UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

LE PIATTAFORME JADEX E JASON PER LO SVILUPPO DI SISTEMI MULTIAGENTE

ANALISI DEL FUNZIONAMENTO E STUDIO COMPARATIVO DELLE PRESTAZIONI

Relatore: Prof. Carlo Ferrari

Laureando: Martini Andrea

ANNO ACCADEMICO 2009-2010

Ai miei genitori, ai miei nonni e a Letizia

SOMMARIO

INTRODUZIONE

1.1 Sistemi multi-agente

I sistemi multi-agente (*MAS, Multi-Agent Systems*) sono sistemi composti da elementi computazionali multipli, conosciuti appunto come *agenti*. Gli agenti sono sistemi computerizzati dotati di due importanti capacità. Innanzi tutto sono (almeno fino ad un certo punto) capaci di azioni autonome, di decidere cioè per conto proprio su ciò che devono fare in modo da portare a compimento i loro obiettivi. In secondo luogo sono altresì capaci di interagire con altri agenti non solo per scambiarsi dati, ma anche per compiere operazioni più complesse di coordinamento, cooperazione e negoziazione.

Oggetto di ricerche da lunga data in intelligenza artificiale, i sistemi multi-agente costituiscono un'interessante tipologia di modellazione di società ed hanno a questo riguardo vastissimi campi di applicazione che si estendono fino alle scienze umane e sociali.

I sistemi multi-agente possono essere visti come un raccordo tra l'ingegneria del software e l'intelligenza artificiale, con ovviamente un apporto fondamentale da parte dei sistemi distribuiti. In rapporto ad un oggetto, un agente può prendere iniziative, può rifiutarsi di obbedire ad una richiesta, può migrare e così via. L'autonomia che ne consegue consente al progettista di concentrarsi quindi sul lato umanamente comprensibile del software.

1.2 Un po' di storia

L'idea delle tecniche di modellizzazione basate sugli agenti è stata sviluppata come concetto relativamente semplice verso la fine degli anni '40. Dal momento che necessitava però di procedure di calcolo molto avanzate per l'epoca non è stata diffusa su larga scala fino ai nostri anni '90.

La storia dei modelli basati sugli agenti va indietro fino alla *macchina di John Von Neumann*, una macchina puramente teorica. La periferica proposta da Von Neumann seguiva istruzioni dettagliate e precise per modellare un'esatta copia di se stessa. Il concetto è stato in seguito migliorato da *Stanisław Marcin Ulam*, matematico e amico di Von Neumann. Ulam ha suggerito che la macchina fosse costruita su carta, come una collezione di celle su di una griglia. L'idea interessò Von Neumann che la disegnò, creando così la prima di quelle periferiche che in seguito sarebbero state conosciute come *automi cellulari*.

Un altro miglioramento è stato introdotto dal matematico *John Conway*, che costruì il celebre *Game of Life*. A differenza della macchina di Von Neumann, il Game of Life di Conway operava seguendo regole molto semplici in un mondo virtuale nella forma di una scacchiera a due dimensioni[1].

Uno dei primi modelli concettuali basati sugli agenti fu il *modello della segregazione* di *Thomas Schelling*, che fu discusso nel suo scritto *Dynamic Models of Segregation* del 1971. Sebbene Schelling usasse originariamente monete e carta millimetrata piuttosto che computer, il suo modello rappresentava di fatto il concetto di base dei modelli basati sugli agenti, visti come agenti autonomi che interagiscono in un ambiente condiviso che permette di osservarne la totalità come risultato emergente.

Agli inizi degli anni '80, *Robert Axelrod* ha tenuto un torneo riguardante le strategie del *dilemma del prigioniero* e le ha fatte interagire in maniera agent-based per determinare un vincitore. Axelrod ha poi proseguito su questa strada sviluppando molti altri modelli basati sugli

[1] Tra gli esempi distribuiti con le piattaforme Jadex e Jason è presente anche un'implementazione per entrambe le piattaforme del Game of Life.

agenti nel campo della scienza politica che esamina fenomeni che spaziano dall'etnocentrismo alla disseminazione delle culture. Verso la fine degli anni '80, il lavoro di *Craig Reynolds* sui *flocking models* (modelli affollati) contribuì allo sviluppo dei primi modelli biologici ad agenti che contenevano caratteristiche sociali. Reynolds tentò di modellare la realtà degli agenti biologici animati, conosciuta come *vita artificiale*, un termine coniato da *Christopher Langton*.

Il primo utilizzo della parola *agente* e l'ideazione di una definizione di come è attualmente usata nel contesto che intendiamo noi è di difficile collocazione. Un possibile candidato è probabilmente lo scritto di *John Holland* e *John H. Miller* del 1991, *Artificial Adaptive Agents in Economic Theory*. Allo stesso tempo, durante gli anni '80, scienziati sociali, matematici, ricercatori e molti altri provenienti da altre discipline svilupparono il *CMOT (Computational and Mathematical Organization Theory)*. Questo ramo si sviluppò come un gruppo di particolare interesse del *TIMS (The Institute of Management Sciences)* e della sua società collegata, la *ORSA (Operations Research Society of America)*. Durante la metà degli anni '90 questo ramo si focalizzò nella progettazione di team efficaci, nella comprensione della comunicazione necessaria per un'organizzazione efficace nonché nel comportamento delle reti sociali. Con la comparsa di *SWARM* nella metà degli anni '90 e di *RePast* nel 2000 (così come anche di altri codici creati ad hoc) il CMOT - più tardi rinominato in CASOS *(Computational Analysis of Social and Organizational Systems)* - incorporò innumerevoli modelli basati sugli agenti. *Samuelson* (2000) è un'ottima visione d'insieme della storia recente, così come anche *Macal*.

Kathleen M. Carley, della Carnegie Mellon University di Pittsburgh, sviluppò un primo modello basato sugli agenti, *Construct*, per esplorare la co-evoluzione delle reti sociali e della cultura. *Joshua M. Epstein* e *Robert Axtell* svilupparono invece il primo modello ad agenti su larga scala, *Sugarscape*, per simulare ed esplorare il ruolo dei fenomeni sociali come le migrazioni stagionali, l'inquinamento, la riproduzione sessuale, il combattimento, e così via.

Alla fine degli anni '90 la fusione di TIMS e ORSA portò alla creazione di *INFORMS (The Institute For Operations Research and The Management Sciences)*. La loro mossa di passare da due meeting ogni anno a uno solamente fu da stimolo al gruppo CMOT per formare una società separata, la *NAACSOS (North American Association for Computational Social and Organizational Sciences)*. Kathleen M. Carley fu una delle persone più attive specialmente per i modelli delle reti

sociali nonché la prima presidente della NAACSOS. In seguito fu succeduta da *David Sallach* della University of Chicago, e da *Michael Prietula* della Emory University. Contemporaneamente all'inizio della NAACSOS, vennero fondate anche la *ESSA (European Social Simulation Association)* e la *PAAA (Pacific Asian Association for Agent-Based Approach in Social Systems Science)*, controparti della suddetta. Queste tre organizzazioni ora collaborano sullo scenario internazionale, ed è grazie ai loro sforzi congiunti se nel 2006 e nel 2008 sono stati tenuti il *First* e il *Second World Congress on Social Simulation*.

Più recentemente, *Ron Sun* ha sviluppato metodi per basare le simulazioni basate sugli agenti a modelli di percezione umana, conosciuti come *cognitive social simulation*. *Bill McKelvey*, *Suzanne Lohmann*, *Dario Nardi* e altri alla *UCLA (University of California, Los Angeles)* hanno apportato interessanti contributi alle azioni organizzative e al decision-making. Dal 2001, la UCLA ha predisposto una conferenza a *Lake Arrowhead*, in California, che è diventata un punto di ritrovo fondamentale per tutti i professionisti in questo settore.

Dopo questa brevissima panoramica inerente la storia dei sistemi multi-agente, nei prossimi due paragrafi di questo capitolo introduttivo al lavoro di tesi andremo a conoscere meglio (anche se in modo non molto approfondito, ma comunque sufficiente a darci un'idea delle potenzialità di questi due linguaggi) quelle che sono le due piattaforme software selezionate per il lavoro di analisi e di confronto, e cioè *Jadex* e *Jason*.

1.3 Jadex

Jadex è una piattaforma basata su *Java* per la creazione di agenti goal-oriented che segue il modello *BDI (Belief-Desire-Intention)*. Il progetto, ideato e sviluppato da *Alexander Pokahr*, *Lars Braubach* e *Winfried Lamersdorf* dell'*Università di Amburgo*[2], mira a semplificare il più possibile la creazione di sistemi ad agenti senza però sacrificare la potenza espressiva del paradigma ad agenti stesso. L'obiettivo è quello di costruire un livello di agenti *(agent layer)* razionale che si collochi immediatamente sopra ad una infrastruttura middleware, e che permetta una costruzione e una gestione intelligente degli agenti in questione.

[2] Vedi [1] nella bibliografia a fine tesi

Incoraggiando una semplice transizione dai sistemi distribuiti tradizionali allo sviluppo di sistemi multi-agente, Jadex permette di utilizzare (dove possibile) concetti e tecnologie orientate agli oggetti e già molto ben affinate e collaudate come *Java* e *XML*, come vedremo in dettaglio nel prosieguo del lavoro di tesi. Inoltre, l'engine di ragionamento di Jadex tenta di superare i tradizionali limiti dei sistemi BDI introducendo goal espliciti: questo permette la realizzazione di un meccanismo di calcolo e di valutazione dei goal, e inoltre facilita lo sviluppo di applicazioni rendendo i risultati di questa analisi facilmente trasferibili al livello di implementazione.

Il sistema è disponibile open source sotto licenza *GNU LGPL*, e fornisce inoltre un'ottima documentazione online oltre che un pacchetto di programmi dimostrativi e di esempi molto esplicativi, utili anche per chi è alle prime armi con questa tipologia di linguaggi.

1.4 Jason

Jason è un interprete Java basato su di una versione estesa di *AgentSpeak* (chiamata *AgentSpeak(L)*, ideata inizialmente da *Anand Rao*). È stato sviluppato da *Jomi F. Hübner* e *Rafael H. Bordini*, rispettivamente della *Federal University of Rio Grande do Sul* e della *Universidade Regional de Blumenau (FURB)*, entrambe brasiliane[3]. I due docenti e ricercatori si sono basati sul lavoro precedentemente svolto con vari colleghi al fine di arrivare, circa cinque anni fa, alla realizzazione di una piattaforma stabile, che è tuttora in continuo aggiornamento.

Uno dei migliori approcci conosciuti per lo sviluppo di agenti cognitivi è il paradigma *BDI (Belief-Desire-Intention)*. Nell'area dei linguaggi di programmazione agent-oriented, *AgentSpeak* è in assoluto uno dei linguaggi astratti più importanti basati sul suddetto paradigma. Il tipo di agenti programmati con *AgentSpeak* sono talvolta detti anche *sistemi di pianificazione reattiva*.

Il *PRS (Procedural Reasoning System)*, o schema di ragionamento procedurale, è stato originariamente sviluppato allo *Stanford Research Institute* da *Michael Georgeff* ed *Amy Lansky*, ed è forse la prima architettura basata sugli agenti ad incorporare specificamente il paradigma BDI: anche ai nostri giorni è considerato uno dei più duraturi approcci allo sviluppo di agenti.

[3] Vedi [2] nella bibliografia a fine tesi

Al momento, *Jason è a tutti gli effetti il primo interprete per una versione così migliorata di AgentSpeak*. Il fatto di essere implementato in Java lo rende quindi uno strumento multi-piattaforma molto *versatile* e *duttile*. È disponibile anch'esso come piattaforma open source, distribuita sotto licenza *GNU LGPL*.

Oltre che interpretare il linguaggio *AgentSpeak(L), Jason* ha anche svariate altre caratteristiche, come ad esempio la gestione dei *plan failures*, la presenza della negazione forte (*strong negation*), il supporto per lo *sviluppo di ambienti*, la possibilità di eseguire un *sistema multi-agente distribuito su di una rete di computer* (con l'utilizzo di *JADE* o *SACI*), funzioni di selezione completamente *personalizzabili* (in Java, ovviamente), e molte altre ancora.

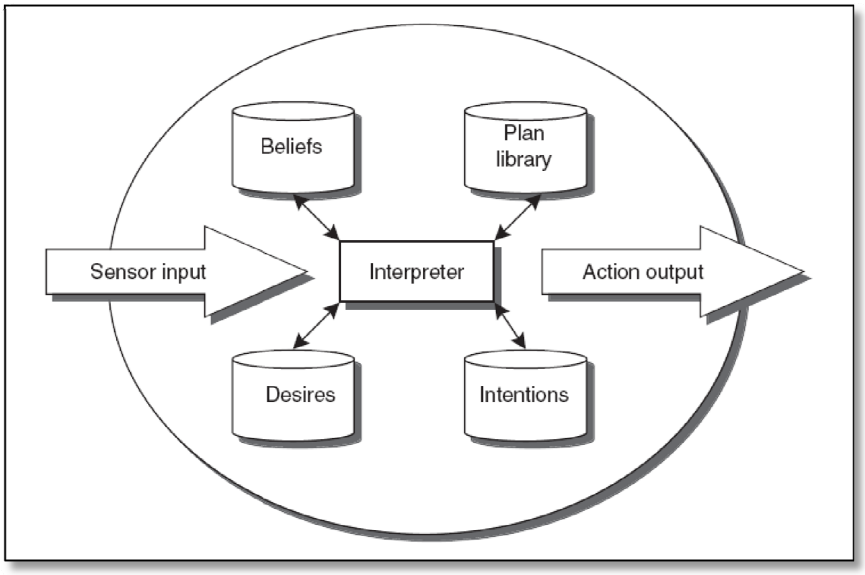

Figura 1.1 Schema di ragionamento procedurale (PRS)

1.5 Analisi e confronto

Dal momento che entrambe le piattaforme *Jadex* e *Jason* sono basate su *Java* e distribuite sotto licenza *GNU LGPL*, l'operazione di confronto che verrà portata avanti nel prosieguo di questo lavoro di tesi risulta molto naturale.

Il lavoro di analisi e confronto sarà suddiviso in *due fasi*. La *prima fase* riguarderà un'analisi di tipo *qualitativo* su quelle che sono le caratteristiche tecniche e le modalità di funzionamento delle due piattaforme, quindi con particolare attenzione a tutte le componenti di ciascuna

piattaforma, al modo in cui queste componenti interagiscono tra di loro e agli aspetti sintattici dei linguaggi. La *seconda fase* prevede invece un'analisi di tipo prettamente *quantitativo*: dal momento che le due piattaforme sono open source, verranno appositamente modificati i file sorgenti delle due piattaforme per permetterci di raccogliere dati riguardanti i tempi di caricamento di ciascuna piattaforma, il carico di memoria iniziale, il tempo di creazione e il tempo di cancellazione di un agente, il tempo che un agente impiega a migrare tra due host della stessa piattaforma e così via.

Le misurazioni che verranno effettuate nella seconda fase saranno scremate e pesate con opportuni *strumenti matematici* che verranno descritti in dettaglio nel prosieguo del lavoro di tesi. Naturalmente, essendo queste misurazioni fortemente influenzate dall'hardware delle macchine sulle quali si effettuano i test, utilizzeremo per le nostre misurazioni tre diverse macchine in modo da cercare di stabilire con buona precisione una linea guida attendibile.

Ricordo infine che l'obiettivo di questo lavoro di tesi non è sicuramente quello di stabilire quale delle due piattaforme sia migliore rispetto all'altra, ma semplicemente quello di fornire una panoramica attendibile delle *potenzialità* che entrambe possono offrire.

ANALISI DELLA LETTERATURA

2.1 Metodi e modelli per l'analisi delle prestazioni di un sistema

Gli enormi progressi tecnologici degli ultimi decenni hanno reso possibile la costruzione di sistemi informatici sempre più complessi e strutturati. Dall'introduzione della definizione di *sistema*, che consiste in un insieme di componenti interdipendenti e che interagiscono per raggiungere un determinato obiettivo, si deduce quella relativa ad un *sistema di elaborazione*, che è un insieme di componenti hardware, firmware e software che permettono l'elaborazione delle informazioni eseguendo programmi utente. Di conseguenza, per l'analisi del comportamento di tali sistemi non è sufficiente applicare un semplice approccio intuitivo e basato sull'esperienza.

Nel seguito verranno introdotti i principali *modelli* e *metodi* per la valutazione delle prestazioni di sistemi. I modelli e le metodologie qui descritte trovano ampia applicazione nella rappresentazione ed analisi non solo di diversi sistemi di elaborazione, ma anche di sistemi di comunicazione e telecomunicazione, così come di sistemi software.

Le prestazioni di un sistema solo valutabili sia *qualitativamente* che *quantitativamente*. L'analisi quantitativa si effettua definendo opportune figure di merito o indici di prestazione, quali ad esempio l'utilizzazione delle risorse o i tempi di risposta forniti dal sistema.

Lo studio della valutazione delle prestazioni di sistemi costituisce un'area di ricerca in informatica che ha portato alla definizione di metodologie e di strumenti per la creazione e l'analisi di modelli di sistemi (*modeling*) e allo sviluppo di tecniche di misurazione e caratterizzazione del carico.

Le metodologie per la valutazione delle prestazioni di sistemi possono essere distinte in due categorie principali, come illustrato nella *figura 2.1*:

- *tecniche di misurazione*
- *tecniche modellistiche*

Figura 2.1 Schema riassuntivo dei metodi di valutazione delle prestazioni di sistemi

Tra le *tecniche di misurazione* si possono identificare:

- **Tecniche di misurazione diretta:** il sistema viene direttamente misurato utilizzando il carico reale del sistema stesso, tramite opportuni strumenti e metodologie.

- **Benchmarking (misurazione con carico artificiale):** le misurazioni vengono effettuate sempre sul sistema reale, ma utilizzando un carico artificiale (detto *benchmark*). Questo approccio ha come principale vantaggio, rispetto al precedente, la ripetibilità del procedimento di misurazione e la possibilità di effettuare misurazioni comparative fra diversi sistemi sotto le stesse condizioni di carico.

- **Prototipazione:** se il sistema su cui effettuare le misurazioni non è disponibile, in quanto non esistente o in quanto le due tecniche precedentemente descritte risultino non applicabili, si può ricorrere alla costruzione di un prototipo su cui effettuare poi le misurazioni. Se il prototipo è realizzato a livello software viene anche detto *emulatore del sistema*. Lo svantaggio di questo tipo di approccio è che, una volta realizzato il prototipo, offre *scarsa flessibilità e modificabilità* per lo studio di scelte alternative.

L'uso dei *modelli* per la valutazione e lo studio del comportamento dei sistemi diventa indispensabile nella fase di progettazione di sistemi non esistenti (per cui le tecniche di misurazione diretta o artificiale non sono applicabili) e in particolar modo nei primi stadi di progettazione in cui è importante poter differenziare tra varie alternative senza comunque dover scendere ad un livello di dettaglio eccessivamente elevato, come invece risulta solitamente necessario nello sviluppo di prototipi.

Un modello è una rappresentazione astratta del sistema che include solo gli aspetti rilevanti allo scopo dello studio del suddetto. Un modello è *definito* ad un determinato livello di astrazione, ovvero il sistema viene descritto con un certo livello di dettaglio, includendo nella rappresentazione solo quelle componenti e interazioni tra esse che si ritengono necessarie allo scopo prefisso. Alla definizione del modello segue la sua *parametrizzazione* (per poter considerare le alternative di studio) e la sua valutazione o soluzione (per ottenere le informazioni relative allo studio del sistema).

I *modelli* possono essere suddivisi in:

- **Analitici:** in tali modelli le componenti e il carico del sistema sono rappresentati da variabili e parametri, mentre le relazioni tra le componenti da relazioni fra queste quantità. La valutazione del sistema effettuata utilizzando un modello analitico richiede il calcolo della sua soluzione tramite metodi analitici o soluzioni numeriche.

- **Simulativi:** riproducono il *comportamento dinamico del sistema* nel tempo rappresentando le componenti e le interazioni in termini di relazioni funzionali. La

valutazione di un sistema tramite un modello di simulazione richiede l'esecuzione di un programma di simulazione, o *simulatore*, che rappresenta l'evoluzione temporale del sistema e su cui si effettuano delle misure per stimare le grandezze di interesse. Se da un lato la simulazione fornisce uno strumento potente per la valutazione di sistemi, dall'altro il suo limite maggiore è costituito dal costo sia di sviluppo e di parametrizzazione che di esecuzione, specialmente se il sistema è rappresentato ad un elevato livello di dettaglio. Inoltre, per le caratteristiche delle misurazioni effettuate negli esperimenti di simulazione, una corretta analisi dei risultati deve utilizzare opportune tecniche statistiche per la stima degli indici di prestazione, spesso di non semplice applicazione.

- **Ibridi:** costituiscono una categoria particolare di modelli per la valutazione delle prestazioni di sistemi. I modelli ibridi *combinano le caratteristiche dei modelli analitici e di simulazione*. Tali modelli sono particolarmente adeguati per rappresentare sistemi di grandi dimensioni eventualmente nell'ambito di metodologie di sviluppo gerarchico.

In letteratura, la tecnica combinata *analitico-simulativa* è prevalentemente presentata come un approccio empirico per l'analisi di sistemi complessi e di grandi dimensioni e non come una vera e propria metodologia.

2.2 Indici di prestazione

Le prestazioni di un sistema di elaborazione possono essere quantificate da *figure di merito* o da *indici di prestazione* che descrivono l'efficienza dello svolgimento delle sue funzioni. Nel caso delle tecniche di misurazione gli indici di misurazione del sistema vengono appunto *misurati*, mentre nel caso relativo alle tecniche modellistiche vengono *calcolati* (applicando e risolvendo modelli analitici) o *stimati* (utilizzando ed eseguendo modelli di simulazione).

Gli indici di prestazione di un sistema di elaborazione che spesso vengono utilizzati sono il *tempo di risposta* (o *tempo di esecuzione*) ed il *throughput*. Il loro utilizzo dipende essenzialmente

dall'ambito specifico in cui si sta effettuando la valutazione delle prestazioni. Nel caso delle prestazioni di calcolatori o di software viene utilizzato il primo, mentre il secondo ha un'applicazione maggiore nell'analisi delle prestazioni di reti o centri di calcolo. Il *tempo di risposta* è il lasco di tempo che intercorre tra l'inizio e la terminazione di un lavoro, mentre il *throughput* rappresenta la quantità di lavoro eseguita nell'unità di tempo.

Nel caso specifico del tempo di risposta, per massimizzare le prestazioni è necessario minimizzare il tempo di esecuzione necessario per svolgere un determinato lavoro. Prestazioni e tempo di risposta possono essere correlati in riferimento ad una stessa entità: infatti la prestazione è il reciproco del tempo di esecuzione. Nel caso di due entità distinte X e Y, se le prestazioni di X sono migliori di quelle di Y allora il tempo di esecuzione di Y è più elevato di quello di X, ovvero X è più veloce di Y.

È possibile correlare anche le *prestazioni* delle due entità in modo quantitativo:

$$n = \frac{Prestazioni\ X}{Prestazioni\ Y} = \frac{T^{Y}_{esecuzione}}{T^{X}_{esecuzione}}$$

Questo significa che le prestazioni di X sono n volte migliori di quelle di Y, o che il tempo di esecuzione di Y è n volte maggiore di quello di X. Tale indice n può anche essere espresso sotto forma di valore percentuale:

$$n(\%) = 100\left(\frac{Prestazioni\ X - Prestazioni\ Y}{Prestazioni\ Y}\right) = 100\left(\frac{T^{Y}_{esecuzione} - T^{X}_{esecuzione}}{T^{X}_{esecuzione}}\right)$$

2.3 Modelli di monitoraggio delle prestazioni delle applicazioni Java

In questa sede si vuole citare la possibilità di usufruire di famiglie di modelli di analisi delle prestazioni basati su *Java*. Nello specifico, questi modelli possono essere applicati per lo sviluppo e la valutazione delle prestazioni di componenti software sviluppati in *Java*, come lo sono ad esempio *Jadex* e *Jason*. Contrariamente ad una implementazione manuale, questi modelli

forniscono una strumentazione automatica in grado di far risparmiare tempo, e che introduce un *overhead* minimo ed una elevata *precisione* nella determinazione delle misure.

La caratteristica principale di questi modelli è quella di porsi al di sopra del sistema operativo, o al più di affiancare la *JVM (Java Virtual Machine)*, a differenza di un usuale strumento di misura scritto in *Java* che lavora al di sopra di essa. Una tale proprietà giustifica la precisione ottenibile riguardo la misura del tempo di esecuzione, in quanto vengono eliminati i ritardi di comunicazione tra la *JVM* e il *sistema operativo*. La granularità dipende quindi solo dagli intervalli di scheduling dei processi definiti dal sistema operativo.

Uno svantaggio di questi modelli riguarda il fatto che essi consentono solo una *visione d'insieme* (e non specifica) relativamente al contesto da andare a misurare: ad esempio, non permettono di andare a determinare due intervalli da cui ricavare una misura temporale.

2.4 Modello di analisi delle prestazioni di sistemi ad agenti mobili

Lo studio di metodologie e strumenti che permettano l'analisi e la predizione delle prestazioni di sistemi caratterizzati dall'estrema complessità degli ambienti hardware e software impiegati è un problema di ricerca aperto. Tali ambienti di calcolo sono altamente *eterogenei* (a livello delle unità di I/O, dei sistemi operativi, delle reti di interconnessione) e caratterizzati da una particolare sensibilità delle prestazioni a fattori esterni (ad esempio il *carico di CPU* e il *traffico sulle reti utilizzate*).

