maniera immediata. Questo *file* lancia solamente il seguente comando *batch*:

```
java -jar C:\eclipse_workspace\RAIS\lib\RAIS-arch.jar
C:\eclipse_workspace\RAIS\conf\RAIS.xml
```

Questo commando lancia il JAR appena creato, che nel *file* MANIFEST specifica come classe di *boot main.RAIS* la quale, come abbiamo già detto nel paragrafo 5.2.1, è la classe dell’infrastruttura che si occupa dello *startup* del sistema. Oltre ad eseguire il JAR, nel secondo parametro del comando, viene anche specificato il percorso del *file* di configurazione XML necessario alla classe *main.RAIS* per creare il profilo di *bootstrap* della piattaforma JADE.

```
C:\Windows\system32\cmd.exe
C:\eclipse_workspace\RAIS>java -jar C:\eclipse_workspace\RAIS\lib\RAIS-arch.jar
C:\eclipse_workspace\RAIS\conf\RAIS.xml


--> RAIS: Remote Assistant for Information Sharing <--


Starting Spring container...
Config file: C:\eclipse_workspace\RAIS\conf\RAIS.xml
 INFO - Loading XML bean definitions from file [C:\eclipse_workspace\RAIS\conf\R
AIS.xml]
...OK!


Getting peer...
16-set-2008 11.44.33 jade.core.Runtime beginContainer
INFO: ----------------------------------------
    This is JADE 3.4.1 - revision 5912 of 2006/11/16 13:09:18
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
----------------------------------------
16-set-2008 11.44.33 jade.core.AgentContainerImpl init
INFO: Startup options dump:
(Profile agents=JxtaDF:com.marsteam.jade.agents.JxtaDF; LSA:it.unipr.aotlab.jade
.dsa.LocalSearchAgent; DSA:it.unipr.aotlab.jade.dsa.DesktopSearchAgent port=1099
 jvm=j2se main=true local-host=192.168.1.3 local-port=1099 mtps=com.marsteam.jad
e.mtp.jxta.MessageTransportProtocol services=[jade.core.mobility.AgentMobilitySe
rvice, jade.core.event.NotificationService, jade.core.messaging.TopicManagement:
jade.core.messaging.TopicManagementService] dump-options=true host=192.168.1.3)
16-set-2008 11.44.35 jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
16-set-2008 11.44.35 jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
16-set-2008 11.44.35 jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
16-set-2008 11.44.35 jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
16-set-2008 11.44.35 jade.core.BaseService init
INFO: Service jade.core.messaging.TopicManagement initialized
Setting keystore password...            Setting PeerID...
Setting identity password...            Joining to PSE membership service...
Creating peer group ID =  clearText:'JADE-GROUP' , function:'A Multi Agents Syst
em framework FIPA compliant'
 INFO [main] (MessageTransportProtocol.java:126) - JXTA MTP is now running...
16-set-2008 11.44.44 jade.core.messaging.MessagingService boot
INFO: MTP addresses:
jxta:Michele
16-set-2008 11.44.44 jade.core.AgentContainerImpl joinPlatform
INFO: --------------------------------------------
Agent container Main-Container@JADE-IMTP://XPS_M1530 is ready.
--------------------------------------------
...OK!
Starting platform...
...OK!
Starting peer...
...OK!
Waiting for peer setting...
- PEER SETTING -
Peer id: LocalPeer@XPS_M1530:1099/JADE
Peer name: Michele
...peer setting ended!


The RAIS system is now ready to start!

JxtaDF: New REGISTRATION request from ( agent-identifier :name LSA@XPS_M1530:109
9/JADE  :addresses (sequence jxta:Michele )).

Peer View
Private Peer View
Peer View
Private Peer View
Peer View
Private Peer View
_
```

Figura 5.8 Il nuovo *startup* del sistema RAIS.

Una volta effettuata la nuova procedura di *startup* (illustrata nella figura 5.8), il comportamento di RAIS è pressoché identico a quello del sistema ottenuto alla fine della prima parte del progetto. Infatti la creazione di un’architettura astratta per il sistema, comporta cambiamenti a livello di configurazione e di *setting* iniziale, mentre le fasi operative vere e proprie non subiscono mutamenti poiché si limitano semplicemente ad utilizzare elementi ed oggetti già inizializzati. Nonostante questo vi è da dire che spesso l’infrastruttura finale così ottenuta, nelle fasi di *testing*, ha presentato risultati comunicativi peggiori rispetto al sistema eseguito in precedenza. Nei vari test infatti è capitato più volte che si avessero problemi di comunicazione con i *peer* remoti. Eseguendo continui *debug* per scoprire a cosa fosse dovuto quest’errore, si è notato che gli agenti avevano problemi nell’apertura delle *pipe* di JXTA. Probabilmente questo può essere dovuto a problemi di aggiornamento delle versioni di JXTA e della VM Java, a nuove regole create per il *firewall* del sistema operativo che limita certi tipi di connessioni oppure alla modalità di *startup* via Spring della piattaforma JADE che non crea i giusti servizi per gli agenti.