Risulta quindi consigliato il ricorso a *metodi predittivi* che possano fornire indicazioni *quantitative*, consentendo di individuare la migliore strategia di impiego delle risorse dedicate all'applicazione. Le esperienze nello sviluppo e nella valutazione di prestazioni di sistemi distribuiti hanno mostrato la sostanziale validità di un approccio consistente nello sviluppo di applicazioni in un ambiente di simulazione piuttosto che sull'architettura target. La necessità di confrontarsi con un sistema complesso, basato sull'interazione di componenti eterogenei e con differenti risorse a disposizione, rende indispensabile l'utilizzo di strumenti di analisi e previsione delle prestazioni che permettano un opportuno dimensionamento del sistema.

L’utilizzo di *strumenti simulativi* in grado di predire il comportamento del sistema in differenti condizioni operative è una tecnica comprovata per raggiungere tale obiettivo. Tali strumenti sono in grado di riprodurre il comportamento del sistema, dal punto di vista prestazionale, nelle sue singole parti e nella sua complessità.

JADEX

Sviluppato da Alexander Pokahr, Lars Braubach e Winfried Lamersdorf

http://jadex-agents.informatik.uni-hamburg.de

3.1 Visione d'insieme

Al giorno d'oggi possiamo scegliere tra una moltitudine di piattaforme per sviluppare applicazioni multi-agente [11]. Tuttavia molte di queste piattaforme sono sviluppate focalizzandosi specificamente su diverse architetture (come quella *cognitiva* o quella a *infrastruttura*, ad esempio), quindi non tutti gli aspetti della tecnologia ad agenti sono equamente coperti. L'applicabilità generale di una piattaforma ad agenti per una grande varietà di domini richiede che siano considerate almeno tre categorie di requisiti: l'*openness* (apertura), il *middleware* e il *reasoning* (ragionamento). L'apertura è strettamente legata alla visione di reti interconnesse di applicazioni originariamente non collegate, mentre gli aspetti inerenti il middleware coinvolgono invece fattori riguardanti la gestione dei servizi, la sicurezza e la persistenza. Il ragionamento, infine, si concentra nei processi di decisione interni agli agenti.

Concordemente a questi aspetti, le piattaforme esistenti possono essere classificate in due gruppi abbastanza distinti. Da un lato, le piattaforme conformi agli standard *FIPA* si rivolgono principalmente alle questioni riguardanti l'apertura e il middleware. Dall'altro esistono invece piattaforme incentrate sul *ragionamento* che si focalizzano prevalentemente sul modello comportamentale di un singolo agente, per esempio per cercare di ottenere più razionalità e immediatezza nei vari goal. Il gap presente tra queste due tipologie di piattaforme è uno dei principali motivi che hanno portato alla realizzazione del motore di ragionamento BDI *Jadex*, che mira a mettere insieme questi due importanti filoni di ricerca.

Al di là di questo obiettivo generale, il design del sistema è comunque guidato da due fattori molto importanti. Da un lato, lo sviluppo del motore di ragionamento è accompagnato da un continuo sforzo atto a migliorare l'architettura BDI in generale. Dall'altro il sistema rispetta l'attuale stato dell'arte riguardante lo sviluppo *object-oriented*, ed è quindi disegnato per essere usato non solo dagli esperti di *AI* ma anche dai normali sviluppatori di software. Proprio per questo lo sviluppo di agenti in *Jadex* si basa su tecniche diffuse come *Java* e *XML*, come vedremo meglio nel prosieguo della tesi.

3.2 Architettura

Questa sezione presenta l'*architettura* del sistema Jadex. Inizieremo con una breve descrizione del modello BDI e dei sistemi collegati, per passare poi a presentare gli aspetti architetturali veri e propri. I concetti base del sistema sono introdotti sottolineando le loro principali caratteristiche e differenze con altri sistemi BDI. Per ultimo andremo a parlare del modello di esecuzione, mostrando come le componenti del sistema interagiscono tra di loro.

3.2.1 Modelli e sistemi BDI

Il *modello BDI* è stato inizialmente concepito da *Michael Bratman* come teoria del ragionamento umano pratico [12]. Il suo successo è basato sulla semplicità con cui si riduce la spiegazione della struttura per i comportamenti umani più complessi ad un molto più semplice atteggiamento motivazionale. In questo modello le cause delle azioni sono legate solamente ai *desideri*, ignorando completamente altri aspetti della percezione come ad esempio le *emozioni*. Un altro punto di forza del *modello BDI* è l'uso consistente di nozioni psicologiche tradizionali, che corrispondono quasi perfettamente al modo in cui la gente comunica attrraverso i normali comportamenti umani.

La teoria BDI di *Rao* e *Georgeff* [13] definisce convinzioni *(belief)*, desideri *(desires)* e intenzioni *(intentions)* come atteggiamenti mentali rappresentati come possibili stati del mondo. Per un agente le intenzioni sono sottoinsiemi delle convinzioni e dei desideri, per essere più chiari

possiamo dire che un agente agisce nei riguardi di alcuni stati che desidera siano veri e che crede siano possibili. *Rao* e *Georgeff* proposero anche varie semplificazioni della teoria che permettessero di essere trattabili computazionalmente, e la più importante dice che solo i *belief* debbano essere rappresentati esplicitamente. I desideri sono ridotti a eventi che sono gestiti da modelli di piani predefiniti, mentre le intenzioni sono rappresentate implicitamente dallo stack di runtime dei piani che devono essere eseguiti.

Nei prossimi paragrafi verrà descritta l'architettura del motore di ragionamento di *Jadex* (che in sostanza segue il modello computazionale *PRS*) sottolineando dove necessario anche alcune importanti differenze con altre piattaforme (sempre di tipo PRS).

3.2.2 Concetti in Jadex

Nella *figura 3.1* viene presentata una visione d'insieme dell'architettura astratta di *Jadex*. Visto da fuori, un agente è una *black box* che riceve e manda messaggi. Come è comune nei sistemi PRS tutti i tipi di eventi, come ad esempio messaggi in arrivo o eventi relativi a goal, servono come input al meccanismo interno di reazione e delibera, che distribuisce gli eventi ai piani selezionati dalla plan library. In *Jadex*, il *meccanismo di reazione e delibera* è l'unico componente globale di un agente. Tutte le altre componenti sono raggruppate in moduli riutilizzabili chiamati *capabilities* (capacità).

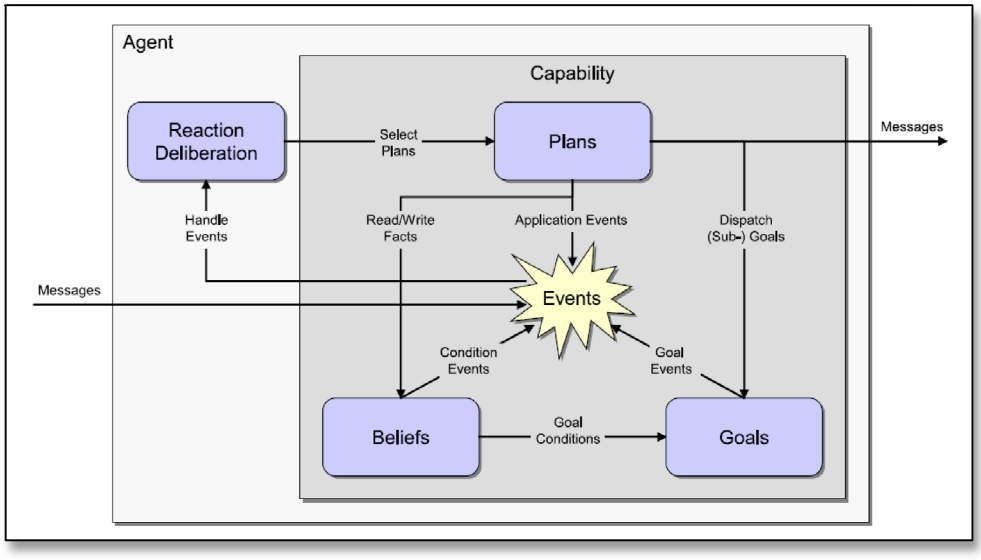

Figura 3.1 L'architettura astratta in Jadex (figura tratta da [5])

Belief Un obiettivo del progetto *Jadex* è l'adozione di una prospettiva di software engineering per descrivere gli agenti. In altri sistemi BDI, i *belief* sono rappresentati in predicati logici di primo ordine (per esempio in *Jason*, che analizzeremo nel prosieguo del lavoro di tesi) o usando modelli relazionali (per esempio in *JACK* e *JAM* [14]). In *Jadex* viene invece impiegata una rappresentazione dei belief di tipo object-oriented, dove oggetti arbitrari possono essere memorizzati come cosiddetti *fatti* (chiamati appunto belief) oppure come cosiddetti *insiemi di fatti* (chiamati belief-sets). Le operazioni contro la belief-base possono essere formulate in un linguaggio di query descrittivo di tipo *set-oriented* (vedi *OQL*, ad esempio). Inoltre, la belief-base non è solamente un data store passivo, ma prende parte attiva nell'esecuzione dell'agente monitorando le condizioni dei belief. Le modifiche dei belief possono quindi portare direttamente ad azioni quali la generazione di eventi o la creazione od eliminazione di goal.

Goal I goal sono un concetto fondamentale in *Jadex*, seguendo l'idea generale che *essi sono desideri concreti e momentanei di un agente*. Per qualsiasi goal a lui assegnato, un agente si impegnerà più o meno direttamente in azioni adatte fino a che non considererà il suddetto goal raggiunto, non più raggiungibile o non più voluto. In altri sistemi PRS i goal sono rappresentati come un tipo speciale di evento. In *Jadex*, invece, i goal sono rappresentati come *oggetti espliciti* contenuti in una *goal-base*, la quale è accessibile alla componente di ragionamento così come anche ai piani se essi hanno bisogno di conoscere o vogliono modificare i goal correnti dell'agente.

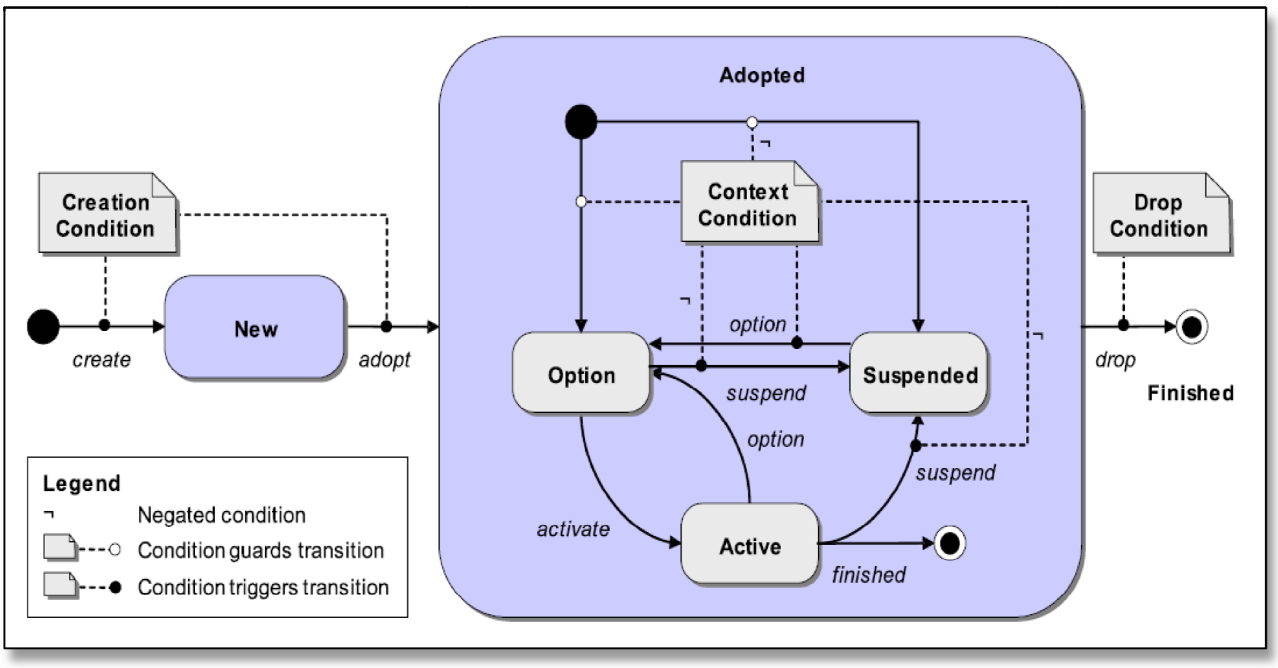

Figura 3.2 Ciclo di vita di un goal in Jadex (figura tratta da [5])

Dal momento che i goal sono rappresentati separatamente dai piani, il sistema può *conservare* i goal che non sono correntemente associati a nessun piano. Di conseguenza, a differenza di altri sistemi BDI, Jadex non richiede che tutti i goal adottati siano consistenti l'uno con l'altro, purché solo i sottoinsiemi consistenti di questi goal siano perseguiti in ogni momento. Per distinguere tra goal appena adottati e goal portati avanti attivamente, viene introdotto un *ciclo di vita del goal* che consiste degli stati *option*, *active* e *suspend* (vedi *figura 3.2*, tratta da [15]). Quando un goal viene adottato diventa un'opzione che viene aggiunta alla *goal-base* dell'agente (come top-level goal, o anche come sotto-goal quando viene creato da un piano).

Jadex supporta quattro tipi di goal che ne estendono il ciclo di vita generico e assumono comportamenti diversi rispettivamente a cosa devono processare. Un *perform goal* è relazionato direttamente all'esecuzione delle azioni, di conseguenza il goal si considera raggiunto quando alcune azioni sono state eseguite (a prescindere dal loro esito). Un *achieve goal* è un goal nel senso tradizionale, definisce cioè uno stato senza però specificare come raggiungerlo (gli agenti si trovano solitamente a dover provare vari piani alternativi prima di raggiungere un goal di questo tipo). Un *query goal* è simile ad un achieve goal ma lo stato desiderato è interno all'agente e riguarda la disponibilità di alcune informazioni che l'agente vuole conoscere. Nei *maintain goal*, infine, un agente tiene traccia di uno stato desiderato e continua ad eseguire piani appropriati per ristabilire il suddetto stato quando necessario. Discuteremo più in dettaglio dei vari tipi di goal nel prosieguo del lavoro di tesi.

Piani I piani rappresentano gli elementi comportamentali di un agente, e sono composti da un *header* e da un *corpo*. La specifica dell'header è simile a quella di altri sistemi BDI e riguarda principalmente le circostanze sotto le quali un piano dovrebbe essere selezionato. Inoltre, nell'header può essere anche dichiarata una condizione di contesto che deve necessariamente essere vera perché il piano continui a rimanere in esecuzione. Il corpo fornisce invece una linea d'azione predefinita, espressa in *linguaggio procedurale*. Questa linea d'azione deve essere eseguita dall'agente quando il piano è selezionato per l'esecuzione e può contenere azioni fornite dall'*API* di sistema (per esempio inviare messaggi, manipolare belief o creare sotto-goal).

Capabilities Le capacità (*capabilities*, appunto), introdotte in [16], rappresentano un meccanismo di raggruppamento per gli elementi di un agente BDI come belief, goal, piani ed

eventi. In questo modo gli elementi strettamente collegati possono essere messi assieme in un modulo riutilizzabile che *incapsula* una certa funzionalità. La capacità inclusiva di un elemento rappresenta la sua portata, e un elemento può avere accesso solamente ad elementi della stessa portata (ad esempio, un piano può accedere solamente a belief o gestire goal o eventi della stessa capacità). Per collegare fra loro diverse capacità, *Jadex* fornisce meccanismi di importazione/esportazione molto flessibili che permettono di definire l'interfaccia esterna delle varie capacità.

3.2.3 Modello di esecuzione

Questo paragrafo ha lo scopo di mostrare l'operato della *componente di reazione e delibera*, dati i concetti BDI di *Jadex* descritti precedentemente. Tutte le funzionalità richieste sono assegnate a componenti separati (illustrati in *figura 3.3*), che verranno descritte una per volta. I messaggi in arrivo sono posizionati nella coda globale dei messaggi degli agenti dalla piattaforma sottostante (nel nostro caso *JADE*, ma come vedremo potrebbe essere anche un'altra piattaforma). Prima che il messaggio possa essere inoltrato al sistema deve essere assegnato ad una *capability*, che è appunto in grado di gestire il messaggio. Se il messaggio fa parte di una conversazione ancora in corso, viene creato un evento del messaggio in arrivo nell'esecuzione della conversazione, in *runtime*. In caso contrario deve essere trovata un'adeguata capacità, e questa operazione viene fatta confrontando il messaggio con i modelli degli eventi definiti nella base eventi di ogni capacità. Il modello che dopo il confronto risulta essere il più adeguato è poi usato per creare un evento appropriato nel raggio della capacità. In ogni caso, l'evento creato è in seguito aggiunto alla lista eventi globale dell'agente.

Il *dispatcher* è il responsabile della selezione di piani applicabili per gli eventi dalla lista eventi. Questa operazione viene portata a termine in *due passi*: innanzi tutto viene generata una lista dei piani validi confrontando l'evento con gli *header* dei piani come definito nella plan-base di ogni capacità, in base a cui solo quelle capacità in cui l'evento è visibile devono essere considerate. Il secondo passo è quello di selezionare un sottoinsieme dei piani validi per essere eseguiti. Riguardo questo secondo passo sorgono varie importanti considerazioni, come ad esempio se tutti i piani applicabili debbano essere eseguiti concorrentemente o se invece l'evento debba essere

trasferito ad un altro piano se il primo fallisce. La decisione di quale piano eseguire è detta *ragionamento di meta-livello* e può essere semplice come selezionare il primo piano dalla lista, ma talvolta anche estremamente complicata come trovare ed eseguire meta-piani per la decisione. A questo proposito, *Jadex* fornisce *impostazioni flessibili* per permettere di influenzare questo processo. Di default, i messaggi sono trasferiti ad un solo singolo piano, mentre per i goal molti piani sono eseguiti in sequenza finchè il goal non è raggiunto o finchè non fallisce (cioè quando nessun piano è più valido). Gli eventi interni sono trasferiti a tutti i piani allo stesso momento, in quanto sono considerati solo come un cambio di notifica e non ci si aspetta nessun valore dato in ritorno. Una volta che i piani sono stati selezionati, vengono posizionati nella *ready list* in attesa di essere eseguiti.

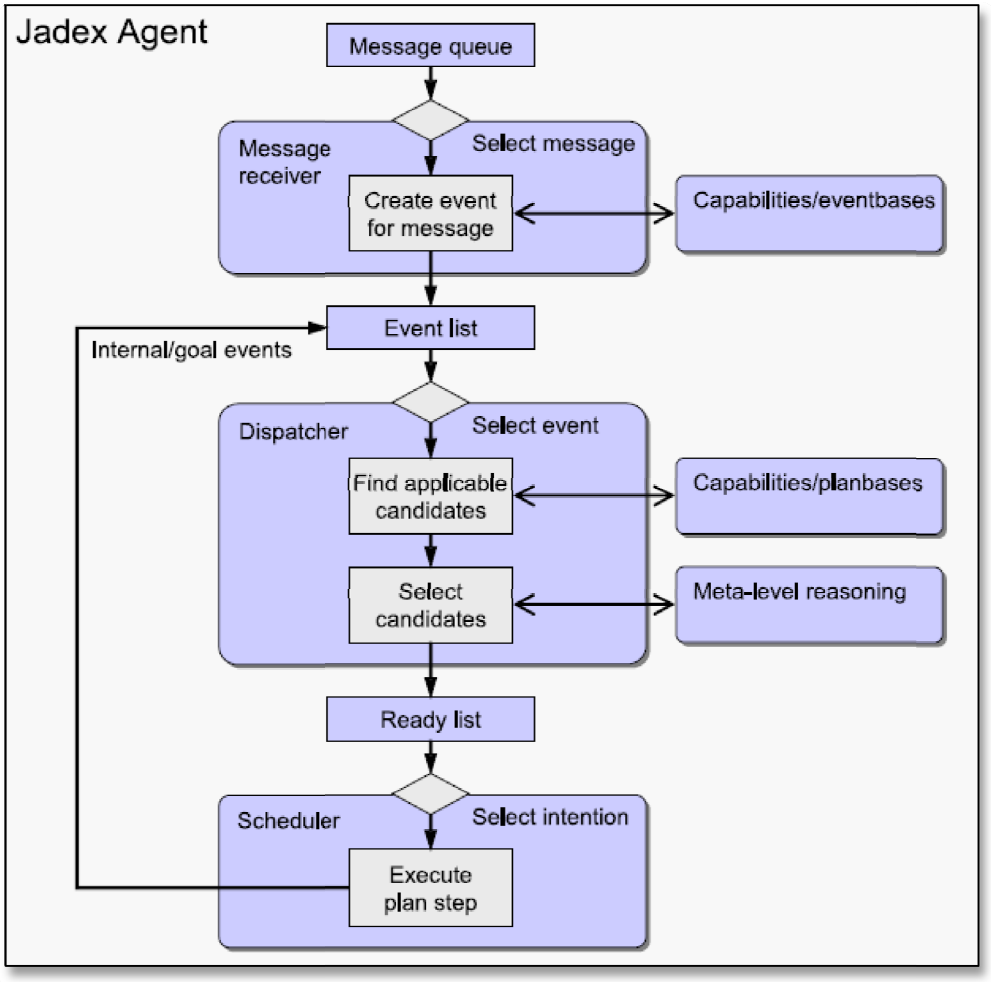

Figura 3.3 Modello di esecuzione in Jadex (figura tratta da [5])

L'esecuzione dei piani è affidata ad uno *scheduler*, il quale seleziona i piani dalla ready list. I piani vengono eseguiti *passo-passo*, e a differenza di altri sistemi PRS la lunghezza del passo del piano dipende anche dal contesto e non solo dal piano stesso. Un piano è eseguito solo fino a che

attende esplicitamente o riguarda significativamente lo stato interno di un agente (ad esempio creando o cancellando un goal). I cambiamenti interni di stato possono essere causati direttamente da effetti collaterali, ad esempio quando un cambiamento di un belief va a modificare la condizione di creazione di un goal. Dopo che un piano attende o viene interrotto, lo stato dell'agente può essere propriamente aggiornato.

3.3 Linguaggio

Jadex non si basa su di un nuovo linguaggio di programmazione ad agenti, e nemmeno ne utilizza o ne rivisita uno già esistente. Diciamo che viene invece scelto un *approccio ibrido* che distingue esplicitamente tra il linguaggio usato per gli agenti statici e quello usato per definire il comportamento degli agenti dinamici. In accordo con questa distinzione, un agente *Jadex* è formato da due componenti: un *Agent Definition File (ADF)* per le specifiche di belief, goal e piani così come dei loro valori iniziali, e dall'altra parte un *piano di codice procedurale* (vedi *figura 3.4*). Per definire gli *ADF* usiamo, come detto in precedenza, il linguaggio *XML*. Le specifiche della struttura *XML* sono integrate da un linguaggio di espressioni dichiarative che possono servire, ad esempio, a specificare le condizioni dei goal. La parte procedurale dei piani (il *corpo*) è realizzata invece in *Java*, e ha accesso alle strutture BDI di un agente attraverso un'interfaccia di programmazione[4].

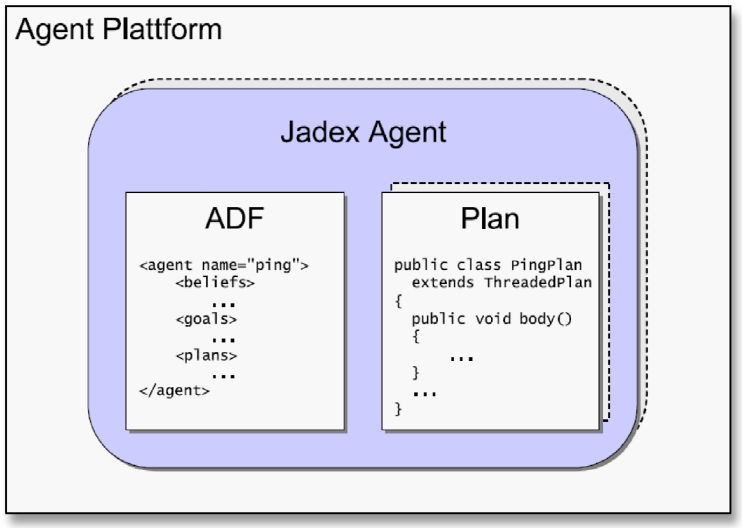

Figura 3.4 Un agente in Jadex (figura tratta da [5])

[4] API, Application Program Interface

3.3.1 Specifiche e aspetti sintattici

Il *metamodello BDI* di *Jadex* definito in schema *XML* è molto esteso, quindi non potrà essere trattato nella sua interezza in questa tesi (si rimanda a [17] per una trattazione più approfondita). Generalmente, il linguaggio corrispondente è stato specificato con due principi di design fondamentali in mente. Il primo obiettivo di design è il *supporto per la rappresentazione esplicita di tutti i tipi di elementi*, siano essi belief, goal o eventi. Di conseguenza ciò richiede che gli utenti scrivano degli *ADF* molto dettagliati, ma d'altro canto permette di attuare dei test di consistenza molto più rigorosi dei modelli degli agenti. Inoltre, certi tipi di guasti possono essere trovati e risolti molto più semplicemente anche in fase di *runtime* (ad esempio, il tentativo di memorizzare un valore dato in un belief non definito può essere subito segnalato e corretto).

Il secondo obiettivo di design è quello di *aumentare il potere espressivo degli ADF* per i seguenti propositi: la creazione arbitraria e complessa di oggetti (ad esempio, valori tra belief o parametri), la descrizione di condizioni booleane (ad esempio, quando un dato goal dovrebbe essere scartato) e la costruzione di query (ad esempio, per il reperimento di valori dal belief-base). Per ottenere i propositi appena descritti viene usato un linguaggio espressivo integrato per specificare parti del modello dell'agente, e non può essere facilmente rappresentato in *XML*. Queste espressioni sono usate nella parte *XML* dell'ADF ogni volta che i valori devono essere ottenuti per certi elementi in runtime (ad esempio valori di belief, condizioni di goal, ecc). Le espressioni devono assolutamente essere *prive di effetti collaterali*, proprio perché sono spesso valutate internamente dal sistema. Il linguaggio è stato studiato per conformarsi perfettamente con la sintassi delle espressioni *Java* (termine di destra dell'assegnazione), estesa con un sottoinsieme delle istruzioni *Object Query Language (OQL)* [18]. La sintassi del linguaggio *OQL* permette di creare query nella classica forma *select-from-where*. Nonostante le query possano essere usate in qualsiasi espressione, sono più utili per viste predefinite su alcuni sottoinsiemi dei belief dell'agente, che può quindi essere valutato anche in fase di *runtime*.