Conclusioni

I risultati finali di questo progetto di tesi sono stati molto buoni e si sono raggiunti tutti gli obiettivi sperimentali che ci si era prefissati ad inizi lavori.
La nuova infrastruttura sviluppata si è rivelata efficiente e semplice da utilizzare, permettendo un rapido e versatile sviluppo del supporto al sistema di *information-sharing* ideato.

Nella prima parte del progetto, grazie all'integrazione JADE-JXTA-RAIS, si è potuti creare un sistema di *information-sharing* estremamente efficiente e dalle marcate potenzialità. Il sistema in questione abbina in maniera ottimale le caratteristiche di intelligenza, autonomia e collaborazione proprie dei sistemi multi-agente (MAS), con quelle di condivisione ed efficienza comunicativa delle reti *peer-to-peer* (JXTA).

Nella seconda parte del progetto, grazie all'uso del *framework* Spring, è stato possibile configurare l'applicazione tramite un *file* scritto in linguaggio XML: questo permette di apportare cambiamenti alla configurazione senza modificare alcun sorgente Java. La sufficiente generalità dell'architettura astratta insieme all'uso del *framework* Spring, permettono di modificare l'implementazione scelta semplicemente cambiando poche righe di codice nel *file* di configurazione XML, senza modificare le classi originarie del sistema.

Per quanto riguarda gli sviluppi futuri, sicuramente la prima operazione da andare ad effettuare è cercare di capire e risolvere l'anomalia operativa segnalata

alla fine dell'ultimo capitolo, poiché un comportamento di questo tipo pregiudica molto le *performance* di un sistema *peer-to-peer*.

Fatto questo, un altro miglioramento che è possibile effettuare riguarda l'interfaccia grafica del *DesktopSearchAgent* ed i messaggi d'avviso per l'utente. Attualmente questa parte è semplice e funzionale ma per una futura distribuzione del sistema ad un'utenza a largo raggio, occorre sicuramente adottare qualche accorgimento in più, in particolare nell'assistenza all'utente e nelle interazioni con esso.

Un secondo aspetto che è possibile considerare è quello di ampliare il numero di servizi messi a disposizione dall'architettura astratta, per dare ancora maggiore versatilità all'architettura nel suo insieme.

Infine un ultimo aspetto degli sviluppi futuri potrebbe riguardare lo sviluppo di meccanismi di supporto per la pubblicazione ed invocazione dei servizi offerti dai *peer*, ispirandosi agli *standard* definiti dai *Web Services*. Lo scopo è quello di permettere a ciascun *peer* di scoprire in maniera automatica i servizi offerti dagli altri *peer*, selezionare i servizi desiderati ed invocarli dinamicamente.

Bibliografia

Vari autori, *Sistemi peer-to-peer: caratteri generali*, Wikipedia - L'enciclopedia libera, http://it.wikipedia.org/wiki/Peer-to-peer, 2008.

L. Gatti, *Sviluppo di una architettura astratta per la realizzazione di applicazioni peer-to-peer*, Tesi di laurea specialistica in Ingegneria Informatica, Università degli Studi di Parma, 2007.

M. Longari, P*rogettazione e realizzazione di un sistema web multi-agente per la pianificazione di itinerari*, Tesi di laurea triennale in Ingegneria Informatica, Università degli Studi di Parma, 2006.

Vari autori, *Project JXTA: Getting Started*, Documentazione originale del progetto JXTA, Sun Microsystems, 2000.

M. Mari – A. Poggi – M. Tomaiuolo, *A Multi-Agent System for Information Sharing*, Progetto di ricerca accademico, Università degli Studi di Parma, 2008.

Vari autori, *JXTA Search: Distributed Search for Distributed Networks*, Documentazione originale del progetto JXTA, Sun Microsystems, 2001.

P. Simonazzi, *Sviluppo di un sistema peer-to-peer per la condivisione di documenti*, Tesi di laurea specialistica in Ingegneria Informatica, Università degli Studi di Parma, 2008.

Vari autori, *Il framework Spring*, Wikipedia - L'enciclopedia libera, http://it.wikipedia.org/wiki/Spring_framework, 2008.

M. Sciabarra, *Spring Framework per Non Credenti*, Java Fishing, http://www.javajournal.it/blog/2006/09/20/spring_framework_per_non_credenti.html, 2006.

V. Grassi, *Jade: Java Agent Development Framework*, Informatica Mobile, http://www.uniroma2.it/didattica/infomob/deposito/jade.pdf, 2008.

M. Agosti – R. Scalise, *jade-jxta: paper*, Progetto di *curriculum* accademico, Università degli Studi di Parma, 2008.