Qui di seguito i concetti BDI fondamentali descritti precedentemente saranno ripresi dettagliandone la loro realizzazione a livello di linguaggio. Questi concetti sono specificati come parte della descrizione di un agente o di una capacità alla stessa maniera. Nella *figura 3.5* sono mostrati gli attributi possibili e i vari sub-tag degli agenti. Ogni tipo di agente è identificato da un

nome e da un *pacchetto*, e può essere messo a disposizione con un testo descrittivo. Inoltre possono essere settate le *proprietà di runtime* e la *classe dell'agente* (nella maggior parte dei casi comunque i valori di default sono più che sufficienti e non necessitano di essere modificati).

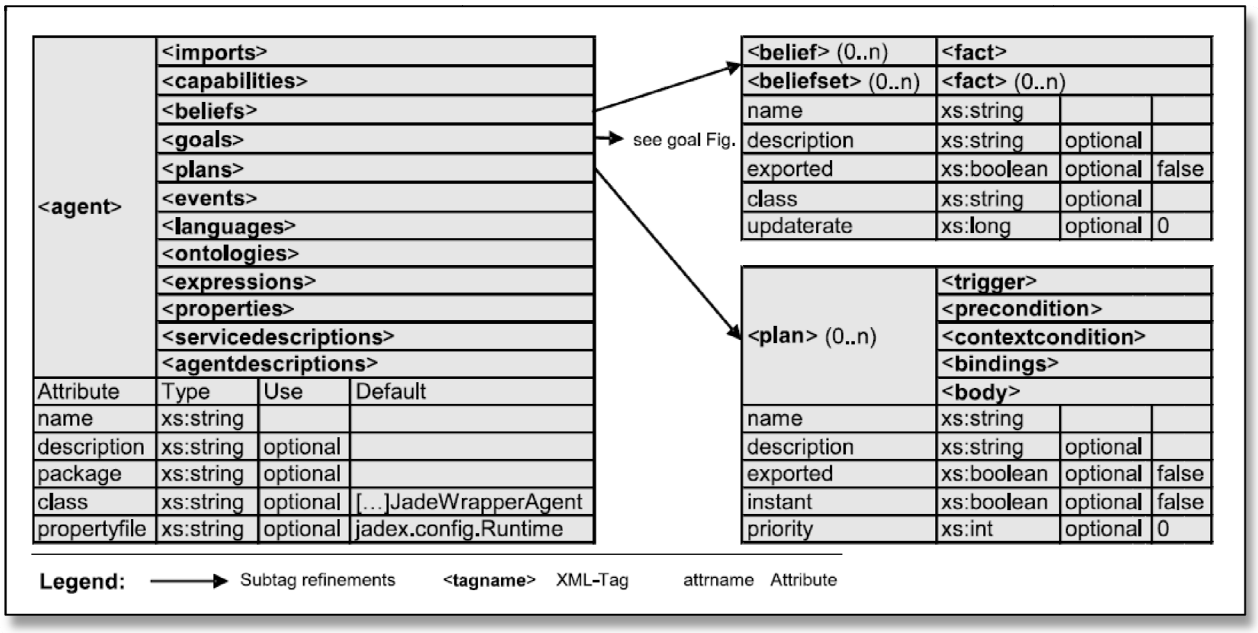

Figura 3.5 Frammento del metamodello XML di un agente in Jadex (schema tratto da [17])

Belief In *Jadex*, i *belief* sono rappresentati in modo object-oriented permettendo ad arbitrari oggetti *Java* di essere memorizzati come *fatti*. Come tutti gli elementi di una capacità, i belief e gli insiemi di belief possono essere forniti con un *nome*, un *testo descrittivo* e un *flag* per l'esportazione (esportare un elemento lo rende accessibile ad agenti o capacità esterni, ma questa funzionalità è disattivata di default). Per i belief e gli insiemi di belief deve inoltre essere definita la classe *Java* per i fatti. Il valore di un fatto deve essere dichiarato nel linguaggio espressivo come statico o dinamico, dove ad esempio i fatti dinamici possono essere utili per rappresentare valori continuamente influenzati dall'ambiente o da aspetti temporali. Il ricalcolo di tali fatti dinamici avviene al momento dell'accesso e anche a intervalli di tempo prestabiliti. In fase di runtime, i belief e gli insiemi di belief sono accessibili da piani interni attraverso operazioni sulla belief-base e anche emanando query *OQL*.

Goal Come descritto precedentemente, in *Jadex* i goal si distinguono in *quattro tipi*: *perform*, *achieve*, *maintain* e *query*. Tutti questi tipi sono basati sul ciclo di vita generico di un goal, e quindi mostrano molte proprietà comuni che sono riassunte in *figura 3.6*.

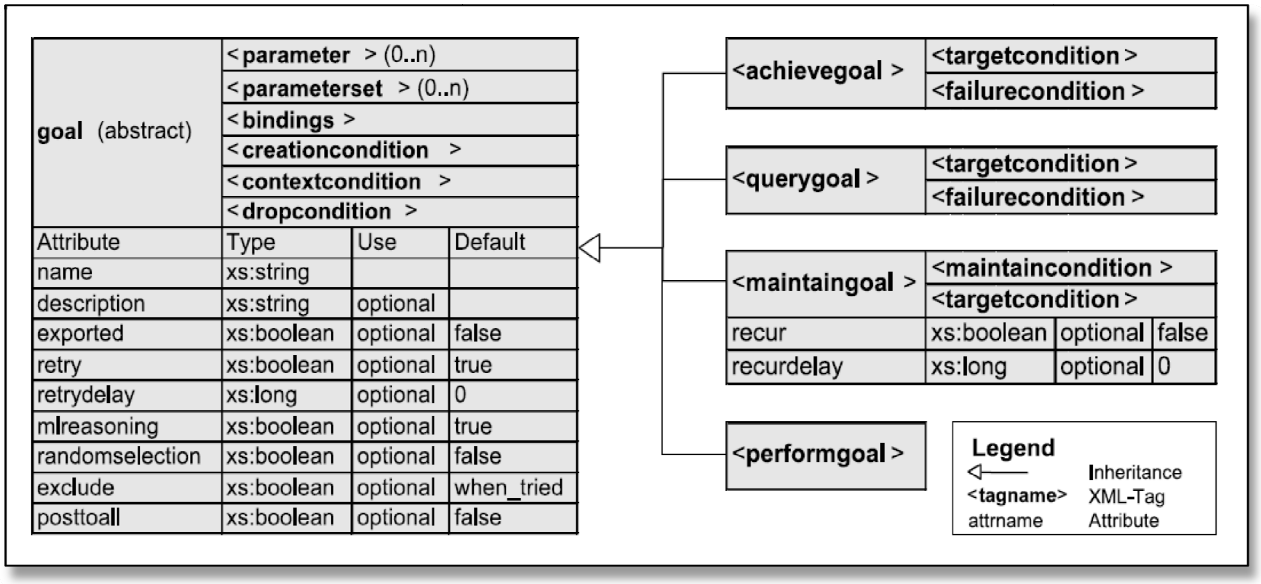

Figura 3.6 Metamodello XML dei goal in Jadex (schema tratto da [17])

Concordemente al ciclo di vita di un goal la creazione, l'abbandono e le condizioni di contesto possono essere specificate come *espressioni booleane*. La personalizzazione dei tipi di goal può essere ulteriormente ottenuta definendo vari parametri (detti *in*-, *out*- e *inout*-) che sono usati per trasferire informazioni tra il creatore di un goal e i piani che lo processano. La processazione in fase di runtime dei goal può essere raffinata usando vari flag BDI, che tra le altre cose controllano se un goal viene riprocessato quando un piano fallisce, se viene usato del ragionamento di meta-livello e se vari piani validi sono testati in sequenza o in parallelo. Una spiegazione più completa si può trovare in [17].

Dal tipo di goal astratto sono poi derivati tutti i tipi di goal concreti. Il più semplice è sicuramente il *perform goal*, che viene usato per eseguire (anche ripetutamente) certe azioni e non richiede nessun tipo di dati aggiuntivi. Un *achieve goal* estende il tipo di goal astratto, aggiungendo il supporto per la specifica di una *condizione di target* e di una *condizione di failure*. La condizione di target è usata per descrivere lo stato che il goal mira a raggiungere come un'espressione booleana. Allo stesso modo, una condizione booleana di failure ha lo scopo di abortire un goal nel momento in cui il suo raggiungimento sia diventato impossibile. Un *query goal* fornisce lo stesso tipo di condizioni, ma assume un comportamento leggermente diverso in quanto viene usato per scopi di reperimento di informazioni.

I comportamenti più complessi sono esposti nei goal di tipo *maintain*, che sono usati per monitorare uno specifico stato e automaticamente tentare di ristabilirlo nel caso diventi invalido. Una condizione di target booleana viene usata per raffinare lo stato che si sta cercando di ripristinare. I maintain goal non vengono abbandonati una volta che sono stati raggiunti, ma rimangono invece inattivi fino a che lo stato monitorato non viene violato nuovamente. Inoltre un maintain goal può essere configurato per tentare di essere ripristinato a intervalli di tempo prestabiliti, quando per qualche ragione ha fallito.

Piani La dichiarazione dei piani in *Jadex* è molto simile ad altri sistemi PRS, e richiede che siano specificate nell'ADF tutte le circostanze sotto le quali un piano è definito valido. Possono essere forniti *trigger*, *eventi interni*, *messaggi*, *goal*, così come *condizioni di stato per i belief*. Per semplificare il raggiungimento dei goal può risultare vantaggioso talvolta creare varie *istanze parametrizzate* di un tipo di piano, e provarle una dopo l'altra fino a che il piano non ha successo. A questo proposito possono essere specificati dei parametri vincolanti che poi verranno usati per la configurazione del piano.

```
01: /** Plan skeleton for an application plan. */
02: public class SomePlan extends jadex.runtime.Plan {
03:
04:   public void body() {
05:     // Plan code.
06:   }
07:
08:   public void passed() {
09:     // Optional cleanup code in case of a plan success.
10:   }
11:   public void failed() {
12:     // Optional cleanup code in case of a plan failure.
13:   }
14:   public void aborted() {
15:     // Optional cleanup code in case the plan is aborted.
16:   }
17: }
```

Figura 3.7 Esempio di ossatura di un piano (tratto da [5])

Il corpo del piano deve essere fornito come espressione per la creazione di un'istanza del piano adatta. Attualmente *Jadex* supporta due tipi di corpi per i piani (*standard* e *mobile*), ed entrambi richiedono che sia implementata una classe *Java*. I corpi di tipo *mobile* hanno vari svantaggi se comparati alle versioni *standard*, ma hanno comunque senso negli scenari in cui gli agenti devono migrare tra i vari host. In *figura 3.7* viene raffigurata l'ossatura di un semplice piano. Come si può notare, è obbligatoria l'estensione di una corrispondente classe *framework* (*Plan*) e l'implementazione di un *metodo astratto* body(), nel quale va inserito il comportamento del piano per il dominio specifico. Oltre al suddetto metodo body(), possiamo notare la presenza di altri tre metodi che possono essere implementati opzionalmente nelle relative circostanze di *successo*, *fallimento* e *aborto*.

3.3.2 Espressività e semplicità

L'obiettivo generale del progetto *Jadex* è quello di fornire un sofisticato modello di ragionamento che permetta di sviluppare agenti intelligenti arbitrariamente complessi. Quindi, nonostante il sistema cerchi di essere il più usabile possibile, non sacrifica comunque l'espressività per la semplicità. Nondimeno, le questioni inerenti il *software engineering* giocano un ruolo fondamentale nel sistema.

Come detto prima, un obiettivo primario del progetto è quello di *facilitare la transizione dallo sviluppo di software object-oriented ad un approccio agent-oriented*. Questo si ottiene facendo affidamento su tecniche già rodate e affinate quando e dove possibile: il sistema infatti si basa principalmente su *Java* e *XML*, quindi lo sviluppatore non dovrà imparare un nuovo linguaggio ma solo affinare conoscenze già acquisite. Un altro vantaggio è che lo sviluppatore potrà continuare ad operare in un ambiente familiare, e dal momento che sono necessari solo file *Java* e *XML* possono essere usati anche ambienti di sviluppo già esistenti ed integrati (come ad esempio *Eclipse*) per sviluppare un agente *Jadex*.

Il sistema fornisce inoltre anche avanzate caratteristiche di *software engineering*, come ad esempio controlli su *riusabilità* e *consistenza*. È possibile anche *incapsulare* le funzionalità di un agente in un modulo riusabile mantenendo comunque il livello di astrazione degli elementi BDI.

L'insieme di tutte queste caratteristiche rende più semplice identificare gli errori (ad esempio, di battitura) nel minor tempo possibile.

3.4 La piattaforma

Questa sezione descrive la realizzazione del motore di ragionamento di *Jadex* e la sua integrazione nella piattaforma *JADE*. La *figura 3.8* mostra i componenti essenziali per sviluppare ed eseguire un agente *Jadex*, e sottolinea anche le dipendenze tra le varie componenti. Le componenti sono state suddivise in *Core System Components* (riga più in alto) che realizzano il motore di ragionamento, in *System Interface Components* (riga di mezzo) che forniscono e definiscono i punti di accesso del sistema, e infine in *Custom Application Components* (riga più in basso) che devono essere fornite dallo sviluppatore di agenti. I collegamenti tra le varie componenti si distinguono invece in *dipendenze di runtime* (per esempio nelle prime due colonne da sinistra), *dipendenze applicabili solamente durante la fase di avvio di un agente* (vedi le componenti nella terza colonna), e dipendenze *risolte al momento dell'ideazione del design* (vedi colonna più a destra).

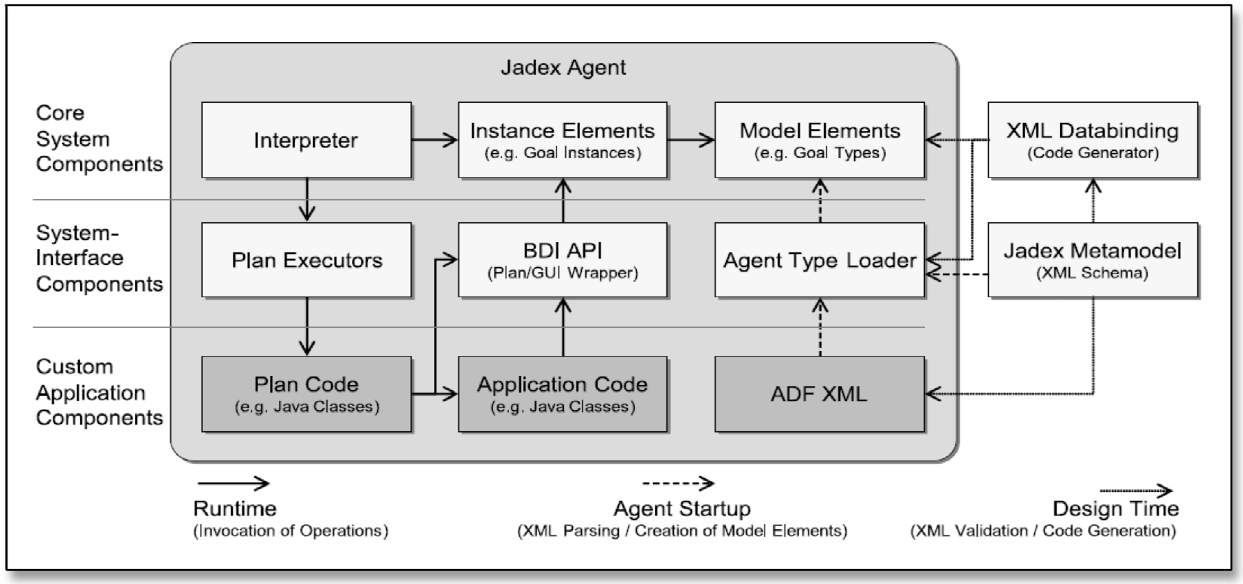

Figura 3.8 Realizzazione del sistema (figura tratta da [5])

Le componenti verranno descritte partendo dall'estrema destra. *Jadex* è basato su di un *metamodello BDI* definito da uno schema *XML*. Questo schema è usato da un lato per validare i

modelli degli agenti specificati negli *Agent Definition Files (ADF*). Dall'altro lato, un *framework di databinding XML (JBind)* viene usato per generare le classi *Java* per gli elementi del meta-modello e per leggere gli elementi *XML*. Quando un agente viene istanziato, l'*Agent Type Loader* generato legge il modello *XML* dell'agente fornito dall'utente e crea automaticamente i corrispondenti elementi del modello.

Da questi elementi in fase di runtime sono poi continuamente create varie istanze, rappresentate appunto dalle istanze degli elementi *(Instance Elements)* nella figura precedente. L'*interprete* opera sulle istanze degli elementi correnti, ed esegue piani per gestire eventi e goal. I *Plan Executors* sono usati per nascondere i dettagli implementativi del piano al resto del sistema. Di default c'è un *Plan Executor* per eseguire il codice del piano scritto in *Java*. Questo codice può avere accesso a qualsiasi altra applicazione o libreria di terze parti. Sia il piano che il codice dell'applicazione hanno accesso al motore di ragionamento attraverso una *API BDI*.

Per l'integrazione in *JADE* il tool di gestione della piattaforma *(RMA)* è stato leggermente esteso per permettergli così di supportare il lancio di agenti *Jadex*, selezionando il corrispondente modello di agente con un selezionatore di file. *L'interprete Jadex è di per sé realizzato come un tipo speciale di agente JADE*, il quale carica un modello di agente fornito all'avvio e crea la sua propria istanza del motore di ragionamento in accordo con le impostazioni stabilite dal modello (belief, goal e piani iniziali). Le funzionalità corrispondenti alle componenti del modello di esecuzione (*message receiver*, *dispatcher*, *scheduler* e altri) sono implementate come comportamenti ciclici, sempre in esecuzione all'interno dell'agente. Questi comportamenti dicono al motore di ragionamento di processare i messaggi in arrivo, nonché di effettuare i ragionamenti interni all'agente. In ogni ciclo di agenti di *JADE*, il motore di ragionamento è chiamato a processare un evento e ad eseguire un passo del piano.

3.4.1 Strumenti e documentazione

La distribuzione contiene una *documentazione completa* per iniziare subito e da usare in seguito come punto di riferimento. Sono inoltre presenti varie applicazioni esemplificative e il loro codice sorgente commentato. Una *guida utente* fornisce una sistematica panoramica di tutte le

funzionalità, e serve anche da manuale di riferimento. Sono inoltre forniti anche i *Java Docs* per la parte di programmazione e un riferimento al modello definito dallo schema *XML*.

Dal momento che un agente *Jadex* è comunque un agente *JADE*, *tutti gli strumenti di runtime forniti dalla piattaforma JADE possono essere usati anche con la piattaforma Jadex*. Per permettere un testing efficace per gli agenti *Jadex* sono stati sviluppati anche vari *tool aggiuntivi*, come un *logger* e un *introspettore*: il primo permette di reindirizzare l'output di un agente ad un singolo punto di responsabilità, mentre l'introspettore permette di visualizzare e modificare i concetti BDI di ogni singolo agente direttamente in fase di *runtime*, oltre che abilitare il *debugging step-by-step*.

3.4.2 Osservanza degli standard, interoperabilità e portabilità

Un aspetto fondamentale dello sviluppo di *Jadex* è stata la necessità di una piattaforma conforme allo *standard FIPA* che supportasse capacità di ragionamento BDI avanzate. Questa conformità è stata ottenuta grazie alla piattaforma *JADE*, che di fatto fornisce sofisticate implementazioni di tutte le più importanti specifiche *FIPA* (*figura 3.9*). Il motore di ragionamento di *Jadex*, realizzato sopra alla piattaforma *JADE*, di per sé supporta solamente *agenti omogenei* (agenti BDI, appunto), ma permette anche ad agenti basati su altri modelli di interoperare.

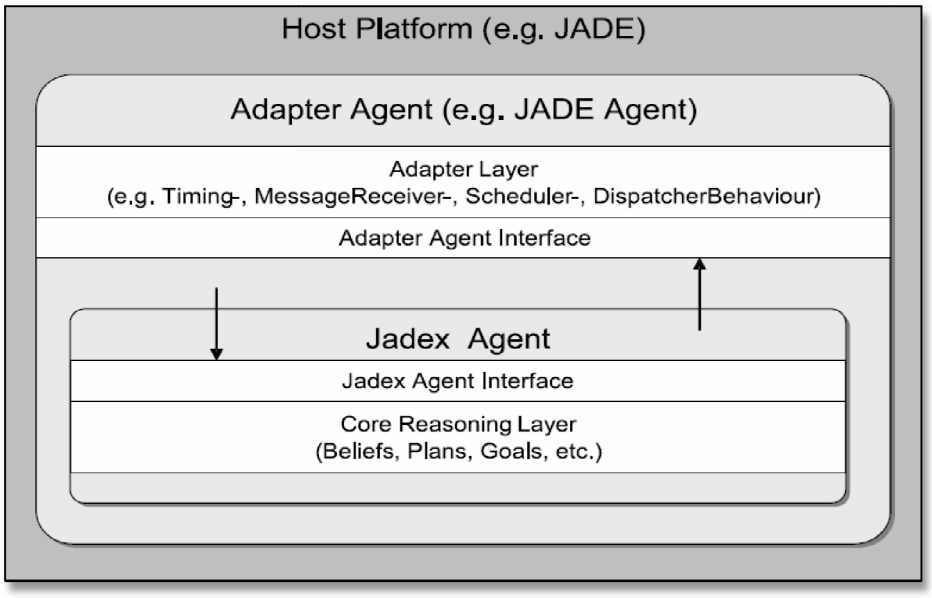

Figura 3.9 Integrazione della piattaforma (figura tratta da [5])

Gli agenti realizzati usando le convenzionali tecniche di programmazione di *JADE* possono essere eseguiti direttamente assieme ad agenti *Jadex* in esecuzione sulla stessa piattaforma. L'interoperabilità con altri tipi di agenti è semplice fintanto che questi agenti aderiscono agli standard FIPA[5].

Il *motore di ragionamento* è stato realizzato come componente separato, con l'obiettivo di limitare intenzionalmente le dipendenze dalla piattaforma sottostante. Per usare il motore di ragionamento su altre piattaforme va realizzato un *adattatore*: questo deve implementare una manciata di metodi usati da *Jadex* (ad esempio, per inviare messaggi) e deve inoltre richiamare il motore di ragionamento al momento opportuno. Per questo motivo, nonostante l'implementazione attuale sia stata realizzata per essere usata con *JADE*, il motore di ragionamento può essere abbastanza facilmente integrato con altre piattaforme ad agenti che aderiscono agli standard *FIPA*, come ad esempio *CAPA* o *ADK*. È anche possibile utilizzare il sistema congiuntamente ad altri ambienti middleware come *J2EE* o *.NET*, qualora gli standard *FIPA* non fossero richiesti.

Il motore è stato come detto realizzato in *Java* (*versione 1.4*) e include i pacchetti di terze parti *JBind* (per il data binding delle parti in *XML* relative agli *ADF* degli agenti) e *Apache Velocity* (per generare il contenuto di alcuni strumenti di dialogo). Per supportare i *dispositivi mobili* è anche disponibile una trasposizione del motore in una versione ridotta e basata su *J2ME/CDC*. Inoltre, tutti i tipi di strumenti e librerie con una *API Java* possono essere usati per fornire capacità aggiuntive.

3.5 Applicazioni supportate dal linguaggio e dalla piattaforma

Jadex è un ambiente di sviluppo multiuso per la creazione di applicazioni multi-agente che permette di costruire agenti con *comportamento reattivo* (basati cioè sugli eventi) e anche agenti con *comportamento deliberativo* (guidati dai goal). Non è limitato ad un target specifico, ma è

[5] In un esempio fornito con la piattaforma, degli agenti Jadex sono stati collegati con successo ad agenti in esecuzione su di una piattaforma CAPA, la quale serve a fornire un modello computazionale per agenti basato sulle reti di Petri.

stato usato per realizzare applicazioni in differenti campi quali *simulazioni*, *scheduling* e *calcoli su dispositivi mobili*.

Jadex ha avuto origine nel progetto tedesco chiamato *MedPAge* (*Medical Path Agents*, vedi [19] e [20]). Cooperando con l'*Università di Mannheim*, il progetto ha lo scopo di valutare i vantaggi dell'uso della tecnologia ad agenti nel contesto delle logistiche ospedaliere. In questo progetto, *Jadex* è usato per realizzare un'applicazione multi-agente per la negoziazioni di schede di trattamento nonché per la simulazione di un modello ospedaliero per testare il meccanismo di negoziazione.

In altri contesti *Jadex* è stato usato per realizzare applicazioni portatili per *PDA*. È stato sviluppato un *task planner mobile* personale per testare il porting di *Jadex J2ME* e per provare l'utilità degli agenti *BDI* su dispositivi mobili [21]. Inoltre, nel progetto *PITA (Personal Intelligent Travel Assistant)* sviluppato alla *Delft University of Technology*, *Jadex* è stato utilizzato per realizzare un prototipo di un'applicazione mobile che simula un assistente di viaggi personale [22].

Oltre a sviluppare specifiche applicazioni ad agenti, *Jadex* viene anche usato a *scopo didattico e per la ricerca riguardante lo sviluppo di software orientato agli agenti in generale*. Grazie al suo linguaggio semplice, basato come detto in precedenza su tecnologie molto diffuse come *Java* e *XML*, nonché all'ampia documentazione e agli innumerevoli esempi applicativi, *Jadex* si presta molto bene anche all'insegnamento agli studenti. È stato applicato con successo a diversi corsi all'*Università di Amburgo*, ed è anche insegnato in altri istituti. Riguardo alla ricerca in sistemi ad agenti, il progetto è stato anche pensato come strumento per aiutare i ricercatori a investigare sui concetti più appropriati nel design e nell'implementazione di sistemi ad agenti. La combinazione di schemi *XML* con tecniche di *databinding Java* permette al metamodello di *Jadex* di essere adattato in maniera molto flessibile ed esteso per scopi di sperimentazione.

3.6 Considerazioni finali

In questo capitolo abbiamo velocemente presentato gli aspetti principali del motore di ragionamento *BDI Jadex*. La realizzazione del sistema è motivata essenzialmente da tre fattori.

Innanzi tutto, il sistema mira a combinare i benefici del middleware ad agenti e dei processi di ragionamento interni degli agenti. In secondo luogo ha come obiettivo quello di migliorare lo stato dell'arte dell'architettura *BDI* rivedendo e migliorando alcuni difetti delle attuali piattaforme *BDI* ad agenti, come ad esempio la rappresentazione implicita dei goal. Infine, il sistema mira a rendere la tecnologia ad agenti molto più usabile sfruttando tecniche di programmazione molto diffuse come *XML*, *Java* e *OQL*.

L'architettura di *Jadex* è in linea di principio simile a quella dei tradizionali sistemi PRS, che prendono appunto in considerazione la processazione di eventi e di goal. Tuttavia ci sono delle differenze che riguardano principalmente la rappresentazione dei concetti *BDI* fondamentali, nonchè a livello di linguaggio. In accordo al requisito di *usabilità*, i belief sono espressi in stile *object-oriented* piuttosto che usando formule logiche o modelli relazionali. Inoltre, i goal sono rappresentati come specifiche entità durevoli. A livello di linguaggio, *Jadex* differenzia tra la descrizione del comportamento di un agente e la sua struttura statica, e per ognuno di questi due aspetti usa diversi linguaggi. La struttura statica di un agente è dichiarata in *XML*, mentre il linguaggio *Java* è usato per la parte di realizzazione del piano.

Attualmente il lavoro svolto per migliorare e sviluppare la piattaforma *Jadex* si sta concentrando su due aspetti fondamentali: *estensioni a concetti interni* e *supporto ad altri tool addizionali*. A livello concettuale sono state sviluppate varie estensioni ai meccanismi *BDI* di base, come ad esempio il supporto alla pianificazione, ai team e alla deliberazione dei goal. Si pensa di utilizzare la rappresentazione esplicita dei goal per migliorare l'architettura *BDI* con una struttura per la delibera dei goal, che servirà a ridurre la necessità di sviluppare agenti con un insieme iniziale di goal troppo consistente. Il lavoro sui *tool* invece si concentra principalmente nel cercare di migliorare l'usabilità del sistema. Per ottenere un maggior livello di usabilità si pensa di sviluppare, tra gli altri, anche un tool di modellazione grafica basato sull'approccio *MDA*[6], nonché un tool per lo *schieramento di sistemi multi-agente*.

[6] Model-Driven Architecture

JASON

Sviluppato da Rafael H. Bordini, Jomi F. Hübner e Renata Vieira

http://jason.sourceforge.net

4.1 Visione d'insieme

La ricerca nel campo dei *sistemi multi-agente (MAS)* ha portato ad una varietà di tecniche che promettono di realizzare complessi sistemi distribuiti. L'importanza di questo fatto è che tali sistemi sarebbero in grado di lavorare in ambienti tradizionalmente non pensati per comuni programmi di computer, in quanto troppo imprevedibili.

Jason ha come obiettivo quello di riprendere uno dei linguaggi di programmazione più eleganti mai apparsi in letteratura; il linguaggio era chiamato *AgentSpeak(L)*, ed era stato introdotto da *Anand S. Rao* nell'ormai lontano 1996 [23]. AgentSpeak(L) è un linguaggio di programmazione logico *agent-oriented* che mira a realizzare sistemi di pianificazione reattivi, e non è nient'altro che un linguaggio di programmazione ad agenti astratto. Il lavoro fatto ha avuto come obiettivo quello di estendere *AgentSpeak* in modo da farlo diventare un linguaggio di programmazione funzionale (permettendo quindi una perfetta integrazione con le più importanti tecniche *MAS*) nonché quello di fornire semantiche operazionali per *AgentSpeak* e molte delle sue estensioni, tenendo sempre in primo piano l'eleganza e anche la rigorosa base formale del linguaggio originale.

Jason è dunque l'interprete per una versione estesa di *AgentSpeak*, in quanto permette agli agenti di essere distribuiti sulla rete attraverso l'uso di *JADE* o di *SACI*[7]. *Jason* è disponibile open source sotto licenza *GNU LGPL*, ed implementa la semantica operazionale di *AgentSpeak*. Un'altra importante estensione è sicuramente quella che permette anche lo *scambio di piani*.

Alcune delle caratteristiche disponibili in *Jason* sono:

- *comunicazioni inter-agente basate sul paradigma speech-act*;
- *annotazioni sulle etichette dei piani*, che possono essere usate per elaborare funzioni di selezione;
- *possibilità di eseguire un sistema multi-agente distribuito su di una rete* (usando *JADE*, *SACI* o anche altri tipi di middleware);
- *possibilità di personalizzare completamente (in Java) le funzioni di selezione* nonché *l'architettura generica di un agente*;
- *semplice estendibilità attraverso azioni interne definite dall'utente*;
- *semplice nozione di ambiente multi-agente*, che può essere implementato in *Java* (può essere anche la simulazione di un ambiente reale, ad esempio per scopi di test prima che il sistema sia effettivamente sviluppato).

Interessante il fatto che molte delle caratteristiche avanzate in *Jason* sono disponibili come meccanismi opzionali e personalizzabili. Essendo *Jason* molto semplice ed elegante si presta anche all'insegnamento didattico di linguaggi *agent-oriented*.

4.2 Linguaggio

Il linguaggio di programmazione *AgentSpeak(L)* è un'estensione naturale della programmazione logica per l'architettura ad agenti *BDI*, e fornisce un framework astratto molto elegante per programmare agenti *BDI*. L'*architettura BDI* è dunque l'approccio predominante per l'implementazione di agenti *intelligenti* o *razionali* [24].

[7] Lo scopo principale di SACI (Simple Agent Communication Infrastructure) è quello di permettere agli agenti distribuiti di comunicare in modo semplice. SACI è formato da una API e da un insieme di strumenti che possono essere usati per lo scopo suddetto.

Un agente *AgentSpeak* è definito da un *insieme di belief* che forniscono lo stato iniziale della *belief-base* dell'agente, la quale è costituita semplicemente da un insieme di formule atomiche di primo ordine e da un insieme di piani che formano la *libreria dei piani*. Prima di spiegare nel dettaglio come sia scritto esattamente un piano, dobbiamo introdurre le nozioni di *goal* e di *triggering degli eventi*. *AgentSpeak* distingue sostanzialmente *due tipi* di goal: gli *achievement goal* e i *test goal*. Gli achievement goal sono formati da una formula atomica con prefisso l'operatore '!', mentre i test goal hanno come prefisso l'operatore '?'. Un *achievement goal* dichiara che l'agente vuole raggiungere uno stato dell'ambiente dove la formula atomica associata risulti vera. Un *test goal* dichiara invece che l'agente vuole verificare quando la formula atomica è (o può essere unificata con) uno dei suoi *belief*.

Un agente *AgentSpeak* è un *sistema di pianificazione reattivo*. Gli eventi ai quali reagisce sono correlati sia a cambiamenti in belief dovuti alla percezione dell'ambiente che a cambiamenti nei goal dell'agente originati dall'esecuzione dei piani innescati da eventi precedenti. Un *evento di triggering* definisce quali eventi possono iniziare l'esecuzione di un particolare piano. I piani sono scritti dal programmatore, così possono essere innescati (*triggered*, appunto) dall'addizione ('+') o dalla sottrazione ('-') di belief o goal (i *comportamenti mentali* degli agenti *AgentSpeak*).

Un piano *AgentSpeak* ha una *testa* (l'espressione alla sinistra della freccia), che è formata da un *evento di triggering* (specificando gli eventi per i quali questo piano è rilevante), e una congiunzione di belief letterali che rappresentano il *contesto*. La congiunzione dei letterali nel contesto deve essere una conseguenza logica dei belief correnti dell'agente se il piano è considerato valido per questo particolare momento temporale (solo i piani considerati validi possono essere scelti per l'esecuzione). Un piano ha anche un corpo, che è sostanzialmente una sequenza di *azioni base*[8] o di (sotto-)goal che l'agente deve conseguire (o testare) una volta che il piano è innescato.

[8] Tali azioni rappresentano operazioni atomiche che l'agente può eseguire ad esempio per modificare l'ambiente. Inoltre, tali azioni sono scritte sotto forma di formule atomiche usando però un insieme di simboli di azione piuttosto che di simboli di predicato.

```
skill(plasticBomb).
skill(bioBomb).
-skill(nuclearBomb).

safetyArea(field1).

@p1
+bomb(Terminal, Gate, BombType) :    skill(BombType)
       <-     !go(Terminal, Gate);
              disarm(BombType).

@p2
+bomb(Terminal, Gate, BombType) :   -skill(BombType)
       <-     ImoveSafeArea(Terminal, Gate, BombType).

@p3
+bomb(Terminal, Gate, BombType) :   not skill(BombType) &
                                    not -skill(BombType)
       <-     .broadcast(tell, alter).

@p4
+!moveSafeArea(T,G,Bomb) ; true
       <-     ?safeArea(Place);
              !discoverFreeCPH(FreeCPH);
              .send(FreeCPH, achieve,
                     carryToSafePlace(T,G,Place,Bomb)).
```

Figura 4.1 Esempi di piani per un agente AgentSpeak (tratto da [5])

La *figura 4.1* illustra un semplice esempio di codice *AgentSpeak* per i belief iniziali e i piani di un ipotetico agente dedito al disarmo delle bombe in un aeroporto. Inizialmente l'agente ha le abilità di disarmare *bombe plastiche* e *biologiche*, ma *non è in grado di disarmare bombe nucleari*; l'agente sa inoltre che $field1$ è una zona sicura dove lasciare una bomba che non è in grado di disarmare. Quando questo agente riceve un messaggio da un altro agente che dice che una bomba biologica è al terminale $t1$, gate $g43$, viene creato un nuovo evento per $+bomb(t1, g43, bioBomb)$.

Un agente per il disarmo delle bombe come questo appena descritto ha *tre piani rilevanti per questo evento* (identificati dalle etichette $p1$, $p2$ e $p3$), dato che l'evento coincide con l'evento scatenante di questi tre piani. Ad ogni modo, solo il contesto del primo piano è soddisfatto ($skill(bioBomb)$), e questo lo rende *l'unico piano valido*. Nei piani $p1$, $p2$ e $p3$ il contesto è usato per decidere quando tentare di disarmare una bomba (nel caso in cui l'agente abbia la capacità di disarmare quel particolare tipo di bomba), quando muoverla in un'area sicura (nel caso non sia in grado di disarmarla) o quando far scattare un allarme di sicurezza (nel caso non sia specializzato a sufficienza). Dal momento che solo il primo piano è applicabile, viene creata una *intenzione* basata

su di esso ed il piano inizia ad essere eseguito, aggiunge un sotto-goal ! $go(t1, g43)$ (i piani per il raggiungimento di questo goal sono stati omessi) ed esegue l'azione base $disarm(bombType)$. Nel piano $p4$ abbiamo un esempio di un *test goal* nel quale l'agente consulta i suoi propri belief per capire dove portare la bomba, e un esempio di un'azione interna usata per spedire un messaggio. Vedremo i dettagli del codice *AgentSpeak* usato nel prosieguo del lavoro di tesi.

4.2.1 Specifiche e aspetti sintattici

La grammatica in *figura 4.2* fornisce la *sintassi AgentSpeak* così come *Jason* la accetta. $< ATOM >$ è un identificatore che inizia con una lettera minuscola oppure con il simbolo ".", $< VAR >$ (ad esempio, una variabile) è un identificatore che inizia con una lettera maiuscola, $< NUMBER >$ è un qualsiasi intero o numero *floating-point*, mentre $< STRING >$ è una stringa racchiusa tra due doppi apici come di consueto.

Le differenze principali dal linguaggio *AgentSpeak* originale sono le seguenti. *Dove nel linguaggio originale erano permesse formule atomiche, quin viene invece usato un letterale.* I termini ora possono essere variabili o liste (con la sintassi *Prolog*), così come interi, numeri floating-point o stringhe. Inoltre ogni formula atomica può essere trattata come un termine, e le variabili possono essere trattate come letterali.

Un'altra importante modifica è che ora le formule atomiche possono avere *annotazioni*. Queste sono liste di termini racchiusi tra parentesi quadre che seguono immediatamente la formula. All'interno della *belief-base* le annotazioni sono usate, ad esempio, per registrare le sorgenti di informazione. Un termine del tipo $source(s)$ viene usato nelle annotazioni a questo proposito; s può essere il nome di un agente (per denotare l'agente che ha comunicato tale informazione) oppure due *identificatori atomici speciali*, cioè $percept$ e $self$, usati per indicare che un belief si è manifestato a partire dalla percezione dell'ambiente oppure dal fatto che l'agente abbia esplicitamente aggiunto un belief alla sua propria *belief-base* dall'esecuzione del corpo di un piano, rispettivamente.

```
agent               →   beliefs plans
beliefs             →   ( literal "." ) *
                        N.B.: viene generato un errore semantico
                        se non si usa un letterale base
plans               →   ( plan ) +
plan                →   [ "@" atomic formula ]
                        triggering event ":" context "<-" body "."
triggering event    →   "+"    literal
                    |   "-"    literal
                    |   "+" "!"literal
                    |   "-" "!"literal
                    |   "+" "?"literal
                    |   "-" "?"literal
literal             →   atomic formula
                    |   "-" atomic formula
                    |   <VAR>
default literal     →   literal
                    |   "not" literal
                    |   "not" "(" literal ")"
                    |   term ( "<" | "<=" | ">" | ">=" | "== | "\\==" | "=" ) term
                    |   literal ( "==" | "\\==" | "=" ) literal
context             →   "true"
                    |   default literal ( "&" default literal ) *
body                →   "true"
                    |   body formula ( "&" body formula ) *
body formula        →   literal
                    |   "!" literal
                    |   "?" literal
                    |   "+" literal
                    |   "-" literal
atomic formula      →   <ATOM> [ "(" list of terms ")" ] [ "(" list of terms ")" ]
list of terms       →   term ( "," term ) *
term                →   atomic formula
                    |   list
                    |   <VAR>
                    |   <NUMBER>
                    |   <STRING>
list                →   "["
                        [ term ( ( "," term ) *
                               | "|" ( list | <VAR> )
                               )
                        ] "]"
```

Figura 4.2 Grammatica per l'estensione di AgentSpeak(L) interpretata da Jason (schema tratto da [5])

I belief iniziali, che fanno parte cioè del codice sorgente di un agente *AgentSpeak*, sono da considerarsi *interni* (quindi come se avessero un'annotazione del tipo $[source(self)]$), a meno che il belief non riporti una qualsiasi annotazione esplicita fornita dall'utente.

I piani hanno anche *etichette*, e l'etichetta di un piano può essere una formula atomica (comprese le annotazioni). Le annotazioni nelle formule usate come etichette per i piani possono essere sfruttate per l'implementazione di funzioni di selezione più sofisticate, e la personalizzazione delle funzioni di selezione è fatta ovviamente in *Java*. Inoltre, dal momento che l'etichetta è parte di un'istanza di un piano nell'insieme delle intenzioni e le annotazioni possono

essere cambiate dinamicamente, abbiamo tutti i mezzi necessari per l'implementazione di funzioni di selezione delle intenzioni molto efficienti.

In *Jason* sono disponibili anche degli *eventi per gestire le situazioni in cui i piani falliscono*, nonostante questi eventi non siano formalizzati nella grammatica. Se un'azione fallisce o non vi è alcun piano valido per un sotto-goal nel piano in esecuzione per gestire un evento interno con l'aggiunta di un goal $+!\,g$, allora l'intero piano che è fallito viene rimosso dalla cima dell'intenzione e viene generato un evento interno per $-!\,g$ associato con questa stessa intenzione. Se il programmatore avesse fornito un piano che avesse un evento scatenante che combaciasse con $-!\,g$ e che fosse anche applicabile, tale piano sarebbe stato posto in cima all'intenzione in modo che il programmatore potesse specificare nel corpo di tale piano come dovesse essere gestita quella particolare tipologia di fallimento. Se tale piano non risultasse disponibile, l'intera intenzione verrebbe scartata e verrebbe stampato un *messaggio di warning* sulla console.

Per finire, le *azioni interne* possono essere usate sia nel contesto che nel corpo dei piani. Ogni simbolo di azione che inizia con ".", o che ha il suddetto simbolo in una qualche posizione, denota un'azione interna. Vi sono azioni definite dall'utente che sono eseguite internamente all'agente. Chiamiamo queste azioni *interne* per creare una chiara distinzione con le azioni che compaiono nel corpo di un piano e che denotano le azioni che l'agente può eseguire con lo scopo di cambiare l'ambiente condiviso.

In *Jason*, le azioni interne sono codificate in *Java* oppure anche in altri linguaggi di programmazione attraverso l'uso di JNI[9], e possono essere organizzate in librerie di azioni per scopi specifici (la stringa a sinistra del simbolo "." è il nome della libreria; le azioni interne standard hanno un nome di libreria vuoto).

Ci sono svariate azioni interne standard distribuite con *Jason*, che comunque non verranno trattate in questo lavoro di tesi per motivi di tempo e spazio (vedi [25] per una lista completa). Per citare solamente alcune tra le più importanti, ricordiamo l'azione interna che implementa la comunicazione tra agenti in stile *KQML* e la classe di azioni interne standard collegate alle *query*

[9] Java Native Interface

relative ai desideri e alle intenzioni degli agenti (nonchè alla capacità degli agenti di poter scartare tali desideri e intenzioni).

4.2.2 Semantiche e accertamento

Come già detto precedentemente, una delle principali caratteristiche di *Jason* è quella di implementare la *semantica operazionale* di un'estensione del linguaggio *AgentSpeak*. Avere a disposizione semantiche formali ci permette ovviamente di dare definizioni precise per le nozioni pratiche di convinzioni, desideri e intenzioni in relazione all'esecuzione di agenti *AgentSpeak*. Per motivi di spazio in questo lavoro di tesi non riusciremo ad introdurre nel suo complesso tutte le semantiche relative ad *AgentSpeak*. Quello che cercheremo di fare sarà invece fornire quelle che sono le intuizioni principali dietro alle interpretazioni dei programmi *AgentSpeak*.

Oltre che la *belief-base* e la *libreria dei piani* l'interprete *AgentSpeak* deve gestire anche un *insieme di eventi* e un *insieme di intenzioni*, e il suo funzionamento richiede *tre funzioni di selezione*. La *funzione di selezione degli eventi* (S_ε) seleziona un *singolo evento* dall'insieme degli eventi; un'altra *funzione di selezione* (S_O) seleziona un'*opzione* (ad esempio, un piano valido) da un insieme di piani validi; una terza *funzione di selezione* (S_I) seleziona infine una particolare *intenzione* dall'insieme delle intenzioni. Le tre funzioni di selezione sono specifiche per ogni agente, nel senso che effettuano selezioni basate sulle caratteristiche di un singolo agente. Qui di seguito lasciamo volutamente indefinite le funzioni di selezione dal momento che le scelte fatte da loro sono *nondeterministiche*.

Le intenzioni sono particolari *linee d'azione* alle quali un agente si è dedicato allo scopo di gestire certi eventi. Ogni intenzione è uno *stack di piani parzialmente istanziati*. Gli eventi, che possono iniziare con l'esecuzione di piani che hanno eventi scatenanti rilevanti, possono essere *esterni* (quando hanno origine dalla percezione dell'ambiente dell'agente) oppure *interni* (quando sono generati dall'esecuzione del piano da parte dell'agente). In quest'ultimo caso l'evento è accompagnato dall'intenzione che lo ha generato (dal momento che il piano scelto per quell'evento sarà posizionato in cima allo stack di quell'intenzione). Gli eventi esterni creano nuove intenzioni, le quali rappresentano distini centri di interesse per l'operato dell'agente nell'ambiente.

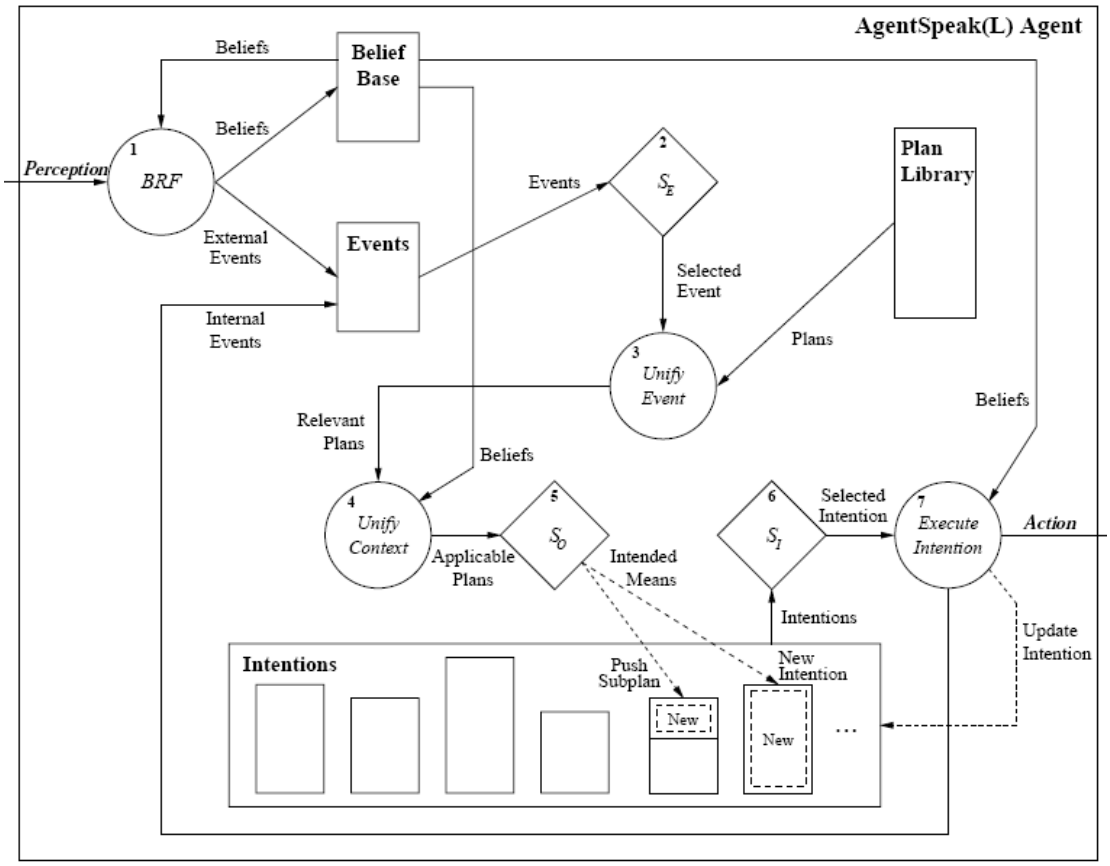

Figura 4.3 Ciclo di interpretazione di un programma AgentSpeak (schema tratto da [5])

Forniamo ora alcuni ulteriori dettagli sul funzionamento di un *interprete AgentSpeak*, facendo riferimento alla *figura 4.3* qui sopra. Questa è comunque solo una rappresentazione degli aspetti fondamentali dell'interprete della definizione originale di *AgentSpeak*, non include quindi le estensioni implementate in *Jason*. Nella figura, gli insiemi (di belief, di intenzioni, di eventi e di piani) sono rappresentati come *rettangoli*. I *rombi* rappresentano l'operazione di selezione (di un elemento da un insieme). I *cerchi* rappresentano parte della processazione coinvolta nell'interpretazione dei programmi *AgentSpeak*.

Ad ogni ciclo di interpretazione di un programma agente l'interprete aggiorna una lista di eventi che può essere generata da percezioni dell'ambiente o dall'esecuzione di intenzioni. Si assume che i belief siano aggiornati dalle percezioni, e quando dovessero esserci cambiamenti nei belief dell'agente questo implichi anche l'inserimento di un evento nell'insieme degli eventi. Dopo che la funzione S_ε ha selezionato un evento, l'interprete deve *unire questo evento con gli eventi scatenanti nelle teste dei vari piani*. Questa operazione genera l'insieme di tutti i piani rilevanti, o *validi*, per quell'evento. L'insieme dei piani applicabili è determinato controllando se la parte

contestuale dei piani in questo insieme si attiene ai belief dell'agente – questo è l'insieme dei piani che possono essere usati in quel particolare momento per gestire l'evento scelto. Poi S_O seleziona un singolo piano applicabile da quell'insieme, ne effettua un *push* in testa allo stack di una intenzione esistente (se l'evento è interno) oppure crea una nuova intenzione nell'insieme delle intenzioni (se l'evento è esterno, ad esempio se generato dalla percezione dell'ambiente).

Tutto ciò che resta da fare a questo punto è *selezionare una singola intenzione per l'esecuzione nel ciclo attuale*. La funzione S_I seleziona una delle intenzioni dell'agente. In cima allo stack di tale intenzione vi è un piano, e la formula all'inizio del suo corpo viene mandata in esecuzione. Questo implica *tre possiblità*: che un'azione base sia eseguita dall'agente nel suo ambiente, che sia generato un evento interno (nel caso la formula selezionata sia un *achievement goal*), oppure che sia eseguito un *test goal* (il che significa che l'insieme dei belief dovrà essere controllato).

Se l'intenzione è quella di eseguire un'azione base oppure un *test goal*, allora l'insieme delle intenzioni deve vinire aggiornato. Nel caso di un *test goal* ci sarà una ricerca nella *belief-base* per un belief atomico in grado di unirsi con la formula nel *test goal*. Se la ricerca ha successo, nuove variabili verranno istanziate nel piano parzialmente istanziato che conteneva il *test goal* (e lo stesso *test goal* verrà rimosso dall'intenzione da cui era stato preso). Nel caso in cui venga selezionata un'azione base gli aggiornamenti necessari per l'insieme delle intenzioni non sono pesanti, basta rimuovere tale azione dall'intenzione. Quando tutte le formule nel corpo del piano sono state rimosse (cioè sono state eseguite), l'intero piano è rimosso dall'intenzione così come l'*achievement goal* che lo ha generato (se questo era il caso). Questa operazione termina un *ciclo di esecuzione* e tutto viene ripetuto nuovamente, inizialmente controllando lo stato dell'ambiente dopo che gli agenti hanno agito su di esso, poi generando gli eventi rilevanti, e così via.

4.2.3 Confronto con altri linguaggi

Finora *Jason* è stato considerato molto poco quando si tratta di sviluppare sistemi orientati agli agenti, anche a causa di metodologie già esistenti e maggiormente rodate come ad esempio *Prometheus* o *JACK* [5]. Va comunque detto che il codice *AgentSpeak* è molto più leggibile se

paragonato ad altri linguaggi come *JACK* o *Jadex*, ed è quindi plausibile che *Jason* sia in grado di fornire un metodo se non altro molto più chiaro per l'implementazione di queste tipologie di design. Punto a favore di *JACK* è però di essere una piattaforma industriale e quindi di contare su molti più strumenti di supporto e su di una documentazione più estesa, ma d'altro canto *Jason* è *open source* mentre *JACK* non lo è.

Un costrutto che ha un importante impatto nel mantenere il giusto livello di astrazione nel codice *AgentSpeak* anche per sistemi molto sofisticati sono sicuramente le *azioni interne*. Le azioni interne ritornano necessariamente un *valore booleano*, e quindi sono rappresentate a livello dichiarativo da un programma logico in *AgentSpeak*. Perciò il modo in cui l'integrazione con i tradizionali linguaggi di programmazione orientati agli oggetti e l'uso di codice ereditato sono raggiunti in *Jason* è notevolmente più elegante rispetto ad altri linguaggi di programmazione ad agenti (anche rispetto a *Jadex*, nel nostro specifico caso).

4.2.4 Altre caratteristiche del linguaggio

Comunicazione in AgentSpeak

Le primitive attualmente disponibili per la comunicazione in *AgentSpeak* sono largamente ispirate dalle primitive presenti in *KQML*[10], ma vengono inoltre incluse alcune nuove primitive di comunicazione riguardanti lo scambio di piani oltre che la comunicazione riguardo le proposizioni. Le primitive disponibili sono brevemente descritte qui di seguito, dove s denota l'agente che *invia* il messaggio, mentre r denota l'agente che *riceve* il messaggio. Da notare che $tell$ e $untell$ possono essere usate sia per un agente che manda informazioni *proattivamente* ad un altro agente, che come risposta a messaggi ask precedenti.

$tell$: s comunica a r di considerare vera la sentenza nel contenuto del messaggio;

$untell$: s comunica a r di non considerare vera la sentenza nel contenuto del messaggio;

[10] Knowledge Query and Manipulation Language (KQML) è un linguaggio e un protocollo di comunicazione tra agenti software e sistemi knowledge-based.

$achieve$: s richiede che r tenti di raggiungere uno stato nel quale il contenuto del messaggio risulti vero;

$unachieve$: s richiede che r tenti di scartare l'intenzione di raggiungere uno stato nel quale il contenuto del messaggio risulti vero;

$tellHow$: s informa r di un piano;

$untellHow$: s richiede che r ignori un certo piano (ad esempio eliminandolo dalla sua libreria di piani);

$askIf$: s vuole sapere se il contenuto del messaggio è vero per r;

$askAll$: s vuole tutte le risposte di r ad una domanda;

$askHow$: s vuole tutti i piani di r per un evento innescato;

Viene usato un particolare meccanismo per ricevere e mandare messaggi in modo *asincrono*. I messaggi sono *memorizzati in una casella di posta* e uno solo di loro viene processato dall'agente all'inizio di un ciclo di ragionamento: lo specifico messaggio che deve essere gestito all'inizio del ciclo di ragionamento è determinato da una *funzione di selezione*, che comunque può essere personalizzata dal programmatore.

Inoltre per processare i messaggi consideriamo una funzione *data*, allo stesso modo in cui le funzioni di selezione sono assunte come date nelle specifiche di un agente. Questa funzione definisce un insieme di messaggi socialmente accettabili: per esempio, l'agente ricevente può voler sapere quando l'agente che invia il messaggio ha oppure no il permesso di comunicare con lui (può essere utile per evitare che l'agente sia attaccato da eventuali agenti di comunicazione maligni).

Le nozioni di *fiducia* possono anche essere programmate nell'agente considerando le annotazioni delle sorgenti di informazione durante il processo di ragionamento dell'agente stesso. Quando è applicato ai messaggi $tell$, la funzione determina solamente se il messaggio debba essere processato oppure no. Quando la sorgente è *fidata* (nel senso limitato usato in questo contesto), la sorgente di informazione per un belief acquisito dalla comunicazione viene annotata con quel belief nella *belief-base*, abilitando ulteriori considerazioni sui gradi di fiducia durante il ragionamento dell'agente.

Quando la funzione per controllare l'accettazione dei messaggi è invece applicata a un messaggio di tipo $achieve$ dovrebbe essere programmata per ritornare un valore $true$ se, ad esempio, l'agente ha una relazione di subordinazione nei confronti dell'agente che invia il messaggio. Ad ogni modo questa relazione non deve venire interpretata con sfumature sociali o psicologiche: il programmatore definisce questa funzione solamente per tenere in considerazione tutte le possibili ragioni per cui un agente debba fare qualcosa per un altro agente (dall'attuale subordinazione a situazioni di puro altruismo).

Come semplice esempio di come l'utente possa *personalizzare* questa relazione di subordinazione in *Jason*, possiamo considerare il fatto che *un robot CPH esegue solamente quello che chiede un altro robot MDS*. Ecco qui di seguito la personalizzazione del pacchetto *CPH*:

```
package cph;
import jason.asSemantics.Agent;

public class CPHAgent extends Agent
{
public boolean socAcc(Message m)
       {
if (m.getSender().startsWith("mds") && m.getIlForce().equals("achieve"))
               {
                      return true;
               }
               else
               {
               return false;
               }
       }
}
```

Figura 4.4 Esempio di personalizzazione del pacchetto CPH (esempio tratto da [5])

Per dotare gli agenti *AgentSpeak* della capacità di processare messaggi di comunicazione dobbiamo annotare, per ogni belief, quale sia la sua *sorgente*. Questo meccanismo di annotazione fornisce un metodo molto elegante per rendere esplicite le sorgenti dei belief degli agenti, porta vantaggi in termini di potenza espressiva e di leggibilità oltre che fornire la possiblità di usare tali informazioni nel ciclo di ragionamento dell'agente (ad esempio, nella selezione di piani per il raggiungimento di goal).

Le sorgenti dei belief possono essere annotate allo scopo di identificare lo specifico agente che aveva precedentemente inviato le informazioni in un messaggio, così come per denotare

belief interni o percezioni (ad esempio nel caso in cui il belief sia stato acquisito attraverso percezioni dell'ambiente). Usando questo meccanismo di annotazione delle sorgenti di informazione si rendono più chiari alcuni problemi nell'implementazione di interpreti *AgentSpeak* relativi a *belief interni* (quelli aggiunti durante l'esecuzione di un piano, per intenderci).

Cooperazione in AgentSpeak

Coo-BDI[11] [26] estende i tradizionali linguaggi di programmazione *BDI* orientati agli agenti sotto molti aspetti: l'introduzione di *cooperazione tra gli agenti* per il reperimento di piani esterni per un dato evento scatenante, l'*estensione di piani* con vari specificatori di accesso, l'estensione di intenzioni per prendere in considerazione il meccanismo esterno di reperimento dei piani e la *modifica dell'interprete* per poter gestire tutti questi compiti.

La *strategia di cooperazione* di un agente Ag include l'*insieme di agenti* con cui Ag si vedrà a cooperare, la *politica di reperimento del piano*, nonché la *politica di acquisizione del piano*. La strategia di cooperazione può evolvere nel tempo, permettendo maggior flessibilità e autonomia agli agenti, ed è modellata da *tre funzioni*:

- $trusted(Te, TrustedAgentSet)$, dove Te è un evento scatenante (non necessariamente di base) e $TrustedAgentSet$ è l'insieme di agenti che Ag contatterà per ottenere piani rilevanti per Te.
- $retrievalPolicy(Te, Retrieval)$, dove $Retrieval$ può assumere i valori $always$ e $noLocal$, il che significa che i piani rilevanti per l'evento scatenante Te devono essere recuperati da altri agenti in ogni caso oppure solo se non sono disponibili altri piani locali rilevanti, rispettivamente.
- $acquisitionPolicy(Te, Acquisition)$, dove $Acquisition$ può assumere i valori $discard$, add e $replace$, nel senso che quando un piano rilevante per Te viene recuperato da un agente di fiducia deve venire usato e scartato, oppure aggiunto alla libreria dei piani, oppure usato per aggiornare la libreria dei piani sostituendo tutti i piani innescati da Te.

[11] Cooperative BDI

Piani Al di là delle componenti standard che costituiscono i piani BDI, in questa estensione i piani hanno anche una sorgente che determina il *primo proprietario* del piano stesso nonché uno specificatore di accesso che determina l'*insieme di agenti con i quali il piano può venire condiviso*. La sorgente può assumere due valori: $self$ (l'agente è in possesso del piano) e Ag (l'agente era originariamente di Ag). Lo specificatore di accesso può invece assumere tre valori: $private$ (il piano non può essere condiviso), $public$ (il piano può essere condiviso con ogni agente) e $only(TrustedAgentSet)$ (il piano può essere condiviso solamente con gli agenti contenuti nell'insieme degli agenti chiamato *TrustedAgentSet*). Il meccanismo chiamato *Coo-AgentSpeak* presente in *Jason* consente all'utente di definire le strategie di cooperazione in stile *Coo-BDI* e si occupa inoltre di tutte le altre questioni, come ad esempio inviare richieste appropriate per i piani.

Un'ulteriore caratteristica rilevante in *Jason* è sicuramente l'*opzione di configurazione* nel caso non ci sia alcun piano applicabile per un evento rilevante. Se un evento è rilevante significa che ci sono uno o più piani nella libreria dei piani che gestiscono questo particolare evento. Se dovesse succedere che nessuno di questi piani fosse applicabile in un certo momento, potrebbe sorgere un problema dal momento che l'agente non saprebbe come gestire la situazione in quel determinato istante.

In *Jason* viene data agli utenti un'*opzione di configurazione* che può essere impostata all'interno del file dove sono specificati i vari agenti e l'ambiente che compongono un sistema multi-agente. Tale opzione permette all'utente di stabilire, per eventi che hanno piani rilevanti ma non applicabili, quando l'interprete debba scartare tale evento completamente ($event = discard$) o reinserire l'evento alla fine della coda degli eventi ($event = requeue$). Grazie al meccanismo di personalizzazione di *Jason*, l'unica modifica necessaria per far fronte a *Coo-AgentSpeak* è una terza opzione di configurazione disponibile agli utenti (non sono necessarie modifiche all'interprete stesso).

Quando viene usato *Coo-AgentSpeak*, l'opzione $event = retrieve$ deve essere usata nel file di configurazione. Questo fa sì che *Jason* richiami la funzione $selectOption$ definita dall'utente anche quando non esiste alcun piano applicabile per un determinato evento. In questo modo parte dell'approccio *Coo-BDI* può essere implementato fornendo una speciale funzione $selectOption$ che si occuperà di recuperare i piani esternamente, quando opportuno.

4.3 La piattaforma

In questo paragrafo descriveremo in dettaglio il funzionamento della piattaforma *Jason* e tutte le sue principali caratteristiche, con un occhio di riguardo anche agli aspetti riguardanti l'*osservanza agli standard*, l'*interoperabilità* nonché le *applicazioni supportate dal linguaggio e dalla piattaforma*.

4.3.1 Caratteristiche principali della piattaforma Jason

Configurare sistemi multi-agente

La configurazione di un sistema multi-agente completo è gestita da un file di testo molto semplice, di cui abbiamo un esempio qui sotto nella *figura 4.5*. Brevemente, l'ambiente è implementato in una classe chiamata *HeathrowEnv*. Il sistema ha *tre tipi di agenti*: cinque istanze di *MDS*, dieci di *CPH* e tre di *BD* (*Bomb Disarmers*). Gli agenti *MDS* hanno una classe di agente personalizzata, mentre gli agenti *CPH* hanno sia una classe di agenti che una classe di architettura personalizzate. La grammatica in *figura 4.6* fornisce la sintassi che può essere usata nel file di configurazione. In questa grammatica $< NUMBER >$ viene usato per numeri interi, $< ASID >$ sono identificatori *AgentSpeak* (che devono iniziare con una lettera minuscola), $< ID >$ è un identificatore e infine $< PATH >$ è necessario per definire i percorsi dei file.

```
MAS heathrow         {

       architecture:
              Centralised

       environment:
              HeathrowEnv

       agents:
              mds    agentClass mds.MDSAgent
                     #5;
              cph    agentArchClass cph.CPHAgArch
                     agentClass cph.CPHAgent
                     #10;
              bd     #3;
}
```

Figura 4.5 File di configurazione per un sistema multi-agente (esempio tratto da [5])

L'$< ID >$ usato dopo la keyword MAS è il *nome del gruppo*. La keyword $architecture$ è usata per specificare quale dei tre agenti generici per le architetture disponibili in *Jason* debba essere usato. Le opzioni attualmente disponibili sono $Centralised$, $Saci$ oppure $Jade$; la seconda e la terza opzione permettono agli agenti di essere eseguiti su diverse macchine distribuite in una rete. È importante notare che l'ambiente dell'utente e le personalizzazioni delle classi rimangono le stesse con entrambe le architeture di sistema.

```
mas             →       "MAS" <ID> "{"
                        [ "architecture" ":" <ID> ]
                        environment
                        agents
                        "}"
environment     →       "environment" ":" <ID> ["at <ID> ]
agents          →       "agents" ":" ( agent ) +
agent           →       <ASID>
                        [ filename ]
                        [ options ]
                        [ "agentArchClass" <ID> ]
                        [ "agentClass" <ID> ]
                        ["#" <NUMBER> ]
                        ["at" <ID> ]
                        ";"
filename        →       [ <PATH> ] <ID>
options         →       "[" option ( "," option ) * "]"
option          →       "events" "=" ( "discard" | "requeue" | "retrieve" )
                        |       "intBels" "=" ( "sameFocus" | "newFocus" )
                        |       "verbose" "=" <NUMBER>
```

Figura 4.6 Sintassi da usare nel file di configurazione (tratta da [5])

Di seguito va specificato un *ambiente*: questo è semplicemente il nome della classe *Java* usata per programmare tale ambiente. Da notare che può essere specificato un *host name* (opzionale) dove l'ambiente può venire eseguito.

La keyword $agents$ è usata per definire l'insieme di agenti che prenderanno parte al sistema multi-agente. Un agente è specificato dapprima dal suo nome simbolico fornito come termine *AgentSpeak* (ad esempio, un identificatore che inizia con una lettera minuscola); questo sarà il nome che gli agenti useranno per riferirsi ad altri agenti nel gruppo (ad esempio, per le comunicazioni e lo scambio di messaggi tra agenti). Può essere poi fornito anche un nome di file (opzionale) al momento dell'inserimento del codice sorgente di *AgentSpeak* (di default, *AgentSpeak* comunque assumerà che il codice sia presente nel file $< name >. asl$, dove $< name >$ è il nome simbolico dell'agente). Ci sono poi anche vari settaggi opzionali per l'interprete *AgentSpeak* disponibile con *Jason*. Possono essere specificate un numero opzionale di

istanze di agenti che fanno uso dello stesso codice sorgente digitando un numero preceduto dal carattere # (vedi esempio in figura). Nel caso in cui siano richieste più di una istanza dell'agente, il nome attuale dell'agente sarà formato dal nome simbolico concatenato ad un indice che indica il numero dell'istanza (iniziando da 1). Così come per la keyword $environment$, la definizione di un agente può terminare con un *host name* dove l'agente sarà eseguito (preceduto dal simbolo @). Questo ovviamente avrà senso solamente se vengono selezionate le architetture *JADE* o *SACI*. Per l'interprete *AgentSpeak* presente in *Jason* sono disponibili anche le seguenti impostazioni (sono seguite dal simbolo " = " e poi da una delle keyword associate, considerando che l'opzione sottolineata è quella che viene usata di default):

$events$: le opzioni sono $\underline{discard}$, $requeue$ o $retrieve$; l'opzione $discard$ significa che gli eventi esterni per i quali non ci sono piani applicabili sono scartati, mentre l'opzione $requeue$ viene usata quando tali eventi vengono reinseriti alla fine della lista degli eventi che l'agente deve gestire. Quando viene selezionata l'opzione $retrieve$, la funzione $selectOption$ definita dall'utente viene chiamata anche se l'insieme dei piani rilevanti risulta vuoto. Questa funzionalità può venire usata, ad esempio, per permettere agli agenti di richiedere piani da altri agenti che possono avere la conoscenza necessaria in una determinata circostanza.

$intBels$: le opzioni in questo caso sono $sameFocus$ oppure $newFocus$. Quando i belief interni vengono aggiunti o rimossi esplicitamente all'interno del corpo di un piano, l'evento associato è un evento scatenante per il piano ed è trattato come interno e quindi i significati risultanti dal piano scelto per questo evento sono posizionati in testa alle intenzioni che hanno generato l'evento. Se viene usata l'opzione $newFocus$, invece, l'evento viene trattato come esterno creando quindi un nuovo centro dell'attenzione.

$verbose$: deve innanzi tutto venire specificato un numero tra 0 e 6. Più alto è questo numero, e maggiori sono le informazioni relative all'agente (o agli agenti se il numero di istanze è maggiore di 1) che verranno stampate a video nella console in cui il sistema era in esecuzione. Il valore di default è 1, e non 0: in questo modo vengono stampate a video solo le azioni che gli agenti eseguono nell'ambiente nonché i messaggi scambiati tra di loro.

Infine possono essere specificate architetture di agenti generiche personalizzate dall'utente oltre che altre funzioni personalizzabili che possono essere usate dall'interprete *AgentSpeak*, con l'utilizzo delle keyword $agentArchClass$ e $agentClass$.

Creare ambienti

Gli agenti in *Jason* possono essere posizionati in *ambienti reali* o *simulati*. Nel precedente caso l'utente ha dovuto personalizzare l'architettura generale dell'agente come verrà ora descritto nel prosieguo di questo paragrafo; in quest'ultimo caso l'utente deve fornire anche un'implementazione dell'ambiente simulato. Questa operazione è fatta direttamente in una classe *Java* che estende la classe base $Environment$ presente in *Jason*. Nella *figura 4.7* vediamo una semplice versione simulata dell'ambiente associato allo scenario dell'aeroporto *Heathrow* (la versione estesa del codice sorgente è contenuta nella distribuzione di *Jason* come esempio pratico delle potenzialità della piattaforma).

Tutte le *percezioni* (ovvero tutto ciò che possiamo percepire dall'ambiente) dovrebbero essere aggiunte alla lista ritornata dalla funzione $getPercepts$; questa è una lista di proposizioni elementari, quindi permette di usare la negazione forte (*strong negation*). Risulta impossibile inviare percezioni specifiche: programmando l'ambiente lo sviluppatore può determinare quali sottoinsiemi delle proprietà dell'ambiente saranno percepibili dagli agenti individualmente.

All'interno dell'architettura generale di un agente si possono ulteriormente personalizzare i belief che l'agente acquisirà dalle sue percezioni. Intuitivamente, *le proprietà dell'ambiente a disposizione di un agente sono associate a tutto ciò che è percepibile nell'ambiente* (ad esempio, se qualcosa è dietro il muro di casa mia non posso vederla). La personalizzazione a livello di architettura generale dell'agente dovrebbe venire usata per simulare *percezioni imperfette*. La personalizzazione delle percezioni individuali di un agente all'interno dell'ambiente è fatta effettuando l'*override* del metodo $getPercepts(agName)$; il metodo standard fornisce semplicemente tutte le proprietà correnti dell'ambiente sotto forma di percezioni a tutti gli agenti. Nell'esempio precedente, invece, solamente i robot *MDS* percepiranno la loro posizione all'interno dell'aeroporto.

```
public class HeathrowEnv extends Environment  {

       Map agsLocation = new HashMap();

       public List getPercepts(String agName)  {
              if     ( ... unattended luggage has been found ... )  {
                     // all agents will perceive the fact that
                     // there is unattendedLuggage
                     getPercepts().add(Term.parse("unattendedLuggage"));
              }

              if     (agName.startsWith("mds")) {
                     // mds robots will also perceive their location
                     List customPerception = new ArrayList();
                     customPerception.addAll(getPercepts())/
                     customPerception.add(agsLocation.get(agName));
                     return customPerception;
              }  else  {
                     return getPercepts();
                     }
       }

       public boolean executeAction(String ag, Term action)  {
              if (action.hasFunctor("disarm")) {
              ... the code that implements the disarm action
              ... on the environment goes here
              } else if (action.hasFunctor("move")) {
              ... the code for changing the agents' location and
              ... updating the agsLocation map goes here
              }
              return true;
      }
}
```

Figura 4.7 Ambiente simulato per lo scenario dell'aeroporto Heathrow (esempio tratto da [5])

La maggior parte del codice per la costruzione di ambienti dovrebbe essere nel corpo del metodo $executeAction$. Qualora un agente tentasse di eseguire un'azione base (quelle cioè che cambiano lo stato dell'ambiente circostante), il nome dell'agente e un termine ($Term$) rappresentante l'azione scelta vengono inviati come parametri a questo metodo. Quindi il codice di questo metodo deve controllare il suddetto termine (il quale è in forma di struttura *Prolog*) che rappresenta l'azione (e qualsiasi parametro) che sta per essere eseguita e controlla quale sia l'agente che sta tentando di eseguire l'azione per poi fare tutto ciò che risulta necessario in quel particolare modello dell'ambiente – normalmente questo significa cambiare le percezioni, ad esempio ciò che è vero o falso dell'ambiente viene cambiato in accordo alle azioni che stanno per venire eseguite.

Da notare che l'esecuzione di un'azione deve ritornare un valore di tipo $boolean$, che indica se il tentativo dell'agente di eseguire quell'azione sull'ambiente sia stato eseguito oppure no. Un piano *fallisce* se ogni azione base tentata dall'agente fallisce.

Personalizzare agenti

Certi aspetti del funzionamento cognitivo di un agente possono essere personalizzati dall'utente effettuando l'*override* di alcuni metodi della classe $Agent$. Cambiando la funzione di selezione dei messaggi l'utente può determinare la preferenza che l'agente deve dare a messaggi ricevuti da certi agenti piuttosto che altri, oppure a certi tipi di messaggi quando diversi tipi di messaggi sono stati ricevuti durante un ciclo di ragionamento. Come esempio della personalizzazione della classe di un agente, consideriamo nuovamente lo scenario dell'aeroporto di *Heathrow*. I robot *MDS* devono dare priorità a eventi correlati a bagagli abbandonati al di sopra ti qualsiasi altro tipo di eventi. Un agente *MDS* personalizzato che effettua l'*override* del metodo $selectEvent$ può implementare questa priorità come segue:

```
public class MDSAgent extends Agent    {

       public Event selectEvent(List evList)   {
              Iterator i = evList.iterator ( );
              while (i.hasNext())   {
                     Event e = (Event)i.next();
                     if
                     (e.getTrigger().getFunctor().equals("unattendedLuggage"))
                            i.remove();
                            return e;
              }
       }
       return super.selectEvent(evList);
}
```

Figura 4.8 Implementazione dell'agente MDS personalizzato (esempio tratto da [5])

Allo stesso modo l'utente può personalizzare le funzioni che definiscono l'architettura generale dell'agente. Queste funzioni gestiscono i seguenti aspetti:

- il modo in cui l'agente aggiornerà la sua *belief-base* data l'attuale percezione dell'ambiente, cioè la cosiddetta *BRF*[12] nella letteratura di *AgentSpeak*;
- il modo in cui l'agente percepisce l'ambiente;
- come l'agente riceve i messaggi inviati da altri agenti;
- come l'agente reagisce all'ambiente (riguardo alle azioni base che compaiono nel corpo dei piani).

[12]Il Belief Revision è il processo di cambiamento dei belief (delle convinzioni) per prendere in considerazione nuove porzioni di informazione. Ci possono essere due tipi di cambiamenti possibili, cioè aggiornamento oppure revisione. La Belief Revision Function è la funzione che adempie a questo compito.

Per la funzione di percezione potrebbe risultare interessante usare la funzione definita nella distribuzione di *Jason* e, una volta che ha ricevuto le percezioni correnti, processare ulteriormente la lista delle percezioni al fine, per esempio, di simulare percezioni difettose. È importante sottolineare che la funzione di revisione dei belief fornita con *Jason* aggiorna semplicemente la belief-base e genera gli eventi esterni (ad esempio, aggiunta e cancellazione di belief dalla belief-base) in accordo con le percezioni correnti, e in particolare *non garantisce la consistenza dei belief*. Dal momento che le percezioni sono inviate dall'ambiente, sta al programmatore dell'ambiente stesso verificare che non appaiano contraddizioni nelle percezioni. Inoltre, se i programmatori *AgentSpeak* aggiungono belief interni nei corpi dei piani, dovrà essere loro premura assicurare la consistenza. Infatti l'utente potrebbe essere anche interessato a modellare un agente *paraconsistente*, cosa che potrebbe essere realizzata molto facilmente.

Supponiamo, ad esempio, che sotto nessuna circostanza un robot *CPH* ottenga il permesso di disarmare una bomba. Per prevenire questo comportamento non desiderato (e potenzialmente pericoloso), anche se il robot abbia già deciso di farlo (ad esempio, nel caso sia stato infettato da un virus software), lo sviluppatore può effettuare l'*override* del metodo act nella classe personalizzata $AgArch$ del robot *CPH* ed assicurare quindi che l'azione selezionata non sia $disarm$ prima di permetterle di essere eseguita nell'ambiente:

```
public class CPHAgArch extends CentralisedAgArch   {

       public void act()  {

              / / get the current action to be performed
              Term action = fTS.getC().getAction().getActionTerm();

              i f ( !action.getFunctor().equals("disarm") )   {
                     / / ask the environment to execute the action
                     fEnv.executeAction(getName(), action));
                     …
              }
       }
}
```

Figura 4.9 Implementazione della classe personalizzata CPHAgArch (tratto da [5])

Definire nuove azioni interne

Un costrutto fondamentale che permette ad *AgentSpeak* di rimanere al giusto livello di astrazione è quello delle *azioni interne*, che permettono una semplice estendibilità e uso del codice proprietario. Come suggerito in [27], le azioni interne che iniziano con il simbolo "." fanno

parte delle *librerie standard* distribuite con *Jason*. Le azioni interne definite dagli utenti dovrebbero essere organizzate in librerie specifiche che forniscono un modo molto interessante di organizzare questo codice, normalmente utili per un'ampia gamma di sistemi diversi. In *AgentSpeak* si ha accesso all'azione utilizzando il nome della libreria seguito dal simbolo "." e infine seguito dal nome dell'azione. Le librerie sono definite come *pacchetti Java* e ogni azione nella libreria dell'utente dovrebbe essere una classe *Java*: i nomi del pacchetto e della classe sono i nomi della libreria e dell'azione.

Nell'esempio della gestione dell'aeroporto, quando un bagaglio incustodito è percepito dai robot *MDS* questi si scambiano un valore tra di loro che rappresenta quanto adatti sono per gestire la nuova situazione. Il robot con il valore a lui assegnato più alto sarà riposizionato per occuparsi del bagaglio incustodito. Ora supponiamo che per calcolare il suddetto valore iniziale sia richiesta una formula molto complessa, e che siano inoltre richiesti in seguito ulteriori controlli e calcoli per aggiornare tale valore; chiaramente i linguaggi di tipo imperativo sono normalmente più adatti per implementare questo tipo di algoritmi. L'utente può quindi usare la seguente classe *Java* per implementare questo algoritmo, e riferirsi ad essa dall'interno del codice *AgentSpeak* con la chiamata $mds.calculateMyBid(Bid)$:

```
package mds;
import ...

public class calculateMyBid implements InternalAction  {

       public boolean execute(TransitionSystem ts, Unifier un,
                                    Term[] args) throws Exception {
              int bid = ... a complex formula ...;

              ... plus complex algorithm and calculations
                     for adjusting the agent's bid ...

              un.unifies(args[0], Term.parse(""+bid))/
              return true;
       }
}
```

Figura 4.10 Implementazione della classe di esempio $calculateMyBid$ (tratto da [5])

È importante che la classe abbia un metodo $execute$ dichiarato esattamente come nell'esempio qui sopra, dal momento che *Jason* utilizza l'introspezione delle classi per richiamarlo. Gli argomenti dell'azione interna sono forniti come un array di termini ($Terms$). Da notare che questo è il terzo argomento fornito al metodo $execute$. Il primo argomento è il sistema di

transizione (così come definito dalla semantica operazionale di *AgentSpeak*) che contiene tutte le informazioni riguardo lo stato attuale dell'agente che sta per essere interpretato. Il secondo argomento rappresenta la funzione di unificazione determinata dall'esecuzione del piano (oppure dal test se il piano è applicabile o meno).

4.3.2 Strumenti e documentazione

Jason viene distribuito assieme ad una *IDE*[13] (*jEdit*) che fornisce un'interfaccia grafica per l'editing della configurazione di un sistema multi-agente così come del codice *AgentSpeak* dei singoli agenti. Attraverso l'*IDE* è anche possibile controllare l'esecuzione del sistema e distribuire gli agenti su di una rete in modo molto semplice. Ci sono sostanzialmente tre metodi di esecuzione:

- **Asincrono**: qui tutti gli agenti sono eseguiti in modo *asincrono*. Un agente passa al suo successivo ciclo di ragionamento non appena ha finito il suo ciclo corrente. Questo è il metodo di funzionamento di default, e come si può facilmente intuire non ha come obiettivo primario la performance;

- **Sincrono**: con questo metodo ogni agente effettua un singolo ciclo di ragionamento in ogni *passo di esecuzione globale*. Quando un agente porta a termine un ciclo di ragionamento informa il controllore di *Jason* e attende per un segnale di c*arry on*. Il controllore di *Jason* attende che tutti gli agenti abbiano finito il loro ciclo di ragionamento attuale, e poi manda a tutti quanti il segnale di *carry on*;

- **Debugging**: questo metodo di esecuzione è simile a quello sincrono. Ad ogni modo il controllore di *Jason* attende anche che l'utente clicchi un tasto *Step* nell'interfaccia grafica prima di mandare il segnale di *carry on* agli agenti. Questo metodo di esecuzione risulta essere molto utile nel caso l'utente debba individuare, analizzare e correggere un eventuale malfunzionamento del sistema.

[13] Integrated Development Environment

```
free. // I'm not currently handling unattended luggage

+unattendedLuggage(Terminal,Gate)  :  true
       <-      !negotiate.

@pn1
+!negotiate : not free
       <-     .broadcast(tell, bid(0)).

@pn2
+!negotiate : free
       <-     .myName(I); // Jason internal action
              +winner(I); // belief I am the negotiation winner
              +bidsCount(1)/
              mds.calculateMyBid(Bid); // user internal action
              +myBid(Bid);
              .broadcast(tell, bid(Bid)).

©pbl // for a bid better than mine
+bid(B)[source(Sender)] :
                     myBid(MyBid) & MyBid < B &
                     .myName(I) & winner(I)
       <-     -winner(I);
              +winner(Sender).

@pb2 // for other bids (and I'm still the winner)
+bid(B)     : .myName(I) & winner(I)
       <-     laddBidCounter;
              !endNegotiation.

©pendl // all bids was received
+!endNegotiation : bidsCount(N) & numberOfMDS(M) & N >= M
       <-     -free; // I'm no longer free
              !checkUnattendedLuggage.

@pend2 // void plan for endNegotiation not to fail
+!endNegotiation : true <- true.
…
```

Figura 4.11 Esempio di piani AgentSpeak per un robot della sicurezza in aeroporto (esempio tratto da [5])

C'è anche un altro strumento molto utile che viene fornito come parte dell'*IDE* e che permette all'utente di ispezionare gli stati interni degli agenti quando il sistema è in esecuzione in modalità di *debugging*. Tale strumento è chiamato *mind inspector,* ed è molto utile perché permette di ispezionare gli stati interni degli agenti in un sistema distribuito.

Jason viene distribuito corredato da una *ricca documentazione online*. Tale documentazione ha in pratica l'aspetto di un *tutorial* del linguaggio *AgentSpeak*, seguito però anche da una ricca descrizione delle caratteristiche e delle modalità d'uso della piattaforma. Tale documentazione è studiata sia per gli utenti alle prime armi ma anche per coloro che hanno già dimestichezza con *AgentSpeak* e che vogliono utilizzare al meglio tutte le potenzialità che *Jason* offre.

4.3.3 Osservanza degli standard, interoperabilità e portabilità

Dal momento che *Jason* è implementato in *Java* non vi è alcun problema di portabilità, anche se per ora questi aspetti non sono stati più di tanto presi in considerazione. Ad ogni modo le componenti della piattaforma possono essere facilmente modificate dall'utente. Ad esempio, al momento ci sono tre architetture di sistema disponibili con la distribuzione *Jason*: una *centralizzata* (il che significa che il sistema viene eseguito su di una singola macchina) e altre due distribuite che fanno uso rispettivamente di *JADE* e di *SACI*. Dovrebbe anche essere ragionevolmente semplice produrre un'altra architettura di sistema che usi altre piattaforme per la distribuzione e la gestione di agenti in un sistema.

4.4 Applicazioni supportate dal linguaggio e dalla piattaforma

Al momento *Jason* è utilizzato per poche applicazioni (alcune delle quali sono descritte qui di seguito) e per semplici progetti universitari portati avanti da studenti. Ad ogni modo, grazie alle sue basi di *AgentSpeak* è chiaramente indicato per una vastissima gamma di applicazioni che fanno uso di sistemi *BDI* (come ad esempio *PRS* o *dMARS*). Nonostante *Jason* abbia un vastissimo raggio di applicazioni nel futuro, come ad esempio la *semantica web* e le *applicazioni grid-based*, una particolare area di interesse è quella della *Social Simulation* [28]: infatti *Jason* è attualmente usato anche all'interno di un grande progetto che ha come scopo quello di produrre una piattaforma fatta su misura a questo proposito. Tale piattaforma è chiamata *MAS-SOC* e include un linguaggio di alto livello detto *ELMS* che serve a definire ambienti multi-agente. Un'altra area applicativa che è stata attualmente solo parzialmente esplorata riguarda l'uso di *AgentSpeak* per definire il comportamento di personaggi animati per l'animazione al computer (o *realtà virtuale*).

4.5 Considerazioni finali

Jason è una piattaforma in continuo aggiornamento ed estensione. L'obiettivo a lungo termine che i programmatori si sono posti è quello di avere una piattaforma che renda disponibili importanti tecnologie emergenti dalla ricerca nell'area dei sistemi multi-agente, ma raggiungere

questo obiettivo in modo tale da evitare che il linguaggio diventi ingombrante nonché avere una semantica formale per tutte le caratteristiche essenziali disponibili in *Jason*.

Attualmente ci sono progetti che mirano ad estendere *Jason* a svariate associazioni, dal momento che la struttura sociale è un aspetto essenziale per lo sviluppo di complessi sistemi multi-agente.

VALUTAZIONE DELLE PRESTAZIONI

5.1 Introduzione alla fase di valutazione

La descrizione (effettuata nel *capitolo 2* della tesi) relativa ai *metodi* e ai *modelli* per l'analisi delle prestazioni di un sistema ha permesso di prendere in considerazione e di analizzare alcune delle alternative possibili presenti in letteratura.

Il metodo scelto per la valutazione delle piattaforme *Jadex* (*versione 0.96*) e *Jason* (*versione 1.3.2*), che in seguito ne permetterà il confronto prestazionale, è quello della *misurazione diretta*, che rientra nella categoria delle tecniche di misurazione. Tutte le misurazioni che verranno effettuate sulle due piattaforme saranno di tipo quantitativo e, in quanto *open source*, saranno realizzate modificando direttamente i codici sorgenti delle due piattaforme ad agenti (i sorgenti sono allegati ad entrambe le distribuzioni, quindi non ci sono stati problemi per il loro reperimento).

Qui di seguito, nel *paragrafo 5.2*, riporto una breve descrizione delle caratteristiche *hardware* e *software* dei PC sui quali sono state effettuate le misure mentre nel *paragrafo 5.3*

riporto l'indicazione della versione *Java* utilizzata, con anche alcune precisazioni su problemi di compatibilità che ho avuto modo di riscontrare durante i test riguardanti la migrazione di agenti su host multipli. Gli strumenti e gli indici che verranno adottati sono brevemente descritti nel *paragrafo 5.4*, mentre dal *paragrafo 5.5* entreremo nel vivo della fase di test illustrando dapprima le tecniche utilizzate per la raccolta dei dati di interesse, per poi passare ad un'analisi che si dividerà sostanzialmente in due parti:

- **analisi su host singolo:** in questa sezione parleremo dei dati raccolti dalle piattaforme in esecuzione su di un *singolo host*, quindi senza tenere in considerazione gli aspetti di migrazione e relativi alla rete. Verranno raccolti dati interessanti per questo frangente, come ad esempio i tempi di caricamento di entrambe le piattaforme, la memoria occupata inizialmente dalle piattaforme, i tempi di creazione di un agente, i tempi di eliminazione di un agente e così via.

- **analisi su host multipli:** in questa sezione parleremo invece degli aspetti inerenti il collegamento di vari host secondari a quello che viene comunemente definito *Main Container*. Dal momento che sia *Jadex* che *Jason* sfruttano le librerie *JADE* per gli aspetti inerenti la migrazione degli agenti, raggrupperemo in un'unica parte l'analisi delle due piattaforme unificando quindi i risultati (in quanto si sono appunto rivelati identici dopo vari test), ma vedremo più avanti nel dettaglio la spiegazione di questa particolare situazione. Verranno raccolti dati interessanti per questo frangente, come ad esempio il tempo di connessione di un container secondario al *Main Container*, il carico di memoria iniziale del *Main Container*, il tempo di avvio del *Main Container*, il tempo di migrazione di un agente tra due host nella rete, il tempo di clonazione di un agente e così via.

5.2 Organizzazione delle misure

In questo paragrafo descriveremo brevemente quelle che sono le caratteristiche, sia *hardware* che *software*, delle macchine che useremo per effettuare tutti i test contenuti nel prosieguo di questo lavoro di tesi.

	PC_1_MAIN	*PC_2*	*PC_3*
Modello	Notebook Acer® Aspire™ 5920G	Desktop HP® Pavilion™ P6373IT	Notebook HP® Presario™ C700
Processore	Intel® Core™ 2 Duo CPU T7500 @ 2,20 GHz	AMD® Phenom™ 8600 Triple-Core CPU @ 2,30 GHz	Intel® Pentium™ Dual CPU T2310 @ 1,47 GHz
Versione BIOS	v1.3811 [31.03.2008]	v5.27 [26.09.2008]	vF.08 [31.01.2008]
Memoria RAM	4,00 GB	3,00 GB	2,00 GB
Sistema operativo	Microsoft® Windows™ Seven Professional	Microsoft® Windows™ Vista Home Premium	Microsoft® Windows™ Vista Home Premium
Tipologia di sistema operativo	Sistema operativo a 64 bit	Sistema operativo a 32 bit	Sistema operativo a 32 bit
Scheda video	NVIDIA® GeForce™ 8600M GT 512MB	NVIDIA® GeForce™ 9300GE 1GB	Mobile Intel® 965 Express™ 384 MB
Scheda di rete	Broadcom® Netlink™ Gigabit 10/100 Mbps Ethernet	NVIDIA® nForce™ 10/100/1000 Mbps Ethernet	Realtek® RTL8139/810x 10/100 Mbps Ethernet

Figura 5.1 Riepilogo delle caratteristiche hardware e software delle macchine utilizzate per i test

Va precisato che per quanto riguarda le misurazioni su *singolo host* tutte e tre le macchine sono state usate per la raccolta dei dati al fine di effettuare un confronto più approfondito e stabilire delle dipendenze hardware e software dei dati raccolti, mentre per quanto riguarda le misurazioni su host multipli *PC_1_MAIN* è stato utilizzato come *Main Container* per tutte le prove effettuate, mentre *PC_2* e *PC_3* sono stati utilizzati principalmente per instaurare una connessione remota e quindi per simulare uno scenario realistico di comunicazione tra entità situate su host diversi (vedi *figura 5.2*).

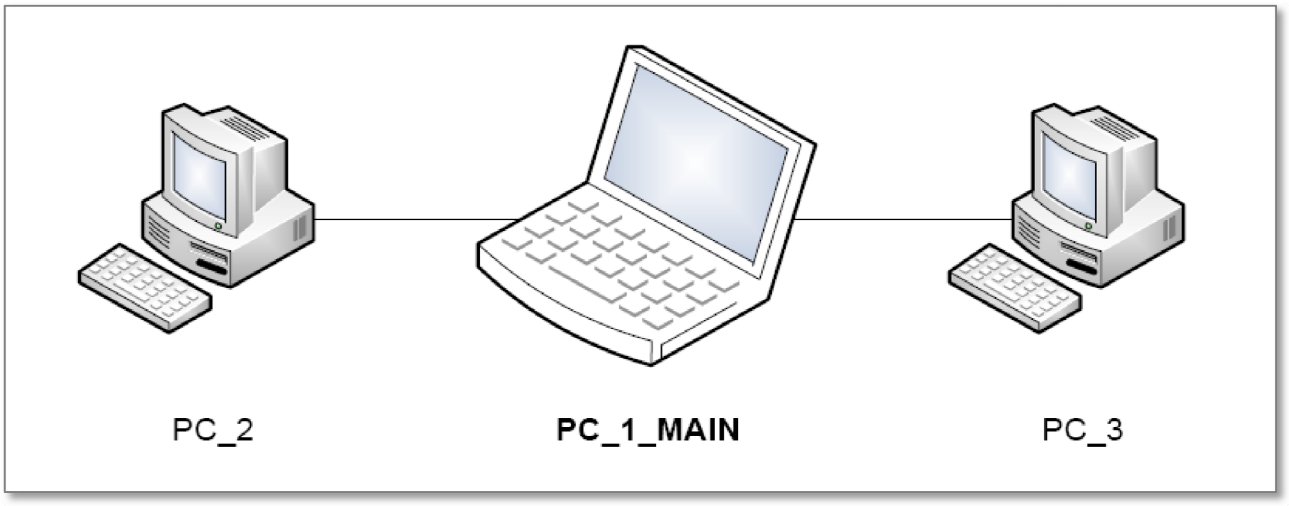

Figura 5.2 Schema della connessione tra le tre macchine a disposizione per la fase di test su host multipli

Per tutte le misure effettuate è stata utilizzata una connessione di rete di tipo *ethernet*, cercando così di evitare una situazione di indisponibilità del collegamento e di ridurre eventuali disturbi dovuti all'utilizzo, per esempio, di reti *WLAN*. È stato inoltre fatto in modo che il traffico di altri host non influisca sulle misure.

5.3 Versione Java utilizzata

Sulle varie macchine usate per i test (descritte nel paragrafo precedente) è stata installata la stessa versione della piattaforma *Java* al fine di cercare di uniformare il più possibile le misurazioni e i dati raccolti. Ricordo che entrambe le piattaforme *Jadex* e *Jason* si basano su *Java*, quindi il loro funzionamento implica necessariamente la presenza nel sistema in cui sono in esecuzione anche della suddetta piattaforma*,* appunto. La versione utilizzata è la seguente:

- *Java 2 SE 5.0 Update 22*

Una precisazione: avevo inizialmente cominciato la fase di test installando su tutte le macchine la versione *Java* più recente al momento della stesura della tesi (*Java 2 SE 6.0 Update 20*), ma quest'ultima creava alcuni problemi in occasione dei test su host multipli dal momento che era stato cambiato il metodo di caricamento delle librerie personalizzate. Questo fatto poteva essere ovviato sostituendo le librerie *JADE*, già integrate nei due programmi nella versione *3.7*, con le più recenti in versione *4.0*, ma questo avrebbe richiesto anche la modifica intensiva di gran parte dei file sorgenti di *Jadex* e *Jason*, cosa assolutamente improponibile e incompatibile con i limiti di tempo a mia disposizione.

La scelta più logica è stata dunque quella di utilizzare le suddette librerie *Java 2 SE 5.0 Update 22*, che comunque non differiscono in maniera eclatante dalle loro sorelle più aggiornate. Ovviamente sono state scaricate e installate su tutte quante le macchine a disposizione sia la versione *SDK (Software Development Kit)* che *JRE (Java Runtime Environment)*, rispettivamente nell'edizione a 64 bit per *PC_1_MAIN* che monta un sistema operativo a 64 bit e nell'edizione a 32 bit per *PC_2* e *PC_3*.

5.4 Strumenti di analisi dei dati e indici di prestazione

Gli strumenti che vengono proposti qui di seguito riguardano sia l'analisi temporale che l'analisi dei carichi di memoria, ed in particolare alcuni di essi si basano sul concetto di tempo di esecuzione: esso risulta definito come il tempo tra l'inizio ed il completamento di un lavoro, ed è il valore che viene percepito da un singolo utente di un calcolatore.

Per l'analisi dei dati ricavati dalle misure effettuate si utilizzeranno nel seguito di questo lavoro di tesi le seguenti funzioni matematico/statistiche:

- **Media aritmetica:** è data dalla somma di uno specifico campione di dati (variabile aleatoria X i cui valori sono indicati con x_i) diviso il numero n dei dati stessi. Indica il centro della distribuzione della variabile aleatoria X, appunto.

  In formula: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$

- **Varianza:** esprime una misura della concentrazione dei valori dei dati attorno al loro valore medio.

  In formula: $\sigma^2 = \frac{1}{n(n-1)}[n\sum x^2 - (\sum x)^2]$

- **Coefficiente di dispersione:** indica di quanto i valori dei dati si discostano dalla media, e spesso viene espresso in percentuale.

  In formula: $\sigma/\bar{x}$ $dove\ \sigma\ risulta\ la\ derivazione\ standard, ovvero\ la\ radice\ quadrata\ della\ varianza\ descritta\ precedentemente$

5.5 Misurazioni su singolo host

Iniziamo ora ad affrontare la parte di analisi prettamente quantitativa della tesi. In questo paragrafo illustreremo varie misurazioni effettuate su entrambe le piattaforme (in esecuzione su di un singolo host) riguardanti sia tempi che carichi di memoria, cercando di fornire all'utente delle informazioni significative che potrà sfruttare per selezionare una delle due piattaforme in base alle sue esigenze personali. Come detto prima, le misurazioni nel caso dell'host singolo sono state effettuate su tutte e tre le macchine a disposizione e poi rappresentate in grafici utilizzando diversi colori in modo da distinguere i risultati ottenuti e permettere di stabilire, se esistono, delle dipendenze hardware o software nei dati raccolti.

Per l'avvio della piattaforma *Jason* dovremo solamente cliccare sull'icona relativa al programma (avendo cura però di impostare le directory delle librerie *Java* e *JADE* dopo il primo avvio), mentre per avviare la piattaforma *Jadex* dovremo digitare, da riga di comando, la seguente stringa:

```
java jadex.adapter.standalone.Platform
```

Notiamo da questa istruzione che in realtà la piattaforma *Jadex* viene caricata allo stesso modo di un agente, infatti la sintassi usata è esattamente la stessa.

5.5.1 Tempi di caricamento delle piattaforme

Qui raccoglieremo semplicemente i dati relativi ai tempi impiegati da entrambe le piattaforme per il caricamento dell'interfaccia grafica specifica e per raggiungere la piena operatività. Va precisato che *Jadex* forniva già questa informazione all'avvio senza bisogno di effettuare alcun tipo di modifica, mentre per quanto riguarda *Jason* abbiamo utilizzato il metodo *Java* **System.currentTimeMillis()** per estrapolare questo dato. Entrambe le misurazioni sono state effettuate in millisecondi.

JADEX			JASON		
PC_1_MAIN	PC_2	PC_3	PC_1_MAIN	PC_2	PC_3
983	1.431	2.948	1.727	2.702	3.780
1.045	1.421	2.870	1.650	2.654	3.759
1.014	1.447	2.855	1.875	2.931	3.604
1.030	1.446	2.917	1.594	3.003	3.726
1.015	1.480	2.890	1.714	2.745	3.685
980	1.399	2.915	1.823	2.893	3.420
1.021	1.402	2.923	1.662	2.898	3.914
994	1.287	2.879	1.690	2.928	3.528
1.003	1.394	2.799	1.721	3.012	3.690
997	1.421	2.870	1.731	2.411	3.501
1.008,20	1.412,80	2.886,60	1.718,70	2.817,70	3.660,70

Figura 5.3a Tempi di avvio, espressi in millisecondi (l'ultima riga contiene la media delle misurazioni)

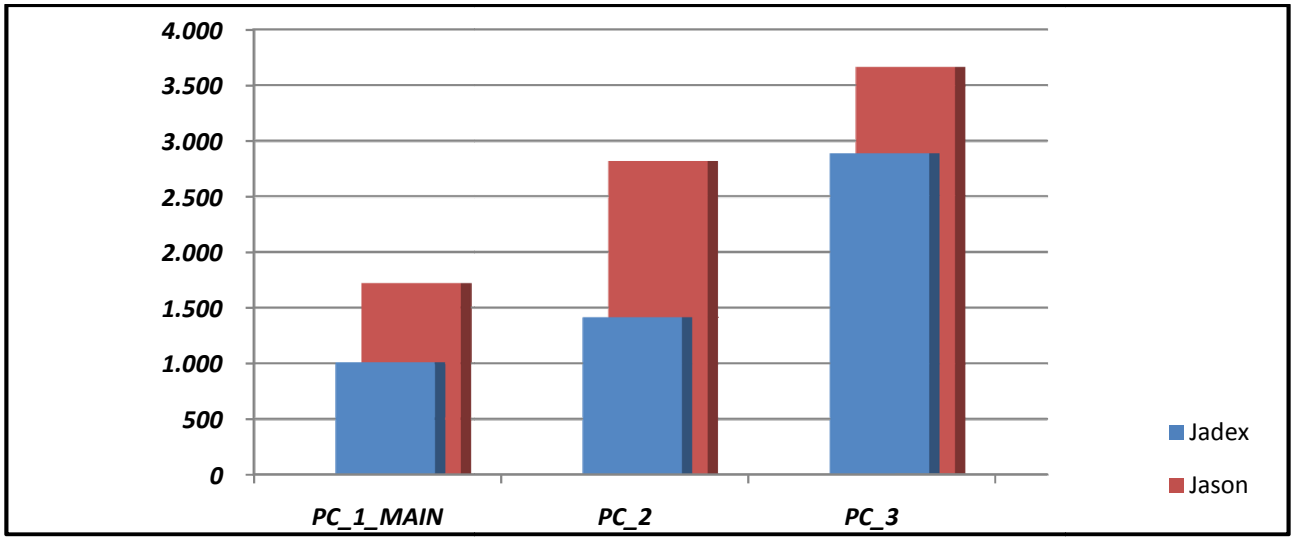

Figura 5.3b Grafico comparativo dei tempi di caricamento delle piattaforme Jadex e Jason

Salta subito all'occhio il fatto che *Jadex* impiega, in ognuna delle macchine usate per i test, un tempo decisamente minore rispetto a *Jason*. Dal grafico in *figura 5.3b* notiamo che la configurazione più performante è quella associata a *PC_1_MAIN*: consultando la tabella dei dati tecnici contenuta nel *paragrafo 5.2* notiamo che *PC_1_MAIN* monta una quantità di memoria *RAM* maggiore rispetto alle altre due macchine, 4GB contro rispettivamente 3GB e 2GB di *PC_2* e *PC_3*. Possiamo concludere che *i tempi di avvio della piattaforma sono strettamente legati al quantitativo di memoria RAM del sistema*, dato che il processore di *PC_2* è comunque più potente del processore di *PC_1_MAIN*. Altro fatto che può sicuramente influenzare i tempi di caricamento è il *sistema operativo*, che in *PC_1_MAIN* è a *64 bit*: questa caratteristica ha permesso l'installazione della distribuzione *Java* specifica per questa architettura.

5.5.2 Carico di memoria iniziale

In questa sezione raccoglieremo i dati inerenti il carico di memoria all'avvio di entrambe le piattaforme. La mia scelta è stata quella di effettuare la *misurazione delle singole piattaforme ad agenti* (escludendo quindi il carico di memoria relativo alla piattaforma *Java*), in modo da considerare il loro peso sul sistema in termini di memoria *RAM* occupata all'avvio (naturalmente questa è destinata ad essere allocata in quantità maggiori se i vari agenti o lo stesso ambiente in esecuzione dovessero richiederlo). Va precisato che *Jason* fornisce già questa informazione all'avvio senza bisogno di effettuare alcun tipo di modifica, mentre per quanto riguarda *Jadex* abbiamo utilizzato il metodo Java **Runtime.getRuntime()** per estrapolare questo dato, facendo in seguito la differenza tra i valori ritornati dai metodi **totalMemory()** e **freeMemory()**. I valori sono stati raccolti in KB (kilobytes).

JADEX			JASON		
PC_1_MAIN	PC_2	PC_3	PC_1_MAIN	PC_2	PC_3
4.261	4.263	4.238	6.892	7.827	6.811
4.251	4.240	4.247	6.861	7.967	6.928
4.247	4.239	4.236	7.081	7.968	6.930
4.253	4.249	4.245	6.867	7.921	6.933
4.251	4.250	4.250	6.980	7.890	6.908
4.252	4.248	4.245	6.992	7.932	6.914
4.259	4.252	4.256	6.892	7.896	6.914
4.259	4.260	4.256	6.895	7.935	6.909
4.261	4.239	4.260	6.950	7.921	6.926
4.257	4.252	4.245	6.987	7.967	6.930
4.255	4.249	4.248	6.940	7.922	6.910

Figura 5.4a Occupazione di memoria, dati espressi in KB (l'ultima riga contiene la media delle misurazioni)

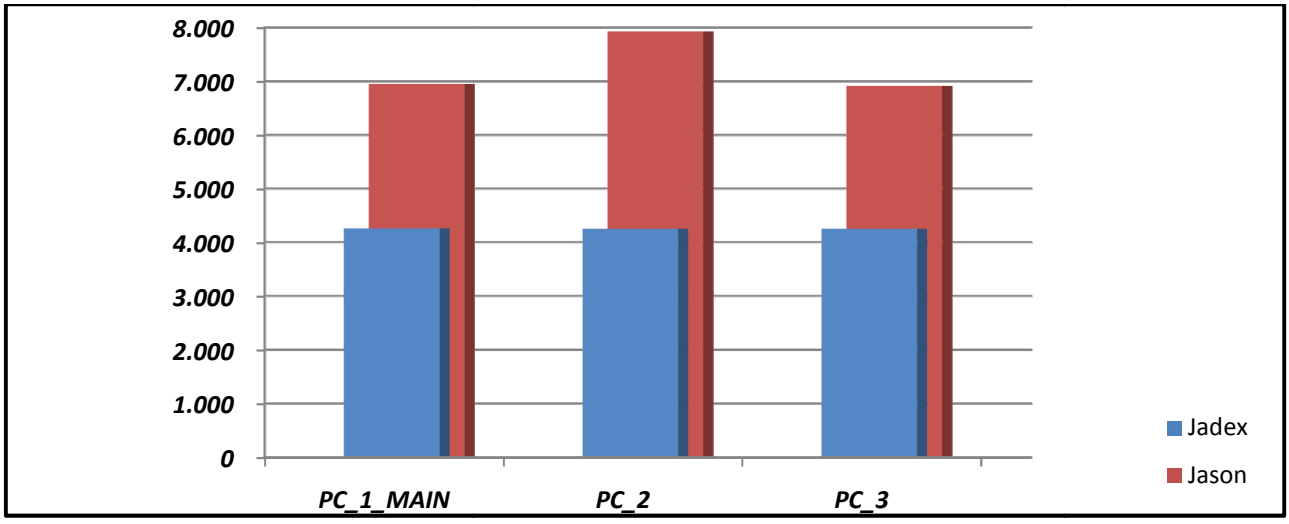

Figura 5.4b Grafico comparativo del carico di memoria RAM inizialmente occupara dalle piattaforme Jadex e Jason

Notiamo facilmente che la piattaforma *Jadex è decisamente più leggera di Jason*, occupando poco più della metà della memoria *RAM*. Un altro fattore interessante da osservare è che *Jadex* ha un carico di memoria uniforme su tutte e tre le macchine usate per i test, mentre notiamo, per quanto riguarda *Jason*, un picco di memoria *RAM* più alto per quanto riguarda *PC_2*. Ciò può essere dovuto al fatto che *Jason* all'avvio deve caricare anche l'editor *jEdit*, il quale potrebbe riservare un diverso quantitativo di memoria *RAM* a seconda del sistema su cui viene caricato.

5.5.3 Tempo di creazione di un agente

Raccogliamo in questa sezione i dati relativi ai tempi di creazione di un agente. I tempi in questo caso sono espressi in microsecondi al fine di avere una maggiore precisione, e sono stati raccolti attraverso il metodo *Java* **System.nanoTime()** e moltiplicati per 1000.

Notiamo dal grafico in *figura 5.5b* una prestazione migliore da parte della piattaforma *Jadex*, anche se non così marcata. Anche in questo caso la macchina più performante tra le tre utilizzate per i test è stata *PC_1_MAIN*, subito seguita da *PC_2* e poi da *PC_3*, a conferma dell'ipotesi che la quantità di memoria *RAM* montata sulle macchine sia un aspetto cruciale per la velocità nell'esecuzione dei compiti delle piattaforme ad agenti.

JADEX			JASON		
PC_1_MAIN	**PC_2**	**PC_3**	**PC_1_MAIN**	**PC_2**	**PC_3**
5.788	8.070	14.897	7.936	10.192	22.054
5.796	9.353	13.000	7.122	8.371	13.178
5.188	6.520	14.637	6.449	8.271	24.624
5.949	6.809	15.483	6.819	9.044	15.387
5.780	7.890	14.045	7.503	10.078	19.233
5.882	7.782	15.123	7.891	9.812	20.980
5.706	6.954	15.089	7.832	9.456	18.451
5.234	8.953	14.325	6.920	9.579	18.900
5.449	7.801	14.560	7.002	8.344	17.283
6.254	8.587	15.753	8.850	9.194	18.789
5.703	7.872	14.691	7.432	9.234	18.888

Figura 5.5a Tempi di creazione di un agente, espressi in microsecondi (l'ultima riga contiene la media delle misurazioni)

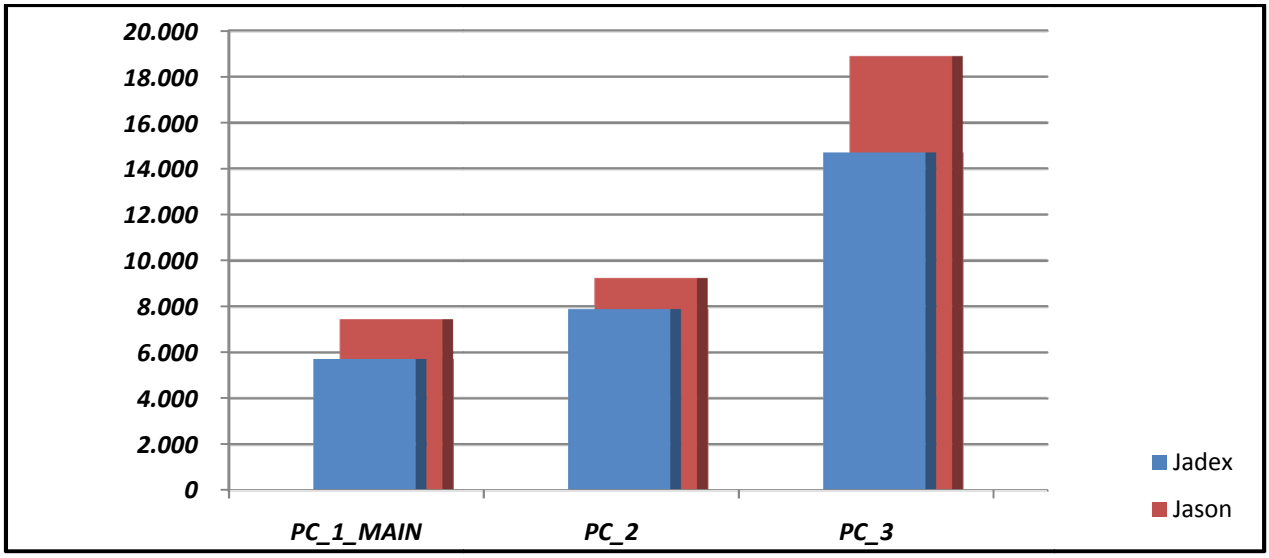

Figura 5.5b Grafico comparativo del tempo impiegato dalle due piattaforme per la creazione di un agente

5.5.4 Tempo di cancellazione di un agente

Raccogliamo qui i dati relativi ai tempi di cancellazione di un agente esistente. Anche in questo caso i tempi vengono espressi in microsecondi al fine di avere una maggiore precisione, e sono stati raccolti ancora una volta attraverso il metodo *Java* **System.nanoTime()** e moltiplicati per 1000. Notiamo sempre una performance sostanzialmente migliore per *Jadex*, anche se nel caso di *PC_1_MAIN* notiamo dal grafico che i risultati sostanzialmente si equivalgono. Salta subito all'occhio anche il fatto che i tempi richiesti per la cancellazione di un agente esistente siano nettamente minori rispetto ai tempi richiesti per la creazione dello stesso.

JADEX			JASON		
PC_1_MAIN	PC_2	PC_3	PC_1_MAIN	PC_2	PC_3
2.634	3.721	3.813	1.921	2.562	3.823
2.163	3.546	3.655	1.843	6.214	4.765
1.910	2.739	4.766	2.274	5.118	4.301
1.482	2.365	3.921	3.006	4.781	7.144
2.080	3.240	3.956	2.994	4.680	5.662
2.148	2.978	4.220	2.873	3.979	5.891
1.661	2.941	4.501	2.440	5.455	6.950
1.852	3.183	4.298	2.234	6.022	5.525
3.024	3.204	3.029	2.981	4.890	4.933
3.797	3.390	3.432	2.586	3.928	6.228
2.275	3.131	3.959	2.515	4.763	5.522

Figura 5.6a Tempi di cancellazione di un agente, espressi in microsecondi (l'ultima riga contiene la media delle misurazioni)

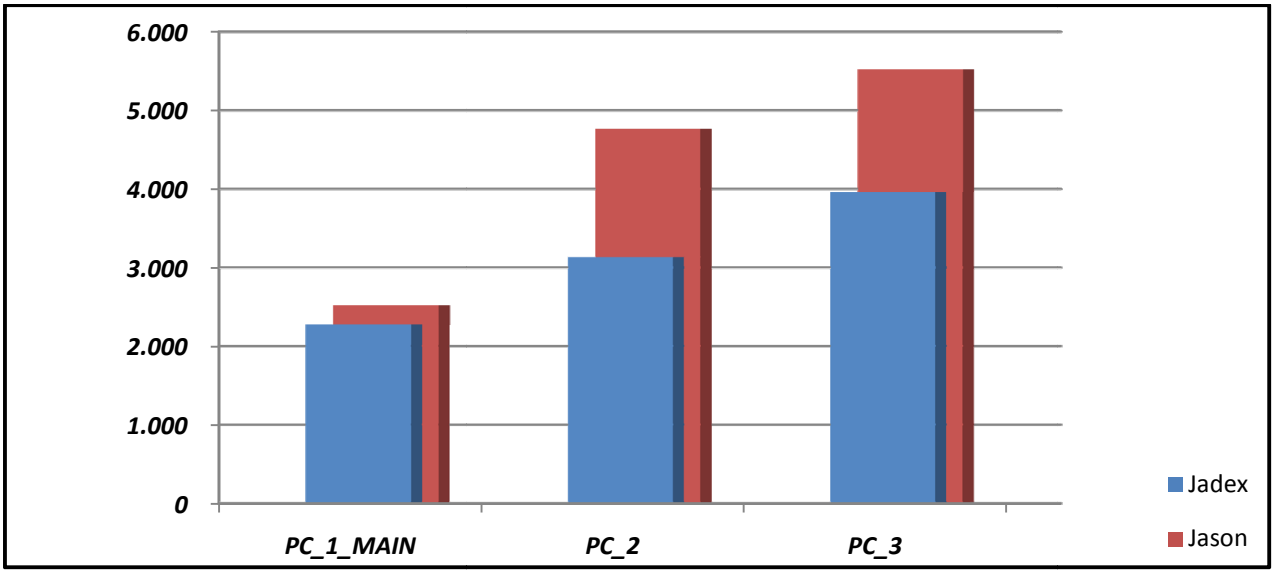

Figura 5.6b Grafico comparativo del tempo impiegato dalle due piattaforme per la cancellazione di un agente esistente

5.5.5 Tempi di sospensione e di resume di un agente

Passiamo ora ad analizzare i tempi richiesti per sospendere un agente (in pratica, per metterlo in standby) e per effettuare il resume di un agente precedentemente sospeso. Come nei casi precedenti, i valori sono stati raccolti in microsecondi al fine di avere una maggiore precisione con l'utilizzo del metodo *Java* **System.nanoTime()** e moltiplicati per 1000.

JADEX			JASON		
PC_1_MAIN	**PC_2**	**PC_3**	**PC_1_MAIN**	**PC_2**	**PC_3**
831	919	1.973	1.211	1.621	1.450
670	907	1.291	979	1.189	1.620
593	830	1.578	1.193	1.204	1.625
578	815	1.711	1.072	1.192	1.847
612	822	1.345	1.098	1.241	1.804
624	807	1.429	1.144	1.306	1.785
599	799	1.593	996	1.297	1.926
581	801	1.100	980	1.190	1.911
610	780	1.098	1.030	1.422	1.945
615	847	1.895	1.055	1.377	2.304
631	833	1.501	1.076	1.304	1.822

Figura 5.7a Tempi di sospensione di un agente, espressi in microsecondi (l'ultima riga contiene la media delle misurazioni)

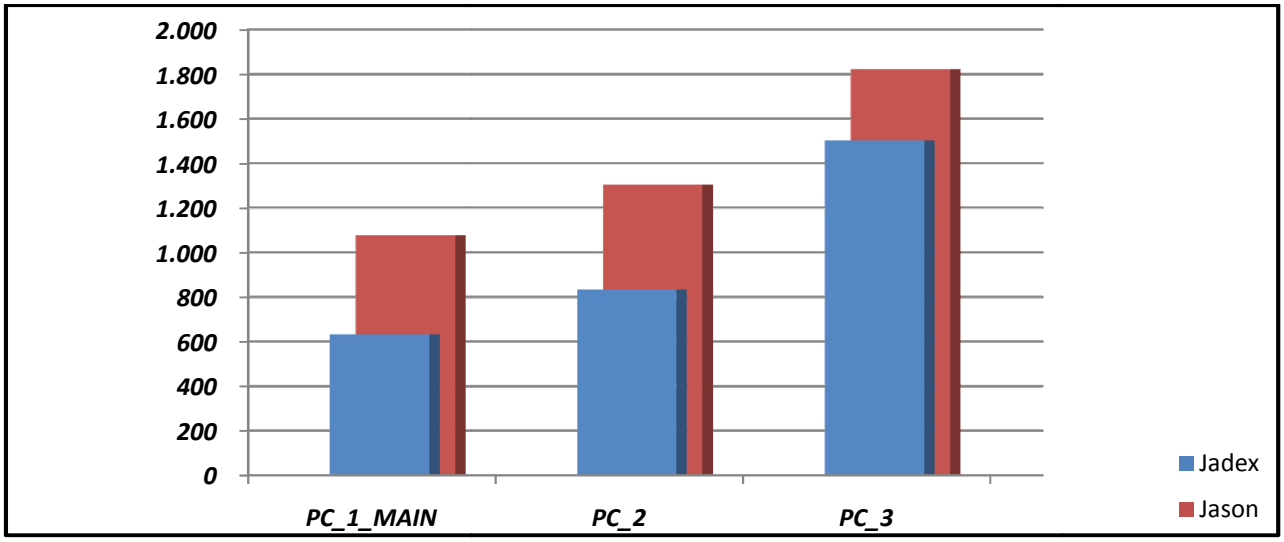

Figura 5.7b Grafico comparativo del tempo impiegato dalle due piattaforme per la sospensione di un agente esistente

JADEX			JASON		
PC_1_MAIN	PC_2	PC_3	PC_1_MAIN	PC_2	PC_3
851	924	1.721	1.294	2.088	2.190
1.225	836	1.770	1.188	1.723	2.461
557	1.259	2.265	1.250	1.694	2.379
507	1.011	1.875	1.096	1.701	1.903
890	1.194	1.900	1.390	2.007	2.415
782	1.081	1.895	1.452	1.916	1.989
956	1.195	1.946	1.366	1.955	1.950
905	968	1.802	1.296	1.832	2.263
832	1.007	1.799	1.680	1.870	2.305
921	1.082	1.678	1.711	1.809	2.608
843	1.056	1.865	1.372	1.860	2.246

Figura 5.8a Tempi di resume di un agente sospeso, espressi in microsecondi (l'ultima riga contiene la media delle misurazioni)

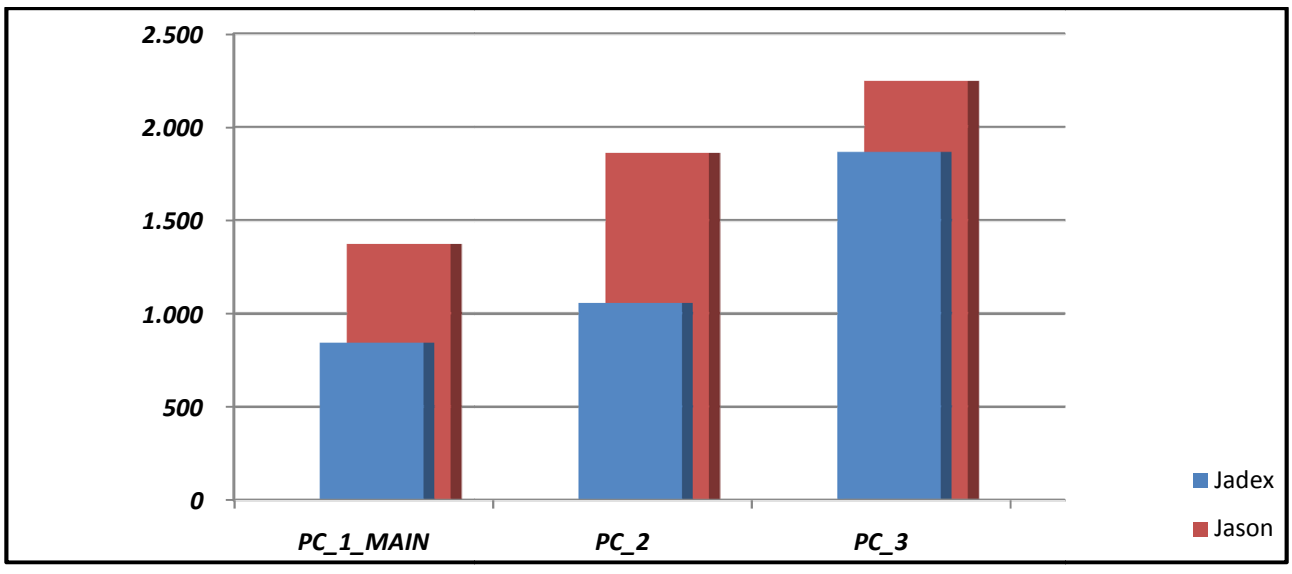

Figura 5.8b Grafico comparativo del tempo impiegato dalle due piattaforme per il resume di un agente sospeso

In questo caso entrambe le misurazioni sono sullo stesso ordine di grandezza, e ancora una volta notiamo una performance migliore da parte della piattaforma *Jadex* su tutte e tre le macchine utilizzate per i test.

5.6 Misurazioni su più host connessi

In questo paragrafo ci porremo come obiettivo quello di raccogliere i dati riguardanti le piattaforme distribuite su vari host, e quindi ovviamente anche i dati inerenti la migrazione degli agenti tra i vari host. Come detto prima sia la piattaforma *Jadex* che la piattaforma *Jason* si basano sulle librerie *JADE* per gli aspetti riguardanti la migrazione degli agenti, e dopo alcuni test specifici mi sono reso conto che i valori raccolti sono perfettamente identici. Quindi quello che andremo a fare ora sarà raggruppare l'analisi delle due piattaforme, tenendo presente che i risultati ottenuti possono essere applicati indifferentemente all'una oppure all'altra piattaforma.

Come già specificato in precedenza (*paragrafo 5.2*, vedi *figura 5.2*) per le misurazioni su più host connessi utilizzeremo sempre la macchina *PC_1_MAIN* come *Main Container*, mentre le macchine *PC_2* e *PC_3* verranno configurate come container secondari per instaurare una connessione remota e quindi per simulare uno scenario realistico di comunicazione tra entità situate su host diversi. Casi a parte rappresentano i *paragrafi 5.6.1* e *5.6.2*, nei quali misureremo rispettivamente il tempo di caricamento del *Main Container* e il carico di memoria iniziale dello stesso: in questi due casi effettueremo le misurazioni su tutte e tre le macchine a disposizione come nei casi discussi nel *paragrafo 5.5* al fine di ottenere una maggiore varietà ed individuare eventuali colli di bottiglia.

Per avviare le due piattaforme ad agenti con il supporto per *host multipli* (e quindi per la migrazione degli agenti tra i vari host) dobbiamo seguire due procedure diverse rispettivamente per ognuna delle piattaforme: per quanto riguarda *Jason* dovremo usare, all'interno del file di descrizione della piattaforma, la dicitura

```
infrastructure: Jade
```

per specificare la tipologia della piattaforma da caricarsi (cioè *JADE*) per poi gestirla da interfaccia grafica, mentre per *Jadex* dovremo utilizzare il comando

```
java jadex.adapter.jade.tools.Starter
```

che serve sostanzialmente a caricare l'adattatore *JADE* (*versione 0.96*, da installarsi come libreria aggiuntiva alla piattaforma standard) al fine di creare il *Main Container*, e in seguito utilizzare il comando

```
java jadex.adapter.jade.tools.Starter –host <main> container
```

per creare un container secondario da collegare al *Main Container* creato precedentemente (`<main>` dovrà appunto essere il nome dell'host del *Main Container*).

5.6.1 Tempo di caricamento del Main Container

In questo paragrafo raccoglieremo i dati riguardanti i tempi di caricamento del *Main Container*, al quale verranno in seguito collegati come nodi i vari container secondari. Come nel caso delle misurazioni per host singolo, le misure verranno effettuate raccogliendo i tempi in millisecondi grazie al metodo *Java* **System.currentTimeMillis()**.

JADEX / JASON - JADE		
PC_1_MAIN	**PC_2**	**PC_3**
1.449	1.929	2.590
1.444	2.008	2.215
1.434	1.870	2.184
1.448	2.114	2.186
1.449	2.010	2.103
1.438	1.970	2.221
1.440	2.012	2.419
1.436	1.978	2.168
1.434	1.990	2.593
1.441	1.899	2.200
1.441,31	1.978,00	2.287,90

Figura 5.9a Tempi di caricamento del Main Container, espressi in millisecondi

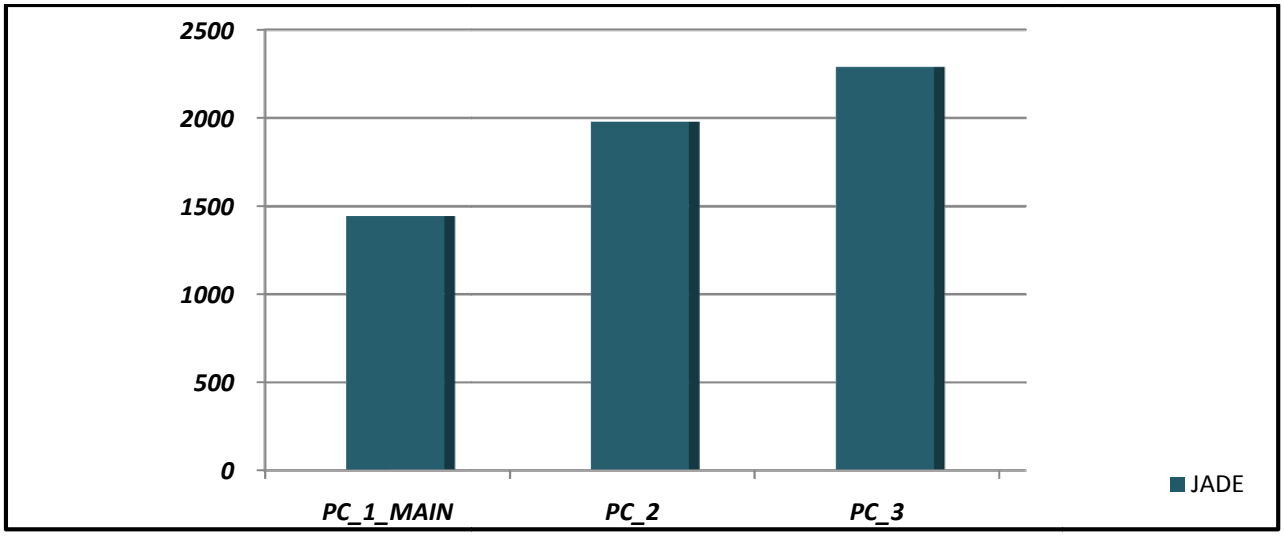

Figura 5.9b Grafico relativo al tempo impiegato dal Main Container su ogni macchina per essere completamente caricato

Notiamo subito che i tempi di caricamento del *Main Container* sono dello stesso ordine dei tempi calcolati nel *paragrafo 5.5.1* e relativi al caso con singolo host, quindi sotto questo punto di vista non ci sono grosse differenze. Anche in questo caso notiamo un incremento dei tempi a seconda della macchina utilizzata, dovuto probabilmente come nei precedenti casi alla differente quantità di memoria *RAM* installata su ognuna delle macchine.

5.6.2 Carico di memoria iniziale del Main Container

Raccoglieremo in questo paragrafo i dati relativi al carico di memoria all'avvio del *Main Container*. Anche in questo caso andremo a raccogliere i dati specifici della piattaforma ad agenti, tralasciando quindi la piattaforma *Java*. Abbiamo utilizzato il metodo *Java* chiamato **Runtime.getRuntime()** per estrapolare questo dato, facendo in seguito la differenza tra i valori ritornati dai metodi **totalMemory()** e **freeMemory()**. I valori sono stati raccolti in KB (kilobytes).

Notiamo subito che, a differenza del caso con singolo host, il carico di memoria iniziale necessario per il caricamento del *Main Container* e delle librerie *JADE* è molto inferiore e più o meno simile in tutte e tre le macchine utilizzate. Non dobbiamo però dimenticare che le piattaforme *Jadex* e *Jason* devono comunque essere caricate per poter utilizzare queste funzionalità, quindi il carico di memoria complessivo equivale alla somma del carico di memoria iniziale del *Main Container* e del carico di memoria iniziale di ognuna delle due piattaforme.

JADEX / JASON - JADE		
PC_1_MAIN	**PC_2**	**PC_3**
592	601	592
587	619	594
588	612	594
588	623	599
587	613	597
590	615	597
590	620	598
591	603	593
591	603	592
589	609	603
589,30	611,80	595,90

Figura 5.10a Dati relativi al carico di memoria iniziale del Main Container, valori espressi in KB

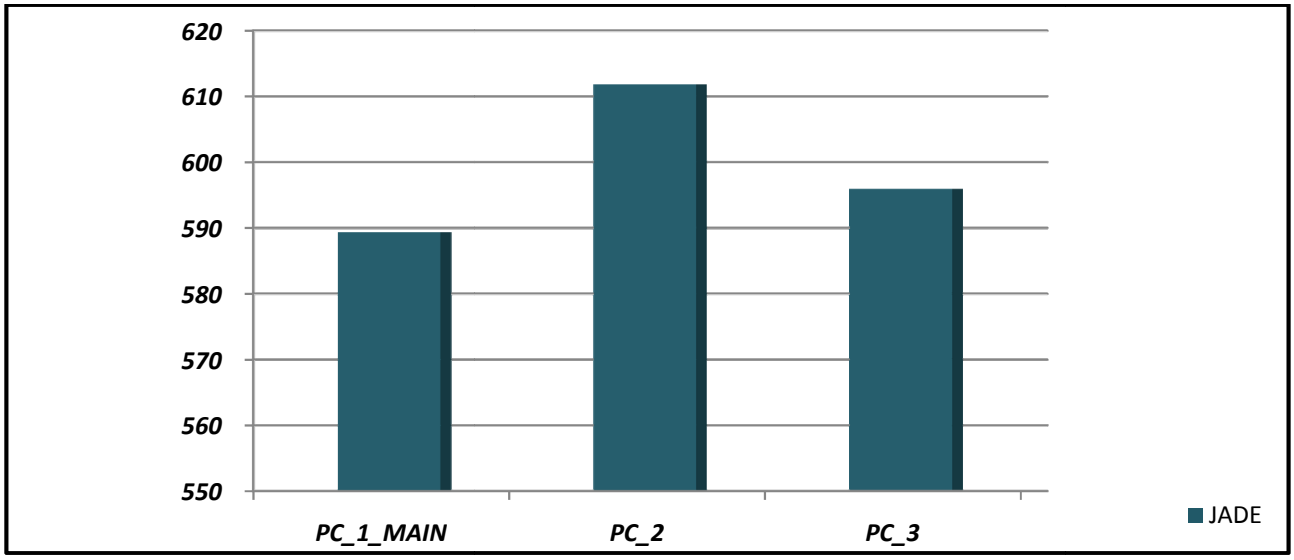

Figura 5.10b Grafico relativo al carico di memoria iniziale del Main Container su ognuna delle tre macchine per i test

5.6.3 Connessione di un container secondario al Main Container

Analizziamo ora i tempi necessari per la connessione di un container secondario al *Main Container*, considerando sia il tempo per la creazione del nuovo nodo che il tempo per la connessione del suddetto nodo al nodo principale (rappresentato appunto dal *Main Container*). In questo caso i valori saranno raccolti in microsecondi al fine di avere una maggiore precisione con l'utilizzo del metodo *Java* **System.nanoTime()** e moltiplicati per 1000. Saranno sostanzialmente effettuate due misurazioni con le macchine a nostra disposizione al fine di individuare eventuali

dipendenze, se esistono: le misurazioni riguarderanno la connessione tra le macchine *PC_1_MAIN* e *PC_2*, e la connessione tra le macchine *PC_1_MAIN* e *PC_3*.

JADEX / JASON - JADE	
Connessione tra PC_1_MAIN e PC_2	**Connessione tra PC_1_MAIN e PC_3**
791.477	834.399
786.991	809.470
768.888	776.418
790.226	767.153
789.936	801.446
792.522	800.953
786.408	796.340
787.788	823.701
790.995	800.912
801.491	773.384
788.672,20	798.417,60

Figura 5.11a Dati relativi ai tempi necessari per la connessione di un container secondario al Main Container. Valori espressi in microsecondi, l'ultima riga contiene la media

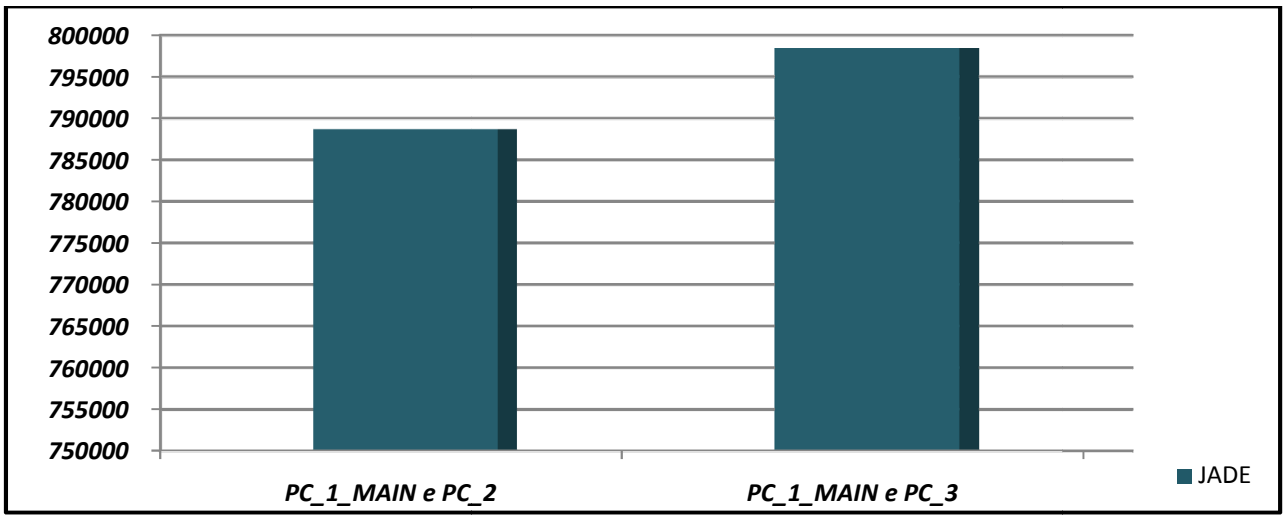

Figura 5.11b Grafico relativo ai tempi necessari per la connessione di un container secondario al Main Container, concordemente alla configurazione descritta nel *paragrafo 5.2*

Alla luce di queste misurazioni notiamo che il tempo di connessione ad ognuno dei due container secondari è pressochè uguale, solo leggermente maggiore nel caso della connessione tra *PC_1_MAIN* e *PC_3*. Questo fatto è plausibile, dal momento che la maggior parte delle istruzioni viene eseguita dal *Main Container* mentre solo una minima parte (relativa all'inizializzazione del nodo) viene eseguita dal *container secondario*.

5.6.4 Tempi di migrazione e di clonazione di un agente

Riportiamo infine qui di seguito i dati raccolti relativi ai tempi di clonazione di un agente e di migrazione di un agente da un host verso un altro host, indipendentemente che questi siano Main Container o container secondari in quanto sono state effettuate varie misurazioni per tutti i possibili casi. I valori saranno raccolti anche questa volta in microsecondi al fine di avere una maggiore precisione con l'utilizzo del metodo *Java* **System.nanoTime()** e moltiplicati per 1000.

JADEX / JASON - JADE	
Clonazione di un agente	**Migrazione di un agente**
475	582
480	487
476	449
475	632
483	602
490	597
478	712
475	468
481	590
492	495
480,50	561,40

Figura 5.12a Dati relativi ai tempi impiegati per la clonazione e la migrazione di un agente
Valori espressi in microsecondi, l'ultima riga contiene la media

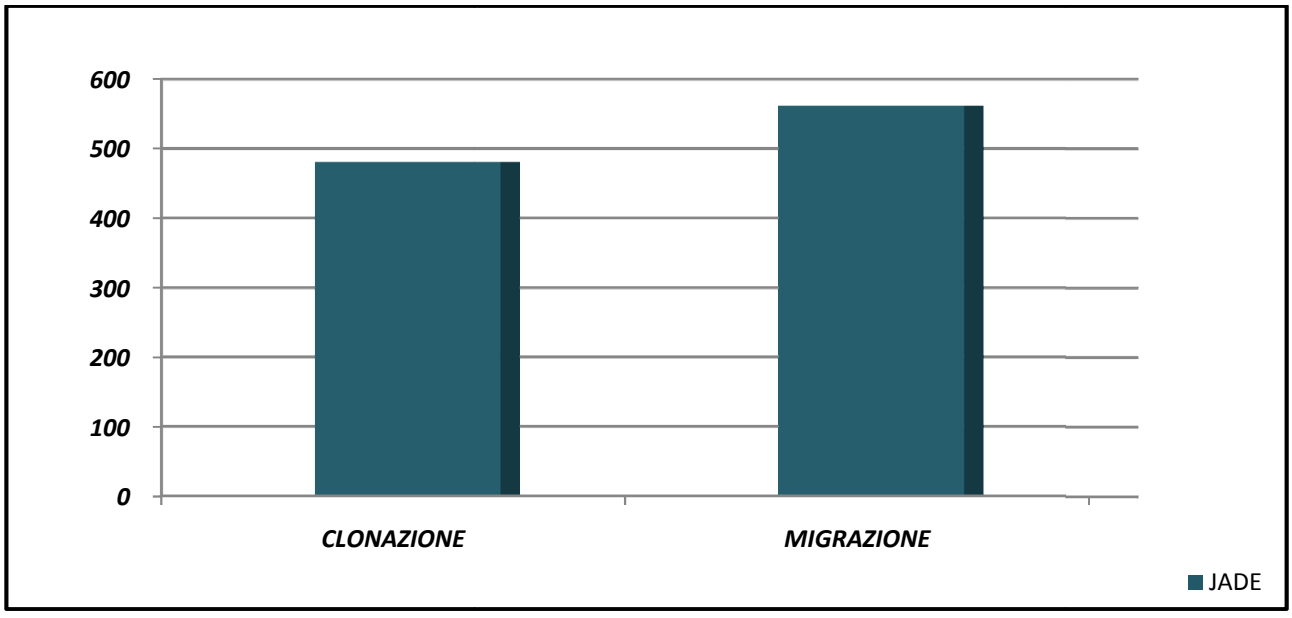

Figura 5.12b Grafico comparativo dei tempi impiegati per la clonazione e la migrazione di un agente

Come salta subito all'occhio, i tempi richiesti sono veramente esigui e pressoché indistinguibili per l'utente finale. Va specificato che tutti i test effettuati sono stati svolti con agenti il più possibile minimali, quindi se si dovessero far migrare (oppure clonare) agenti di dimensioni più consistenti (che portino con se dei dati, ad esempio) il tempo aumenterebbe indefinitamente. Quello che mi premeva calcolare in questa sede era il tempo impiegato dalle librerie *JADE* per portare a compimento la lista di azioni necessarie per il raggiungimento dello scopo.

Va precisato anche che, per come sono strutturate le librerie *JADE*, non è possibile che una stessa macchina faccia migrare nello stesso momento due agenti, quindi l'operazione che verrà effettuata sarà di tipo sequenziale a meno che gli agenti che devono migrare non siano fisicamente su due macchine diverse (o su due container diversi, a patto che i due container non siano istanziati su di una stessa macchina). Sarebbe possibile ovviamente implementare un nuovo metodo nelle librerie, chiamandolo ad esempio **moveMultipleAgents()**, adibito a far migrare allo stesso tempo più agenti (ovviamente rispettando i limiti fisici imposti dal canale di connessione tra gli host), ma non sarà comunque oggetto di questo lavoro di tesi.

CONCLUSIONI

L'obiettivo di questo lavoro è stato quello di analizzare le due piattaforme ad agenti *Jadex* e *Jason* e di confrontare le loro prestazioni e performance, dapprima a livello qualitativo e in seguito a livello quantitativo dopo aver effettuato vari test su macchine diverse al fine di stabilire, se esistono, eventuali colli di bottiglia o dipendenze.

Dalla prima fase di analisi qualitativa sono emersi risultati molto interessanti: innanzi tutto, essendo entrambe le piattaforme basate su *Java*, la modifica dei sorgenti e l'integrazione di nuovi moduli nelle piattaforme risulta possibile e anche relativamente semplice, cosa assolutamente non da sottovalutare. Per quanto riguarda i due linguaggi nello specifico, *Jason* implementa una versione migliorata di *AgentSpeak* - *AgentSpeak(L)*, appunto - mentre *Jadex* si basa prevalentemente su *XML* per la descrizione degli agenti. Dal momento che, per esigenze relative al lavoro di tesi da svolgere, ho avuto a che fare con entrambe le tipologie di linguaggi, posso affermare che *AgentSpeak(L)* risulta essere molto più chiaro e di facile assimilazione, soprattutto quando si devono scrivere formule complesse e strutturate. Come detto, entrambe le piattaforme si appoggiano alle librerie *JADE* per quanto riguarda la parte relativa agli host multipli e alla migrazione degli agenti. Nonostante ciò *Jason* può sfruttare anche la tecnologia *SACI*, altro aspetto da non sottovalutare e che potrebbe rivelarsi utile a molti sviluppatori.

Passando alla parte di analisi quantitativa del lavoro di tesi, dai dati raccolti emerge abbastanza palesemente che *Jadex* risulta migliore sotto tutti gli aspetti considerati rispetto a *Jason*, soprattutto per quanto concerne il carico di memoria iniziale e i tempi di caricamento della piattaforma (motivo questo che ha portato anche alla realizzazione di una versione di *Jadex* apposita per dispositivi portatili[14]). I tempi relativi alla creazione di un agente, alla cancellazione di un agente, alla sospensione e alla ripresa sono puramente indicativi in quanto possono cambiare a seconda del tipo di agente: come detto, ho cercato di effettuare i test su modelli di agenti minimali in modo da estrapolare il tempo di calcolo impiegato dalla piattaforma.

Aspetto che però va sicuramente a favore di *Jason* è la frequenza con cui vengono rilasciati aggioramenti per la piattaforma rispetto a *Jadex*, fattore questo che indica una maggiore attenzione da parte degli sviluppatori che continuano a seguirlo costantemente (anche durante la stesura della tesi è stato rilasciato un aggiornamento, la *versione 1.3.2*, il che mi ha portato necessariamente a ricominciare la fase di raccolta dati dall'inizio). Va detto comunque che è disponibile una versione beta più aggiornata di *Jadex*, cioè la *versione 2.0 RC*, ma dal momento che per questa versione non è contemplato il supporto per l'adattatore *JADE* ho dovuto necessariamente basarmi sulla *versione 0.96 ufficiale*, altrimenti non avrei potuto utilizzare le funzioni di migrazione degli agenti. Vanno inoltre ricordati i problemi che ho dovuto affrontare a causa della mancanza di aggiornamenti rilasciati da *Jadex* in merito alla compatibilità con l'ultima versione di *Java* disponibile (vedi *paragrago 5.3* per una descrizione più dettagliata del problema incontrato), problemi che mi hanno rubato parecchio tempo non tanto per essere risolti ma piuttosto per essere individuati in quanto non descritti da nessuna parte.

Per concludere posso affermare che, nonostante *Jadex* risulti molto più performante di Jason nell'assolvere i propri compiti nonché molto più snello a livello di carico di memoria e di tempi di avvio iniziali, giocano sicuramente a suo sfavore gli aggiornamenti sporadici della piattaforma (la *versione 0.96* utilizzata, nonché la versione ufficiale più recente, risale al *15 giugno 2007*). *Jason*, da parte sua, risulta essere meno performante rispetto a *Jadex* e leggermente più pesante in memoria e lento all'avvio (non dimentichiamo comunque che la piattaforma integra anche un comodo editor creato appositamente), ma ha però dalla sua aspetti positivi come il

[14] Vedi [21] per maggiori informazioni e dettagli più approfonditi

linguaggio *AgentSpeak(L),* che risulta essere estremamente intuitivo ed elegante nella stesura di programmi, e una frequenza di aggiornamenti rispettabilissima. Naturalmente, come detto prima, ci sono poi aspetti positivi comuni ad entrambe le piattaforme, e cioè il fatto di essere *open source* (sotto licenza *GNU LGPL*) e di essere basate entrambe su *Java*.

Possiamo affermare, a titolo puramente indicativo, che gli utenti che abbiano già maturato una discreta esperienza con *XML* sicuramente si troveranno molto più a loro agio con *Jadex*, mentre gli utenti che già conoscono *AgentSpeak* sarebbero probabilmente più propensi a preferire *Jason* in quanto il linguaggio *AgentSpeak(L)* è un adattamento del suddetto. Per il resto le potenzialità offerte dalle due piattaforme sono assolutamente le stesse dal momento che si basano entrambe sul modello BDI, e sia per l'una che per l'altra possiamo trovare anche svariati tool che possono semplificare il lavoro degli sviluppatori.

RINGRAZIAMENTI

Volevo porgere un sincero ringraziamento al mio relatore, Prof. Carlo Ferrari, per la Sua costante disponibilità e per l'interesse mostrato verso gli argomenti sviluppati in questo mio lavoro. Sicuramente questi fattori hanno contribuito in modo significativo al raggiungimento di un risultato finale migliore di quanto potessi inizialmente auspicare.

Un grazie anche a tutti i miei colleghi laureandi che come me hanno affrontato, e alcuni affrontano tutt'ora, questa sfida. In particolare ringrazio il mio collega e amico Francesco Fassina che più di una volta mi ha dato una mano a capire e risolvere i problemi che si venivano a presentare durante la stesura della tesi e l'analisi delle due piattaforme.

Ringrazio ovviamente i miei genitori Ivano e Lorella, i miei nonni e tutti i miei parenti e amici per il loro vivo ed incessante sostegno alla mia causa universitaria: hanno saputo consigliarmi ed incoraggiarmi nei momenti più difficili ed è anche merito loro se sono riuscito a raggiungere questo traguardo. Un ringraziamento speciale va infine a Letizia, per avermi accompagnato moralmente e con pazienza in questo lungo e difficile cammino.

Grazie a tutti quanti.

BIBLIOGRAFIA

[1] Sito web ufficiale della piattaforma Jadex
http://jadex-agents.informatik.uni-hamburg.de

[2] Sito web ufficiale della piattaforma Jason
http://jason.sourceforge.net

[3] Sito web ufficiale della piattaforma JADE
http://jade.tilab.com/

[4] F. Bellifemine, G. Caire, D. Greenwood
Developing Multi-Agent Systems with JADE
Ed. John Wiley & Sons, Ltd

[5] R. H. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni
Multi-Agent Programming: Languages, Platforms and Applications
Ed. Springer

[6] R. H. Bordini, J. F. Hübner, M. Wooldridge
Programming Multi-Agent Systems in AgentSpeak using Jason
Ed. John Wiley & Sons, Ltd

[7] D. Hales, B. Edmonds, E. Norling, J. Rouchier
Multi-Agent-Based Simulation III
Ed. Springer

[8] A. M. Uhrmacher, D. Weyns
Multi-Agent Systems: Simulation and Applications
Ed. CRC Press

[9] M. Wooldridge
An Introduction to Multi-Agent Systems
Ed. John Wiley & Sons, Ltd

[10] M. Mantione
Analisi delle prestazioni di sistemi ad agenti mobili per la rimodulazione dell'organizzazione e dei comportamenti
Tesi di laurea dell'Università di Padova, A.A. 2006-2007

[11] E. Mangina
Review of Software Products for Multi-Agent Systems
http://www.agentlink.org/resources/software-report.html

[12] M. Bratman
Intentions, Plans and Practical Reason
Harvard University Press, Cambridge, Massachusetts, 1987

[13] A. Rao, M. Georgeff
BDI Agents: From Theory to Practice
In V. Lesser, *Proceeding of the First International Conference on Multi-Agent Systems (ICMAS'95)*
The MIT Press, Cambridge, Massachusetts, 1995

[14] M. Huber
JAM: A BDI-Theoretic Mobile Agent Architecture
In O. Etzioni, J. Müller, J. Bradshaw, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99)*
ACM Press, New York, 1999

[15] L. Braubach, A. Pokahr, D. Moldt, W. Lamersdorf
Goal Representation for BDI Agents Systems
In *Proceedings of the Second Workshop on Programming Multi-Agent Systems: Languages, frameworks, techniques, and tools (ProMAS04)*, 2004

[16] P. Busetta, N.Howden, R. Rönnquist, A. Hodgson
Structuring BDI Agents in Functional Clusters
In N. R. Jennings, Y. Lespérance, Intelligent *Agents VI, Proceedings of the 6th International Workshop, Agent Theories, Architectures, and Languages (ATAL'99)*
Ed. Springer, 2000

[17] A. Pokahr, L. Braubach, W. Lamersdorf
Jadex: Implementing a BDI-Infrastructure for JADE Agents
EXP – In Search of Innovations, 2003

[18] M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, F. Velez
The Object Data Standard: ODMG 3.0
Morgan Kaufmann Publishers Inc., 2000

[19] T.O. Paulussen, N. R. Jennings, K. S. Decker, A. Heinzl
Distributed Patient Scheduling in Hospitals
In G. Gottlob, T. Walsh, *Proceedings o0f the Eighteenth International Joint Conference on Artificial Intelligence(IJCAI-03)*
Ed. Morgan Kaufmann, 2003

[20] T. O. Paulussen, A. Zöller, A. Heinzl, A. Pokahr, L. Braubach, W. Lamersdorf
Dynamic Patient Scheduling in Hospitals
In M. Bichler, C. Holtmann, S. Kirn, J. Müller, C. Weinhardt, *Coordination and Agent Technology in Value Networks*
GITO, Berlin, 2004

[21] M. Harbeck
BDI-Agentensysteme auf mobilen Geräten
University of Hamburg, Germany, 2005

[22] M. Beelen
Personal Intelligent Travelling Assistant: a distributed approach
Delft University of Technology, 2004

[23] A. S. Rao
AgentSpeak(L): BDI agents speak out in a logical computable language
In W. Van de Velde, J. Perram, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*
Ed. Springer-Verlag, 1996

[24] M. Wooldridge
Reasoning about Rational Agents
The MIT Press, Cambridge, MA, 2000

[25] R. H. Bordini, J. F. Hübner, et al.
Jason: A Java-based AgentSpeak interpreter used with SACI for multi-agent distribution over the net
Manuale, versione 0.9.5, Febbraio 2007

[26] D. Ancona, V. Mascardi, J. F. Hübner, R. H. Bordini
Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange
In N. R. Jennings, C. Sierra, L. Sonenberg, M. Tambe, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004)*
ACM Press, 2004

[27] R. H. Bordini, A. L. C. Bazzan, R. O. Jannone, D. M. Basso, R. M. Vicari, V. R. Lesser
AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling
In C. Castelfranchi, W. L. Johnson, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002)*
ACM Press, 2002

[28] J. Doran, N. Gilbert
Simulating societies: An introduction
In N. Gilbert, J. Doran, *Simulating Society: The Computer Simulation of Social Phenomena*
UCL Press, Londra, 1994